UNIVERSITY OF COPENHAGEN

This thesis has been submitted to the PhD School of The Faculty of Science,

University of Copenhagen

MEOW

Enabling Dynamic Scheduling of Scientific Analysis

David Marchant

`d.marchant@ed-alumni.net`

Supervisors: Brian Vinter, Kenneth Skovhede, James Avery

May 31st, 2021

ACKNOWLEDGEMENTS

First and foremost I would like to thank Brian Vinter, James Avery, and Kenneth Skovhede who have each acted as supervisors during this project. Throughout my time at university I have heard constant horror stories of supervisors who were never seen or offered no help, and I am glad to say that I can share in none of these. You have each provided consistent, valuable guidance without which this project would definitely not have succeeded.

I am deeply indebted to Rasmus Munk, with whom I have worked almost constantly. More than anyone you have helped guide me through life in Denmark and have been extremely patient in putting up with my distracting comments in our time working together. I would also like to thank the other members of the eScience group, Carl Johannes-Johnsen, Alberte Thegler, Rene Lowe Jacobsen, Jonas Bardino, and Martin Rehr. Each of you have been so welcoming and made the eScience group a wonderful place to work.

I would also like to thank Erik Lauridsen and Hans Fangohr, who supervised my secondments to Xnovo and EuXFEL respectively. You and your teams were each a pleasure to work with. Jon Kerridge and Kevin Chalmers also deserve thanks for helping me find and accept this PhD position, without your input I'd still be in Scotland.

Special thanks should also go to my friends Corrie Gibb, Alex Kiker, Vic Hutchinson, Angus Ruddick, and Angus Barker, along with my family Iain, Clive and Diana. Thank you all for your help and support, I've really appreciated the regular contact despite moving to a new country.

Finally I would like to thank Laura Murray and TJ Marsden. The help and support you've both provided through thick and thin has been invaluable to me. Every difficulty I've encountered you've helped me overcome, and I truly am indebted to you both. Thank you.

This project is developed as part of Multiscale, Multimodal and Multidimensional imaging for Engineering (MUMMERING)[66]. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604.

ABSTRACT

To manage complex scientific data processing, the concept of workflows has been adapted from the world of business. This is not a recent innovation, and so a vast range of different tools have arisen to the point that all manner of specialised needs and use-cases can be accommodated by one tool or another. However, one requirement that has been recently identified as lacking in the current crop of workflow management tools is the need to be adaptable at runtime. This could be for a variety of reasons, such as the exploratory nature of scientific workflows, error handling, or human-in-the-loop interactions.

This lack of dynamic support is caused by most workflow systems being built in a static, top-down paradigm where all of the constituent parts of a workflow are identified and scheduled before any processing takes place. Therefore, to meet the need to be dynamic a new, bottom-up paradigm of scientific analysis is proposed. This new system is known as Managing Event Oriented Workflows (MEOW), and uses file system events to schedule scientific analysis on a continuous basis. An implementation is provided within the Python package *mig_meow*. This provides definitions for a variety of MEOW constructs, as well as widgets for use within Jupyter Notebooks. The aim of this is to make an accessible system for new users to manage their analysis, as well as provide a variety of provenance about whatever processing has occurred. MEOW is designed to work primarily with the Minimum intrusion Grid system to enable shareable, repeatable, and completely dynamic scientific analysis. However, it is also capable of working in a reduced manner as an independent system in its own right.

As well as MEOW, a variety of supplementary teaching materials are presented to make the learning of new users easier. Supporting work, such as an investigation into converting between Static and Dynamic workflows is also presented, as is a system for integrating remote cloud resources into existing scientific applications.

Ultimately this thesis acts as support to the scientific work of researchers, by providing a tool for automating large amounts of scientific processing in an adaptable manner.

RESUMÉ

Workflow konceptet blev introduceret med inspiration fra erhvervslivet for at kunne håndtere kompleks videnskabelig databehandling. Dette er ikke en ny opfindelse, hvilket medfører at der allerede eksisterer en lang række workflow værktøjer, der hver især dækker forskellige behov og brugstilfælde. For nylig er der blevet identificeret særligt ét krav i de foreliggende værktøjer til styring af workflow, hvilket er kravet om at kunne tilpasse sig undervejs i workflow eksekveringen. Dette krav kan opstå på baggrund af adskillige årsager, såsom en udforskende tilgang i videnskabelige workflows, fejlhåndtering, eller human-in-the-loop interaktioner.

Manglen på dynamisk understøttelse skyldes, at de fleste workflow-systemer bygges i et statisk top-down paradigme, hvor de enkelte dele identificeres og planlægges før databehandling finder sted. Vi designer og implementerer et nyt system til at dække dette dynamiske behov. Systemet benytter sig af et bottom-up paradigme til at beskrive og håndtere videnskabelige analyser. Dette nye system bliver kaldt Managing Event Oriented Workflows (MEOW) og gør brug af filsystemhændelser til løbende at planlægge videnskabelige analyser. En implementering gøres tilgængelig i Python pakken `mig_meow`, der eksponerer definitioner for adskillige MEOW-konstruktioner såvel som widgets der kan benyttes i Jupyter Notebooks. Hensigten er at lave et forståeligt system for nye brugere til at administrere deres analyser og levere en oversigt over hvilke databehandlingsopgaver der er blevet foretaget. MEOW er primært beregnet til at interagere med Minimum intrusion Grid (MiG) systemet, for at muliggøre reproducerbar og helt igennem dynamisk videnskabelig analyse, som kan deles mellem brugere og organisationer. MEOW kan anvendes som et selvstændigt værktøj i en reduceret form.

Udover MEOW præsenteres der også en vifte af supplerende undervisningsmateriale, som er designet til at gøre det lettere at undervise nye brugere i at benytte MEOW. Hertil følger en undersøgelse for hvordan man kan omlægge mellem statiske og dynamiske workflows, samt et system til at integrere eksterne cloud-ressourcer ind i eksisterende videnskabelige programmer.

Ultimativt fungerer den her afhandling som støtte til forskeres videnskabelige arbejde, ved at præsentere et værktøj til at automatisere store mængder videnskabelig databehandling med en tilpasningsdygtig tilgang.

THESIS OUTLINE

This thesis is presented as part of the final submission for the PhD of David Marchant at Niels Bohr Institute (NBI), part of Københavns Universitet(University of Copenhagen) (KU). The main piece of work it presents is Managing Event Oriented Workflows (MEOW), a framework for designing inherently dynamic analysis systems, and `mig_meow`, an implementation of that framework. This is a novel approach, and differs significantly from other common tools for running scientific analysis which usually rely on static workflows. Other notable work in this thesis is the development of `corc`, a tool for allowing cloud scheduling to be integrated into existing scientific applications.

This thesis has been broken down into several parts. The main ones are:

- Part i introduces the aims and objectives of the thesis. MEOW as a design framework is introduced and explained.
- Part ii addresses the implementation of `mig_meow`. A users ability to run their own scheduling system is described, using either the MiG or an inbuilt `WorkflowRunner`.
- Part iii illustrates the functionality and correctness of MEOW and `mig_meow` through a number of examples and tests.
- Part iv considers the other work relevant to the thesis not already covered. This includes integrating a cloud orchestration tool into existing x-ray simulation software, a converter between the Common Workflow Language and into MEOW, as well as several teaching materials developed during the project.
- Part v bring together and evaluates all of the work within the thesis. This is also where the successes and failures of the project are identified, and a number of use cases are put forward for where I expect MEOW to be most useful in the future.

CONTRIBUTIONS

This section lists the various contributions made to science outlined within the thesis. Each is briefly summarised, along with details on where to find out more. The core contributions of this thesis are:

1. **MEOW** - Created a framework for designing an event-driven scheduling system for scientific analysis. Users define Patterns and Recipes to identify processing to be perform, and events under which said processing should be undertaken. This processing can then trigger further processing in a non-predetermined manner to achieve an extremely dynamic structure. See Part i.

2. **`mig_meow`** - Created a python package for designing MEOW systems. These can be run locally on a users machine or on the Minimum intrusion Grid (MiG), a grid computing manager. Also contains a variety of Jupyter Notebook widgets to assist in MEOW construct creation. See Part ii.

3. **MEOW on the MiG** - Added support for MEOW constructs to the MiG, along with the ability to take provenance reports. Completed in collaboration with Rasmus Munk. See Part ii.

The following additional significant contributions were made in support of the above listed core contributions:

1. **'Managing Event Oriented Workflows'** - Wrote a paper first outlining MEOW and `mig_meow`, along with a use case from MUMMERING. Completed in collaboration with Rasmus Munk, Elise Brenne and Brian Vinter. Presented by me at the XLOOP workshop[99], part of SC20[85]. See Appendix A.

2. **'Further Developments in Event Oriented Workflows'** - Wrote a paper describing the additional developments of the `WorkflowRunner`, and provenance reporting through the MiG. Completed in collaboration with Rasmus Munk, and Brian Vinter. See Appendix B.

3. **CWL to MEOW** - Created a translator between CWL workflows and MEOW, and vice versa. See Chapter 21.4.

4. **MEOW teaching materials** - Wrote teaching materials for MEOW and `mig_meow`, both for workshops and for self-study. See Chapter 20.6.

5. **`corc` with McStas** - Integrated `corc`, a tool for scheduling processing on cloud resources, into McWeb, a GUI for use with both McStas and McXtrace, x-ray and neutron simulation software. Completed in collaboration with Rasmus Munk. See Chapter 23.4.

6. **'Cloud enabling educational platforms with corc'** - Co-wrote a paper outlining the use of `corc` in an academic context. Completed in collaboration with Rasmus Munk and Brian Vinter. See Appendix D.

7. **Supervision** - Supervised Masters Project of Niels Andreas Tyndeskov Voetmann. Completed in collaboration with Kenneth Shovhede, Carl-Johannes Johnsen. See Part 22.3.

NOTATION WITHIN THIS DOCUMENT

Throughout this thesis, some consistent notation has been adopted so as to help make reading it easier. These notation decisions have been highlighted here.

- Programming constructs have been highlighted in a console font. For example, '`copy`' would refer to a library known by that name, whilst 'copy' would refer to the act of replicating something.
- The MEOW constructs Patterns and Recipes are always referred to with capital letters to highlight that they are specific definitions. When they are implemented within `mig_meow` they are referred to as `Patterns` and `Recipes` as these are an implementation of the abstract definitions.
- Within code samples, ellipsis are used to show where code has been cut for brevity, but where it is important to show that some processing is taking place in that spot.
- When the contents of a Jupyter Notebook is being displayed, only the contents of the code cells will be shown. This is as none of the Jupyter Notebooks shown in this thesis have significant markdown contents, and the non-code contents can effectively be ignored.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

corc Cloud Orchestrator. v, vi, 110, 121–123, 125, 150, 243, 244

AEC2 Amazon Elastic Cloud Compute. 122
API Application Programming Interface. 41
ASIC Application-Specific Integrated Circuit. xix
AWS Amazon Web Services. 121

CPU Central Processing Unit. 39, 139
CSP Communicating Sequential Processes. 49–52
CWL Common Workflow Language. v, vi, 110, 115–118, 125, 241, 242

DAG Directed Acyclic Graph. xx, 5, 14–21, 23–25, 32, 35, 67, 68, 77, 79, 108, 116, 128, 149
DTU Danmarks Tekniske Universitet(Technical University of Denmark). 123

EBNF Extended Bachus-Naur Form. 27, 34
ESR Early Stage Researcher. 3, 4, 6, 97, 129, 147

FPGA Field-Programmable Gate Array. xix
FUR Framework for Uploading research data. 110, 119–121, 123, 125, 129, 135, 140, 141, 146, 150

GPU Graphics Processing Unit. 4, 39, 140
GUI Graphical User Interface. vi, 14, 15, 18, 24, 25, 123

HDF5 Hierarchical Data Format 5. 13, 120, 135, 140, 146
HITL Human-in-the-loop. 9
HPC High Performance Computing. 3, 4, 121, 130, 139
HTTP HyperText Transfer Protocol. 64, 65

ITN Initial Training Network. 3

JSON JavaScript Object Notation. xix, 28, 45, 58, 64, 65, 94

KU Københavns Universitet(University of Copenhagen). v, 5, 6, 37

McStas Monte Carlo Simulation of Triple Axis Spectrometers. vi, xvii, 121, 123
McWeb Not a true acronym, but a web UI for running McStas and McXtrace. vi, 110, 123, 125
McXtrace Not a true acronym, but the X-ray version of McStas. vi, xvii, 121, 123
MEOW Managing Event Oriented Workflows. v–vii, xii, xx, 2, 16, 17, 22, 26–35, 37, 38, 40–44, 46, 49, 56, 58–67, 69, 71–73, 76–80, 85, 87, 88, 90, 93–99, 101, 103–108, 110–120, 123, 125, 127–142, 144–150, 174, 221, 234, 239, 240, 245
MiG Minimum intrusion Grid. v, vi, xx, 5, 37–40, 42, 44, 46, 49, 53, 55, 58–66, 69, 74, 77–79, 85, 87, 91, 93–95, 97–99, 101, 104–108, 110, 114, 116, 119, 120, 123–125, 128–131, 133–135, 139–142, 144–147, 149, 150, 221, 222, 234, 239, 240
MPI Message Passing Interface. 17
MUMMERING Multiscale, Multimodal and Multidimensional imaging for Engineering. ii, vi, xix, 3–6, 40, 97, 114, 129, 147, 149, 218, 219

NBI Niels Bohr Institute. v

OCI Oracle Cloud Interface. 121, 122, 217

PyPI Python Package Index. 42

regex REGular EXpression. 28, 41, 43, 105

SFTP Secure File Transfer Protocol. 62
SSH Secure SHell. 63, 95, 145
SSHFS Secure SHell File System. 62–64, 66
SWMS Scientific Workflow Management System. 5, 6, 9–14, 16–18, 20–23, 32, 34, 35, 56, 63, 69, 96, 99, 114, 115, 127–130, 134, 142, 150

UI User Interface. xvii, 13, 16, 123
UNICORE UNiform Interface to COmputing REsources. 18
URL Unique Resource Locator. xix, 64

WP2 work package 2. 3, 4, 6, 218, 219

YAML YAML Ain't Markup Language. 48, 54, 57, 61, 73, 90, 92, 115, 117, 122, 123

GLOSSARY

checksum A mechanism to validate if data has been modified or is inconsistent with a previous state. Some algorithm is used to reduce a large amount of data down to a single number or string, with the same data input always resulting in the same result. If different data is used a different result will be produced, so this can be used as a quick check on if data has changed between calculations. 12

cURL A command line tool for transferring data between resources, using a URL syntax. 39

deadlock A term used with concurrent and parallel computing for a multi-process program that cannot progress as all processes require action from some other process. In this scenario, the system will never progress and is stuck as no part is able to act first. xix, 11, 13, 49–52, 55

dynamic A term used within this thesis to refer to workflows not created from a pre-defined structure. Instead the workflow is an emergent property of a series of connected jobs. v, vi, 2, 5, 6, 9–12, 16, 19–23, 25, 28, 34, 35, 37, 42, 56, 59, 69, 80, 95, 107, 108, 111–113, 115, 116, 118, 119, 125, 128, 137, 142, 145, 149, 150

FPGA Stands for Field-Programmable Gate Array. An integrated circuit board comprised of a collection of configurable logic blocks. These can be reconfigured repeatedly after manufacture in contrast to dedicated hardware such as an ASIC. 139, 140

job An singular piece of processing, either done in isolation or as a constituent part of a step, and the atomic part of a workflow. A job is collection of processing code, along with the necessary input data for it to be processed. Output is expected to be produced from this analysis. As an atomic unit of the workflow, each job can be completed in isolation, though their inputs and outputs are often managed according to steps. xix, xx, 5–9, 11, 12, 14, 16–21, 23–28, 32–35, 38–40, 43, 44, 46, 49, 54–56, 58, 61–68, 72, 76–79, 84–88, 91, 92, 94, 95, 97–102, 104, 106–108, 110, 115, 116, 118, 122, 123, 128–134, 136–142, 144–146, 149, 150, 221, 222, 234–240

Jupyter Notebook A JSON document that can express a mix of runable code, text, and rich media. Allows for very easily sharing and presentation of scientific data. Execution environments can be provided in many languages, most notably in Python. Can also refer to older Jupyter implementations, that have now been superseded by JupyterLab and so within this thesis Jupyter Notebook will only refer to the documents themselves. vi, vii, xix, xx, 28–31, 37, 41, 45, 46, 54–56, 58, 61, 62, 64, 65, 69, 73, 75, 77, 79, 82, 88–90, 92, 94–96, 129, 134, 135, 141, 144, 147, 149, 220, 223, 224, 226, 227, 229

JupyterHub A server service for spawning instances of Jupyter Notebook or JupyterLab. 40

JupyterLab A web based environment and interface for hosting a number of Jupyter Notebooks. Allows for easy extension of the interface through the use of plugins. xix, 29, 30, 40, 41, 64, 77, 94

livelock Both similar and opposite to deadlock. livelock occurs when a multi-process program will not progress as each process is consistently giving up control to other processes. In this scenario, the system will never progress as no part is willing to act first. xix, 49–51

Marie Skłodowska-Curie Actions An EU funding scheme for researchers, with a focus on fostering international collaboration between different universities, research facilities and private companies. It is most relevant here as the sole funder of this project as part of the MUMMERING project. 3

parameter sweep A common scientific use case, where multiple otherwise identical jobs are required, each with a different instance across a range of parameter values. 44

Pattern A MEOW workflow construct. Defines the conditions under which processing should be triggered. The is done by defining a path against which file events are matched. Also defines the parameters for the processing itself, including which Recipe is to be used. vi, vii, xx, 26–29, 31–34, 38, 42, 45, 46, 48, 53, 56–66, 68, 69, 72–77, 79–81, 83–85, 88–91, 95, 101, 102, 106, 115–117, 120, 129, 131–137, 144–149

race condition A problem within multi-processed systems where an invalid state is reached by multiple processes acting on the same data at the same time. Each process acts as though they alone are using the data, and so the final result does not account for the multiple processes that have. 11, 13, 45, 49, 50, 65, 130, 131, 142, 150

Recipe A MEOW workflow construct. Defines the processing itself. This is done within `mig_meow` using Jupyter Notebooks. vi, vii, xx, 26–34, 38, 42, 43, 45, 46, 48, 53, 56–62, 64–66, 68, 69, 72, 73, 75–77, 79, 81, 83–85, 88–92, 95, 96, 101, 117, 129, 131–134, 136, 137, 139, 141, 144, 146–149

Rule A MEOW workflow construct. Not directly made by a user, a Rule is created when a Pattern is registered, along with the Recipe stated in said Pattern. It is the list of currently created Rules against which events are compared, and in the event of matches that jobs are scheduled. xx, 31–34, 53, 54, 59, 74–79, 92, 101, 102, 105, 131, 146

static A term used within this thesis to refer to DAG based workflows, where the structure is defined ahead of time and kept to through the workflow run. v, 2, 10, 19, 21, 23, 24, 32, 35, 56, 67, 87, 108, 110, 112–115, 117, 125, 128, 129, 137, 149, 150, 174

step A constituent part of a workflow. Each step represents one phase of the processing. It may consist of one, or more jobs scheduled in parallel. Each job scheduled as part of the step must have the same processing code, though may have different input parameters and so may produce different, but related output. xix, xx, 7, 9, 13–16, 18, 19, 23, 24, 27, 34, 35, 67, 69, 79, 81, 87, 115–117, 129, 133, 137, 140

sub-workflow A constituent part of a workflow that is itself a workflow. 15, 16

tomography The science of non-destructively imaging a 3D object by taking a series of 2D slices through the object using penetrating waves, most commonly X-rays. These slices can then be reassembled into a 3D model to be used in further scientific analysis. 3–6, 8, 9, 26, 80, 87, 110, 137, 147–150

VGrid Legacy term for a workgroup. xx

workflow An ordered progression of analysis conducted on data. Made up of one or more steps. v, vi, xix, xx, 2–25, 27, 34, 35, 37, 42, 46, 49, 59, 67, 69, 87, 107, 110–119, 128–130, 132, 136–138, 142, 146, 149, 150, 241, 242

workgroup A term within the MiG for a structure within the file system. This is a shared location in which numerous users can enrol and share data, or access to resource. Membership can be controlled and so workgroups are an effective way for projects, departments or just individuals to organise their work. Also sometimes referred to as a VGrid. xx, 39, 53, 59, 61–67, 74, 77, 78, 85, 87, 96, 132, 145

Part I

INTRODUCTION AND DESIGNING MEOW

1 INTRODUCTION

This is the first of the three main parts of this thesis. In it, we will establish the motivations and objectives for the thesis as a whole. We then go on to discuss the background and theoretical underpinning for the work presented in the subsequent two parts. By the end of this part, we will have completed the necessary research into the first objective of this thesis, *'design a framework to express automated, dynamic workflows'*(see Section 2.2).

The main work presented in this thesis is Managing Event Oriented Workflows (MEOW), and is fundamentally created as a response to a perceived deficiency in existing scientific workflow management tools. As such, we will start by considering what a scientific workflow is, and what currently available tools there are for managing them. These tools will be considered in the context of a series of requirements that have been set out by a number of authors, with slight modifications made and presented here. The reason for these modifications is that a shortcoming has been identified in many of these tools, that being that they are static.

Static here means that any workflows designed in these systems is difficult to adapt at runtime, and so a new system is needed that better accommodates change. In other words, we need a workflow system that is dynamic. Exactly what this means, and how we might achieve this is outlined at length in the later half of this part. In addition, some of the initial implementation work is presented so that we may better understand the final requirements set out for a dynamic system. By the end of this part, readers should have a full understanding of why MEOW was designed, what context it is operating in, and what requirements we will be using to judge its success in later parts.

2

MOTIVATION AND OBJECTIVES

2.1 AN INTRODUCTION TO MUMMERING

The work presented within this document is presented as part of the Multiscale, Multimodal and Multidimensional imaging for Engineering (MUMMERING) project[66]. MUMMERING is an Initial Training Network (ITN), part of the European Union's Marie Skłodowska-Curie Actions for funding research and fellowships. As such, any discussion of objectives should first be put in the context of MUMMERING, and its motivations. As stated on the front page of the mummering.eu website, the MUMMERING mission statement is:

> *'The overarching goal of MUMMERING is to create a research tool that encompasses the wealth of new 3D imaging modalities that are surging forward for applications in materials engineering, and to create a doctoral programme that trains 15 Early Stage Researchers (ESRs) in this tool.'*

MUMMERING is a tomography focused ITN, with positions offered to improve certain aspects of data retrieval or analysis. Even within the relatively small area of tomography, it is still possible to be highly specialised, as demonstrated by the fifteen ESR positions available as part of MUMMERING. These specialists were divided into five different work packages, as shown in Figure 2.1.

I was part of work package 2 (WP2), titled *'Data Management and HPC'*. This work package was intended to support the other work packages by developing and providing a platform for large amounts of data analysis and storage in a collaborative, robust manner. This is as data from scientific experiments has been increasing exponentially within the last decade[104]. For example, CERN[13] is already producing more than 115 PB annually[15]. Processing this volume of data demands the

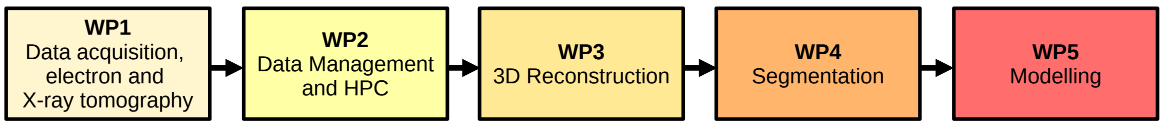


Figure 2.1: The MUMMERING work packages, and where they relate to a tomography workflow. Note that work packages 6-10 are not shown as they were administrative in nature.

use of dedicated hardware, such as GPUs, clouds, or grids. This can in turn lead to a demand for specialist HPC knowledge and so WP2 is tasked to provide a platform allowing for others within MUMMERING to access and use dedicated hardware without needing an obstructive amount of previous knowledge.

As well as providing access to HPC resources, the platform should foster collaboration. That even a relatively small subject like tomography can be divided into fifteen distinct ESRs demonstrates a need for specific tools to support collaboration amongst tomography specialists. This demonstrates a need for tools to support collaboration amongst tomography specialists, so that they can share their data, work and results effectively. Although not a concern at the start of the project, in light of Covid-19 and the frequency with-which researchers are being asked to work from home, this is only becoming a more urgent need.

It is also worth noting that collaboration is not just conducted between individual researchers. It can also be between institutions and organisations such as is frequently the case in MUMMERING. For instance, several universities, research facilities and private companies all contribute in some way to MUMMERING, providing support for ESRs, research, resources, data, and/or teaching. Therefore, these tools should allow specialists across disciplines and institutions to share their data, work and results effectively.

The use of such a platform should not be constrained to just MUMMERING however. For instance, facilities such as CERN, EuXFEL, or MAX IV[57] provide experimental apparatus to external researchers. These researchers may only be physically present at the facility relatively briefly, but will require access to data or processing for some time, perhaps even remotely. In light of this, it should be taken that these needs for greater collaboration and easier HPC access is not unique to MUMMERING or tomography, and is expected to be broadly applicable to any scientific field. Therefore, a central motivating factor in WP2 was to enable easy collaboration across specialisms and organisations.

2.2 OBJECTIVES FOR MY PROJECT

At its inception, this project and its goals were set out in the MUMMERING grant agreement, with the two relevant to me being:

> *Deliverable 2.1: Basic workflow framework, expected month 18.*
> *Deliverable 2.2: Training Materials for the workflow, expected month 20.*

The expected resultant work from these deliverables is summed up within the MUMMERING grant agreement as Task 2.1, *'Automating data analysis through workflows'* and Task 2.3 *'Total data management'*. These tasks will now be summarised, though a full description of both as presented in the MUMMERING Grant Agreement and are shown in F. Task 2.3 is to provide a platform capable of supporting big data analysis and storage in a collaborative manner, such as the Minimum intrusion Grid (MiG). This is a grid management solution for, among other things, managing compute resources at KU. As the university gains more resources, they can be enrolled in the MiG to be used either for storing, processing data or both. This has proven to be an effective way of granting users access to all manner of specialised hardware, as well as assisting them in their collection, management, processing and storage of large amounts of data. A one stop-shop for storage and processing is especially necessary thanks to stories like the one occasionally told story within the eScience group. This concerns scientists running long-running, expensive experiments at facilities such as CERN or MAX IV that produced terabytes of data. This data would then be retrieved from the facility on a collection of hard disks or USB drives, that would be simply carried home in a plastic bag which does nothing to ensure it was kept securely, completely, or correctly. A solution that would remove this would be highly sought after. For instance, letting users access their data at their home machine whilst keeping it within a grid system such as the MiG would be an effective solution to this, and a highly motivating example for this work.

In addition to this, Task 2.1 is to automate the creation and execution of scientific analysis that requires several individual stages of processing. This is the case within tomography, but is broadly applicable to any complex scientific analysis. To aid in this processing researchers will frequently use scientific workflows, with them being managed by a Scientific Workflow Management System (SWMS). Commonly, they use a Directed Acyclic Graph (DAG) to define a workflow, with the nodes representing jobs and edges dependencies between them. This dependency can be many things, but most commonly it is a data dependency such as the input to one job depending on the output of another. This is elaborated in more depth in Section 4.4. For now we can content ourselves with the knowledge that a Scientific Workflow Management System (SWMS) derives from the DAG all of the constituent jobs and what resources they should be mapped to before any processing has taken place. This is sufficient for many workflows, though in some scientific use cases there is a specific need for the workflow structure to be dynamic[56], [102].

Of note here, is that prior to this project event-driven triggers were added to the file system within the MiG. This allows for some MiG interactions to be automated following defined file events[7] using a construct called a `Trigger`. A feature of these `Triggers`, was that it allowed for new jobs to be scheduled on the MiG. These `Triggers` could be used for a new workflow system that was much more dynamic than a traditional, DAG-based SWMS. This is as it would be possible to use the `Triggers` to schedule a job which produces output directly back into the MiG file system, which could in turn trigger further events. Several jobs could then combine to form a complete workflow.

Objective 1	Design a framework to express automated, dynamic workflows
Objective 2	Implement an automated, dynamic workflow system
Objective 3	Integrate the automated, dynamic workflow system into the collaborative big data platform developed within WP2
Objective 4	Create training material for the automated, dynamic workflow system

Table 2.1: Core thesis objectives. These objectives are listed approximately chronologically, and so this should not be taken as an order of priority.

Where this differs from the traditional SWMS is that each individual job is scheduled in isolation, and so can be scheduled, skipped, cancelled, or modified at any time without it affecting other jobs. This would be a very good basis for a workflow that could meet the needs set out in [56] and so was one of the core motivations for my work.

Of course I was not the only ESR within WP2, Rasmus Munk was also a part of it and also based at KU. His research was derived from Task 2.2, to create a framework big data platform on which to host the workflow system, whilst I took Task 2.1 as my own starting point. Meanwhile both of us worked on Task 2.3, to integrate the results of Tasks 2.1 and 2.2 together. Whilst the cold reality of funding a project through the EU meant that these deliverables and tasks were unmovable, they were ultimately only a framework for my work. Following the initial investigations and background research that will be discussed in Chapter 3.2, I aimed to answer the following core research question:

> ***Is it feasible to create a tool for the automatic creation of dynamic scientific workflows, available within a big data capable platform?***

In order to properly answer this broad question, four core research objectives for this thesis were derived, as shown in Table 2.1.

2.3 SUMMARY

This work exists within the MUMMERING project, which aims to create a tool for facilitating tomography analysis. Within that aim, my project aims to create a tool for the automatic creation of dynamic scientific workflows, available within a big data capable platform. As will be explained in the following chapters, such a system does not currently exist outside of this project, so this will be an active investigation into the feasibility of such a system. This research goal was also expressed as four core objectives, shown in Table 2.1. These objectives were inspired by the initial MUMMERING Grant Agreement, but filtered through my own research into current workflow management systems and their shortcomings. They will be the criteria against which this project is judged. Therefore we will use this within this thesis to assess the completeness of the final submission.

3

BACKGROUND

3.1 SCIENTIFIC WORKFLOWS

It is no secret that the size of scientific experiments have been increasing exponentially in recent decades[104]. This involves both an increase in the size of data sets, and the amount of processing required during analysis. As of 2019, CERN[13] was storing roughly 330 petabytes (PB) of data from its experiments[14], and generating around 115 PB additionally every year[15]. While CERN is an extreme example, the broader trend is supported by other facilities such as EuXFEL[33], who generated 3 PB of data in 2019 and anticipate an increase to 100 PB in 2023[83]. Similarly, ESRF[31] generated 8 PB in 2019, which is expected to increase to 50 PB by 2023[83].

Managing all of this data and processing it is no small feat. This has led to the adoption of workflows as a concept within science. Workflows are traditionally a business concept, where certain small tasks are linked together to achieve a greater goal. These smaller tasks will usually need to be completed in a certain order, and can almost be repeated by rote. This gives us the traditional taxonomy of a workflow, shown in Figure 3.1. Here we can see that a workflow exists at the top of the hierarchy and is comprised of one or more steps. Each step may be comprised of one or more jobs, which would be the atomic unit of a workflow.

An example of a workflow could be the hiring of new staff. This might consist of posting adverts for a new position, waiting for responses, filtering applications, conducting interviews, and making offers of employment. Each of those items would be a step in the larger hiring workflow. Each step is dependently linked, and cannot be completed out of order. In contrast to the steps, jobs are often not interdependent and can be completed in any order within the step, such as interviewing several candidates. Obviously this workflow may differ wildly from organisation to organisation, but all new hires will need to go through it, and any manager can automatically carry it out it correctly by simply following the steps of the workflow. This property of automation is what has proved useful to

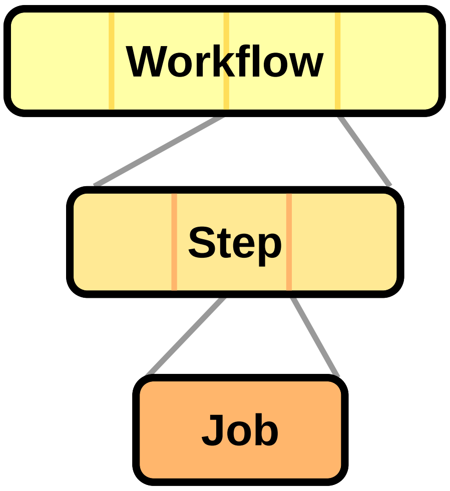


Figure 3.1: Taxonomy of a workflow.

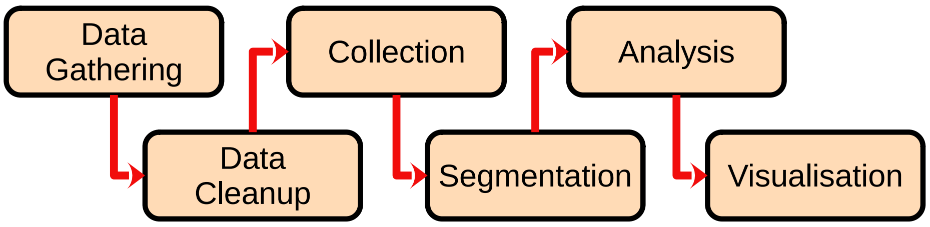


Figure 3.2: A sample tomography workflow.

researchers, who can use workflows to manage the analysis of their data without having to manually set up each individual job on each individual data file.

For instance, within tomography a researcher will usually take multiple scans of a physical object as the starting point for their analysis. Once the scans have been taken, there are still multiple steps that must be undertaken before the researcher can start drawing conclusions and publishing their work. At this point, they can identify a workflow. A series of individual tasks have been dependently linked together, as is shown in Figure 3.2.

Creating such a workflow has many benefits for a researcher. Most obviously, it is a handy way of organising work, and formalising the process undertaken in the same manner as is done in business workflows. It also allows for easier sharing of analysis techniques, both between data sets and between different researchers. Constructing a workflow also allows for the identification of jobs. For instance, in the tomography workflow cleaning each individual raw data file into a usable format would be many jobs, one for each file. This is as each file can be cleaned independently, e.g. they have no dependency on each other.

3.2 A NEED FOR DYNAMIC

Workflows may well be a useful import from the world of business, but there is one key difference in their use in science. Within business they are usually rigid constructs with little need for adaption. This is not necessarily the case with a scientific workflow. Though it may be that certain predictable analysis is sometimes carried out, a far more common characteristic of scientific workflows is that they are exploratory in nature[23]. By this is meant that the researcher running the workflow will not have a complete understanding of the experiment space, and so is running their workflow as an experiment to gain more knowledge.

This exploration of the experiment space means that scientific workflows have the unique need to be dynamic. In the tomography example it may be that a researcher decides their segmentation algorithm needs to be replaced, or that different parameters are needed. In this case any jobs scheduled as part of the segmentation step would need to be replaced. A researcher could also decide that the final visualisation step is not needed, and so those jobs could be cancelled, or that they want to conduct further analysis so more jobs are added.

This is something of a contrived example as a researcher is unlikely to have so many drastic changes during some relatively simple tomography analysis, but many scientists are now requesting more Human-in-the-loop (HITL) interaction[56]. This is where a human become a necessary step within the workflow, such as by identifying significant data sets to examine further, or to continuously monitor output to check results are valid. In this sort of setup jobs may be added, re-run, changed, or removed at any point in the workflow.

A further need to be dynamic is error handling. As in an exploratory workflow we do not yet have a complete understanding of the problem space, there is a much higher likelihood of errors occurring within jobs. There could be a variety of responses here such as, the job being re-run, edited to remove the error, re-scheduled on a different resource, or just forgotten about and ignored. No matter the strategy undertaken to address the error, the workflow will need to be altered at runtime. This need to accommodate change is why the word dynamic was added to the objectives in Section 2.2. Users have made repeated requests for it and modern SWMS are expected to adopt it. As no system seemingly has been made from the ground up to accommodate this new paradigm, it has become a central motivation for this project.

4

RELATED WORK

We have established the importance of workflows to the scientific community, and explained their specific need to be dynamic. We will now begin to look at some of the current tools available to researchers for constructing their workflows. These are usually referred to as a Scientific Workflow Management System (SWMS). Before we get onto examining some of the prominent options currently available, it will be worth first quickly considering what it is that such a system is required to do.

4.1 REQUIREMENTS FOR SCIENTIFIC WORKFLOWS

Just as there are many competing projects to develop SWMS tools, there are also many different suggestions for the formal requirements of such a system. Some of these are intended as a complete list of requirements such as [104], [58], [56] and [81]. Others are intended as more specific requests, outlining individual features they feel are currently missing or under served, such as [56] or [12].

The most complete requirements found were those presented by McPhillips et al in [58]. These requirements cover the general behaviour of a SWMS, as well as many of the edge cases presented in other papers. Though the requirements are stated as applicable to any SWMS, they are presented in the context of the specific Kepler[1] SWMS. As Kepler is designed very much in the static paradigm elaborated in Section 4.2, it does not have any real accounting for the need to be dynamic. Therefore, though the work of McPhillips et al will form the basis of the requirements presented here, they have been modified slightly and other contributions have been brought in. These modified requirements are here presented in alphabetical order, as they are not hierarchical, and should all be adhered to.

The most complete requirements found were those presented by McPhillips et al in [58], though they still have certain limitations. McPhillips et als requirements are stated as applicable to any SWMS, and so cover the general behaviour of a SWMS, as well as many of the edge cases presented in other papers. However, they are presented in the context of the specific Kepler[1] SWMS, and are designed to demonstrate Kepler's utility and completeness. Kepler is designed very much in the static paradigm elaborated in Section 4.2, and so it is unsurprising that the requirements it is shown to meet does not include any significant mention of a need to be dynamic. Regardless, the work of McPhillips et al will form the basis of the requirements presented here, and all of their requirements have been adopted. However, the explicit need to be dynamic, as explained in Section 3.2, has been introduced. Several smaller points, clarifications and influences have also been brought in from [104], [56], [81], and [12] to form a complete list of requirements. These complete requirements are here presented in alphabetical order, as they are not hierarchical, and should all be adhered to equally for a SWMS to be considered complete.

4.1.1 *Automatic Optimisation*

A SWMS should be able to take advantage of the hardware available to it without the user having to possess a deep knowledge of concurrent and parallel computing. For instance, if multiple processing cores are available, the SWMS should be able to automatically identify what jobs can be run in parallel and processes them accordingly. These jobs must be able to complete without then causing concurrent and parallel problems such as deadlock or race conditions. Though a user may be able to provide detailed guidance on how this is done if they are sufficiently expert, it should not be required.

4.1.2 *Clarity*

It should be easy to create workflows using a SWMS, such that the workflow becomes self explanatory. The tools provided to create a workflow should make as much sense as the produced workflow. Clarity should not be confused with confinement however, and a solution here should not be to enable a clear workflow only by so restricting the possible options. This would make for a clear, but overly rigid system and so should be avoided.

4.1.3 *Predictability*

Somewhat related to clarity, it should be easy for a user to be able to anticipate what the output of their workflow will be, before any processing has started. This does not mean that we will know what the outcome itself is, only that they can predict what outcomes will be reached. It should also be clear what data and processing is needed at each stage so that collaborators working on the same workflow can easily identify how their work should link together. It is worth remarking that part of the need for dynamic systems is that sometimes workflows are inherently unpredictable in their outcome.

Nevertheless, the unpredictable or non-deterministic elements should be contained in a predictable manner. For example, it may be that a researcher is running a workflow on a large set of inputs and does not know which inputs are valid. It is unpredictable which jobs for which input will complete, but it is still predictable that jobs will run on each data set and their contents will determine their success or not, rather than the alignment of the sun.

4.1.4 *Recordability*

A SWMS should provide a record of a workflow once it has run. The record should also include any changes made to the workflow at runtime, and what caused those changes. Additional details such as a record of what hardware was used to run the processing, may also be of use. This is especially true due to the requirement for Automatic Optimisation, which can mean that parallel processing is commonly used. This can muddy simple reporting such as logs, with different processes logging interspersed in a manner that can be hard to parse for those not familiar with parallel processing. For this reason any records provided should be presented in a manner that users can easily understand.

4.1.5 *Reportability*

The record of a workflow should allow users to check the scientific validity of their results. For instance, the record should clearly show what processing has taken place on what data, and produced what output from what input. As part of any good science is being able to show where your results came from, and how they were calculated, the SWMS should support this fully. This may also mean providing some form of check on the data through mechanisms such as checksums, to provide users with a way of verifying if data has been modified since the workflow has been completed.

4.1.6 *Responsiveness*

A SWMS should respond accordingly to changes within its state, even as the workflow continues. This can cover a fully dynamic system such as the one presented in this thesis, or simple error handling within a more traditional workflow system. Where such responses are made, they should be made correctly so as change the appropriate part of the workflow without affecting irrelevant parts. For instance, if multiple jobs are run in parallel and one fails, that failed job should be handled without necessarily cancelling or modifying all parallel jobs. The SWMS should also be able to alert a user to important developments such as processing completing or errors being encountered. It is especially important as scientific workflows can run for days or more, and so simply having a UI that requires constant monitoring is not sufficient. The ability to send texts, emails or some other remote notification is therefore required.

4.1.7 *Reusability*

It should be straightforward to use a workflow repeatedly on related problems. This can mean the ability to run it on different input data, or on a different hardware setup. It can also mean that it should be easy to construct a new workflow from an old one, without having to start from scratch each time. Individual elements should be easy to add, modify and remove between runs, or during runs where appropriate.

4.1.8 *Scientific*

Somewhat obviously, though often overlooked in formal requirements is that a good SWMS should support science. This means that it should be possible for researchers to be able to process their data using the SWMS, using data formats, processing techniques and workflow structures they are familiar with. However, in contrast to this it should also take some steps to help ensure good practices in data formatting, processing and workflow structuring. For instance, it should make sure to support commonly used scientific formats such as HDF5[42].

4.1.9 *Well-formedness*

The SWMS should make it easy to create a valid workflow. That is, that as well as being clear, its structure should be sound and avoid common concurrent and distributed problems such as deadlock or race conditions. This structure should also be clear, so that it is easy for a user to see what steps flow into which other steps. This relates closely to requirements for clarity and predictability. Problems within the workflow should also be highlighted, or be easily identifiable.

4.2 CURRENT WORKFLOW MANAGEMENT TOOLS

Unsurprisingly, a large number of tools for constructing and managing scientific workflows already exist. Though many dedicated SWMSs exist, a number of ad-hoc solutions are also commonly used and will also be briefly examined. These workflows are not presented in any particular order, other than to group similarly constructed systems together.

One commonly used system is Apache Airflow[41]. This is a very mature system, with a complex feature set that can be used for either relatively simple workflows, all the way up to hugely complex structures. Though it is not designed primarily to work with scientific problems, it is designed to process large data sets and so is generally appropriate. Apache Airflow is typical of many SWMSs, in that it relies on a user constructing their workflow through the use of one or more Directed Acyclic Graphs (DAGs). A DAG is a type of linked graph, in which nodes are directionally connected such that a loop is never formed. Thus, a DAG is an easy analogue for a workflow, with a defined start and end, and with the dependencies between the different steps trivial to identify. The DAGs themselves are constructed by the user either through a web interface, or programatically. More than one DAG may be provided as individual steps within the workflow may be smaller workflows themselves.

Apache Airflow is also typical of Scientific Workflow Management Systems (SWMSs), in that it uses a data-flow model. This is a method of structuring a workflow and is contrasted with a control-flow model. In a control-flow model, data is kept in place, and control of that data is passed to a series of processes. These processes may make decisions to alter the flow. It is possible that processes do not even modify the data before passing on control. This is analogous to the flow of control within a linear script. In a data-flow model, a pipeline of processing is constructed and data is passed from process to process. Each process will have defined inputs and outputs, and should always perform some modification on the input data. The data-flow model is a very good fit for a DAG based system such as Apache Airflow, and is in fact used by all of the systems talked about in this section.

A number of GUI interfaces are provided by Apache Airflow so that a user can see the overall structure of the workflow and verify how individual steps link together. A number of reports can be generated both before and after the workflow has run, showing how the processing progressed and to get real-time feedback about the status of the workflow. Workflow computation takes place on a number of workers, with a separate scheduler identifying the constituent jobs of the workflow. The overall structure of the system is shown in Figure 4.1. This system is designed to be used in conjunction with a web server, and can be easily integrated with applications such as Kubernetes[51] for the scheduling of jobs on remote resources. Despite this, it is possible to deploy an Apache Airflow workflow on a local machine, with individual processes acting as the server, scheduler and workers.

The ability of Apache Airflow to integrate with remote processing, relatively easy to use workflow construction, and wide ranging reporting and visualisation make it one of the best available systems for workflows operating at scale. It is also easy to see how Apache Airflow meets the requirements set out in Section 4.1, especially regarding the ease-of-use requirements.

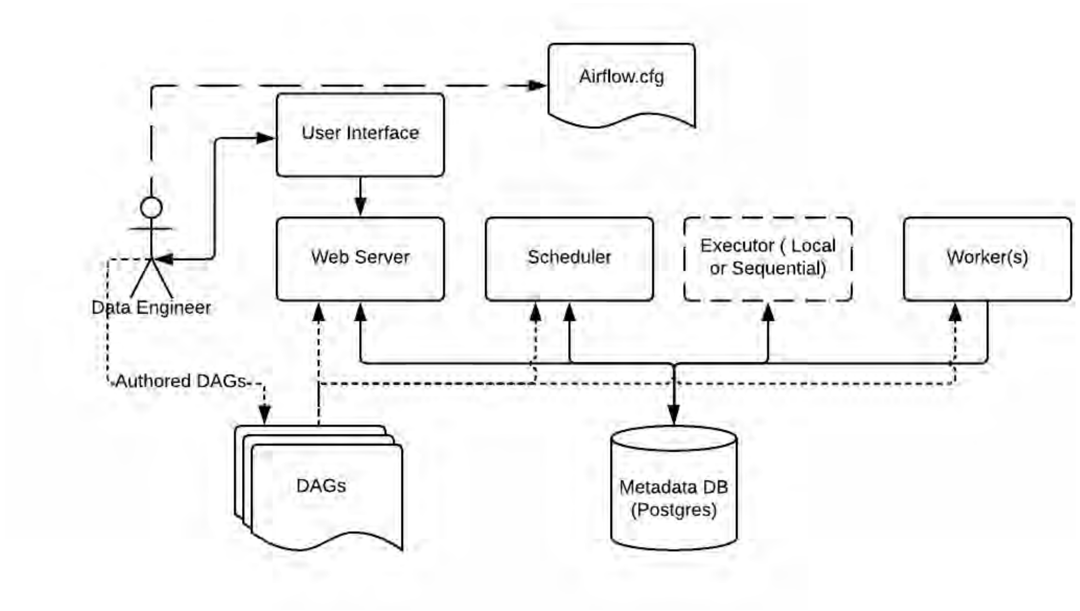


Figure 4.1: Overview of the basic Apache Airflow architecture. This diagram was taken from https://airflow.apache.org/docs/apache-airflow/stable/start.html.

Another commonly used system, and one more specifically targeted at scientists is Kepler[1]. Outwardly, Kepler operates in a similar manner to Apache Airflow, though a user in Kepler will primarily construct their DAGs using a drag-and-drop GUI. This makes for a very intuitive way of constructing workflows through the use of *actors*. An *actor* is a processing component, used to represent a step in a workflow. These can be user defined though over 350 come packaged with Kepler as a quick-start aide. An *actor* can be placed into the GUI, at which point connections can be made to other *actors*, denoting a data transfer. As these *actors* are placed and connected a workflow DAG begins to form. Much like Apache Airflow, processing can be scheduled either locally, or remotely using Kepler's integration with Globus[37], or other available grid technologies.

An important feature of Kepler is that its *actors* can be workflows themselves, allowing for the nesting of workflows. This means that extremely complex workflows can very easily be broken down into digestible, re-usable chunks for easy understanding and re-use. Collaboration is another key feature of Kepler, which also provides a component repository for easy publishing and sharing of workflows. Though Kepler's GUI provides an easy way to construct and view the workflow, it provides far less reporting than is available in Apache Airflow.

Although many workflow systems, such as Keplar allow for sub-workflows, few have specific accommodation for sub-workflows not written in the same system as themselves. An exception to this is the ambiguously named Hybrid Workflows system built on COMPS[6], presented in [102] and [80]. This presents two types of workflow, in-situ and task-based. In-situ workflows are run within a single resource. This resource may be anything up to a supercomputer, but the point is that external data transfer does not occur within the workflow. Task-based workflows however are large, task parallel batches of processing that may take place over all manner of remote resources, or be processed locally. Hybrid Workflows uses Decaf[29] to run in-situ workflows on performance systems, in their case a supercomputer. These in-situ workflows are managed by PyCOMPS[87], which runs each Decaf workflow as a step in a larger task-based workflow. This allows for a large chain of analysis, with individual steps tailored to their specific hardware needs, and allows users to exploit the benefits of both types of workflow.

Pegasus[22] is another research focused workflow system, with a strong feature set and robust implementation. It is also a DAG based system in which a user can programmatically define processing tasks. Pegasus will then automatically identify dependencies between tasks and construct a DAG automatically. As in other systems, tasks may be individual workflows allowing for very complex workflows to be formed. There is slightly more emphasis in Pegasus on error recovery and fault-tolerance as more reporting is provided to help debug and manage errors. Running jobs will emit *workflow events* which can form the basis of custom monitoring systems, showing that Pegasus does the most to meet the requirement for responsiveness of the systems presented so far.

As with Apache Airflow and Kepler, Pegasus is designed as an end to end workflow management tool. It will manage your data from the very beginning of the workflow, and will manage the sending of any required data to resources, as well as the retrieval of any output data. These resources may be local to a user machine, or more commonly could be remote resources such as cloud or grid infrastructure. Other similar systems include but are not limited to Taverna[72], Dask[82], DagOn*[64], Askalon[35], DVega[89], and Condor[17]. These are all reasonably similar to those discussed already, and all are DAG based systems which allow for complex interactions and dependencies between jobs. Most are designed as complete workflow management solutions. Where they tend to differ most is in the level of reporting and feedback provided to users, though these differences often seem to be more down to UI design rather than as a function of radically different underlying systems.

One final workflow system that is worth considering is WED-flows[36]. WED-flows is interesting as it is an event-driven workflow system, that is perhaps the most similar example to the MEOW system presented in this thesis. In WED-flows, data is processed according to user defined trigger conditions. From one of these triggers a series of processing activities are started, in a control-flow fashion. The descriptions of WED-flows[36] are unclear on whether a DAG is specifically used or not in the creation and scheduling of processing tasks. It would certainly be a task appropriate for a DAG, and there is no more dynamic scheduling within those processing tasks than in any other SWMS. These sub-workflows themselves are not event-driven, and it is only the scheduling of the entire workflow itself that is undertaken in response to an event. For this reason we can conclude

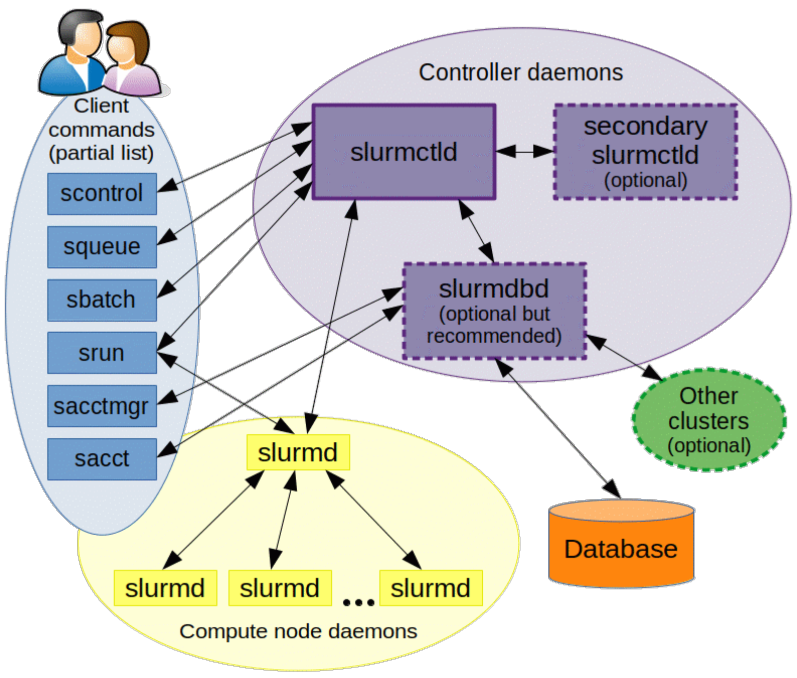

Client commands (partial list)
scontrol
squeue
sbatch
srun
sacctmgr
sacct
Controller daemons
slurmctld
secondary slurmctld (optional)
slurmdbd (optional but recommended)
Other clusters (optional)
Database
slurmd
slurmd
slurmd
...
slurmd
Compute node daemons

Figure 4.2: Overview of the basic Slurm architecture. This diagram was taken from https://slurm.schedmd.com/overview.html.

that although outwardly similar, WED-flows in fact share more characteristics with the previously discussed workflow systems than it does with MEOW.

4.3 COMMONLY USED NON-WORKFLOW TOOLS

As well as SWMS systems, there are also a number of smaller utilities used by researchers to automate their analysis to a lesser degree. Many of these smaller systems can be grouped together as workload managers, as they can be used to schedule large amounts of jobs on large computation systems such as grids or cloud resources. Examples of these include Slurm[103], WLM[97], Torque[90], and OGE[74]. As Slurm is broadly representative of these systems and we will be using it for benchmarking later, we will only consider it in further detail. Slurm consists of a variety of daemon processes, with a single main controller referred to as `slurmctld` as shown in Figure 4.2. Users can contact this controller daemon through command line tools to add jobs to the queue. Meanwhile, additional compute daemons can be started on whatever resource are available. These will also contact `slurmctld` and request one or more jobs from the queue. A large variety of optional extras and overheads can be provided to this system such as MPI support[88], job accounting or database integration.

What Slurm does not do is support complex workflow structures such as a DAG. For this reason it is not counted as a SWMS, though is commonly used by researchers to manage the execution of large amount of processing data on their resources. It is also commonly used within the already described

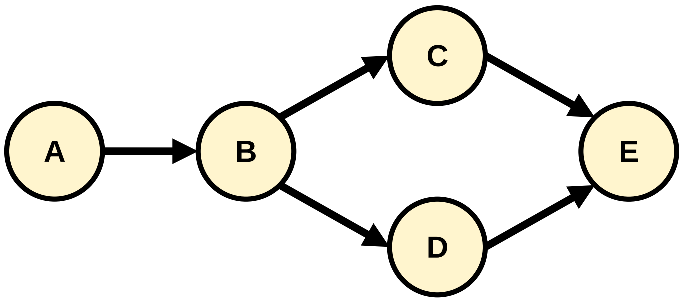


Figure 4.3: An example DAG.

SWMS systems as a way of managing their own execution. Other examples of these systems include Globus[37], Kubernetes[51], UNICORE[9], Decaf[29], parsl[5], cwltool[20], MapReduce[21], as well as running manually constructed workflows in custom scripts. Obviously this is quite a range of tools that are unified more by not being a SWMS rather than by sharing any specific quality. Unlike the dedicated workflow tools previously discussed these will usually, though not always, lack any sort of GUI and provenance reporting. They also often provide little inbuilt error handling, or automatic data management. Nevertheless, they are often used by researchers to conduct large scale analysis that could be automated with a workflow.

4.4 DAGS, AND A FALSE DICHOTOMY

Throughout all this discussion of the available workflow systems, DAGs have been mentioned time and again. These were briefly explained, but are worthy of a more extensive examination. A DAG is a type of graph containing one or more nodes. As in other graphs, these nodes can be linked via edges. In a DAG, these edges are directed, in that the specify a single direction. So for instance in Figure 4.3, we can see that the edges have arrows showing a clear direction from node to node. This gives each node a hierarchy, and creates dependencies between the nodes. For example, in Figure 4.3 we can see that node A logically must come first as it alone has no dependencies, and that node E must come last as it depends directly or indirectly on all other nodes.

Hopefully, it should be apparent how a DAG is a useful tool for managing workflows, as if a series of steps can be mapped to nodes it becomes trivial to identify dependencies and data links between jobs. A DAG used in this way is constructed before any processing has taken place, as part of the job identification and scheduling stage. The limitation of this is that the processing then needs to keep to this DAG structure, and jobs cannot be added or removed without altering the DAG, which cannot be done without restarting the workflow. For this reason, this method of workflow construction will be

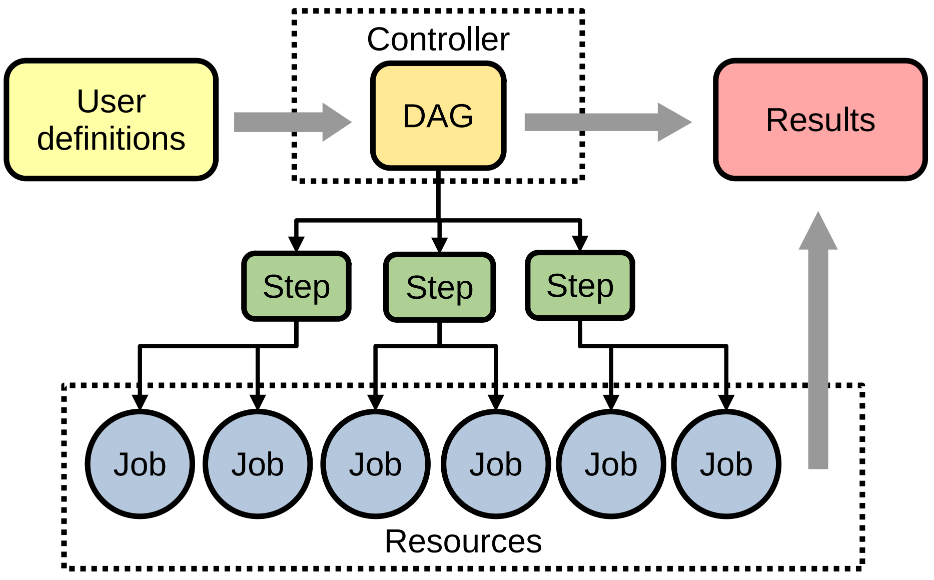


Figure 4.4: A top-down workflow structure. Not shown is any data transfer or dependency between jobs.

referred to throughout this paper as static. This reflects their structure being entirely defined ahead of time, with no scope for change.

Formally, this approach could also be referred to as a top-down approach as a central workflow controller is determining the entire structure ahead of time and dictating it to individual jobs, as is shown in Figure 4.4. Here we can see that a user submits their workflow definitions to the system, either via a DAG, or something that from which a DAG can be derived. In either case the workflow controller will use this DAG to derive the steps. These steps in turn become the basis for identifying jobs, which are scheduled on resources. No data transfer or dependency between jobs has been shown in Figure 4.4, but at some point the final job will complete and any output will become part of the workflow results. The DAG will also form part of the results, independent of any job processing, as it is a very informative and crucial part of the workflow construction.

This static nature conflicts with the stated desire for scientific workflows to be dynamic outlined in 3.2. Another annoying feature of a DAG is that it cannot loop. Observe an example in Figure 4.5. This shows a loop of dependencies and so cannot by considered a DAG, and would instead be known as a Cyclic Graph. Semantically, what this means is that if a workflow system were to be based on a DAG, then it will *always* be defined ahead of time, **must** have a defined beginning and end, and can **only** progress for a finite amount of time. It must also **never** repeat a step or go backwards.

As a final note, this contrast between static and dynamic is not intended as a statement per-se about the capabilities of either approach. As will be seen below, various static workflows are capable of being dynamic to some degree, and it is similarly possible to construct a static, unchanging workflow in a dynamic system. These terms have merely been adopted to demonstrate the base assumptions and

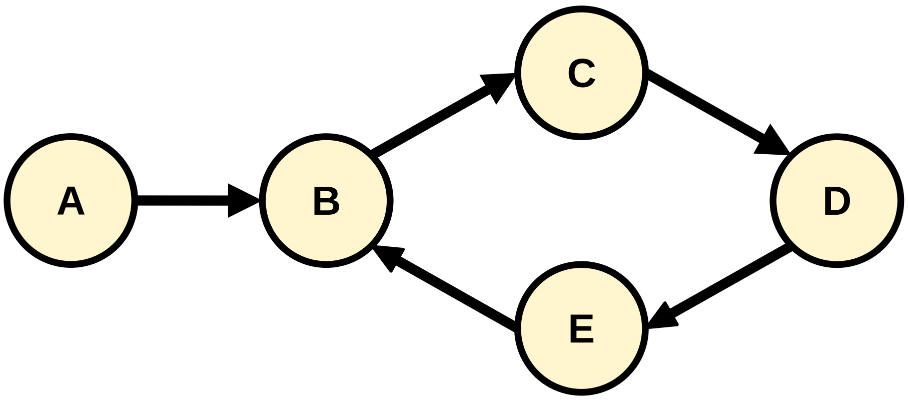


Figure 4.5: An illegal DAG. Note that the loop displayed here means this is not a DAG, and would instead be a Cycle Graph.

approaches used in the workflow construction. For this reason I have noted that it is a false dichotomy, but a useful one nonetheless.

4.5 THE LIMITS OF STATIC, AND THE POSSIBILITIES OF DYNAMIC

The semantic limitations of a DAG laid out in Section 4.4 are obviously ridiculously strict, and were a SWMSs to implement all of them as so written, scientific workflows would be extremely limited and definitionally incapable of meeting either the responsiveness or scientific requirements of a SWMSs, as outlined in Section 4.1. This is due to the exploratory nature of scientific workflows and their need to be dynamic. However, the use of DAGs does allow for the answering of all of the other requirements. Though individual systems may focus on some over others, as a field all requirements have been met in principle, even if debate is still be ongoing as to how well each individual system does so. The only major hole is this need to be dynamic and so we could conclude that we need a new solution to meet this need.

However, before we do so it is worth considering that although all examined SWMSs use a DAG, they do not always keep to all of the strict semantic restrictions. For instance, a common feature is for some degree of error handling within workflows at runtime. As an example, DVega[89] allows for swapping in and out of jobs in response to errors. This tends to be of fairly limited use to an exploratory workflow however as errors may be unpredictable.

Many workflows also make frequent use of loops or continuous processes. This is often represented within the DAG by simply unrolling the loop, so that if a process is to run 10 times it is simply represented by 10 single processes. Such an approach inherently means that only loops of determinate length can be scheduled. There are a few examples of seemingly infinite workflows such as in the Wifire[2] system. This is something of a hybrid system, with finite workflows repeatedly being scheduled and output. These can then be used as the input for the next. In a manner similar to WED-flows, Wifire schedules smaller, static workflows repeatedly. Where WED-flows uses an event driven system to schedule this analysis, Wifire uses a variety of systems such as events, timers and manual input to schedule their analysis. This can give the appearance of infinite loops of workflow processing, but it is loops of static systems and has only solved this problem through a large, bespoke manager above the underlying workflow management.

As well as this, a common feature within existing SWMSs is the ability to branch or make decisions. This would be a case where depending on if condition A is met or not, either processing B or processing C will be scheduled. Often in the larger systems this is an explicit feature, such as Apache Airflow which provides a `BranchPythonOperator` specifically to enable this. Even in systems that do not offer branches as an option, such as cwltool, it is not hard to create similar functionality through cleverly made jobs.

What the prevalence of looping and branching show is that even those SWMSs that explicitly state they use DAGs as the basis of their system, are not bound by the limitations of the DAG. This is as presumably to do so would be massively limiting, and not allow a SWMS to properly address the rest of the requirements set out in Section 4.1. The two relevant requirements here are to be *Responsive*(Section 4.1.6) and *Scientific*(Section 4.1.8) which together express a need to be dynamic. This need is relatively new within workflows and yet we have seen that the current crop of tools have managed to implement some of the key aspects, such as the ability to handle errors at runtime, and create looped or branching workflow structures.

These accommodations to dynamic structuring are by no means standard features however, with many systems not implementing them at all, and no system going further than the three areas of error handling, loops or branching. For example, no current SWMS has the means to properly assemble a workflow at runtime in response to some ongoing events or analysis. No current system will automatically reschedule jobs if the input data is updated at a later date. No current system truly accommodates the ongoing structure of a live analysis system. There is no doubt that there exist cunning ways around all of these problems with the use of existing SWMS, along with various tools and tricks such as scripting, cron jobs[19], careful user management or other such solutions. We will not get into any of these as solution will each be as esoteric as they are inherently bespoke.

We can conclude from this that the provision of dynamic structures is a gap in the existing crop of SWMSs. Though it is partially met, the use of DAGs and their inherently static nature means that any provision of dynamic workflow structures is very much an after-thought within a SWMS, applied as a patch to a system that would rather operate in a constant, unchanging manner. An alternative to this model would be a dynamic model, where flux is accepted as a fundamental characteristic of a scientific workflow. The next chapter will outline MEOW, which demonstrates such a dynamic workflow structure. It will allow for the need to be dynamic to be fully met, allowing for users to create all manner of weird and wonderful structures in their workflows. Users will be able from the ground up to create systems that run forever, assembling a workflow at runtime, and rescheduling analysis automatically. None of these features are inherently supported by existing systems. As well as this, the accommodation of branching and looping structures will be inherently supported from the ground up, rather than being an inconvenient fit to a rigid, linear model. This is expressed in the core research question of this thesis, is it feasible to create a tool for the automatic creation of dynamic scientific workflows (Section 2.2). By then end of this thesis we will see that indeed it is feasible to do so, through the use of an event driven model the responds to changes in both data files and its own internal state. This contrasts strongly with existing SWMSs and so we can say that the state of the art has been extended, through such a dynamic system.

5

DYNAMIC WORKFLOWS

In Section 3.2 we established that researchers require dynamic workflow management tools to assist in their research. Then, in Section 4.5 we saw that the existing SWMSs are insufficient in this area, it is time to set out a possible alternative. This alternative will reject the use of static DAGs as a starting point, and instead create a new structure designed from the ground up to be dynamic. This is as DAGs are inherently static, and so would make an odd choice as the foundation of what is supposed to be a completely dynamic system. The hope is that this will allow researchers to create fully dynamic systems for exploring experiment spaces.

5.1 WORKFLOWS WITHOUT DAGS

As identified in Section 4, a SWMS has an inherent need to be responsive to ongoing workflows, as well as an increasing need to be adapted at runtime by users. Although some of the more mature systems have some capacity for dynamic structuring, this is often a later feature patched onto an inherently static model. A different approach is needed to design a system that fully enables dynamic analysis in all its forms, with steps, jobs and data being added, removed, or modified at runtime. These are all use cases in the need to be dynamic as outlined in Section 3.2, and so we need a base that accommodate such fundamental changes at runtime. Here, an event based system for constructing emergent workflows is proposed as one such solution.

This new solution will use a bottom-up approach as the foundation for its design. Rather than a singular controller identifying, scheduling and assigning all workflow jobs, this responsibility should be done in isolation. By this it is meant that the system does not actively construct an entire workflow ahead of time, merely that it identifies individual jobs, as is shown in Figure 5.1. In this Figure we can see that as before, a user provides some definitions. These are not reduced to a DAG but, and are instead used as the basis for individual job scheduling. Once jobs have completed, they may be linked together into steps. These steps, along with any output from the jobs can then form the results of the

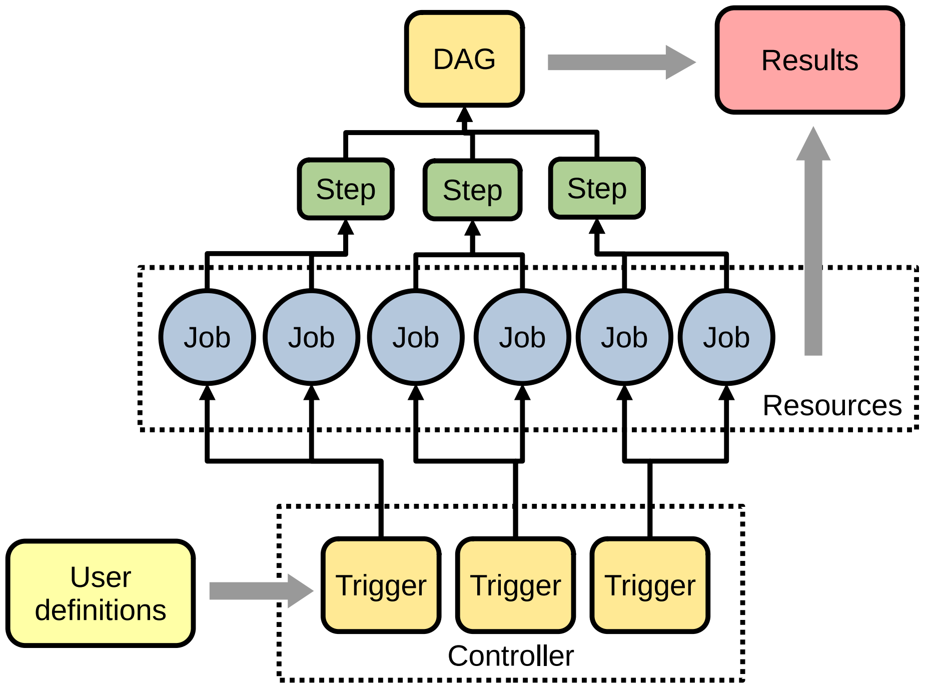


Figure 5.1: A bottom-up workflow structure.

emergent workflow. The word emergent has been highlighted as it is central to this new approach, and is fundamental to understanding it. To re-iterate, in a bottom-up workflow system, a user does not actually construct a workflow directly, they merely create the conditions such that individual jobs can be scheduled.

A core difficulty in a bottom-up system is how to identify and start individual jobs according to a set of steps without a DAG. Recall that one of the key benefits of a DAG is that it becomes extremely easy to identify and schedule jobs in a clear, dependent order. This will not be possible in a bottom-up approach as the jobs are created in isolation. Within a static system is makes sense that the inception of a new workflow is user input, either by running a script or pressing a button in a GUI. This is as the DAG gives the workflow all the necessary knowledge to schedule the jobs in order.

In contrast to this, in a bottom-up approach the controller will only be able to schedule individual jobs on an ongoing basis, as is shown in Figure 5.1. A good way of doing this would be to use events, as demonstrated by WED-flows in [36]. WED-flows uses a small set of trigger conditions, under which static workflow are processed. We can use a similar concept, only rather than scheduling small, static workflows, only individual jobs will be scheduled. Events are a good option here as they can be continuously responded to in real-time, with no prior determination or necessary knowledge of what order they will occur in. This is analogous to event handling in more traditional programming, were code will wait for some trigger either from a user or some underlying process. Events also have the advantage of not needing continuous user monitoring or input, as would be necessary if a user was to manually schedule individual jobs one after another in response to ongoing developments.

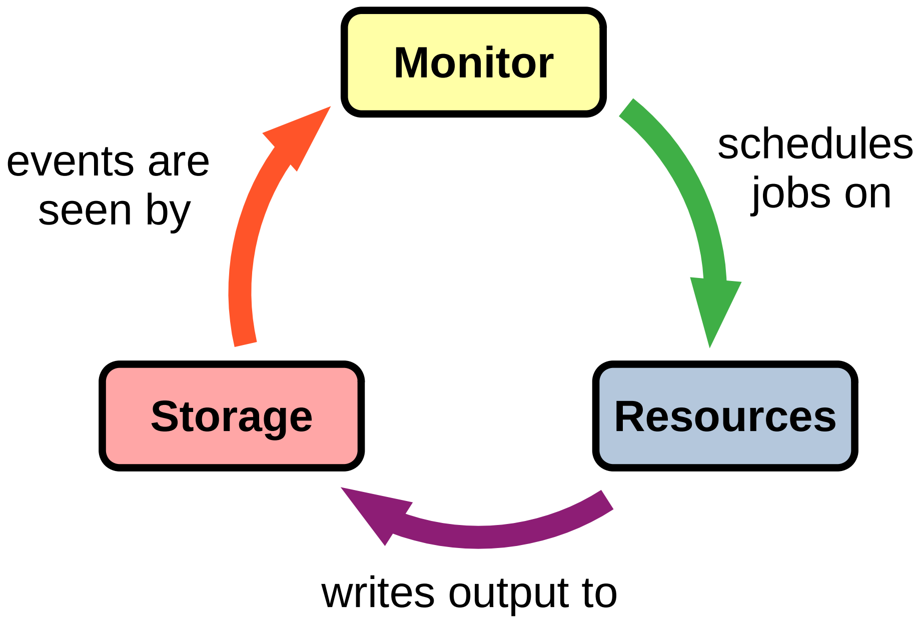


Figure 5.2: General design of an event based scientific analysis system.

To facilitate the change from a centrally controlled workflow, to an event-based system, the workflow system itself has to function as an event monitor. This means that the system will not have an easily defined beginning and end, as would be the case in a top-down workflow. In an event based system, we have no way of knowing when events will occur, and so the system has to run continuously. Jobs will produce output, which will be written back into the monitored system. This may trigger further events and so the processing continues. Note that the actual events have not been defined here, and could be anything. Examples include file events such as files being created or modified, or streaming events such as data arriving across a network. They could be GUI events such as button presses or hardware events such as switches being flipped. In each case some software monitor could listen for these events, and respond to them by scheduling some appropriate processing. A high level view of such a system is shown in Figure 5.2.

This is a simple structure but means that we can't now use any of the benefits of a DAG, namely the easy identification of a complete workflow. As will be discussed in the following chapters and into Part ii, we have not thrown the baby out with the bathwater, and it will still be possible to meet the requirements set out in Section 4.1. It will more difficult to meet some of these than it would have been with a DAG, as we will see. However, this shift is still justified as this prior knowledge and predictability is exactly what we may need to fight against to implement a truly dynamic system. By throwing out all of the utility of a DAG it is hoped not that we have needlessly thrown away an extremely useful tool. Instead it may be that we have escaped a potential progress trap, where the utility of DAGs in solving certain problems has meant we cannot solve others that are just as valid.

5.2 MEOW: MONITORING, NOT CONTROLLING

An architecture such as that presented in Figure 5.2 would be fundamentally different to those discussed in Section 4.2. As a result, before we can implement such an event based system, we need to define a framework that such an implementation can be built on. This framework has been given the straightforward name of Managing Event Oriented Workflows, referred to simply as MEOW. It would allow users the means to match events to processing code. These matches would be managed by a MEOW system and from them, jobs would be scheduled.

To do this we will adopt two core concepts, Patterns and Recipes. These are at the very heart of how this system is designed and so will now be defined. These can been created as part of MEOW as a way of describing what we need to do within an event based scheduling system. We need to be able to flag what events should be triggers for some job scheduling, and we need to define what processing will actually take place in those jobs. These could both be done at the same time, but intuitively it was felt that job code would often be long, complex, and probably shared across a variety of job. Therefore, having it as a separate, shareable object would be of use.

5.2.1 *Recipes: Defining what work is scheduled*

Within MEOW, a Recipe is quite simply some code that will be used to define job processing. It could be an algorithm, routine or any other set of instructions used to express scientific analysis. In the context of tomography this might be an segmentation algorithm, or 3D reconstruction script. Any single, sequential code file could in theory be a MEOW Recipe. As a final clarification, this does not mean that a Recipe could not start scheduling threads or other non-sequential elements, or that it could not utilise extensions such as libraries. It only means that at the highest level, the analysis needs to expressed by a single code file, which we will refer to as a Recipe.

5.2.2 *Patterns: Defining when to schedule*

Whilst a Recipe defines job processing, a Pattern defines the conditions under which a job will be scheduled, and the necessary inputs given to any jobs that are scheduled as a result of said Pattern. This means that a Pattern defines the selection criteria for what events will lead to jobs, and which ones can be ignored. How this is done is a matter of implementation, and not is not tied to this conceptual definition of a Pattern. However, in the implementations that will be presented in the coming chapters we will be using file events as the basis for our system. These are events broadcast

1	<meow>::={<pattern>}
2	<pattern>::=<event> <recipe> <input> <output> <age> <variables>
3	<event>::="some_event_identifier"
4	<recipe>::="some/recipe.file"
5	<input>::=<directory> | <file>
6	<output>::=<directory> | <file>
7	<directory>::="some/directory/location"
8	<file>::="some/file.path" | "some/file.path" <file>
9	<age>::="old" | "new" | "all"
10	<variables>::={<variable>}
11	<variable>::="some_variable = some_value"

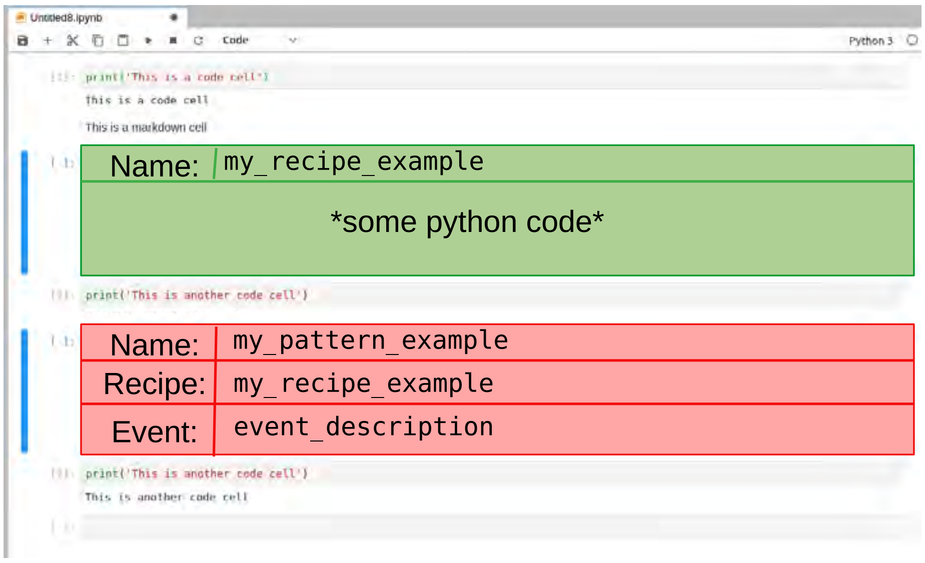
Table 5.1: Initial sketch of MEOW design in EBNF.

by the operating system when a file is created, modified, moved or deleted. In this case the role of the Pattern is to identify which file paths should act as triggers, so if any files events occur at that path, some processing is scheduled. As well as the scheduling conditions, a Pattern also defines what goes into the scheduled job. Which Recipe to use is therefore included within a Pattern, along with any variables to be passed to Recipe to create some distinct and meaningful output.

5.2.3 *First Forays in MEOW*

In order to schedule a job from an event, a user needs to be able to identify events to MEOW, identify processing code to MEOW, and to link them together. In general workflow terms, each of these links would represent a step. To help plan out the initial design a design sketch was made in Extended Bachus-Naur Form (EBNF)[96]. The produced notation is available in Table 5.1. As this sketch later needed to be revised, it will not be covered in extensive detail, with the same applying to the implementation details in the following section.

The initial sketch presented in Table 5.1, is somewhat abridged EBNF, as it lacks proper definitions for an *event*, *recipe*, *directory*, *file*, and *variable*. We can see that two key concepts in MEOW have already been identified, the Pattern and the Recipe. These will remain core to Managing Event Oriented Workflows (MEOW) throughout, though will be expressed differently going forward. At their most basic, a Recipe is the processing code used in a job, whilst the Pattern is a construct for matching events to Recipes thereby scheduling jobs. A user defines several Patterns, each of which contains a number of attributes. Definitions in the sketch that contain strings, such as "some_event_identifier" in definition 3, are meant to denote implementation specific items, rather than a specific string to be matched. For instance, based on what sort of events an implementation of MEOW is built on, the manner of identifying matching events may differ and so is left ambiguous within this sketch.



```
[1]: print('This is a code cell')
     This is a code cell
     This is a markdown cell
```

Name: my_recipe_example

some python code

```
[1]: print('This is another code cell')
```

Name: my_pattern_example
Recipe: my_recipe_example
Event: event_description

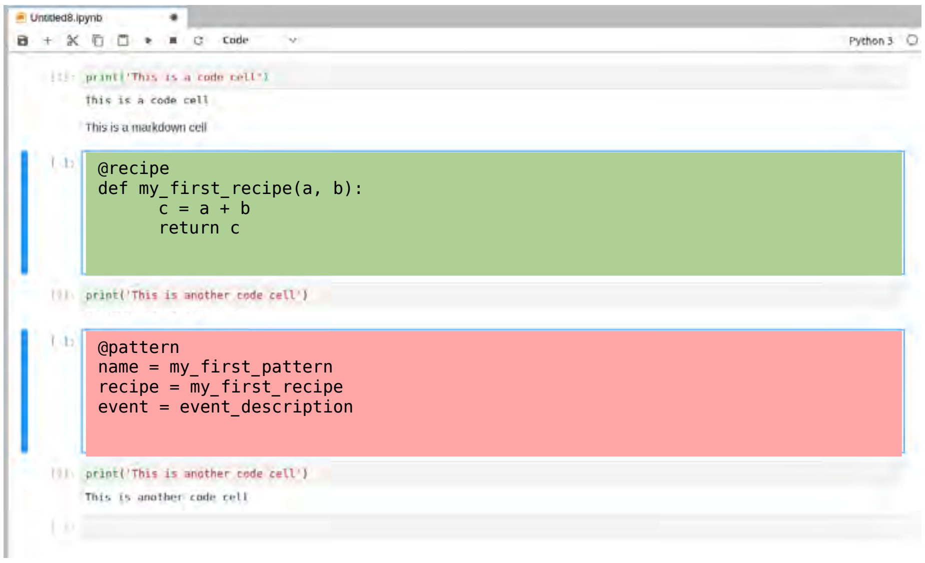
```
[1]: print('This is another code cell')
     This is another code cell
```

Figure 5.3: Initial MEOW mockup. There are two new cell types, a Recipe cell in green, and a Pattern cell in red.

5.2.4 *Attempts at an Implementation*

With this rough plan as a guide, thoughts were turned to how to start implementing MEOW. It was decided that file events would be a potential candidate for the system. File events include files being created, modified, moved, and deleted. These are intuitive events that can easily be understood by a user, making the system easier to comprehend and plan for, as per the requirements in Section 4.1. File events would also match well to a dynamic system for scientific analysis, as analysis is fundamentally going to be run on input data. Users are going to require new jobs when they either get new data (create a new data file) or update their data (modify a data file). They may also expect new jobs when they create or modify analysis algorithms but these can also be saved as files and responded to in the same way as input data. Therefore all of the relevant changes that might need to be monitored can be expressed through file events. For these reasons, event identification was whittled down to paths. This made matching somewhat easy as users could define a REGular EXpression (regex), and any events that occurred at a path that matched that regex would result in a job being scheduled.

Having settled on file events, a first attempt at implementation was made. This design was later discarded, with later implementations borrowing little from it, though is worth quickly going through as it illustrates how the later design was derived. To provide both a platform for implementation, as well as a familiar interface, it was decided to use Jupyter Notebooks[48]. These are already an extremely common form of writing scientific analysis[95]. Jupyter Notebooks are JSON documents that are organised into cells. These cells can express code, output from running said code, or accompanying text and images. The code itself can be written in many different languages, though



```
[1]: print('This is a code cell')
     This is a code cell
     This is a markdown cell
[ ]: @recipe
     def my_first_recipe(a, b):
           c = a + b
           return c
[1]: print('This is another code cell')
[ ]: @pattern
     name = my_first_pattern
     recipe = my_first_recipe
     event = event_description
[1]: print('This is another code cell')
     This is another code cell
[ ]:
```

Figure 5.4: Secondary MEOW mockup. Here there are two new decorators inserted into code cells, with a Recipe decorator in the green cell, and a Pattern decorator in the red cell.

the mostly commonly used kernels are Python 3, Python 2 or R. The JupyterLab interface provides a web based interface for creating Jupyter Notebooks, and can be extended with relative ease through the use of extensions. Therefore, it was decided that the creation of a JupyterLab extension could expand the utility provided by Jupyter Notebook to allow for the creation of MEOW constructs. This would be by expanding the types of cells to include specific cells for Patterns and Recipes. Here a *recipe* cell would contain code in the same manner as a regular *code* cell, but contain an additional field for a name so that a Pattern could refer to it. Meanwhile a *pattern* cell would contain a Pattern definition according to the grammar proposed in Table 5.1. A mockup of what this might look like is presented in Figure 5.3. Aside being astoundingly ugly, this mockup is not fully complete, as many of the Pattern attributes from Table 5.1 have not been included.

However, creating different cell types would be a fundamental change to the operation of Jupyter Notebooks and would require far more than a simple JupyterLab extension. This would firstly not be a insignificant amount of work to achieve, but would also mean the Notebooks themselves would not be usable with non-MEOW systems, as they would contain custom cell types that would make them incompatible with the standard Jupyter Notebooks format. For these reasons, this idea was abandoned. A quick followup was created, that did not need a JupyterLab extension, but instead would use decorators within *code* cells to flag them as either Patterns or Recipes. The entire Notebook would then be read by the MEOW system and the relevant constructs extracted. Another mockup of this design is shown in Figure 5.4.

Again, this idea was abandoned in fairly short order as it would somewhat alter the use of a Jupyter Notebook. Jupyter Notebooks are intended as easily shared platforms for running code and viewing their output. They use a standard format and most commonly created, displayed, edited and run through JupyterLab. They are so common mainly as a function of their utility, and ease of sharing. Any decrease to that shareability would have to overwhelmingly increase the utility to be of value. Introducing decorators on cells would only be correctly interpreted by a MEOW based system, and would potentially break any non-MEOW based system. Therefore we would have massively decreased the shareability. Whilst I am a great proponent of MEOW, I do not think the added utility of its use is enough to cast aside the Jupyter Notebooks ability to be shared so easily, and so it was once again back to the drawing board.

During a conversation with several researchers at the MAX IV Laboratory, Zdenek Matej suggested that rather than individual code cells, Recipes would be far better suited to being whole Jupyter Notebook files. This is as researchers will already have most of their analysis expressed in Jupyter Notebooks, and so being able to use those directly would significantly decrease the barrier to entry. Entire files rather than just cells would also be more fitting, as most scientific analysis will require more code than is often written in a single cell. Recall also that in the definition given in Section 5.2.1, Recipes are just collection of code. This could just as easily be an entire Jupyter Notebook as a single cell. Because of the previous misgivings about using decorators breaking the shareability of Jupyter Notebooks, and with the user request for Jupyter Notebooks-as-Recipes, a third and final approach to implementing MEOW was created.

6

MEOW

Having gone through two significant design iterations, a third approach was finally settled on for MEOW. This approach would implement a Python 3 library that could be added to a Jupyter Notebook kernel, providing necessary MEOW constructs for creating Patterns and Recipes.

6.1 A FRAMEWORK FOR EMERGENT WORKFLOWS

It may seem counter-intuitive to go back a step now and talk about the theoretical underpinning of MEOW at this point, when we are just about to get into the meat of the implementation. However, over the last two iterations of the MEOW design, it was still relatively up in air as to how Patterns and Recipes would actually interact. Although some conceptual definitions for both were presented in Sections 5.2.1 and 5.2.1, we need to start defining how they will be implemented. As a starting point, the hierarchy of a MEOW system is shown in Figure 6.1.

From this we can see that a MEOW system is still comprised of Patterns and Recipes, and that they are still separate constructs. Recall from the conceptual definitions in sections 5.2.1 and 5.2.2 that Recipes define what code to schedule, whilst Patterns define when to schedule it. However, as a Pattern can refer to a Recipe that does not exist, a new construct has also been introduced, the Rule. A Rule will be created any time a Pattern exists and refers to a Recipe that also exists, and will take its attributes from both the Pattern and Recipe used to create it. Any events will then be compared against

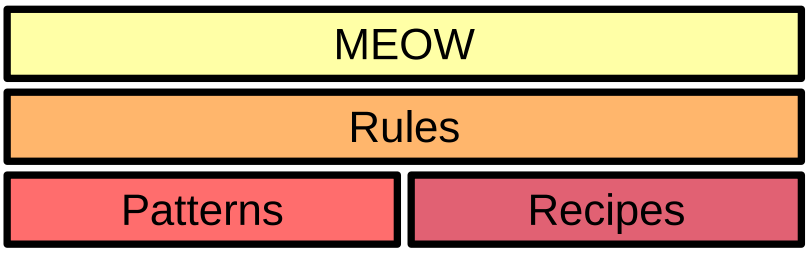


Figure 6.1: MEOW hierarchy.

all existing Rules with any matches leading to the scheduling of the defined processing. Logically, this is not completely necessary as Patterns on their own can be used to filter which events should lead to processing. However, the filtering of events is expected to be a computationally expensive operation and so we only want to check filters that we know can be used to actually schedule jobs. By this we mean that an associated Recipe exists, and so we have this additional construct where we know if any event matches occur against a Rule, by necessity, a job can be scheduled.

Together, Patterns, Recipes and Rules form the basis of MEOW, as was shown in Figure 6.1. Within an implementation, the basic premise is still that users will only define Patterns and Recipes. Recipes are users analysis code, and should be the same algorithms that they already been work with. Patterns define what Recipes are used to create jobs according to what data files. When defined correctly these combine to form a Rule. Meanwhile, the MEOW system will listen to a file system for any events. Any events in the monitored system are compared against the current list of Rules. If an event matches one of the Rules, then one or more jobs are scheduled. The composition of these job is determined by the Pattern and Recipe used to create the Rule. These jobs will be processed and may produce output that may in turn trigger further job scheduling. In this way a chain of scientific analysis is formed without the use of a static DAG. For example, if a user had some segmentation algorithm and then needed to apply it to some raw data, they would put the segmentation algorithm into a Recipe. They would then construct a Pattern which would match to the storage locations of all the data files. The system would then automatically schedule a job for each data file using the segmentation algorithm.

6.2 FINAL REQUIREMENTS

To achieve such a system, specific requirements were derived from our original Pattern and Recipe conceptual definitions, as well as from the requirements for a SWMS outlined in Section 4.1. Most importantly was the requirement to for Responsiveness. This meant that the system had to be able to not just respond to events by scheduling new processing, but that it should respond to internal state events. For instance, if a Pattern is deleted by a user then we should cancel any scheduled jobs that have not completed as we can assume they are no longer needed. Equally, if a Pattern or Recipe is ever modified then we should assume this means that any jobs created is now outdated and should be replaced by new scheduling using the updated Pattern or Recipe. These simple ideas were logically thought through until the system of requirements presented in the three tables were arrived at. The hope is that any system that implements all of these, would by necessity be capable of meeting that requirement for Responsiveness that static systems struggled so much with.

R1	A MEOW Recipe has a name that is uniquely identifiable between Recipes.
R2	A MEOW Recipe expresses some runnable code.
R3	Whenever a MEOW Recipe is created, if any previously created Patterns stated the Recipe name, then a MEOW Rule must be created for every stating Pattern.
R4	If a MEOW Recipe is ever deleted, any MEOW Rules created from it must also be deleted immediately.
R5	If a MEOW Recipe is ever modified, it should be treated in the system as though the original MEOW Recipe was deleted and a brand new one has been created.

Table 6.1: MEOW Recipe requirements.

P1	A MEOW Pattern has a name that is uniquely identifiable between Patterns.
P2	A MEOW Pattern states the name of a Recipe to be used for processing jobs.
P3	A MEOW Pattern describes a filter against which events can be applied.
P4	A MEOW Pattern defines any variables to be passed to the Recipe.
P5	Whenever a MEOW Pattern is created, if a Recipe with the name stated in the Pattern has already been created then a MEOW Rule must be created.
P6	If a MEOW Pattern is ever deleted, any MEOW Rule created from it must also be deleted immediately.
P7	If a MEOW Pattern is ever modified, it should be treated in the system as though the original MEOW Pattern was deleted and a brand new one has been created.

Table 6.2: MEOW Pattern requirements.

M1	A MEOW Rule must be uniquely identifiable.
M2	A MEOW Rule must inherit the event filter parameter from the MEOW Pattern that caused its creation.
M3	A MEOW Rule must link the MEOW Pattern and MEOW Recipe that caused its creation.
M4	A MEOW system must monitor a single file system or single subset of a file system.
M5	All subsets of the monitored system must also be monitored.
M6	To monitor means to identify any and all events within the monitored system.
M7	Any identified events must be compared against every currently created MEOW Rule.
M8	Events are matched to MEOW Rules by comparing the event properties to the MEOW Rule event filter.
M9	Any matching events must schedule one or more jobs, according to the MEOW Pattern definition.
M10	Whenever a MEOW Rule is created, it must be able to check within the system, would any existing event sources match the MEOW Rule, were they created now. If so, the source must be treated as though they were just created.
M11	If a MEOW Rule is ever deleted, any jobs scheduled from it must be cancelled.

Table 6.3: MEOW general requirements.

The individual requirements for Recipes and Patterns are shown in Tables 6.1 and 6.2 respectively. General requirements for Rules, as well as the rest of the system are shown in Table 6.3. They have not been expressed in EBNF as they are expressing more defined, contextual objects than is really suitable to EBNF.

Of note are some significant differences since the first attempts at defining MEOW. Firstly, it is worth noting that whilst any file event that matches the Patterns trigger filter is an implied input, output is not defined at all. This is deliberate as to define an output is to limit results to easily predicted outcomes. This would be counter to the exploratory nature of scientific workflows, and is ill suited to a dynamic system. This contrasts strongly with the previously discussed SWMSs, where outputs are a necessary part of defining any workflow.

A second major difference is that we have introduced specific requirements for the updating and deletion of Patterns and Recipes. This is actually an essential step in making the resultant workflow as responsive as possible. Previously it was made possible for individual jobs to be created, deleted, or modified at any point due to their isolated nature within the event driven system. However, this is a lower scope than what is actually possible within MEOW. By applying the requirements R3, R4, R5, P5, P6, P7, M10 and M11 in combination, we can also make it so that entire steps can be created, modified or deleted at any time. As the emergent workflow is derived directly from the steps, we can now run a workflow system that is completely dynamic, where the very structure of the workflow itself can be changed at runtime by the jobs within it.

Of final note is that the presented MEOW requirements meet the first objective of this Thesis, as presented in Section 2.2. By defining the MEOW requirements, along with the accompanying structure in Figure 6.1, we obtain a design for an automated, dynamic workflows. This is important as such as design was the first objective from Table 2.1, and the first part of addressing the core research question of investigating the creation of an dynamic workflow system. Though the described design perhaps goes beyond the requests for a dynamic workflow as stated in [56] and [102], it doubles down on the possibilities of an event driven system as presented in [36]. This makes MEOW a truly novel SWMS, as it allows for the possibility of a self-modifying workflow, truly addressing for the first time the requirement for SWMSs to be responsive.

7

SUMMARY

This part has set out the theoretical foundation for MEOW, and the requirements for such a system. This has been presented as, and is, a dynamic alternative to the existing static SWMSs. Many of these systems use DAGs, or some other equivalent to structure their workflows before any processing can progress. Though changes are sometimes possible at runtime, they are often limited in scope or require tedious management. MEOW is designed in a new bottom-up paradigm that rejects the static definition of complete workflows in favour of identifying individual steps and scheduling those jobs in isolation. By doing this, each job is free to be executed in isolation and so can succeed or fail without affecting subsequent steps.

The requirements of a dynamic system have been outlined, as has the initial attempt at implementing it. Together, these should contextualise the subsequent part of this thesis, in which the final implementation is presented. Recall that the first objective for this thesis was to *'design a framework to express automated, dynamic workflows'*(see Table 2.1). This has been achieved via the definition of MEOW, and so we can conclude that the first objective has been met. We are now ready to progress to implementing MEOW in accordance with later thesis objectives.

Part II

TOOLS FOR EMERGENT WORKFLOWS

8

INTRODUCTION

This is the second of the three main parts in this thesis. By the end of this part, it will be shown how the second and third thesis Objectives (see Section 2.2) were addressed. These were to *'Implement an automated, dynamic workflow system'* and to *'Integrate the automated, dynamic workflow system into a collaborative big data platform'*. This is primarily done through a newly created python package, `mig_meow`. This is designed primarily to work with KU's grid solution, the Minimum intrusion Grid (MiG). As such this will first be explained, along with a number of essential packages used throughout the implementation. Once this last bit of background is out of the way we can finally consider `mig_meow` more fully, and how a user can use it to establish MEOW constructs.

To help users design these constructs, `mig_meow` has been designed to work with Jupyter Notebooks, and so a number of widgets designed specifically to work within them are also presented. Users can create MEOW systems to function locally, or on the MiG. First we will consider how a user can use the `WorkflowRunner` to construct and run their analysis on their local machine. Secondly we will expand the context to include the MiG, and shall demonstrate how the widgets have been designed to interact with it.

9 BACKGROUND AND DIRECTION

We will now turn to how MEOW was implemented. The first implementation we will be looking at is the Python package `mig_meow`. This was created from scratch for this project, and is designed as a tool for users to create MEOW Patterns and Recipes within a Python environment. These could then be used to run analysis on a users local machine. In addition, `mig_meow` was also intended as a front-end for how users would create Patterns and Recipes for use on the MiG. As the same constructs would be used for both the local analysis and on the MiG, it followed that the local system should mimic the relevant functionality of the MiG.

9.1 MINIMUM INTRUSION GRID

We will return to talking about `mig_meow` in the following chapters, but first we should fully establish what the MiG is and how the relevant sections operate. The MiG is a mature, stand-alone grid solution. It connects users to disparate storage and processing resources. The core goal of the MiG is, as the name suggests, minimum intrusion[49]. This means minimal requirements for both users and resources to enrol and start using the grid. For users they only require an X.509 certificate[98] accepted by the grid, and a web browser to access it. Enrolled resources also only require an X.509 certificate and a local grid user. These are very small requirements compared to other grid solutions and ensure ease of use even across large institutions.

A sketch of the overall architecture of the MiG can be seen in Figure 9.1. It shows how users connect to a central 'black-box' server which administers the entire grid. By centralising in this way it is relatively easy to manage a large amount of disparate resources and access to them. Any significant software installs can be kept to the server itself, as resources are essentially just given jobs to execute. This also ensures that maintaining or updating the grid is relatively simple as only the central server needs to be updated without having to wrangle a myriad of different resources administrators to update together. What can also be seen in this Figure is the variety of ways in which a user can interact

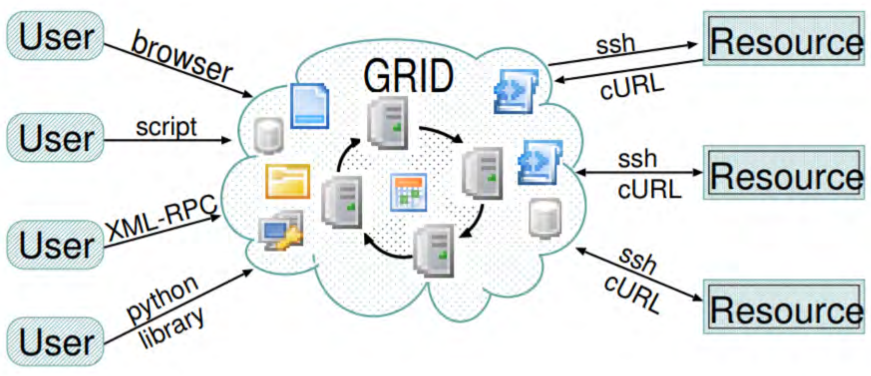


Figure 9.1: Architecture of the MiG. Taken from [10].

with the MiG. The scripting possibilities are of particular note as they allow for the connection of scientific instruments to the MiG via a user account. This means that instruments can be set up to directly output their data onto the MiG.

Job execution on the MiG is done via the resources. Resources may be all manner of hardware, with recent innovations even meaning remote cloud systems may be usable as resources[69]. More regularly, resources will be more traditional compute nodes running several CPUs, or GPUs. As part of their enrolment on the grid, these resources will loop through a simple script that will regularly poll the central server for jobs. The server will identify a suitable job, if there is one, and send the necessary files for its execution out to the resource via cURL. Once the files are transferred, the job is executed and the output files are sent back.

Of course some jobs may require additional software or libraries to be present on the resource. These may be installed by the resource administrator at any time, and the grid alerted to the presence through *Runtime Environments*. These are a MiG construct, where installed software, tools or libraries are flagged as being present on the resource. Jobs may be setup with *Runtime Environment* requirements, and so will only be scheduled on resources that match those requirements.

From a user perspective all data uploaded to the MiG by them is accessible from within their own user directory. As well as this, users can setup workgroups, to share their data[10]. A workgroup is a collection of data, to which access can be controlled on a per-user basis. Access to resources can also be controlled on a per-user basis, but this could quickly become an exhaustive process, so access to resources can be granted on a per-workgroup basis as well. This helps make workgroups an effective way of managing projects as any user can share their data across institutions and all share access to the same resources. Workgroups can also be organised in a hierarchical manner, for further granularity in access to data and resources.

Another significant feature on the MiG is the addition of a trigger system[7]. This was added shortly before this project and allows for automatic responses to MiG file events, such as data being

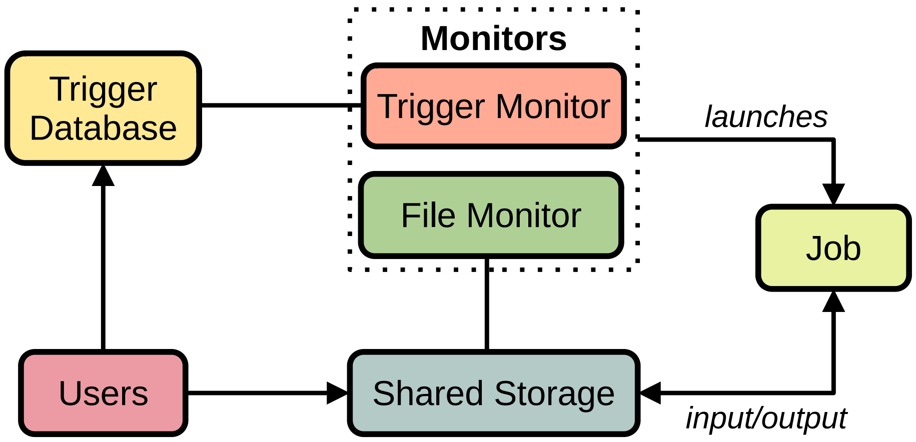


Figure 9.2: Design overview of the MiG trigger system.

created, updated or deleted. Possible responses are common MiG tasks such as scheduling jobs or manipulating files directly. The design for this trigger system is shown in Figure 9.2. Users can create triggers, which are stored within the MiG. A monitor will track any changes within the trigger storage and keep an up to date collection of triggers in memory. At the same time the file system is monitored for events. These events are compared against the triggers in the MiG and in appropriate cases, some task is undertaken. In the diagram this is shown as job scheduling though other command line tasks such as moving data are also possible at this stage.

Of final relevance here, is that the MiG integrates JupyterHub[67], [69], a service for spawning Jupyter instances for a number of users. Using this, users can spawn their own JupyterLab instance. This is especially useful here as an invaluable tool for supporting users research on the MiG. It also provides for a useful platform for MEOW integration. As noted in Section 5.2.4, JupyterLab is designed as an extendable environment and so is well suited to any additions deemed necessary.

9.2 ESSENTIAL PACKAGES

Three non-standard Python 3 packages were necessary to achieve essential functionality for MEOW. These are already present within the MiG project. However, we will now discuss them individually to see if they are still suitable for the `mig_meow` package. These packages are `watchdog`[93], `papermill`[76], and `notebook-parameterizer`[71], and each will be briefly explained here. Note that both `watchdog` and `papermill` are previously existing packages, whilst `notebook-parameterizer` was developed within MUMMERING by Rasmus Munk.

9.2.1 *watchdog*

In order to identify system events in the MEOW implementation, the Python package `watchdog` can be used. It uses the `inotify`[46] API for listening to file events within given directories. This allows for efficient identification of events, without having to repeatedly conduct some form of manual scan. Functions for handling `on_created`, `on_modified`, `on_moved`, and `on_deleted` events are all exposed by the `PatternMatchingEventHandler`, though it itself offers no actual implementation and relies on being overridden by a specific implementation. This suits our requirements perfectly as we can exactly implement how the event path matching actually occurs.

9.2.2 *papermill*

The Python package `papermill` is a tool for executing Jupyter Notebooks either from the command line or within a Python script. Every code cell will be run, in order, with any generated output saved into the Jupyter Notebook in exactly the same way as if it was run manually be a user through the JupyterLab interface. It is also possible to execute a Jupyter Notebook with different input parameters which can passed at runtime. This parameterization means that Jupyter Notebooks can be run repeatedly with different inputs, making `papermill` an important tool in automating their use.

9.2.3 *notebook-parameterizer*

One potential limitation of `papermill` is that it requires a user to identify parameters ahead of time, by grouping them together into a single, tagged cell. Whilst this is not the greatest barrier to entry, it was seen as preferable if users could get away with making no specific preparations to their Jupyter Notebooks. To solve this the package *notebook-parameterizer* could be used. This is a Python based command line tool to replace parameter definitions within Jupyter Notebooks. It does this by reading in a Jupyter Notebook and looking at each line in turn within code cells. If the line contains a parameter definition, then it is replaced with a new value. This avoids having to specifically define new cell tags, and means that almost any Jupyter Notebook as is, can be used as input for `papermill`. One limitation of this system is that parameter definitions in the original Jupyter Notebook are identified using regex. This means that matches are potentially fickle, and though they have been extensively tested there is always the possibility that some unusual parameter definition will not be replaced. That is not expected to happen with any regularity though, and so this potential problem will be tolerated for now.

10

MIGMEOW

The central part of implementing MEOW was in a Python library called `mig_meow` which provides an implementation of MEOW. In this chapter we will examine and explain some of this package, with particular focus on how you can use it to define MEOW constructs, and use them at a local level. This will also lead to the completion of the second thesis objective, to *'Implement an automated, dynamic workflow system'*(see Table 2.1).

10.1 A PYTHON PACKAGE FOR MEOW

With all the definitions, motivations, and first explorations out of the way it is finally time to talk about the core implementation of MEOW, `mig_meow`. This is a Python 3 package, containing definitions for MEOW constructs, as well as functions allowing for their export to the MiG. This makes it compatible with existing MiG code, and usable as a module in in regular Python environments. It is available on PyPI[63], as well GitHub[62].

As the name implies, `mig_meow` is primarily intended as a tool for users to design MEOW workflows and deploy them on the MiG. For this reason it will inherit some behaviour and internal structure from the MiG, most notably it will use the same file event system as its basis. What `mig_meow` is not, is an internal part of the MiG. The MiG functionality is contained entirely within the MiG code itself, available as part of its own repository[61]. These internal MiG alterations will be explained more in Chapter 11.4.

10.2 PATTERNS WITHIN `MIG_MEOW`

The most basic job of `mig_meow` is to provide users with a way of defining Patterns and Recipes. Patterns are implemented as a Python class, as a user-friendly way of modelling the MEOW construct. It would also allow for a number of helper functions to be bundled in with the class definition as will be shown shortly. A sample of the `Pattern` constructor is shown in Listing 10.1. This shows that either a `str` or a `dict` can be used as input. A `str` is used as the name of a new `Pattern`, whilst a `dict` is used to input a complete `Pattern` definition. We can see on lines 7 and 17 that this input is verified so that non-sensible data is not read in. Of note are the various properties of the `Pattern` object that are assigned. These are only shown for the `str` input, but the same ones are used in the case of the `dict`, only with more verification and checking. These parameters are taken directly from the requirements set out in Section 6.1, Table 6.2. Most obviously this gives us the `name`, which all `Patterns` must be given as part of their instantiation as a means of identification. The uniqueness is not covered here per se, but all `Patterns` are stored in a `dict` with the names as keys, thereby enforcing uniqueness within `mig_meow`.

```
1  class Pattern:
2      ...
3      def __init__(self, parameters):
4          # if given only a string use this as a name, it is the basis of a
5          # completely new pattern
6          if isinstance(parameters, str):
7              valid_pattern_name(parameters)
8              self.name = parameters
9              self.trigger_file = None
10             self.trigger_paths = []
11             self.recipes = []
12             self.outputs = {}
13             self.variables = {}
14             self.sweep = {}
15         # if given dict we are importing from a stored pattern object
16         elif isinstance(parameters, dict):
17             valid, msg = is_valid_pattern_dict(parameters)
18             ...
```

Listing 10.1: Code sample showing the constructor for a `Pattern` within `mig_meow`.

The `trigger_paths` parameter is the filter against which events will be matched. It is a complete path, expressed as a literal string, with one wildcard, the '*' character. This character is used to denote any number of other non-separator characters and would be equivalent to '\S*' in regex. Although the `trigger_paths` are stored as a list, and the name implies that there is more than one, currently only the first of these paths is used to match file events. The list is a legacy structure, when it was thought that multiple paths may be used within the same `Pattern`. This may be returned to in the future and so has not been completely removed. Similarly to this is the `recipes` property, which lists by name the Recipes to be used by the `Pattern`. It was envisioned that multiple Recipes may be somehow combined into one meta-recipe but this was never implemented and so the property remains named recipes. Only the first Recipe will be used.

The rest of the properties shown in Listing 10.1 are different forms of variables to pass to the resultant jobs. The variables to be passed to any scheduled jobs are given by the `variables` property, which is stored as a dict. The `outputs` property is similar to this, though these variables are used to identify potential job outputs for the visualisation in the `WorkflowWidget`, discussed in Section 10.4. As outputs are not required by MEOW, and in fact would go against the spirit of the system,

MEOW keyword	MiG keyword	example
PATH	+TRIGGERPATH+	‘workgroup/dir1/dir2/file.ext’
VGRID	+TRIGGERVGRIDNAME+	‘workgroup’
DIR	+TRIGGERDIRNAME+	‘workgroup/dir1/dir2’
REL_PATH	+TRIGGERRELPATH+	‘dir1/dir2/file.ext’
REL_DIR	+TRIGGERRELDIRNAME+	‘dir1/dir2’
FILENAME	+TRIGGERFILENAME+	‘file.ext’
PREFIX	+TRIGGERPREFIX+	‘file’
EXTENSION	+TRIGGEREXTENSION+	‘.ext’
JOB	+JOBID+	*some job id*

Table 10.1: Variable keywords available in MEOW. All examples given are for a triggering file at the ‘workgroup/dir1/dir2/file.ext’ path, within the ‘workgroup’ workgroup. Note that the job id is not dependent on the triggering path and so cannot be predicted from it.

these `outputs` are not used for anything other than the visualisation. Therefore, we can say there is in fact no semantic difference between the `outputs` and the `variables`.

To increase the usability of MEOW, support was added for certain keywords in the `variables` and `outputs`. These are mostly the keywords present in the MiG trigger system, though have been renamed to be more usable. Using them enables runtime variable construction within jobs, using parameters from MEOW definitions. A complete list of keywords are shown in Table 10.1. These keywords can be inserted either on their own, or within larger strings. In either case, they will be replaced at runtime with a string based on how the job was scheduled according to its MEOW defintions. For example, a user could create a variable A, with a value of ‘some/path/{FILENAME}’. If a resultant job was triggered by a file at ‘input/data.txt’, then at runtime A would have the value ‘some/path/data.txt’. This allows for the construction of *variables* without a user being able to predict the exact value at the start of processing. This is especially useful for *outputs*, where output paths can be created based on the input.

As well as conventional variables, users can also use the `sweep` property to define a parameter sweep of variables. A parameter sweep is a term for scheduling multiple otherwise identical jobs, who each have a different value for the same parameter. This parameter is usually taken from a range of values, such as to try multiple values within a simulation to see which ones give accurate results. Within the `Pattern` they are stored as a specific property as they require more information than a standard variable. Namely, they store a `dict` that defines the name of the variable, as well as the complete range of values over which shall be swept.

Finally is the `trigger_file` property. This acts as a variable name for the triggering file, that being the file that caused an event which matches the path in the `trigger_paths` property. The triggering file itself is passed as a variable to the job, and so the `trigger_file` is required to act as a variable name with the job. Although the attributes outlined over the last few paragraphs are

1	add_single_input(input_file, regex_path, output_path=None)
2	add_param_sweep(name, sweep_dict)
3	add_variable(variable_name, variable_value)
4	add_recipe(recipe)
5	integrity_check()

Table 10.2: Selected Pattern method signatures.

far from the most complex properties ever defined in a Python class, it is still unlikely that a user will be able to correctly guess the expected format to store them all without some help. For this reason a number of methods are also provided within the Pattern. These are not described in detail, though some of their signatures are shown in Table 10.2. Each comes with the appropriate checks and verification so that only values which make sense can be assigned, and provides feedback when this is not the case. The fifth function, integrity_check is different in that is checks all defined properties within the Pattern for errors, and checks that values are given for essential properties such as the trigger_paths. If there are no errors, and all necessary parts are defined then integrity_check will return a Tuple of the form (True, None). If there are problems it will return a Tuple of the form (False, str), with the str being a debug message explaining what had failed the check.

10.3 RECIPES WITHIN MIG_MEOW

Recipes are in some ways much simpler than Patterns, as they can already be expressed by a Jupyter Notebook. This is as according to the requirements set out in Section 6.1, Table 6.1, only two properties are actually necessary in a Recipe. Those are that it has a name, and some code. These properties are entirely covered by a Jupyter Notebook, which has a filename and can, by its very nature, express runable code in its dedicated code cells.

To register a Jupyter Notebook with the mig_meow, the function register_recipe(source, name=None) has been provided. This takes a Jupyter Notebook as a source, given by a path to the Notebook document itself. Optionally a name can be provided, else the filename is used as a name. The JSON within the notebook is read in and saved alongside the source path, and Recipe name in a new dict.

The reason we need these additional steps of reading in the document and saving their state separately to the file system, is that the Jupyter Notebook is still 'live' in file system and can be modified or run at any time. This would potentially make quite a mess of race conditions, as the same document could then be saved, read, or modified by several processes at the same time. By saving a separate copy of the Jupyter Notebook we can avoid this completely, even if it does mean a certain amount of data replication.

10.4 WIDGETS AND DESIGN AIDS

So far we have discussed how a user may define `Patterns` and `Recipes` through the use of specific functions and classes. However, as has been mentioned previously the main intended method for users to do so is within a Jupyter Notebook. This allows for the possibility of using `ipywidgets`[47] to create custom widgets that can be run in the Jupyter Notebook. These are graphical objects available within the notebook browser itself that can express, modify, and respond to the current state of the running code. Naturally, this makes them perfect as a user friendly way for creating `Patterns` and `Recipes`, as the definitions could be accompanied with visualisations.

To accomplish this, three widgets are defined as part of `mig_meow`. These are used to define MEOW objects and send them to the MiG, to monitor MiG jobs, and to provide final reporting on MiG workflows. As the MiG interactions will not be considered until Chapter 11.4, the later two widgets will not be examined until then. This leaves the first, and largest widget for now, that being the `WorkflowWidget`.

The `WorkflowWidget` is designed as a complete environment for creating, reading, updating and deleting both `Patterns` and `Recipes`, both at a local level and on the MiG. As can be seen in Figure 10.1, it has a large area for visualising the currently defined Patterns and Recipes. Below this is an area for modifying the internal state of the widget. Forms can be accessed to create, modify or delete `Patterns` and `Recipes`, with any changes immediately reflected in the visualisation. This gives users up to date feedback on what they are defining, and how it is expected to behave.

Do note that the visualisation is just an intelligent guess at what Patterns will link. It makes assumptions such as if one Pattern outputs to a directory and one is triggered by that same directory then they are linked. This seems like a plausible assumption and so we can construct a visualisation of some expected emergent workflows, if users use the system in a straightforward, predictable manner. However, users are fully capable of creating more complex structures such as a Pattern that only triggers on certain file types within a given directory. Depending on how this is set up this may appear as a link or may not. This is only highlighted here to demonstrate that this visualisation is just that. It is not used to identify or schedule jobs, and should be treated only as a rough guide to what has been defined.

The visualisation contains two types of node. Firstly are the coloured circles. These express `Patterns`, with their name labelling them so that they can be identified. If a user scrolls over them

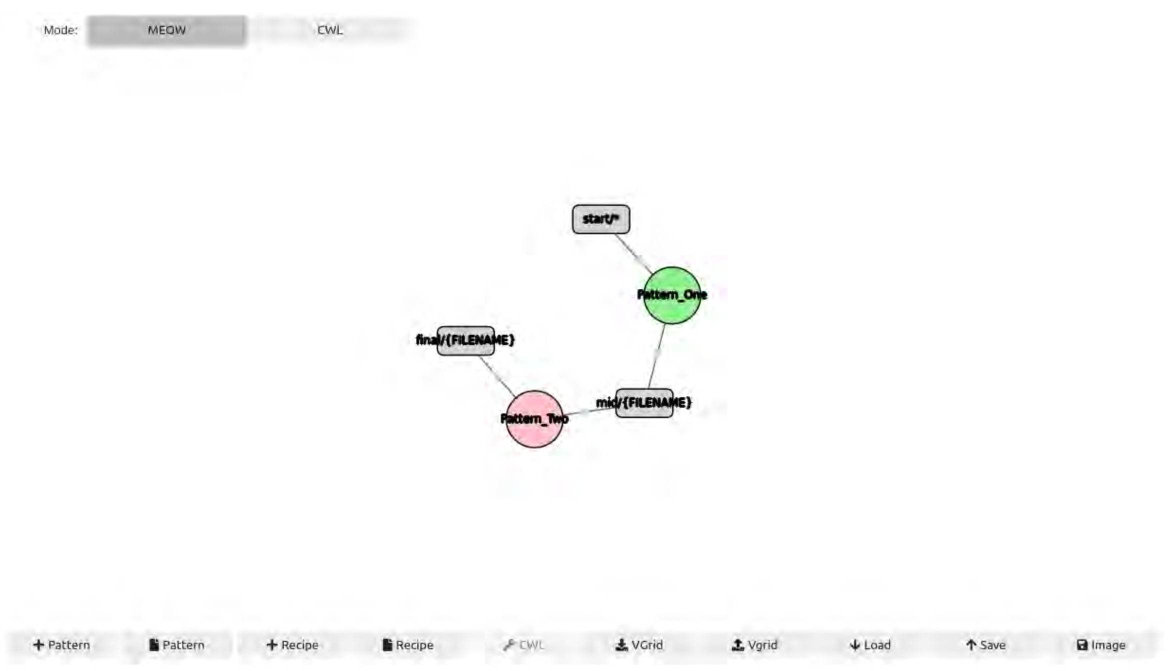
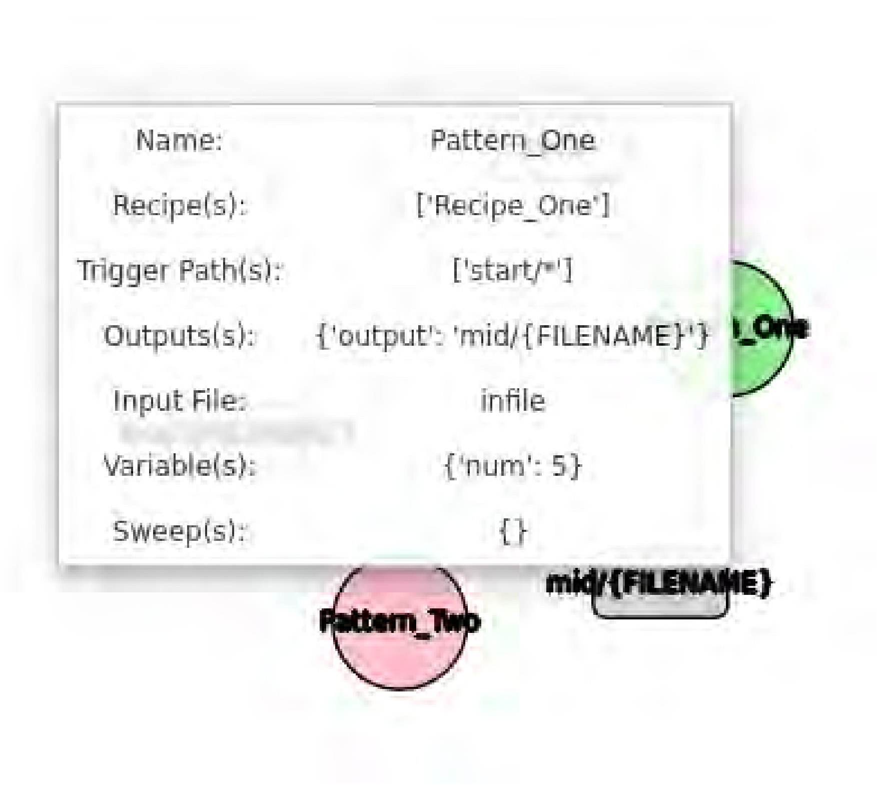


Figure 10.1: Overview of the `WorkflowWidget`. In this example some `Patterns` and `Recipes` have already been defined.

a pop-up will appear showing the internal details of the `Pattern`, as can be seen in Figure 10.2. The colour of the circle denotes if the `Recipe` defined in the `Patterns recipes` property is currently defined or not. If no `Recipe` sharing the specified name exists then the circle is coloured red, and if it does, it is green. This is to clearly highlight what `Patterns` in the currently displayed system are actually capable of triggering or not. In a potentially difficult to debug system like this, where working out why an event did not produce processing can be unclear, a user needs all the help they can get and this colouring is of clear assistance. The second type of node are grey rounded rectangles. There show file locations both as input and as output. The directed edges, or arrows show potential routes through the system, with file locations connected via `Patterns`. The connections are possible due to the `outputs` defined as part of a `Pattern`. Because those `outputs` are only a user provided guide, this visualisation should also only be taken as an expectation, and not as gospel for what will actually happen.

An example form used in the widget is shown in Figure 10.3. It shows the form for creating a new `Pattern`, with similar ones being present for registering new `Recipes`, or modifying existing definitions. Each parameter is shown on the left, with values entered on the right. Observe that the top parameter to be entered, the name, has had the help button on the far right clicked. This is available for each parameter and in each case will toggle the display of some text explaining what the parameter is along with an example value. For parameters that can accept more than one value, there are additional buttons on the right to add or remove additional input boxes as necessary. There are two buttons at the bottom of each form, one to apply all changes and one to cancel the creation/modification with

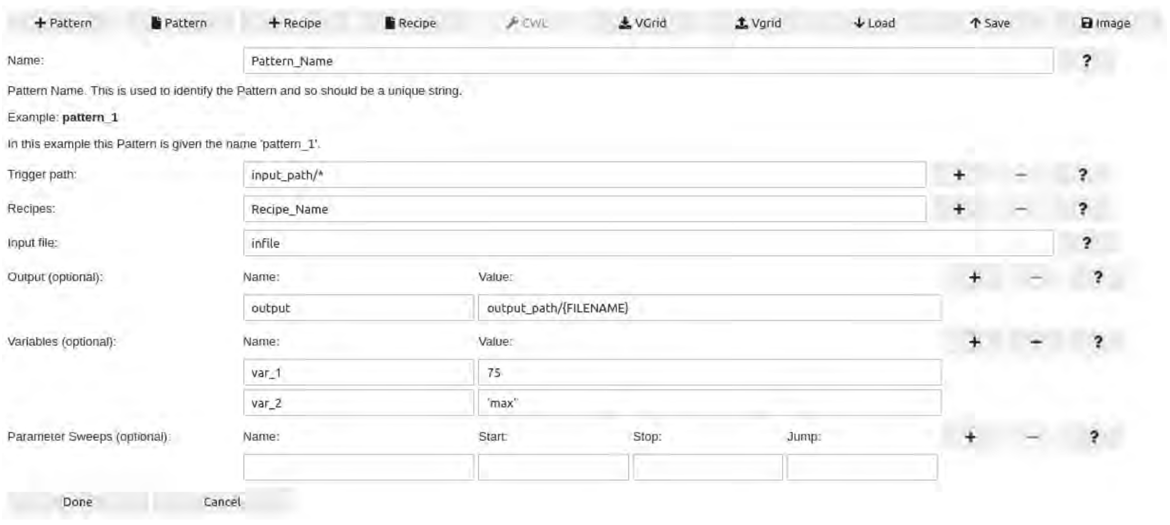


Name:	Pattern_One
Recipe(s):	['Recipe_One']
Trigger Path(s):	['start/*']
Outputs(s):	{'output': 'mid/{FILENAME}'}
Input File:	infile
Variable(s):	{'num': 5}
Sweep(s):	{}

Figure 10.2: `Pattern` feedback within the `WorkflowWidget` visualisation.

no changes applied. The only significant difference between the creation and modification forms is that the modification form features a drop-down list at the top. This is used to select from all of the currently created `Patterns` and `Recipes`. Once a `Pattern` or `Recipe` has been selected a form will be generated that is identical to the one used for creation, save that the name input is missing, and the form is pre-populated with the existing parameter values.

Once created, `Patterns` and `Recipes` can be saved into the local file system, where they are saved as YAML[101] dictionaries into files. YAML was picked here as it is a robust, commonly used format that users are probably already familiar with. These files can be freely copied, modified, or even created from scratch by a user as they would any other file. The files can be read in by `mig_meow`, making this another way for users to create `Patterns` and `Recipes`, or to share their definitions with others. Both reading and writing can be done through the `WorkflowWidget` or programmatically. If done through the `WorkflowWidget`, the widget will first identify any Pattern and Recipe files it could import and inform the user what it is has found. They are then given the choice to continue with the import or not, as unique naming means any currently defined constructs by the same name will be overwritten.



Name: Pattern_Name

Pattern Name. This is used to identify the Pattern and so should be a unique string.

Example: **pattern_1**

In this example this Pattern is given the name 'pattern_1'.

Trigger path: input_path/*

Recipes: Recipe_Name

Input file: infile

Output (optional):	Name:	Value:		
	output	output_path/{FILENAME}		
Variables (optional):	Name:	Value:		
	var_1	75		
	var_2	'max'		
Parameter Sweeps (optional):	Name:	Start:	Stop:	Jump:

Figure 10.3: The form used to create a `Pattern` with the `WorkflowWidget`.

10.5 THE LOCAL RUNNER

As a final significant point when talking about `mig_meow` at the local level, it is worth looking at the `WorkflowRunner`. This is a means for a user to run a MEOW system on their local machine. The `WorkflowRunner` was actually a fairly late addition to the project and had the specific goal of mimicking the functionality of the MiG, so that readers without access to the MiG could run their own analysis and see how the MiG would behave. It will be presented here as it can of course be used entirely independently of the MiG, and as it implements all of the previously mentioned MEOW definitions, is a simple but functional workflow processing system in its own right.

10.5.1 *CSP in* `multiprocessing`

The MiG uses a central queue to manage all jobs within the grid, with multiple processes able to add jobs. Any number of different resources can poll the queue for a suitable job, and so multiple processes may also read and remove jobs. To mimic this functionality the `WorkflowRunner` will need to be composed of multiple processes running concurrently. This will however present a number of design challenges, such as avoiding deadlock or race conditions.

To avoid these problems, a design model was used as this could act as a guide for good practice. In this case, the Communicating Sequential Processes (CSP)[43] model was used. This is as CSP can provide guarantees against deadlocks, livelocks and certain race conditions by rigidly sticking to the client-server design pattern, first presented in [40], and slightly updated [94] and [55]. This design pattern defines two types of communicating processes:

1. **A client process**. In any communication, this is the process that instigates the communication. If the server responds to this initial communication, then the client guarantees that it will handle this reply immediately.

2. **A server process**. A server process will never instigate communication. If it receives a request from a client and is expected to respond, the server commits to responding within a finite amount of time.

It is possible for a single processes to act as a client and server at the same time, but they must remain consistent in their behaviour within their communication to any other single process. Any communications can be plotted in a graph, and labelled so that we can see which end is acting as a clients and which a server. As long as no continuous circle of clients and servers oriented the same way is formed, then according to the model outlined in [55], deadlock and livelock can never occur.

A Python based implementation of CSP exists, called `PyCSP`[11], but it has not been maintained for several years and so was not deemed acceptable. To get a CSP based implementation within Python, without building the whole framework from the ground up meant using some lower level constructs in a CSP-like manner. This was achieved by using the `multiprocessing`[65]. This lets us start multiple processes concurrently, and offers constructs for them to communicate. It is part of the Python Standard Library, and so it is reasonable to expect it to be both widely supported and maintained. Despite not using an actual CSP implementation, the CSP model is still of use to us. This is as it can provide some of the core design principles used to assess the correctness of our design. For instance, according to CSP, no processes should share access to data, so by definition, we will avoid all race conditions on the underlying data. As we will see later in Section 26.2.1, this does not eliminate all race conditions, only those on in-memory values.

Any communication of data between processes in CSP is done via the `Channels` construct. In `multiprocessing` this can be done through `Pipes` or `Queues`, though they both function somewhat differently to `Channels`. Where a `Channel` is intended as a synchronous, one way means of communication, both `Pipes` and `Queues` are asynchronous, and can be used bi-directionally. The main difference between `Pipes` and `Queues` is that a `Pipe` is much quicker, though is limited to linking two processes, whilst a `Queue` can be shared amongst many. As all inter process communications will be defined ahead of time, and can be expressed as one to one connections, `Pipes` stand out as the better choice due to the increased performance. As they are a stand in for `Channels`, `Pipes` shall be used in a strictly one-way fashion. This means that the only conceptual difference between `Pipes` and `Channels` is if they are synchronous or not. However, note that in the process definitions outlined above, synchronicity was never a required feature for us to guarantee freedom from deadlock or livelock, and so this difference can be tolerated.

A very useful property of CSP is that if process interactions are setup appropriately, then the system is guaranteed to be free of deadlock and livelock. Although we are not using CSP directly, we can apply its principles to achieve the same result. The key way in CSP to avoid deadlock, is to avoid a circular loop of dependent communication[50]. As can be seen in Figure 10.4, this has been done as there is a linear hierarchy of primary processes communication. There are in fact only two types of communication within the runner system. Firstly, there is one-way communication such as from the *State Monitor* to *Admin*. Secondly, there is two-way communication with an expected response, such as between a *Worker* and a *Timer*. In these cases a response is always given by the queried process, and that response is generated as soon as a query is received. We have therefore stuck to the requirements set out in [55] and so can state that this system will not suffer from deadlock or livelock.

As a final note, CSP systems often use an `ALT` construct to make a choice between several input channels. The `multiprocessing.connection` package contains the very similar `wait` function, though it must be used precisely so as to replicate the necessary characteristics of an `ALT`. The `wait` function will hang a process until one or more input connections have a message ready to be received. This differs from an `ALT`, which should only ever return one connection. If multiple connections are ready in an `ALT` at the same time, it will make a choice between them. This priority system will be useful within our `WorkflowRunner` design, so we need to use the `wait` function deliberately to replicate this CSP feature. An example of how this was done is shown in Listing 10.2. In this example there are three potential input `Pipes`, from the *state monitor*, *user* and *file monitor* processes. In the unlikely event that two or more readers are available at the same time, they will be prioritised in that order. The key part to note is the use of the `elif` statements, so that only one of the inputs is selected.

```python
def administrator(...):
  ...
  while True:
    ready = wait([
      from_state,
      from_user,
      from_file
    ])

    if from_state in ready:
      input_message = from_state.recv()
      ...
    elif from_user in ready:
      input_message = from_user.recv()
      ...
    elif from_file in ready:
      input_message = from_file.recv()
      ...
```

Listing 10.2: An abridged ALT-like `wait` statement.

Using a CSP based approach to design the `WorkflowRunner` means we can create a well formed, multi-process system. It will avoid common pitfalls of concurrent and parallel computing such as

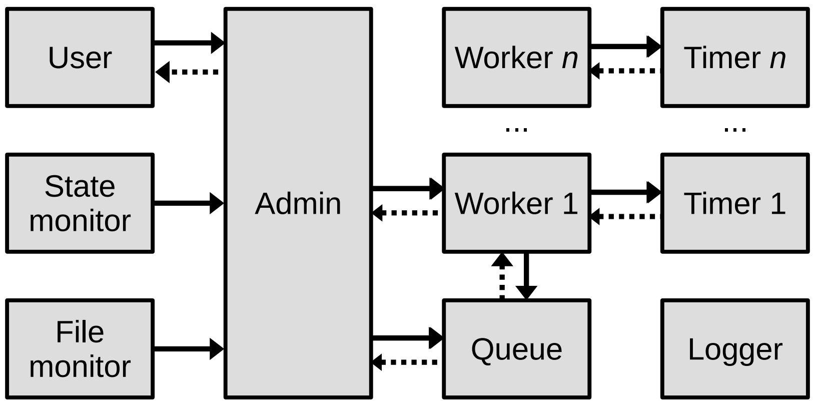


Figure 10.4: Process structure of the `WorkflowRunner`, showing individual processes and their interactions. Note that in addition, the admin, state monitor, file monitor, queue and worker processes also can send messages to the logger process, though these connections have not been shown for brevity. Secondary connections used only for replies are shown in dotted lines. Zero to *n* workers are created based on user input. Taken from [53].

deadlock, and each process should be free to respond in a short amount of time to any messages, helping to ensure the system is responsive and predictable.

10.5.2 *Outlining the Local Runner*

Using the CSP design principles, a multiple process structure was devised, as shown in Figure 10.4. Each process will be explained fully in the following paragraphs. These descriptions are an expanded version of the text first presented in Appendix B.

10.5.2.1 *The USER process:*

The *User* process is the base process in which the constructor for the `WorkflowRunner` is called, and from which the `WorkflowRunner` object is returned. Within the constructor, all other processes are setup and started. The `WorkflowRunner` object is then used as the entry point for any user interaction, with each sending an appropriate message to the `Admin` process. A response is always expected from the `Admin`. An exhaustive list of all provided functions will not be provided here, though they include all necessary functions for the creation, updating and deleting of `Patterns` and `Recipes`, and for monitoring the continued status of the `WorkflowRunner`.

10.5.2.2 *The STATE MONITOR process:*

Both the *State Monitor* and *File Monitor* implement classes that inherit from the `watchdog`[93] class, `PatternMatchingEventHandler`. The `PatternMatchingEventHandler` responds to system events, according to given sub-paths from a watched directory. In the case of the *State Monitor*, this is the hidden '.workflow_runner_data/' directory, with the sub directories 'patterns/' and 'recipes/'. These locations are used to store files defining Patterns and Recipes, in a manner similar to how they are stored on the MiG. These files can be altered and updated at any time either by direct user interaction, or through using functions from the *User* process. In either case, this monitor will catch any changes and send any updates to the `Admin` process. No response is ever expected from the `Admin`, so the *State Monitor* process should never be blocked, and is therefore always able to process new events in a timely fashion.

10.5.2.3 *The FILE MONITOR process:*

The *File Monitor* is very similar to the *State Monitor* process, though its monitors the base data directory. The base directory is equivalent to a workgroup on the Minimum intrusion Grid (MiG). As the *File Monitor* does not know what `Patterns` and `Recipes` have been established, it can do relatively little processing of events itself. All it can do is filter out irrelevant events, such as 'delete' events, or bunch together repeated events at the same file location so as to not spam the `Admin` process. Whenever an appropriate event is identified, it is sent to the `Admin` to be checked against the registered `Patterns` and `Recipes`. This differs from the MiG, where a more complex, overlapping structure is achieved through careful use of some shared state and locks. This complexity is not necessary here as the `WorkflowRunner` is not intended to manage a whole grid systems worth of events, so sending all events is sufficient for the scale of a local problem.

10.5.2.4 *The ADMIN process:*

By far the most complex process is the *Admin*. It maintains the in-memory state of the runner, in which all currently registered `Patterns` and `Recipes` are stored. Updates to this state are provided by the *State Monitor* process, ensuring that the in-memory state is up to date with the saved state expressed in the Pattern and Recipe files. `Patterns` and `Recipes` can also be added, removed, or modified via user interaction from the *User* process. Any changes will result in the appropriate update to the file state, with files being added, removed, or updated. This will in turn generate more updates from the *State Monitor*. To prevent a circular loop of events creating file writes which are interpreted as events, files are strictly only written by the *Admin* process if a change has occurred to its in-memory state.

As `Patterns` and `Recipes` are received from the *State Monitor*, appropriate Rules are maintained in accordance with the requirements set out in Section 6.2. This means the *Admin* will always have an up to date list of Rules. This is important as the *Admin* process will also receive input from the *File Monitor*. These events will be compared against the current list of Rules. If the event path matches the event filter attribute of a Rule, then the *Admin* will create a new job, and send its ID to the *Queue*, so that it may be processed. Creating a new job consists of creating a unique ID, with a corresponding job directory created to store job files. These files are a new Jupyter Notebook file, created as a copy of the appropriate `Recipe`, along with two YAML files. The first of these files is the *parameters file*, which contains the variables defined by the appropriate `Pattern`, such as the triggering path. The second of these contains and the jobs meta information, such as when it was scheduled, and its current running status.

As well as this core functionality, the *Admin* deals with requests from the the *User* process. Aside from the previously mentioned adding or modifying `Patterns` and `Recipes`, users may also query the current state, or running status of the `WorkflowRunner`. Some requests, such as to query the current queue composition require further messages to be sent to the *Queue* before a response can be generated, but a response is inevitable and provided as soon as possible. The *Admin* process utilises a `wait` statement to stand by until receiving input from either the *State monitor*, *User*, or *File Monitor* processes. These three inputs are prioritised in the order given, so that if multiple are available at the same time, only the first is read and processed.

Messages from the *State Monitor* are always of the highest priority as a fresh state will always be needed by the *Admin*. Changes in the state file system will also be finite in nature, as a user is incredibly unlikely to make so many changes to `Patterns` and `Recipes` so as to swamp the runner whilst it is ongoing. Secondly, are messages from the *User* process. These are secondary as they will be requests from a human, and so will be conducted on a human time-frame. This means that they do not need to be responded to within nanoseconds and so can wait behind any *State Monitor* updates.

Lastly, this leaves the *File Monitor*. This may produce a theoretically infinite number of messages as there is no limit on the number of files created or updated by jobs or users. Despite this being an unlikely use case, it is nevertheless a possibility and should be accounted for, therefore it must be the lowest priority as anything behind it could be eternally starved in this scenario.

10.5.2.5 *The WORKER process:*

Jobs are executed within the *Worker* processes. The amount of *Workers* to be spawned is determined by the user, and at least one is needed if the workflow runner is to process jobs. Each *Worker* has its own `Pipes` from the *Admin* and to the *Queue*. By default a *Worker* starts in a stopped state, and will only start when told to do so by the *Admin*. This allows users to setup and experiment with `Patterns` and `Recipes` before any processing takes place. Once a worker is started, it will request a job from the *Queue* process. If a job is available, the ID will be sent to the *Worker*, and its definition files are read from the job directory created by the *Admin*.

The job itself is processed by first parameterising the input notebook using the python module `notebook-parameterizer`[71]. Parameters are extracted from the *parameters file*, created by the *Admin* during job creation. The parameterised Jupyter Notebook is then run using `papermill`[76] in the same manner as is done on the MiG. Once execution has been completed, the job files are copied into a separate job output directory where they can be individually inspected. Jobs may produce output directly into the data directory, monitored by the *File Monitor*, in the same manner as can be done within the MiG.

If no job was available in the *Queue*, the *Worker* sends a notification to its *Timer* process to start sleeping. If a job completes, or the *Worker* is notified by the *Timer* that its sleep is over, it will poll the *Queue* for another job. This polling of the *Queue* will loop until the *Worker* is manually stopped by user input.

10.5.2.6 *The TIMER process:*

To prevent spamming the *Queue* process with requests for new jobs, each *Worker* has its own *Timer* process. This process will wait for a start signal from their *Worker* and then sleep. Once the sleep is over, it will send a signal to the *Worker* as a prompt to request a job again from the *Queue*. Each *Timer* will sleep for a different length of time, so that if multiple *Workers* are waiting at the same time, they should not all wake together and spam the *Queue*. By having the timer in a separate process rather than internal to the *Worker*, the *Worker* is still free to receive messages from the *Admin*, which would not be the case if it itself were sleeping.

10.5.2.7 *The QUEUE process:*

The *Queue* process acts as a buffer for all jobs that have not yet been processed by a *Worker*. It accepts messages either from the *Admin* or any of the *Worker* processes. From the *Admin*, the *Queue* will either receive the identity of a new job to be added to the queue, or a request for the current composition of the queue. Alternatively, any of the *Workers* may request the identity of a new job to execute. In any case, a response is always immediately generated and sent. It was necessary to separate the queue into its own process, rather than having it stored within the *Admin*, as otherwise there would be a risk of deadlock between *Workers* and the *Admin*. This is as each could potentially be the source of new communication and so a circular loop of interaction would occur. A separate queue solves this problem.

10.5.2.8 *The LOGGER process:*

Every non-*Logger* process except the *Timer* processes has a `Pipe` to send messages to the *Logger*. These are messages to be written to a log file for debugging, and/or printed to the console if the appropriate flags are set during `WorkflowRunner` creation.

10.5.3 *Using the* `WorkflowRunner`

The `WorkflowRunner` can be created either within a Jupyter Notebook, or as part of a regular Python 3 script. In either case it can be run as daemon process, meaning it will not block further processing in the containing script. This is significant as unlike in a static SWMS, a MEOW system is continuous and lacks a defined end. By being able to run as a daemon, users can start the `WorkflowRunner`, and then run additional commands. These could then for instance, interact directly with the `WorkflowRunner` itself, or could be used to setup new files to trigger MEOW events.

Feedback from the system can be gained mostly from regular messages to `sys.stdout`, which will usually take the form of print messages in a terminal or Jupyter Notebook output cell. More in depth debug messages can also be produced to a log file, though this is not enabled by default and will require a user to set a flag during `WorkflowRunner` creation. Other arguments that can be set during creation as the `WorkflowRunner`, are the locations for the different state, job and data files. These can be changed to be whatever the user needs, and so could be within each other, or even the same directory. This is not advised behaviour for a beginner, but is highlighted at this point as by doing something like this it would be possible to create very interesting, self modifying systems. For example, it would be possible to create a system where certain jobs create or modify Patterns, and so create even more dynamic analysis algorithms.

A more regular use case would be to use the `WorkflowRunner` to run a simple MEOW system. This could be done by creating some Patterns and Recipes programmatically within a script and then calling the `WorkflowRunner`, as is shown in Listing 10.3. This has the advantage of being an almost entirely contained script, in that it has all of the `Pattern` and `Recipe` definitions contained within one document that can easily be re-run, shared, or modified as needed. In this example the Recipe is registered directly from a separate Jupyter Notebooks, shown in Appendix G. This Jupyter Notebook would need to be kept alongside the script for it to work. However, this could be avoided by constructing the relevant Jupyter Notebooks also within the script using a package such as `nbformat`[70]. For brevity, this has not been shown.

```python
import mig_meow as meow

add5 = meow.Pattern('Add_5')
add5.add_single_input(
    'infile',
    'initial_data/*')
add5.add_output(
    'outfile',
    '{VGRID}/int_1/{FILENAME}')
add5.add_variable(
    'extra',
    5)
add5.add_recipe('addition')

patterns = {
    'Add_5': add5
}

recipes = {
    'addition': meow.register_recipe('add.ipynb', 'addition'),
}

meow.WorkflowRunner('Test', 1, patterns=patterns, recipes=recipes)
```

Listing 10.3: An example of the `WorkflowRunner` created with `Patterns` and `Recipes`.

```python
import mig_meow as meow

patterns = {
    'Add_5': meow.read_dir_pattern('Add_5')
}

recipes = {
    'addition': meow.read_dir_recipe('addition'),
}

meow.WorkflowRunner('Test', 1, patterns=patterns, recipes=recipes)
```

Listing 10.4: An example of the `WorkflowRunner` created with previously made `Patterns` and `Recipes`.

```yaml
input_file: infile
input_paths:
- initial_data/*
output:
  outfile: '{VGRID}/int_1/{FILENAME}'
parameterize_over: {}
recipes:
- addition
variables:
  extra: 5
```

Listing 10.5: The YAML file 'Add_5' expressing a `Pattern` object.

```yaml
recipe:
  *Jupyter notebook code not shown for brevity*
source: add.ipynb
```

Listing 10.6: The abridged YAML file 'addition' expressing a `Recipe` object.

An alternative approach when using the `WorkflowRunner` is to use pre-defined Patterns and Recipes, such as from the `WorkflowWidget`. An example of this is shown in Listing 10.4, with the

`Pattern` and `Recipe` files in 10.5 and 10.6 respectively. Note that in the case of the `Recipe` file, the actual file is much longer as it contains the entire source Jupyter Notebook JSON definition within the recipe entry. This has again been excluded for brevity.

Regardless of how the `WorkflowRunner` is set up, it can be used to easily run a MEOW system on users local machine. This will be demonstrated in detail throughout Part iii. It is intended primarily as a tool for testing Patterns and Recipes before they are deployed to the MiG. For this reason it closely mimics the behaviour and setup of the MiG. This means that the `WorkflowRunner` will have some delays thanks to the jobs not being processed until a *worker* process signals that it is ready to start executing. It is also noteworthy that the `WorkflowRunner` has very limited job reporting in its current state. This is as it is somewhat simplistic, only giving real-time feedback via printing to console, or via logging. Currently there is no way for a user to get a complete report of the analysis carried out. This contrasts heavily with the tools available for running processing on the MiG. Regardless, the `WorkflowRunner` can be used to effective run a MEOW based system, both as a test bed for MiG interactions, and as an analysis platform in its own right.

11 MIG

Despite the utility of the `WorkflowRunner`, it was never intended as the primary provider of MEOW systems. That duty falls to the MiG. As previously discussed, some of the groundwork was already in place on the MiG at the beginning of the project, most notably the trigger system[7]. However, new developments needed to be created to accommodate the creation of Patterns and Recipes, and to interact with `mig_meow`. In addition, further considerations such as security and authorisation would also need to considered on a shareable, widely accessible system. However, once implemented, these could then be considered an answer to the thesis objectives to *'Integrate the automated, dynamic workflow system into the collaborative big data platform'*, as stated in Section 2.2.

11.1 MEOW DEFINITIONS ON THE MIG

The MiG already has several features that will be of use to a MEOW system. The workgroups already act as an online, shareable, collaborative space for storing and processing data. Secondly, the system of triggers recently added to the MiG would be an ideal foundation for the MEOW implementation. These are created on a per-user, per-workgroup basis. By that, it is meant that all triggers must be created within a workgroup and always act within that workgroup, but that triggers are not shared between users of that workgroup. This is not a perfect fit for the Rules as described in Section 6.2 and so some modifications are needed. Ideally, we need a trigger system that is based within a workgroup, but that is shareable by all users within that workgroup

An abridged section of the MiG structure is shown Figure 11.1. It shows that the MiG is comprised of workgroups which each in turn contain triggers. Between these two concepts have been inserted the MEOW constructs of Patterns and Recipes. This would directly mirror the definition shown in Figure 6.1, with the MiG triggers acting as MEOW Rules. Note that within some papers and documentation, MiG triggers are referred to as rules, and that this is what the MEOW construct was named after. For

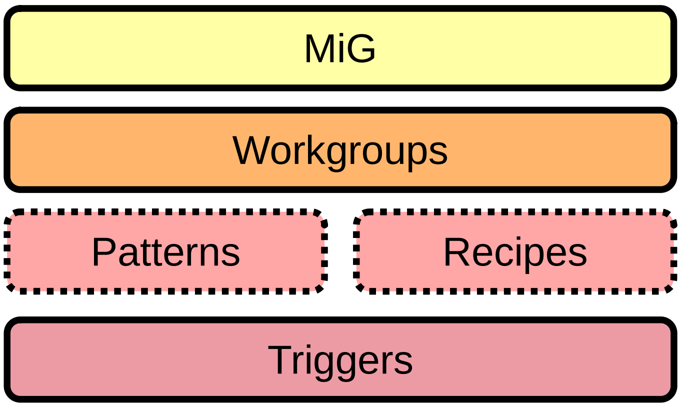


Figure 11.1: Internal structural hierarchy of the MiG, with added MEOW constructs. Structures added by this project are shown with a dotted border. Note that only the parts directly relevant to this project are shown.

clarity, within this thesis the word rule will always be used to refer to the MEOW construct, while the constructs previously existing on the MiG will always be referred to as triggers.

Before we get to explaining how Patterns and Recipes would be added to the MiG, it is worth mentioning how they were not. The most obvious way one would assume would be to import `mig_moew` and use the definitions it provides. However, `mig_meow` was actually only started some time into the MiG implementation of MEOW and so was not available at the start to provide any definitions. For this reason, the implementation of Patterns and Recipes on the MiG are different to those explained in Chapter 10.5.3, though they share the same requirements and characteristics.

The main significant difference is that Patterns on the MiG are not modelled as a class object, but by a `dict`. This was as the use of `dicts` over custom classes was preferred within the MiG project. However, the MiG Pattern `dicts` express all of the same parameters as the `mig_meow` `Pattern` class. Recipes are a much closer match as in both the MiG and `mig_meow` they are `dicts`, expressing the same parameters. Both Patterns and Recipes on the MiG contain a number of additional parameters to those specified in the MEOW requirements. These are for characteristics such as the type of object they represent, or recording who first registered them. All of these parameters are used internally to the MiG and are mostly just used for internal validation so shall not be considered further.

Within the MiG, Patterns and Recipes are defined along with the rest of the MEOW code in a single module at `mig.shared.workflows`. This is in accordance with previous conventions within the MiG project, where similar functionality is grouped into modules, and definitions that are used in numerous places are contained within the `mig.shared` package.

First and foremost `mig.shared.workflows` allows for the creation of Patterns and Recipes. This is done in accordance with the requirements set out in Section 6.2, and so once appropriate Patterns and Recipes have been created, triggers will also be created. As well as a standard MiG

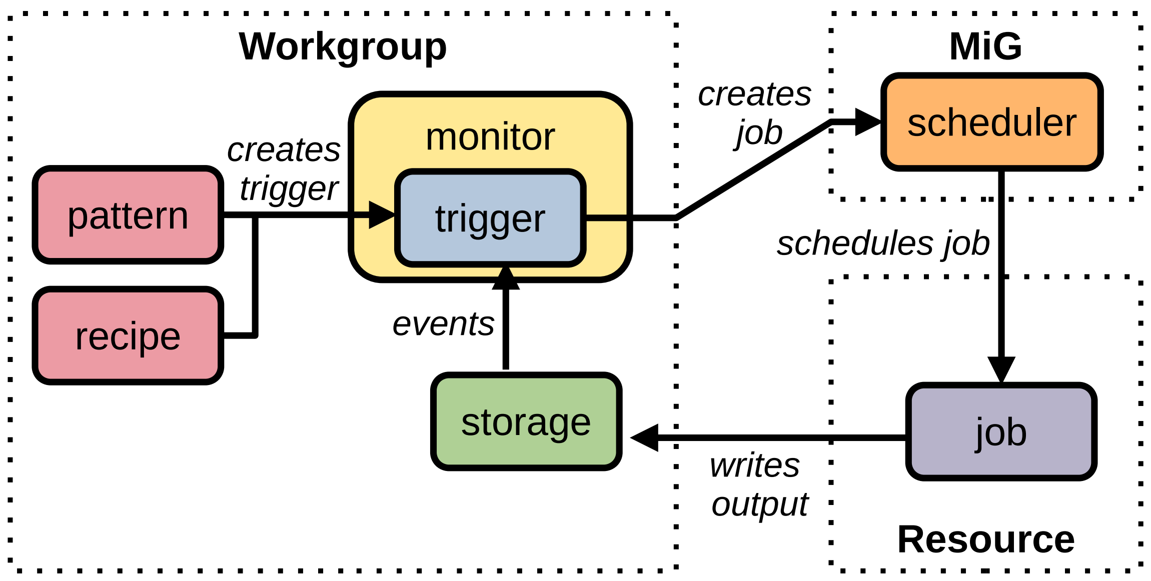


Figure 11.2: MEOW job processing on the MiG.

trigger, two additional files are also created. These are the *parameters file* and the *task file*. The *parameters file* is the same as the one created within the `mig_meow` `WorkflowRunner`, in that it is a YAML file containing the variables defined within the appropriate Pattern. Meanwhile, the *task file* is a new Jupyter Notebook file, used to execute any resultant jobs. This was created for the same reasons as the job file in the `WorkflowRunner`, to avoid users directly altering the job code after registering a Recipe and so producing unexpected output. An additional reason in this case is that when the Recipe implementation was first created on the MiG, it was seem as an option that they may be defined in multiple ways. For instance, a Python script may be registerable as a recipe, but would be run as a Jupyter Notebook. Some automatic conversion would have to take place, which would be done be the creation of the *task file*. It could also be possible for a Pattern to specify multiple Recipes, which would then be sewn together into a single *task file*. To be clear, none of this has been implemented and currently the *task file* is a straight copy of the Recipe specified during trigger creation.

The job identification, scheduling and processing using the `WorkflowRunner` can be taken as a guide for the same on the MiG. Although the MiG does include a number of additional steps, these are mostly validation and authorisation steps that are not needed on the `WorkflowRunner` due to its local nature. The additional steps do not significantly alter the functionality to an authorised user, so the process for creating MEOW jobs can be expressed as shown in Figure 11.2. This shows how within a MiG workgroup, a user can create Patterns and Recipes. From these definitions, MiG triggers are created which will monitor the workgroup file structure for events. If an event occurs at a path that matches a trigger, then it will create a new job and place it in the MiG job queue.

Any resources currently available to the MiG will be periodically polling the MiG queue for potential jobs, and if any are available they will be claimed by the resource and executed. During job execution the job *task file* is parameterised according the *parameter file* using the `notebook_parameterizer` package. The resultant Jupyter Notebook is then run using `papermill`. To achieve this, all jobs created from MEOW based triggers, will be created using the same template, shown in Appendix H. This template is a single, multi-line string into which variables will be substituted using the *template* `dict`. This `dict` contains a number of variables to substitute in, but the most significant here are the *execute* lines. These are the actual lines of code that are executed to run a job. In most MEOW jobs there are two of these, with an example of them shown in Listing 11.1

```
notebook_parameterizer task_file -o job_notebook.ipynb -e
papermill job_notebook.ipynb job_result.ipynb
```

Listing 11.1: MEOW job execution lines example.

A significant difference in job execution between the `WorkflowRunner` and the MiG, is how access to the storage file system is achieved. Within the MiG, the file storage can be mounted within a resource using SSHFS. This allows for the acces to and management of the workgroup files from a resource via an SFTP connection. This means that if a user needs to access other files or directories, they do not have to manually copy them across to the resource, but can instead navigate to them relative to the base workgroup. Similarly, output can be written straight into the file system without having to be predefined, in accordance with the requirements of MEOW.

The trigger functionality was not directly altered during MEOW development as the MiG is a live system, and users may be using the triggers as they are. For this reason any triggers created from MEOW definitions are still locked to their user, and so are only visible to their creator. When Patterns and Recipes are created on the MiG they are stored on a per-workgroup basis, and can be retrieved by anyone who has access to that workgroup. Workgroups already have different levels of users, known as owners and members. Owners can view and edit and Patterns and Recipes in a workgroup, whilst members can only view any, but only edit their own. In either case, all other currently created Patterns and Recipes within the workgroup are used for trigger identification and creation whenever a Pattern or Recipe is created or modified on the MiG.

11.2 IDENTIFYING MEOW OUTPUTS ON THE MIG

Whilst it has been mentioned that some logging of MEOW jobs takes place, additional reporting functionality needs to be added to the MiG. This is used to address a fundamental shortcoming of the event driven model MEOW has adopted. That is, that it is hard to say with certainty what processing has taken place, and what jobs have triggered further jobs. Whilst it can be easy to see which job has been triggered by exactly what Pattern in the first step in a chain of processing, it can be difficult to see which subsequent jobs were triggered from what outputs. This is compounded by the lack of a requirement for a Pattern to identify any output ahead of time. Furthermore, a specific need for provenance reporting was identified in the requirements of any SWMS in Section 6.2.

To address this a report would need to be assembled by the MiG, or at least enough information kept so that one could be assembled by mig_meow. As the MiG already keeps an abundance of data about jobs and their makeup, the only significant information that needs to be identified for such a report to be composed is the output of a MEOW job. If this could be found out, then it would become a trivial task to match the output one job up to the triggering event of another.

Identification of outputs from a specific job was at first complicated by the design decision to allow resources to mount the MiG workgroup file storage within the resource. Whilst this would allow for the easy input and output of workgroup data, it would also mean that output could be produced anywhere within the workgroup. A solution such as regular walks through the workgroup was very quickly rejected as unworkable, because it would have to be conducted extremely often to catch all changes. It would also take a very long time to run even once in any significant data storage, such as is often the case in a scientific data store like the MiG. It was also seen as unworkable to use the existing event monitors built into the MiG as the amount of processing they need to do should be kept to a minimum. This is as they should be quickly responding to any events, and by conducting some extensive analysis of each event to determine the specific provenance of what actually caused it would add considerable slowdown to even handling.

The eventual solution came from the very problem itself. By mounting the MiG file system into the resources using SSHFS, the resource and job gained access to the entire workgroup file system and could output anywhere within it. However, any writes back to the MiG would be over SSH. If these writes could be identified somehow, then any outputs from a job could be identified. In order to accommodate some SSH functionality on the MiG, the Python package paramiko[77] was already in use. This is a package implementing SSH, and provided decorators for several SSH operations, most relevantly the write operation. By using this decorator, any write operations through the SSHFS mount could be intercepted.

As well as this, the mounts themselves would also help solve the problem. This is as, as part of the setup for the mount, an ID is created by the MiG which is used to authenticate the resources eventual request for the mount. This means that each time a resource makes a mount request, it is possible to trace back which job it is that is making the request. In combination with the paramiko write decorators, it is therefore possible for the MiG to identify any write operations through SSHFS and see if they came from a MEOW job. These writes could then be saved into a log, and so a record of

MEOW job interactions could be assembled. To facilitate this, during the scheduling of a MEOW job, a log file was created under the name of the SSHFS connection ID. When a write was carried out through an SSHFS connection it is then a very quick to see if a log file by that connections ID exists. If so then log the write in that file. Regardless of if a log file is present, the normal write operation will always take place uninhibited. This quick check is a handy feature as it means only relevant writes are logged, and without unduly slowing down any other writes via SSHFS connections.

11.3 FOUNDATIONAL INTERACTIONS

So far we have described in detail various points of the MEOW implementation on the MiG, but have danced around how a user actually creates Patterns and Recipes. It is also not clear how a user would see what MEOW constructs are currently defined in a workgroup, if they are only capable of seeing their own triggers. Nor is it apparent how they could accurately track what jobs have been scheduled from the shared Patterns and Recipes, or see how they interact and link. All of this is achieved through the interactions with `mig_meow`.

The MiG can already act as a host for the JupyterLab service, and so can provide any user with access to a powerful Python environment, with pre-installed software to provide an interface for MEOW definitions. As the Jupyter service runs separately to the MiG server code, a method of interaction will need to be developed. The most apparent way of doing this was via JSON based HTTP requests to the MiG. These requests could be sent by `mig_meow` itself, and so should require minimal user interaction. Fortunately, as part of the spawning of Jupyter Notebooks, it is possible to provide environment variables, so it is easy to define and provide the URL to which requests can be made. A token is also generated during the spawning process which is tied to the user spawning the notebook. This is refered to as the `WORKFLOWS_SESSION_ID`, and is sent with all requests to the MiG. It is used to authenticate any received requests, as only those with a valid `WORKFLOWS_SESSION_ID` will be responded to. Through the use of these environment variables, it is possible to make requests from `mig_meow` within a Jupyter Notebook spawned by the MiG, to the MiG itself without any additional input from the user.

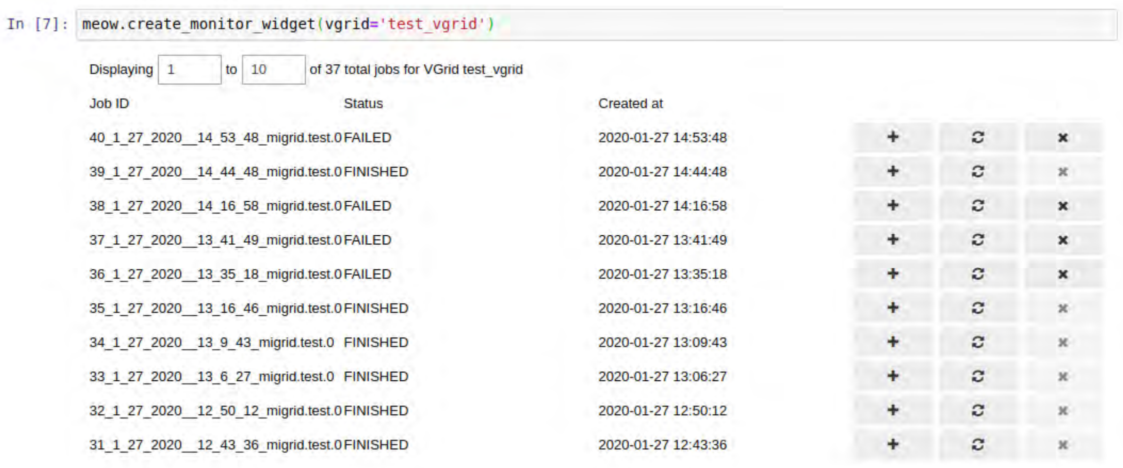
This system is not without its flaws. As the URL and `WORKFLOWS_SESSION_ID` are both passed as environment variables, it is not difficult for a user to find and copy them to other locations than just MiG spawned Jupyter Notebooks. This is seen as a small problem however as only authorised users are able to get to a point where they can spawn a Jupyter Notebook, and can only access data that they would otherwise have access to. Therefore the the unauthorised sharing of a `WORKFLOWS_SESSION_ID` or URL is not seen as a potential avenue for a malicious attack. Another possibility worth considering is that a malicious actor could guess another users `WORKFLOWS_SESSION_ID` and so gain access to data they are not authorised to access. This is also seen as being a minimal risk, as the `WORKFLOWS_SESSION_ID` is a series of 32 random alpha-numeric characters making it very unlikely that it would be randomly guessed.

Requests themselves would be received and handled on the MiG by a new module, known as `mig.shared.functionality.jsoninterface`. This is again, a location appropriate within the established structure of the MiG, and provides a generic interface for receiving and handling incoming JSON requests. The `jsoninterface` module provides functionality for users to remotely create, read, update and delete Patterns and Recipes. Requests for these operations can then be made by users within a Jupyter Notebook, using `mig_meow` functions, and with a response generated by the MiG. This means that Patterns and Recipes can be read from the MiG by various users at once, or that a user could delete a Pattern while another edits it. To help avoid any potential race conditions here, and to properly track individual MEOW constructs, `persistence_ids` are used. These are additional properties in both the Pattern and Recipe `dicts` and are used to track MEOW constructs on the MiG, but also when they are read out by a user. When MEOW constructs are read back in, it is the `persistence_id` that is used to map them to any existing definition, rather than the `name`. As the `persistence_id` is never exposed to the user it should never be manually altered and so is a much better way of tracking persistent objects across potential race conditions. The `persistence_id` is much like the `WORKFLOWS_SESSION_ID`, in that it is a long series of random alpha numeric numbers so will be extremely difficult for a user to guess and overwrite the wrong object.

As well as Pattern and Recipe operations, the `jsoninterface` also supports the reading of individual job data, as well as the reading of provenance reporting from MEOW jobs. These will be explained more fully in Section 11.4.

11.4 INTERACTING VIA WIDGETS

With response mechanisms for HTTP JSON requests built onto the MiG, and the necessary environment variables automatically loaded into spawned Jupyter Notebooks, all that remains is for a user to have some easy way of making requests. This is primarily intended to be done through widgets, provided by `mig_meow`. The main widget is the `WorkflowWidget`, has been introduced in Section 10.4. As well as the functionality already described in that section, it also contains the ability to load in Patterns and Recipes defined on a given MiG workgroup. This is done by providing the name of the workgroup during the creation of the `WorkflowWidget`. The `WorkflowWidget` can then be used to view, update or delete existing Patterns and Recipes, or to create entirely new ones. As the



```
In [7]: meow.create_monitor_widget(vgrid='test_vgrid')
```

Displaying 1 to 10 of 37 total jobs for VGrid test_vgrid

Job ID	Status	Created at
40_1_27_2020__14_53_48_migrid.test.0	FAILED	2020-01-27 14:53:48
39_1_27_2020__14_44_48_migrid.test.0	FINISHED	2020-01-27 14:44:48
38_1_27_2020__14_16_58_migrid.test.0	FAILED	2020-01-27 14:16:58
37_1_27_2020__13_41_49_migrid.test.0	FAILED	2020-01-27 13:41:49
36_1_27_2020__13_35_18_migrid.test.0	FAILED	2020-01-27 13:35:18
35_1_27_2020__13_16_46_migrid.test.0	FINISHED	2020-01-27 13:16:46
34_1_27_2020__13_9_43_migrid.test.0	FINISHED	2020-01-27 13:09:43
33_1_27_2020__13_6_27_migrid.test.0	FINISHED	2020-01-27 13:06:27
32_1_27_2020__12_50_12_migrid.test.0	FINISHED	2020-01-27 12:50:12
31_1_27_2020__12_43_36_migrid.test.0	FINISHED	2020-01-27 12:43:36

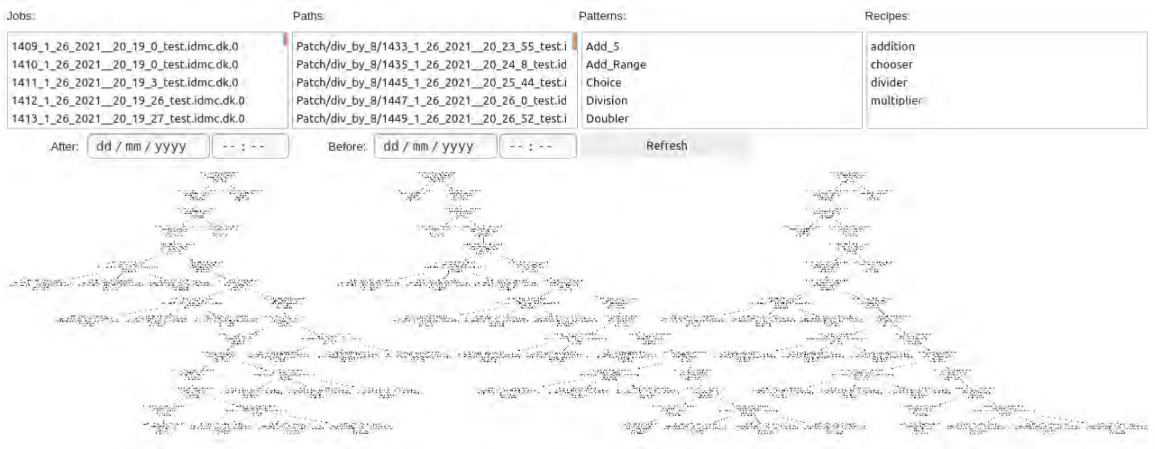
Figure 11.3: An example of the `mig_meow` `MonitorWidget` main view.

`WorkflowWidget` reads a copy of the MEOW constructs in the workgroup, any changes made will have to be manually written back to the MiG for changes to take effect on the system. This can be done via a simple *write* button. As mentioned, a user can use the `WorkflowWidget` to view any and all Patterns and Recipes in any workgroup they are a member of, and can view visualisations such as Figure 10.2 to get a complete view of the potential job scheduling.

The other two significant widgets provided by `mig_meow` are the `MonitorWidget` and the `ReportWidget`. These are both widgets designed to make it easier for users to see how the MEOW system is behaving/has behaved. Specifically, they allow users to identify the jobs that have been scheduled, and to see how they interact within a MEOW context. The `MonitorWidget` is perhaps more straightforward, with an example of it shown in Figure 11.3. This widget will simply provide the user with the details of all jobs scheduled according to MEOW within a given workgroup.

The `MonitorWidget` itself shows a list of all jobs scheduled according to Patterns and Recipes in a given workgroup. These can be sorted to display only those between certain intervals, so that a potentially huge list can be managed in a reasonable manner. Individual jobs can be then displayed in detail, so that specifics of its composition can be inspected. Jobs that are still to run, or ongoing can also be cancelled, whilst jobs that have already been completed can be re-run if necessary. In contrast to the `WorkflowWidget`, the `MonitorWidget` is designed as a real-time reflection of the current MiG status. To achieve this a polling method is adopted, where every 60 seconds an updated list of jobs will be sought from the MiG. This means that the job list can be taken as a reasonable reflection of the current job status.

The final widget provided by `mig_meow` is the `ReportWidget`. This is a proof of concept implementation for displaying a report demonstrating the provenance of all MEOW processing within a workgroup. As described in Section 11.2, using the log of all MEOW job writes via SSHFS, it is possible to determine each jobs output. When combined with the file event that triggered it, it



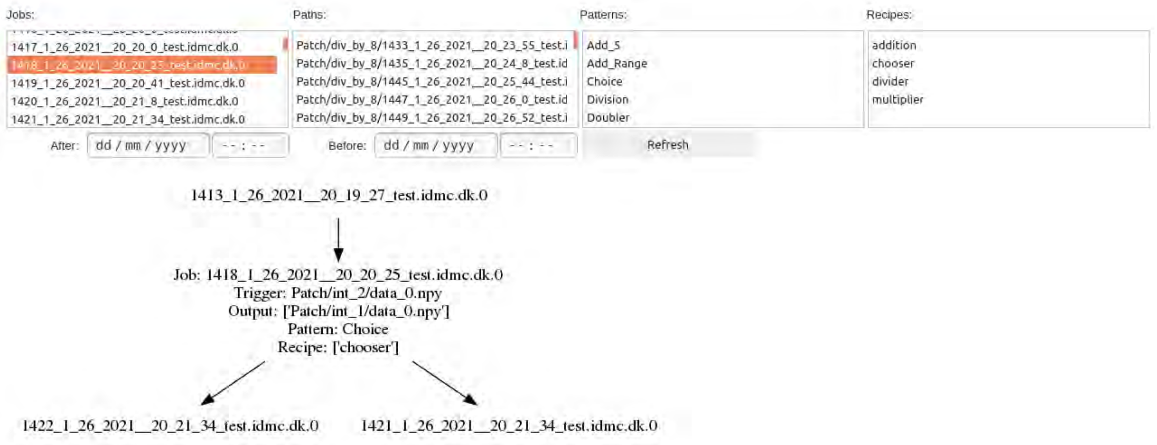
Jobs:
1409_1_26_2021__20_19_0_test.idmc.dk.0
1410_1_26_2021__20_19_0_test.idmc.dk.0
1411_1_26_2021__20_19_3_test.idmc.dk.0
1412_1_26_2021__20_19_26_test.idmc.dk.0
1413_1_26_2021__20_19_27_test.idmc.dk.0

Paths:
Patch/div_by_8/1433_1_26_2021__20_23_55_test.i
Patch/div_by_8/1435_1_26_2021__20_24_8_test.id
Patch/div_by_8/1445_1_26_2021__20_25_44_test.i
Patch/div_by_8/1447_1_26_2021__20_26_0_test.id
Patch/div_by_8/1449_1_26_2021__20_26_52_test.i

Patterns:
Add_5
Add_Range
Choice
Division
Doubler

Recipes:
addition
chooser
divider
multiplier

Figure 11.4: An example of the `mig_meow` `ReportWidget` with full report DAG.



Jobs:
1417_1_26_2021__20_20_0_test.idmc.dk.0
1418_1_26_2021__20_20_25_test.idmc.dk.0
1419_1_26_2021__20_20_41_test.idmc.dk.0
1420_1_26_2021__20_21_8_test.idmc.dk.0
1421_1_26_2021__20_21_34_test.idmc.dk.0

Paths:
Patch/div_by_8/1433_1_26_2021__20_23_55_test.i
Patch/div_by_8/1435_1_26_2021__20_24_8_test.id
Patch/div_by_8/1445_1_26_2021__20_25_44_test.i
Patch/div_by_8/1447_1_26_2021__20_26_0_test.id
Patch/div_by_8/1449_1_26_2021__20_26_52_test.i

Patterns:
Add_5
Add_Range
Choice
Division
Doubler

Recipes:
addition
chooser
divider
multiplier

1413_1_26_2021__20_19_27_test.idmc.dk.0

Job: 1418_1_26_2021__20_20_25_test.idmc.dk.0
Trigger: Patch/int_2/data_0.npy
Output: ['Patch/int_1/data_0.npy']
Pattern: Choice
Recipe: ['chooser']

1422_1_26_2021__20_21_34_test.idmc.dk.0

1421_1_26_2021__20_21_34_test.idmc.dk.0

Figure 11.5: A example of a filtered `ReportWidget` DAG.

becomes easy to construct a DAG of the MEOW processing. Despite being a DAG, this is very different to the type of DAG used in static workflow systems. This is because it shows specific job dependencies rather than generalised steps, but also because it can only be constructed *a posteriori* of the job processing actually taking place.

The report itself is assembled by the `ReportWidget`, which will read the job logging from a given workgroup and construct a DAG. Figure 11.4 shows a complete report, showing a complete emergent workflow for a large amount of interacting jobs. Though it is hard to make out any job details in the final image, it shows that three files within the 'Patch/initial_data/' directory were catalysts for three separate chains of processing. These triggered three jobs, which each produced output. This triggered further jobs until eventually 115 jobs were completed. From this report is easy to see how the different jobs relate to one another, and what caused them to be scheduled.

One limitation of the large report is just that, its size. The report has quickly become unwieldy. For this reason the abilty to edit it down to something more manageable has been added to the widget. Above the report are six filters, with which a user can edit down the displayed report. This can be done by filtering jobs by name, triggering path, Pattern, Recipe, or before/after given times. These can be used in combination, so a user could for instance display all jobs triggered by a particular Pattern after a certain time. A filtered example is shown in Figure 11.5. In this case only a singular job has been selected, so that we can get some idea of what information is displayed. In this case it is the catchily named job '1418_1_26_2021__20_20_25_test.idmc.dk.0', displayed in the middle of the DAG. We can see that it was triggered by output from a previous job, '413_1_26_2021__20_19_27_test.idmc.dk.0', and that the triggering file was 'Patch/int_2/data_0.npy'. We can also see that one output file was proudced, but that it triggered two separate jobs.

Regardless of filtering or size of report, the `ReportWidget` will always produce the report as a .png or .pdf file, depending on user input. This allows the report to be easily saved by users for their own reading, or inclusion in any reports of their own. The produced report looks identical to those shown in Figures 11.4 and 11.5, save the absence of the filtering inputs at the top. In addition, a text based report is also generated which could be used to decode job interactions if that is a preferred way of working, though it is somewhat dense and not the advised technique. In any case, all of the most important information about each job is included, so that a user can quickly see what processing was triggered and why, and what each job produced.

12 SUMMARY

This part has outlined how MEOW was implemented. This was done in a new python package, `mig_meow`, along with additions to the larger MiG project. These allow users to create and edit Patterns and Recipes, the essential building blocks of any MEOW analysis. This can be done programmatically, or can be done through a number of helper widgets if a user is using Jupyter Notebooks. The main widget is the `WorkflowWidget`, which also integrates with the MiG so that any MEOW constructs in a given workgroup can be created, viewed, edited or removed. This also includes a visualisation of how it is expected that the various steps derived from Pattern and Recipe definitions will interlink.

Another significant aspect that has been presented is the `WorkflowRunner`. This is a construct within `mig_meow` that allows for users to run their MEOW analysis without using the MiG. This is intended only as a learning tool, and for illustration purposes for those who do not have access to the MiG. It is still capable of running a complete analysis system, it just lacks some of the MiGs more specific features. One of these is the ability to produce a MEOW provenance report, detailing all of the analysis undertaken according to MEOW definitions.

We have now answered most of the thesis objectives first presented in Section 2.2. Both the `WorkflowRunner` and the MiG implementation can be taken as answers to the second thesis objective, to *'Implement an automated, dynamic workflow system'*. By integrating MEOW into the MiG the third objective to *'Integrate the automated, dynamic workflow system into a collaborative big data platform'*, was also met. This is significant as it goes some way to answering the core research question of the thesis. As we have managed to create a tool for the automatic creation of dynamic scientific workflows within the MiG it seems only sensible to conclude that this was feasible. A system has been constructed that can meet all of the requirements of a SWMS first set out in 4.1. This will now be demonstrated in the Part iii, along with with a variety of investigations and benchmarks demonstrating the utility and cost of using such a system.

Part III

EXAMPLES AND TESTS

13

INTRODUCTION

This is the last of the three main parts of this thesis. In it, three examples of how MEOW might be used as presented, along with a description of the testing undertaken to ensure the correctness of the system. The first example to be presented here is a very basic one, with no scientific value. It is presented as a complete and thorough explanation the various stages of a MEOW analysis and how it behaves. The second example is an actual scientific use case adapted to make use of MEOW in its analysis. Lastly is a hypothetical example showcasing MEOWs ability to modify itself at runtime.

These are then followed by a short description of some of the tests that were carried out during development. Most significantly is a small investigation into the `watchdog` package, to confirm that it is indeed sufficient for the needs of MEOW. By the end of this part, readers should have a more grounded understanding of MEOW and the sorts of problems it is capable of solving. The ideas and implementations outlined in the previous two parts should be reinforced by the concrete examples set out in this.

All of the source code used in the examples in this part is available within [34].

14

A FOUNDATIONAL EXAMPLE

This chapter will demonstrate MEOW scheduling on some very basic processing. This purpose of this is to allow us to demonstrate the key functionality of MEOW, and to do so at such a fundamental level that there is little room for misunderstanding. Apologies in advance if this chapter seems trivial, or dwells too long in explaining a system that you the reader have already understood. However, from experience, a great many people when encountering MEOW for the first time do not grasp the actual sequence of processing, or how jobs causally relate to one another. Therefore, this example is presented in as complete a form as possible.

14.1 PROBLEM OUTLINE

The processing that will take place within this example is trivial, as the processing itself is not the focus here. Therefore, what we will be doing is taking a simple text file in a .txt format, and append another line of text to it in each job.

The focus of this example is to demonstrate the structure of MEOW analysis, and how defined Patterns and Recipes will result in job processing. To do this we will create an initial Pattern that will schedule some processing on some raw data that has been created ahead of time. A second Pattern will also be created, which will take the output of the first Pattern as input. The final output produceed by the second Pattern will therefore have been processed by both Patterns. A single Recipe will be created, and will be used by both of the Patterns. The desired ordering of processing is shown in Figure 14.1.

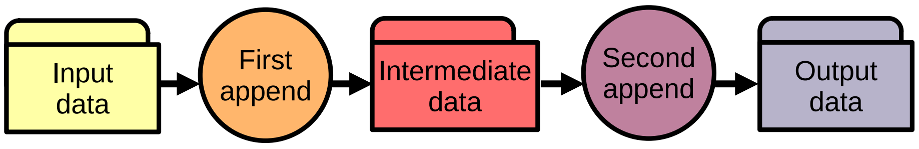


Figure 14.1: The structure of the foundational example. Directories are shown as folders, while Patterns are shown as circles.

14.2 DEFINED RECIPES

Only a single Recipe was needed for this analysis. This was defined in a Jupyter Notebook called *append_text.ipynb*. The code from this Jupyter Notebook is shown in Listing 14.1. This code is a very simple algorithm for reading in a text file, appending a line, and writing the output to a new file.

By default, when the *append_text.ipynb* Jupyter Notebook is registered with MEOW to form a `Recipe`, the `Recipe` would be given the name *append_text*. In this case we will overrule this, and give the `Recipe` the name *Append*. This is done in Listing 14.4, and is not part of the Jupyter Notebook itself.

```python
import os

# Default parameters values
# Data input file location
infile = 'in_dir/default.txt'
# The line to append
extra = 'This line comes from a default pattern'
# Output file location
outfile = 'out_dir/default.txt'

# load in dataset. This should be a text file
with open(infile) as input_file:
    data = input_file.read()

# Append the line
appended = data + '\n' + extra

# Create output directory if it doesn't exist
output_dir_path = os.path.dirname(outfile)
if output_dir_path:
    os.makedirs(output_dir_path, exist_ok=True)

# Save added array as new dataset
with open(outfile, 'w') as output_file:
   output_file.write(appended)
```

Listing 14.1: The *append_text.ipynb* code cell contents.

14.3 DEFINED PATTERNS

A pair of Patterns will need to be defined to implement the structure shown in Figure 14.1. These will simply be named *Pattern_One* and *Pattern_Two* as this nicely shows in what order they will apply to the data, and are shown in Listings 14.2 and 14.3 respectively.

Both Patterns are defined in YAML files, with their names taken from the filenames.. *Pattern_One* uses the *Append* Recipe, which from Listing 14.1 uses *infile* as its input, which should be whatever file caused the event seen by MEOW. Therefore, we need to set the `input_file` parameter in *Pattern_One* to be *infile*, as this is the name of the variable in the *Append* Recipe used to define its input. Related to this we need to define the actual path to read in our input file, that being the value assigned to *infile*. This gives us the `input_paths` parameter being set to *start/**. The use of the wildcard '*' character here means that our Rule will trigger on any file within the *start/* directory. Setting the *extra* variable is done using the `variables` parameter, and is set to *'This line was added by Pattern_One'*.

```
input_file: infile
input_paths:
- start/*
output:
  outfile: '{VGRID}/first/{FILENAME}'
parameterize_over: {}
recipes:
- Append
variables:
  extra: This line was added by Pattern_One
```

Listing 14.2: *Pattern_One* file contents.

The *outfile* variable was added using the `output` parameter so that if this Pattern was loaded into the `mig_meow` `WorkflowWidget` a visualisation could be derived. It could alternatively have been added within the `variables` parameter and the Pattern would function exactly the same with regards to processing, but the visualisation would not show an output location. To define a new output location at runtime, we will use the keywords set out in Table 10.1. As we want our output to be written back into the file storage, we should start our path with the {VGRID} keyword. This is the base directory within the `WorkflowRunner`, and the base directory of a MiG workgroup. Within that base directory we shall output to the hardcoded *first/* directory, but shall use the {FILENAME} keyword so that different input files will produce different output files. This gives us the final value for *outfile* as *{VGRID}/first/{FILENAME}*, which completes *Pattern_One*.

Pattern_Two is very similar to *Pattern_One*, though the `input_paths` is now set to *first/**. This will overlap with the output of *Pattern_One*, so any output from *Pattern_One* will automatically trigger *Pattern_Two*. The *outfile* and *extra* variable values are also updated in *Pattern_Two*, so that the appended line actually reflects who processed it, and it is output to a new location. This means that our final result will be available within the *second* directory.

```
input_file: infile
input_paths:
- first/*
output:
  outfile: '{VGRID}/second/{FILENAME}'
parameterize_over: {}
recipes:
- Append
variables:
  extra: This line was added by Pattern_Two
```

Listing 14.3: *Pattern_Two* file contents.

14.4 USING WORKFLOWRUNNER

To run the analysis we use the defined Patterns and Recipe in conjunction with the `WorkflowRunner`. We will now provide an exhaustive description of how this is done, and the changes that will happen in the system as it progresses.

14.4.1 *Setting up*

The `WorkflowRunner` can be setup and run as a single script, with the Pattern and Recipe definitions read in directly from the previously displayed files, as is shown in Listing 14.4. Pattern definitions are read in and stored in a single `dict` in lines 4-9. We have not constructed a definition file for the *Append* `Recipe`, as it is registered directly from a Jupyter Notebook directly.

```
1  import mig_meow as meow
2  
3  # Read in the Pattern definitions and setup Patterns dict
4  first = meow.read_pattern('Pattern_One')
5  second = meow.read_pattern('Pattern_Two')
6  patterns = {
7      'Pattern_One': first,
8      'Pattern_Two': second
9  }
10 
11 # Register the Recipe notebook and setup Recipes dict
12 append = meow.register_recipe('append_text.ipynb', name='Append')
13 recipes = {
14     'Append': append
15 }
16 
17 # Start the local runner
18 runner = meow.WorkflowRunner(
19     'basic',            # Base directory to monitor
20     1,                  # Number of worker processes
21     patterns=patterns,
22     recipes=recipes
23 )
```

Listing 14.4: The Python script *foundational_example.py* used to run the foundational example within a `WorkflowRunner`.

The final part of the script sets up and starts the `WorkflowRunner` itself. In this case we can see the base directory for our analysis is called *basic*, and that only a single worker will be started. We also pass our Pattern and Recipe `dicts` to the `WorkflowRunner` so that it can construct the relevant Rules from the very beginning. All files used and produced in this running are available at [34], within the *foundational/WorkflowRunner/* directory.

Before running this script Pattern definition files, as well as the *append_text.ipynb* Jupyter Notebook. These will all be placed next to the script, whilst a base directory ´basic’ will also be created. This is the directory that the `WorkflowRunner` will be monitoring. Before we run the script, we will

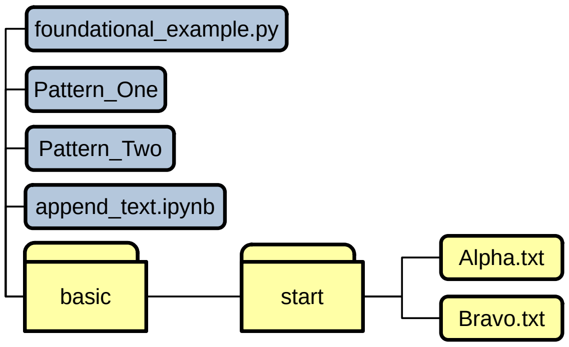


Figure 14.2: The file structure of the foundational example before the `WorkflowRunner` is started. Files for defining the MEOW system are shown in blue, whilst the starting state of the data is shown in yellow.

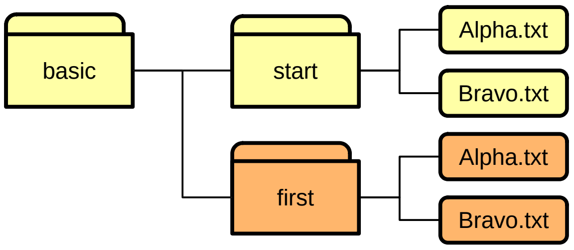


Figure 14.3: The file structure of the foundational example once all `WorkflowRunner` processing according to *Pattern_One* has completed. Note that the files used to create the `WorkflowRunner`, shown in blue in Figure 14.2 are still present but have been removed from this diagram for brevity.

creeate two data files, *start/Alpha.txt* and *start/Bravo.txt*. These will each contain a single line of text stating *This is the starting state of Alpha* or *This is the starting state of Bravo*. This structure can be seen in Figure 14.2. The structure shown can be placed anywhere within a users local machine, and would only require a Python environment with `mig_meow` to run.

14.4.2 *Running the `WorkflowRunner`*

The `WorkflowRunner` can be started by invoking the *foundational_example.py* script shown in Listing 14.4. From the given Patterns and Recipe, the `WorkflowRunner` will create two Rules. As part of creating these Rules, the runner will check if any existing files would trigger them, were they created now. The two files, *start/Alpha.txt* and *start/Bravo.txt* are two such files, as they would trigger *Pattern_One* according to its `input_path` of *start/**. This will result in two jobs being scheduled, one in response to each. The job triggered by the *Alpha.txt* will produce an output file at *first/Alpha.txt*. This file will contain two lines, as shown in Listing 14.5. The second job will have output a similar file at *first/Bravo.txt*, with the contents showing that it has also been modified by *Pattern_One*.

```
This is the starting state of Alpha
This line was added by Pattern_One
```

Listing 14.5: The contents of the output file *first/Alpha.txt*.

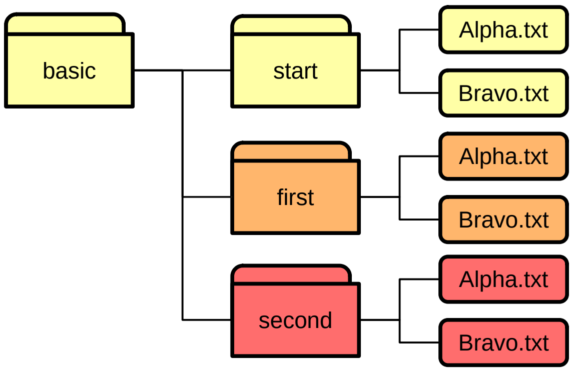


Figure 14.4: The file structure of the foundational example once all `WorkflowRunner` processing has completed. Note that the files used to create the `WorkflowRunner`, shown in blue in Figure 14.2 are still present but have been removed from this diagram for brevity.

The state of the system after the first round of processing is shown in Figure 14.3. As the two output files of the first two jobs are written, they will trigger the Rule created from the second Pattern, *Pattern_Two*. This will schedule two further jobs, one for each of the two files *first/Alpha.txt* and *first/Bravo.txt*. These will produce output in the same way as the first two jobs, though as defined by the parameters in *Pattern_Two*, they will output to the *second/* directory. Once these two jobs are complete, the file structure will appear as shown in Figure 14.4.

The state of the *second/Alpha.txt* file is shown in Listing 14.6. This clearly shows that the file shown in Listing 14.5 has now also been processed by *Pattern_Two*. This will conclude the job processing, as no further Patterns have been created to respond to these final files. The `WorkflowRunner` itself is still running however, so if new files were added to either the *start/* or *first/* directory then new jobs would be scheduled.

```
This is the starting state of Alpha
This line was added by Pattern_One
This line was added by Pattern_Two
```

Listing 14.6: The contents of the output file *second/Alpha.txt*.

14.5 USING THE MIG

To run the example on the MiG the `WorkflowWidget` was used. This was run within a Jupyter Notebook started using the MiGs DAG service[67]. The same Patterns and Recipe definitions were used as in the previous example, so will not be re-explained here. A workgroup was set up to host the files and act as a base for the MEOW system. The newly created workgroup was given the name *basic*, so that no modifications would need to be made to our existing definitions.

A JupyterLab instance was started on the MiG, and within a Jupyter Notebook, an instance of the MEOW `WorkflowWidget` was started. This will be used to manage the MEOW analysis.

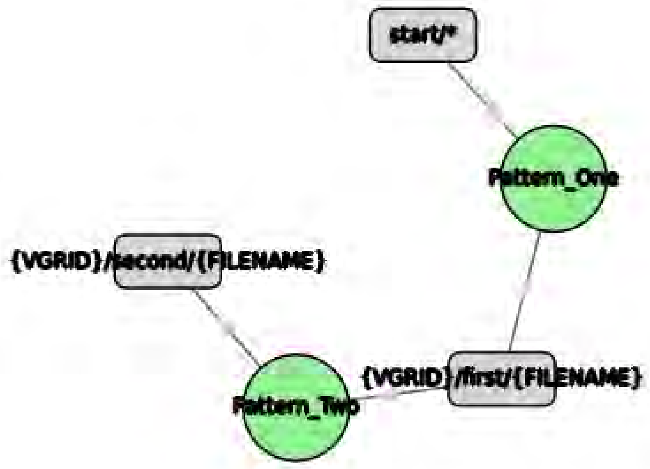
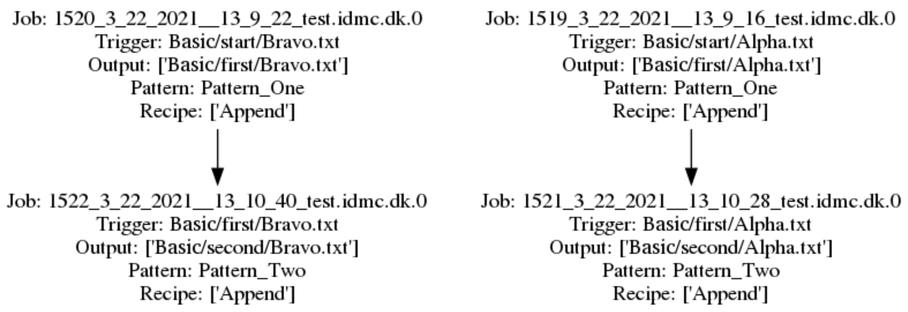


Figure 14.5: The `WorkflowWidget` visualisation of the foundational example.



Job: 1520_3_22_2021__13_9_22_test.idmc.dk.0
Trigger: Basic/start/Bravo.txt
Output: ['Basic/first/Bravo.txt']
Pattern: Pattern_One
Recipe: ['Append']

Job: 1522_3_22_2021__13_10_40_test.idmc.dk.0
Trigger: Basic/first/Bravo.txt
Output: ['Basic/second/Bravo.txt']
Pattern: Pattern_Two
Recipe: ['Append']

Job: 1519_3_22_2021__13_9_16_test.idmc.dk.0
Trigger: Basic/start/Alpha.txt
Output: ['Basic/first/Alpha.txt']
Pattern: Pattern_One
Recipe: ['Append']

Job: 1521_3_22_2021__13_10_28_test.idmc.dk.0
Trigger: Basic/first/Alpha.txt
Output: ['Basic/second/Alpha.txt']
Pattern: Pattern_Two
Recipe: ['Append']

Figure 14.6: The foundational example provenance report generated as part of the `ReportWidget`.

Rather than using it to create new `Patterns` and `Recipes` from scratch, the definition files used in the `WorkflowRunner` were used. Once the `Pattern` and `Recipes` definitions used in the `WorkflowRunner` were loaded into the `WorkflowWidget`, they created the visualisation shown in Figure 14.5. As both `Pattern` nodes are green, we can see that the necessary `Recipe` has been registered, and so the appropriate Rules will be created as soon as we register these definitions with the workgroup.

To trigger processing on the MiG, the *Export to VGrid* button was pressed in the `WorkflowWidget`. This will register each MEOW construct in turn, and give a feedback message to the widget which will be displayed. Unlike in the `WorkflowRunner`, merely registering the MEOW constructs is enough for the system to begin, as the MiG is a live system. Therefore, if the input data had already been placed in the *start* directory, jobs would already have been scheduled. To start scheduling we need to upload the files *start/Alpha.txt* and *start/Bravo.txt* to the workgroup. By uploading these files, we will trigger two jobs as expected, which will output and trigger two more.

Once these jobs have completed, we are able to view a provenance report using the `ReportWidget`. Such a report can be seen in Figure 14.6. As expected, this shows the four jobs which have been scheduled on the MiG. The method of identifying job outputs outlined in Section 11.2 currently is only implemented within the MiG, so such a report is not possible within the `WorkflowRunner`. The same analysis structure will have been kept to, and the two DAGs shown could represent either system.

14.6 CONCLUDING THE FOUNDATIONAL EXAMPLE

This example has shown how MEOW can be setup in a basic fashion to schedule analysis jobs on data through the use of Patterns and Recipes. These can be easily defined by a user, and when done so appropriately will overlap and cause chains of processing. In this case a chain of two steps was demonstrated but there is no upwards limit on how long the chain of processing can go, just as there is no requirement that any chain is formed at all.

This example has also demonstrated the event driven nature of the system, with jobs being scheduled in direct response to files being created or modified. This is especially true of the MiG which is a continuous, live system and so any additional data added to either the *start/* or *first/* directories will schedule new processing until the appropriate Patterns or Recipes are unregistered. This shows that MEOW is especially suited to continuous, repeated tasks such as are often present in scientific analysis.

Lastly, this example has shown an exhaustive account of how a user can setup and use a MEOW system both locally and in conjunction with the MiG. This methodology can be applied and adapted to any processing that takes input data as files, and produces further output. `Patterns` may be defined through definition files as has been shown here. Meanwhile `Recipes` can be simply defined through Jupyter Notebooks that are then registered. In both cases these simple definitions are combined by the underlying MEOW system to create Rules to manage file events.

15

A SCIENTIFIC EXAMPLE

This second example will show a potential use case for a MEOW system in a scientific setting. It is a modification of the example first shown in the paper *Managing Event Oriented Workflows*. This paper is available in full in Appendix A, or can be viewed as originally published at [54]. Some modifications have been made to the start of the analysis to better illustrate the dynamic nature of MEOW. The example presented here will also be a demonstration of the ease with which branching or looping structures can be included, as well as the possibility for scheduling new data retrieval from within the MEOW system.

15.1 PROBLEM OUTLINE

This analysis processes artificial tomography scans of aluminium foam[86]. This foam contains a large amount of unevenly sized pores. Each scan image will be segmented into aluminium and air, with this then used to conduct some analysis of the composition of the foam. However, the segmentation and analysis are computationally taxing tasks, which we only want to conduct on valid datasets. For this reason we can add an initial check, to make an educated guess as to if the data is valid or not. This will be done for each dataset individually, without prior knowledge of which datasets are valid and which are not.

Unlike in the original example, we will not use pre-generated data but will generate artificial input data as the MEOW system is running. This is to simulate a physical experiment producing data on an ongoing basis. This data will still be read in by our initial porosity check, which will either accept or deny each data file in the same manner as before. A random uncertainty was introduced to our data generation, so each generated data file will have a one in three chance of containing too many pores. We can therefore expect roughly a third of initially generated data to be rejected. Where previously this rejected data was ignored, a Pattern will be added that will schedule the generation of new data to replace this rejected data. This new dataset is generated in the same manner as the initial generation,

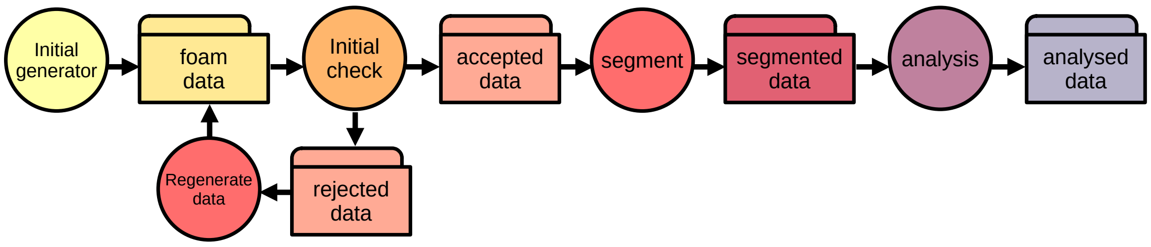


Figure 15.1: The structure of the revised scientific example. Directories are shown as folders, while Patterns are shown as circles.

and so will also have a one in three of being too small. It will therefore also be checked in the same manner as the initially generated data. If it is accepted then it will be segmented and analysed in the same manner as any other data, but if rejected then it will re-generated again and again until a suitable data file has been generated.

The resulting structure is shown in Figure 15.1. Note that the initial data generation and the secondary re-generation of rejected data are separate steps. Both of these will use the same processing to generate individual datasets, so the difference is not too important.

15.2 DEFINED RECIPES

Although four distinct Recipes would be needed for this analysis, three of them could be lifted straight from the example presented in [54]. These are the Recipes for checking the data, segmenting it, and the final analysis. As they are unaltered from their previous incantation, they will not be examined in depth and a simple recap of their purpose is provided here.

The first of these we will call *recipe_check*. It uses a two-component Gaussian Mixture Model, fitted to a small sample (around 1%) of the intensity data. This provides a rough idea of the air-to-aluminium ratio through the model component weights. This data will then be output to one of two locations depending on the found ratio. This is shown in Appendix I.

The second Recipe is called *recipe_segment*. In the first step of the segmentation process, noise is reduced using a Gaussian filter. The filter kernel size is defined as a variable whose value is set in the corresponding Pattern. Thereafter, the image is segmented using Otsu thresholding [75]. Finally, a morphological closing operation is performed to remove possible remaining single-voxel noise. This is shown in Appendix J.

The third reused Recipe is called *recipe_analysis*. It investigates the pore size distribution. The individual pores are identified using the watershed algorithm [24] with local peaks in a distance transform of the segmented data as seeds. This is shown in Appendix K. The fourth Recipe to be used is a completly new Recipe. This is the Recipe to generate new datasets using the Python package foam_ct_phantom [78] as well as the ASTRA toolbox [91]. As the code in this notebook is quite long, an abridged view of it is shown in Listing 15.1, though the full code can be seen in Appendix L. This Jupyter Notebook is used to create individual datasets as replacements for ones rejected within *recipe_check*.

```
 1  # Variables to be overridden
 2  dest_dir = 'foam_ct_data'
 3  discarded = 'discarded/foam_data_0-big-.npy'
 4  utils_path = 'idmc_utils_module.py'
 5  gen_path = 'generate_foam_module.py'
 6
 7  ...
 8
 9  # Randomly determine if dataset will be ok, or have too few pores
10  def get_dataset_type(name):
11      num = random.randint(1, 3)
12      if num == 1:
13          name = name.replace('--', 'X-few-')
14          return (gen.generate_foam, nspheres_per_unit_few, name)
15      else:
16          name = name.replace('--', 'X-ok-')
17          return (gen.generate_foam, nspheres_per_unit_ideal, name)
18
19  # Create a dataset for a given filename
20  def create_random_dataset(name):
21      generator, spheres, filename = get_dataset_type(name)
22      dataset = generator(spheres, vx, vy, vz, res)
23      os.makedirs(dest_dir, exist_ok=True)
24      np.save(os.path.join(dest_dir, filename+'.npy'), dataset)
25
26  ...
27
28  # Generate replacement dataset
29  create_random_dataset(filename)
```

Listing 15.1: Abridged contents of the *generator.ipynb* code cells.

Datasets are generated by first determining randomly if the dataset that will be generated is going to contain pores that are either too small, or ideally distributed. The random determination is done on lines 9 to 17. Of note is that on lines 13 and 16 we update the filename of the generated file to reflect the suitability of the data. This is never used in the actual calculations and is only so that we can quickly check if the algorithm is functioning as expected. The actual process of creating the dataset itself is on lines 21 to 24, where it is generated and saved. The implementation of this function is provided in an ancillary module, *generate_foam_module.py*. The relevant function, generate_foam is shown in Listing 15.2. This uses a random seed to generate datasets, using part of the Astra tooldbox.

```python
def generate_foam(nspheres_per_unit, vx, vy, vz, res):
    def maxsize_func(x, y, z):
        return 0.2 - 0.1*np.abs(z)

    random_seed=random.randint(0,4294967295)
    foam_ct_phantom.FoamPhantom.generate('temp_phantom_info.h5',
                                         random_seed,
                                         nspheres_per_unit=nspheres_per_unit,
                                         maxsize=maxsize_func)

    geom = foam_ct_phantom.VolumeGeometry(vx, vy, vz, res)
    phantom = foam_ct_phantom.FoamPhantom('temp_phantom_info.h5')
    phantom.generate_volume('temp_phantom.h5', geom)
    dataset = foam_ct_phantom.load_volume('temp_phantom.h5')

    return dataset
```

Listing 15.2: Function `generate_foam` used to generate foam data.

The code shown in Listing 15.1 is for the re-generation of data following a dataset being rejected by the *recipe_check*. However, exactly the same algorithm, functions and variables are used to generate the initial datasets, as is shown in Appendix M. The only difference is that rather than creating a single dataset on line 29, a loop is added to create a given number of datasets, each with an individual name. The other minor change is on lines 13 and 16. Within the regeneration code, an X character is added to the filename as a way of tracking how many times a dataset has been regenerated. By doing this, we can quickly count how many times a file has been regenerated. As with the note about data suitability in the filenames, this will have no effect within the actual processing itself and is only there for quick and easy verification of results.

15.3 DEFINED PATTERNS

Much like the Recipes, there is no need for change in the three Patterns inherited from the earlier example. As such, the definitions shown in Listings 15.3, 15.4, and 15.5 are all identical to those used in [54].

```yaml
input_file: input_filename
input_paths:
- foam_ct_data/*
output:
  output_filedir_accepted: '{VGRID}/foam_ct_data_accepted/'
  output_filedir_discarded: '{VGRID}/foam_ct_data_discarded/'
parameterize_over: {}
recipes:
- recipe_check
variables:
  porosity_lower_threshold: 0.8
  utils_path: '{VGRID}/idmc_utils_module.py'
```

Listing 15.3: *pattern_check* file contents.

```
input_file: input_filename
input_paths:
- foam_ct_data_accepted/*
output:
  output_filedir: '{VGRID}/foam_ct_data_segmented/'
parameterize_over: {}
recipes:
- recipe_segment
variables:
  input_filedir: '{VGRID}/foam_ct_data/'
  utils_path: '{VGRID}/idmc_utils_module.py'
```

Listing 15.4: *pattern_segment* file contents.

```
input_file: input_filename
input_paths:
- foam_ct_data_segmented/*
output:
  output_filedir: '{VGRID}/foam_ct_data_pore_analysis/'
parameterize_over: {}
recipes:
- recipe_analysis
variables:
  utils_path: '{VGRID}/idmc_utils_module.py'
```

Listing 15.5: *pattern_analysis* file contents.

The Pattern, *pattern_regenerate*, is a new Pattern to schedule new data generation following a dataset being discarded. As such, it has an `input_path` of *foam_ct_data_discarded/**. This will allow it to trigger for every single discarded dataset. As the resultant job will output to the *foam_ct_data* directory, a loop will form between *pattern_regenerate* and *pattern_check*. This may go on infinitely but as each track through the loop only has a one in three chances of occurring, this can be tolerated.

```
input_file: discarded
input_paths:
- foam_ct_data_discarded/*
output:
  dest_dir: '{VGRID}/foam_ct_data'
parameterize_over: {}
recipes:
- recipe_generator
variables:
  gen_path: '{VGRID}/generate_foam_module.py'
  utils_path: '{VGRID}/idmc_utils_module.py'
```

Listing 15.6: textitpattern_regenerate file contents.

15.4 USING WORKFLOWRUNNER

Once the Patterns and Recipes have each been defined, we can construct a script to use them in a `WorkflowRunner`. This script is shown in Listing 15.7.

The final results of the analysis are available in [34]. From this we can see that a total of 71 jobs were identified and run throughout the `WorkflowRunners` lifetime. Of the 20 initially created datasets, 13 were acceptable results. Each of the seven rejected results were then re-generated and three were then accepted. All of the four twice rejected results were accepted on the second regeneration. This resulted in 20 valid segmentation jobs and a further 20 analysis jobs being conducted as required.

15.5 USING THE MIG

The MEOW system was set up on the MiG as before, though this time the workgroup *Patch* was used to run it. This is simply as it was already set up, and had pre-existing access to processing resources with the necessary software dependencies installed. The `WorkflowWidget` produces the visualisation shown in Figure 15.2.

```python
import mig_meow as meow

# Setup dict of all Patterns
patterns = {
    'pattern_check': meow.read_dir_pattern('pattern_check'),
    'pattern_segment': meow.read_dir_pattern('pattern_segment'),
    'pattern_analysis': meow.read_dir_pattern('pattern_analysis'),
    'pattern_regenerate': meow.read_dir_pattern('pattern_regenerate')}

# Setup dict of all Recipes
recipes = {
    'recipe_check': meow.register_recipe(
            'scientific/initial_porosity_check.ipynb', 'recipe_check'),
    'recipe_segment': meow.register_recipe(
            'scientific/segment_foam_data.ipynb', 'recipe_segment'),
    'recipe_analysis': meow.register_recipe(
            'scientific/foam_pore_analysis.ipynb', 'recipe_analysis'),
    'recipe_generator': meow.register_recipe(
            'scientific/generator.ipynb', 'recipe_generator')}

# Start the local runner
runner = meow.WorkflowRunner(
    'scientific', 1, patterns=patterns, recipes=recipes)
```

Listing 15.7: The Python script *scientific_example.py* used to run the scientific example within a `WorkflowRunner`.

Once all of the Patterns and Recipes had been set up, the *initial_generation.py* script was uploaded to the workgroup. This is shown in Appendix M. In this case, 15 of the initial data files were generated acceptably on the first try. Of the remaining five, four were accepted on the first re-generation whilst the last was only accepted after three re-generations.

As displaying all of these on a single report would make it rather large, a select dataset has been shown. This is in Figure 15.3 and shows the path taken for the *foam_data_15* datasets. We can see that the initial dataset triggered the *pattern_check*, which we can tell if failed as the output was written to the *foam_ct_data_discarded/* directory. This has triggered *pattern_regenerate*, which created a new dataset which was output to the *foam_ct_data* directory. This time the check was passed and the segmentation and analysis were performed on the data as expected.

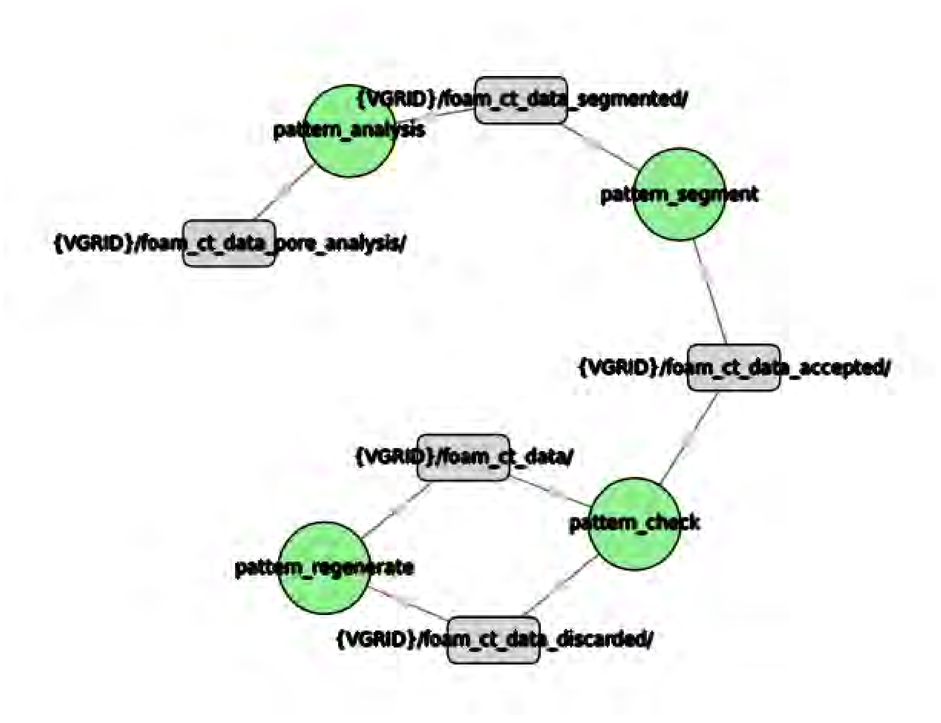
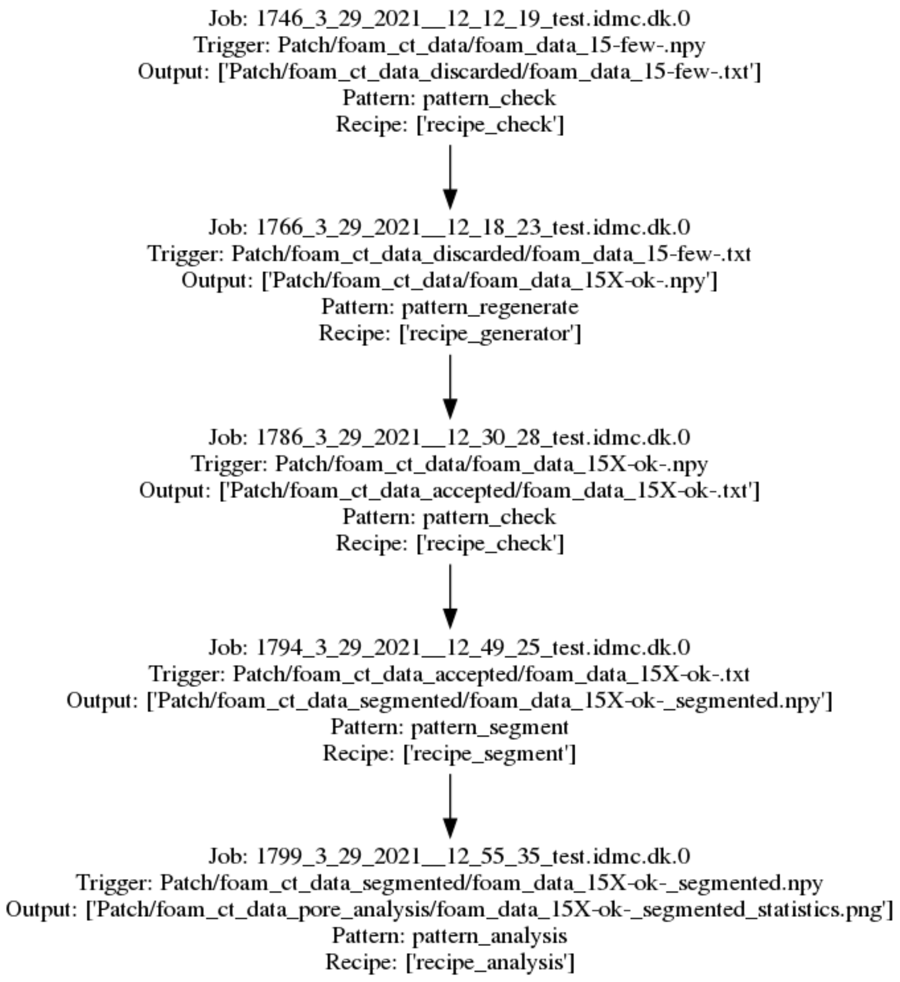

{VGRID}/foam_ct_data_segmented/
pattern_analysis
pattern_segment
{VGRID}/foam_ct_data_pore_analysis/
{VGRID}/foam_ct_data_accepted/
{VGRID}/foam_ct_data/
pattern_check
pattern_regenerate
{VGRID}/foam_ct_data_discarded/

Figure 15.2: The `WorkflowWidget` visualisation of the scientific example.


Job: 1746_3_29_2021__12_12_19_test.idmc.dk.0
Trigger: Patch/foam_ct_data/foam_data_15-few-.npy
Output: ['Patch/foam_ct_data_discarded/foam_data_15-few-.txt']
Pattern: pattern_check
Recipe: ['recipe_check']

Job: 1766_3_29_2021__12_18_23_test.idmc.dk.0
Trigger: Patch/foam_ct_data_discarded/foam_data_15-few-.txt
Output: ['Patch/foam_ct_data/foam_data_15X-ok-.npy']
Pattern: pattern_regenerate
Recipe: ['recipe_generator']

Job: 1786_3_29_2021__12_30_28_test.idmc.dk.0
Trigger: Patch/foam_ct_data/foam_data_15X-ok-.npy
Output: ['Patch/foam_ct_data_accepted/foam_data_15X-ok-.txt']
Pattern: pattern_check
Recipe: ['recipe_check']

Job: 1794_3_29_2021__12_49_25_test.idmc.dk.0
Trigger: Patch/foam_ct_data_accepted/foam_data_15X-ok-.txt
Output: ['Patch/foam_ct_data_segmented/foam_data_15X-ok-_segmented.npy']
Pattern: pattern_segment
Recipe: ['recipe_segment']

Job: 1799_3_29_2021__12_55_35_test.idmc.dk.0
Trigger: Patch/foam_ct_data_segmented/foam_data_15X-ok-_segmented.npy
Output: ['Patch/foam_ct_data_pore_analysis/foam_data_15X-ok-_segmented_statistics.png']
Pattern: pattern_analysis
Recipe: ['recipe_analysis']

Figure 15.3: The `ReportWidget` for the jobs scheduled from *foam_data_15* datasets.

As in the case of all previous examples, the full output of this experiments are available in [34]. Additionally, the full report produced by the `ReportWidget` has also been provided as a reference for all the job scheduling that took place on the MiG.

15.6 CONCLUDING THE SCIENTIFIC EXAMPLE

This example has demonstrated a number of MEOW features and use cases. Most obviously, it has demonstrated how a MEOW system can be used to automate a complicated series of scientific analysis tasks. Although only a tomography example has been presented, it should be apparent how similar workflows in other data analysis disciplines could also use MEOW as a basis. This example has also demonstrated how easy it is to accommodate branching analysis paths. Consider that when each dataset is checked within the *pattern_check* step. It is unpredictable as to if each dataset will be accepted or rejected, yet the system can easily accommodate each. No special definition needs to be inserted, or errors managed.

In addition, the example has shown how we can use loops in our scheduling to great effect. In particular, it is worth noting that these loops can be infinite in nature. This contrasts strongly with many of the loop implementations of the static systems described in Section 4.2. Obviously this should be used with care, as unwanted infinite loops will consume resources. This will be mitigated somewhat on the MiG, which already has a number of features to limit how many resources each workgroups can access, and how long individual jobs can run for. No such management exists within the `WorkflowRunner`, and so it is perfectly possible to make an infinite loop of processing. Despite this being seen as a desirable feature in MEOW, one potential improvement in future work would be a warning system so that a user does not create such a system accidentally. As this is by no means a trivially discarded problem we will return to discussing it in further detail in Section 26.2.1.

16

A SELF MODIFYING EXAMPLE

As a final example, we will examine one possibility for new analysis structures enabled by MEOW. Namely, the ability for a MEOW system to be self-modifying, and construct, modify or remove MEOW constructs at runtime. A toy example is presented as a demonstration of the core functionality and as a potential inception for further ideas in the reader.

16.1 PROBLEM OUTLINE

In this example a user wishes to apply a filter to image data. However, the particular filters regularly change, even though the fundamental process does not. The users needs can be met with MEOW, by designing a system that will take configuration inputs to create new Patterns. Each of these Patterns will apply different filters to different data, according to their configuration. In this system the user only needs to manually write a single Pattern, and any subsequent requirements will be met by the MEOW system itself.

This problem will therefore demonstrate how we can construct MEOW Patterns from within a MEOW system. While Recipes perform the actual analysis, assembling a Jupyter Notebook programmatically has been demonstrated numerous times before[18], and so will be omitted here.

What we will create is a single Pattern and Recipe, which will construct new Patterns, based on user provided configurations. The structure for this system is shown in Figure 16.1. Here we can see that our single Pattern will respond to any configuration files placed in the *confs* directory. This Pattern will trigger jobs that will construct new Patterns, which will monitor different locations for data. These subsequent Pattern will then schedule jobs as would be expected in any other Pattern.

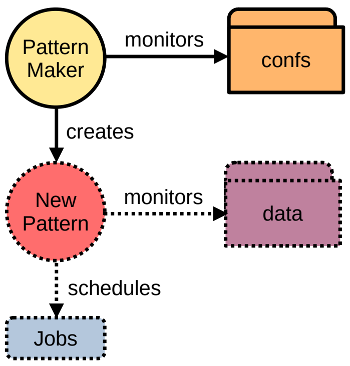


Figure 16.1: The structure of the self-modifying example.

16.2 DEFINED RECIPES

Before we define a Jupyter Notebook for assembling new Patterns, let us define the Recipe which the assembling Pattern will use. The code cell contents of this Jupyter Notebook is shown in Listing 16.1. In this Jupyter Notebook, image data is read in, a filter is applied to this image, and the filtered image is saved as a new file. The key part is on lines 15-17. Here is where the filter command is created from the arguments provided to the Jupyter Notebook. A valid filter command will be any of the filters available as part of the Python `ImageFilter` module[45], part of `Pillow`[79].

```
1  # Variables to be overridden
2  input_image = 'Patch.jpg'
3  output_image = 'Blurred_Patch.jpg'
4  args = {}
5  method = 'BLUR'
6
7  from PIL import Image, ImageFilter
8  import yaml
9  import os
10
11 # Read in image to apply filter to
12 im = Image.open(input_image)
13
14 # Construct the filter command as a string from provided arguments
15 exec_str = 'im.filter(ImageFilter.%s' % method
16 args_str = ', '.join("{!s}={!r}".format(key,val) for (key,val) in args.items())
17 exec_str += '(' + args_str + '))'
18
19 # Apply constructed command as python code
20 filtered = eval(exec_str)
21
22 # Create output directory if it doesn't exist
23 output_dir_path = os.path.dirname(output_image)
24 if output_dir_path:
25     os.makedirs(output_dir_path, exist_ok=True)
26
27 # Save output image
28 filtered = filtered.save(output_image)
```

Listing 16.1: Contents of the *filter_recipe.ipynb* code cells.

To construct the Pattern that will use the Recipe defined by *filter_recipe.ipynb*, we will need a second Jupyter Notebook. This is shown in Listing 16.2. This Jupyter Notebook also has a relatively simple progression of processing, with a configuration YAML file being read in. The contents of the YAML file are then parsed and used to constuct a new `Pattern` programmatically. Once this new `Pattern` is complete, it is written to a specified directory. This is the directory where the `WorkflowRunner` will store the MEOW constructs, and is monitored by its *State Monitor* process. By writing a new `Pattern` directly to this location, we can insert it directly into the state of the `WorkflowRunner`.

```python
# Variables to be overridden
meow_dir = 'meow_directory'
filter_recipe = 'recipe_filter'
input_yaml = 'input.yml'

# Names of the variables in filter_recipe.ipynb
recipe_input_image = 'input_image'
recipe_output_image = 'output_image'
recipe_args = 'args'
recipe_method = 'method'

# Imports
import yaml
import mig_meow as meow

# Read in configuration data
with open(input_yaml, 'r') as yaml_file:
    y = yaml.full_load(yaml_file)

# Assemble a name for the new Pattern
name_str = '%s_%s' % (
    y['filter'],
    '_'.join("{!s}_{!r}".format(key,val) for (key,val) in y['args'].items()))

# Create the new Pattern
new_pattern = meow.Pattern(name_str)
new_pattern.add_recipe(filter_recipe)
new_pattern.add_single_input(recipe_input_image, y['input_path'])
new_pattern.add_output(recipe_output_image, y['output_path'])
new_pattern.add_variable(recipe_method, y['filter'])
new_pattern.add_variable(recipe_args, y['args'])

# Register the new Pattern with the system.
meow.write_dir_pattern(new_pattern, directory=meow_dir)
```

Listing 16.2: Contents of the *pattern_maker_recipe.ipynb* code cells.

16.3 DEFINED PATTERNS

At the start of the experiment, only a single `Pattern` is defined. This is shown in Listing 16.3. Mostly, it is just defining variable names expected within the *pattern_maker_recipe.ipynb* Jupyter Notebook.

```
input_file: input_yaml
input_paths:
- confs/*.yml
output: {}
parameterize_over: {}
recipes:
- recipe_maker
variables:
  filter_recipe: recipe_filter
  meow_dir: self-modifying
  recipe_args: args
  recipe_input_image: input_image
  recipe_method: method
  recipe_output_image: output_image
```

Listing 16.3: *pattern_maker* file contents.

16.4 USING WORKFLOWRUNNER

In order to use the defined Pattern and Recipes with a `WorkflowRunner`, the script shown in Listing 16.4 was used. The only significant difference to previous `WorkflowRunner` instances is that on line 16, the state directory for the `WorkflowRunner` is manually set to the *self-modifying* directory, which is the same that the base file directory given on line 15. Putting the state directory in the same place as the base file directory makes it easiest to access from within the jobs, which makes updating the `WorkflowRunner` state easier. This does mean there will be two monitors listening to the same file structure, so there may be some slowdown in responding to events due to each event being caught and processed twice. For a small example like this on a local system, this will not create a significant overhead, though the problem will become more pronounced if used on something larger such as the MiG, and so is not generally advised.

```
1  import mig_meow as meow
2  
3  # Setup dict of all Patterns
4  patterns = {
5      'pattern_maker': meow.read_dir_pattern('pattern_maker')}
6  
7  # Setup dict of all Recipes
8  recipes = {
9      'recipe_filter': meow.register_recipe(
10         'self-modifying/filter_recipe.ipynb', 'recipe_filter'),
11     'recipe_maker': meow.register_recipe(
12         'self-modifying/pattern_maker_recipe.ipynb', 'recipe_maker')}
13 
14 # Start the local runner
15 runner = meow.WorkflowRunner('self-modifying', 1, patterns=patterns,
16     recipes=recipes, meow_data='self-modifying')
```

Listing 16.4: The Python script run_self_modifying_example.py used to run the self-modifying example within a `WorkflowRunner`.

When the `WorkflowRunner` is initially created, no additional data is present within the *self-modifying/* directory, and so no jobs are scheduled. A user can also easily see the Patterns and

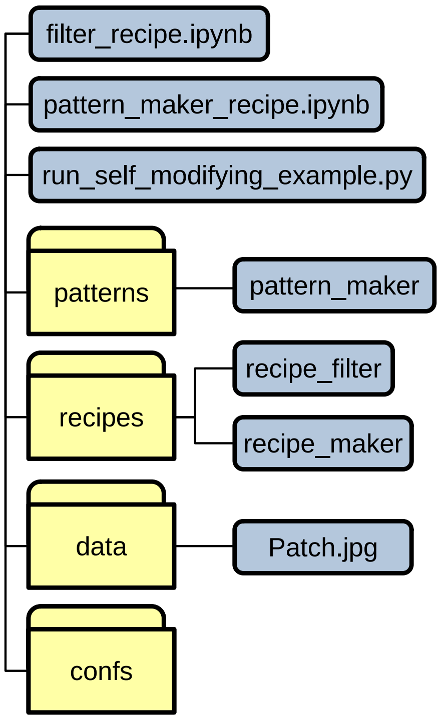


Figure 16.2: The file structure of the self-modifying example before any configuration files are added to the `WorkflowRunner`.

Recipes that are registered in the system as they will each be in a *patterns/* and *recipes/* directory within *self-modifying/*. Before we start initiating jobs, we can also create a directory, *confs*, which *pattern_maker* will be monitoring for configuration files for new filters. We can also add some image data in a *data/* directory. Just as in the previous examples, we could have done this after starting the `WorkflowRunner`, but this will mean processing can start immediately. This will give us the overall structure shown in Figure 16.2.

To start scheduling a job we will need a configuration file to place in the *confs* directory. Such a file is shown in Listing 16.5. This file is a YAML file containing a number of variable defintions, which match up to the expected inputs in the *pattern_maker.ipynb* Jupyter Notebook. If this file is placed into the *confs* directory, then the Rule created by *pattern_maker* will trigger, and a job will be scheduled.

```
input_path: data/*.jpg
output_path: '{VGRID}/GaussianBlurred/{FILENAME}'
filter: GaussianBlur
args:
    radius: 2
```

Listing 16.5: *input.yml* file contents.

This newly scheduled job will use the parameters specified in *input.yml* to create a new `Pattern`, which will be given the name *GaussianBlur_radius_2*. This will be saved into the *self-modifying/patterns/* directory and is shown in Listing 16.6. As the Rule derived from this `pattern` will monitor the `data/` directory, and we have already placed an image file, *Patch.jpg* in said directory, a job will be

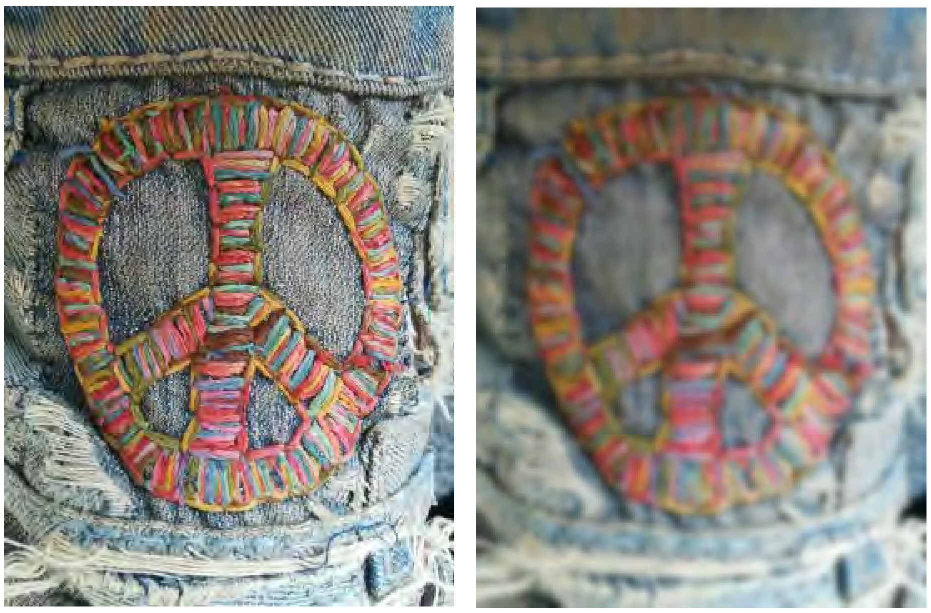


Figure 16.3: Comparison of the input and output *Patch.jpg* data, used in the self-modifying example. Input data from *data/Patch.jpg* is shown on the left, with output data at *GaussianBlurred/Patch.jpg* on the right.

immediately scheduled. In accordance with the already presented definitions, this will produce output which will be saved into the *GaussianBlur* directory. The sample input and output data used in this example are shown in Figure 16.3.

```
input_file: input_image
input_paths:
- data/*.jpg
output:
  output_image: '{VGRID}/GaussianBlurred/{FILENAME}'
parameterize_over: {}
recipes:
- recipe_filter
variables:
  args:
    radius: 2
  method: GaussianBlur
```

Listing 16.6: *pattern_maker* file contents.

16.5 USING THE MIG

Unlike the previous examples, we will need to make some modifications to the presented code before we can get this analysis running on the MiG. This is as we cannot simply manipulate the MEOW state storage location on the MiG, as this part of the system is entirely hidden from the user. This is an intentional security feature, so that users do not manipulate data they do not have access to, or corrupt the state of a live system.

To gain access to the MEOW state through the JSON messaging used in the `WorkflowWidget`, we need a *WORKFLOWS_URL* a valid *WORKFLOWS_SESSION_ID*. The *WORKFLOWS_URL* can be manually entered for now, as this is unchanging, but the *WORKFLOWS_SESSION_ID* is only generated within JupyterLab instances spawned on the MiG. To achieve this, slight modifications will need to be made to the *pattern_maker*, which we will now call *pattern_maker_mig*. It is shown in Listing 16.7. Mostly it is the same as before, but with the addition of the *workflows_session_id*, *workflows_url* and *workgroup* variables. These will all be used in the modified *pattern_maker_recipe.ipynb* Jupyter Notebook so that it can communicate directly with the MiG. As the *WORKFLOWS_SESSION_ID* is a security feature within the MiG, it has not been shown here, but was used in the actual example run.

```
input_file: input_yaml
input_paths:
- confs/*.yml
output: {}
parameterize_over: {}
recipes:
- recipe_maker_mig
variables:
  filter_recipe: recipe_filter
  meow_dir: self-modifying
  recipe_args: args
  recipe_input_image: input_image
  recipe_method: method
  recipe_output_image: output_image
  workflows_session_id: *redacted*
  workflows_url: https://test-sid.idmc.dk/cgi-sid/jsoninterface.py?output_format=json
  workgroup: '{VGRID}'
```

Listing 16.7: *pattern_maker_mig* file contents.

An abridged section of the recipe Jupyter Notebook, *pattern_maker_recipe_mig.ipynb* is shown in Listing 16.8. The full code contents is shown in Appendix N. All of these changes are to enable line 14, where the newly created `Pattern` is sent to the MiG via a JSON request.

```
1  # Variables to be overridden
2  workflows_url = 'https://test-sid.idmc.dk/cgi-sid/jsoninterface.py?output_format=
       json'
3  workflows_session_id = '*redacted*'
4
5  import mig_meow as meow
6
7  . . .
8
9  # Setup environment variables for meow to workgroup communication
10 os.environ['WORKFLOWS_URL'] = workflows_url
11 os.environ['WORKFLOWS_SESSION_ID'] = workflows_session_id
12
13 # Register the new Pattern with the system.
14 meow.export_pattern_to_vgrid(workgroup, new_pattern)
```

Listing 16.8: New code used in *pattern_maker_recipe_mig.ipynb*.

As expected, once the `Pattern` and `Recipes` have been registered with the MiG, no jobs were scheduled until a file was added to the *confs* directory. The same data file was added as before, and the

Pattern *GaussianBlur_radius_2* was created on the MiG. This itself did not schedule any processing until the file *data/Patch.jpg* was added, at which point a second job was scheduled. This produced identical results to those shown in Figure 16.3, and so will not be repeated again here. As in other examples, all results are available at [34].

This is not an ideal solution however, as it depends on exposing security features of the MEOW system. This problem is limited in scope, as in order for a user to get to this stage they will need to have access to the MiG in order to spawn a Jupyter Notebook with the *WORKFLOWS_URL* and *WORKFLOWS_SESSION_ID*. Therefore, as long as users are sensible with this credentials they will not be exposing data that would otherwise be secure. It is suspected that drawing attention to these variables which are otherwise hidden may encourage users to share them and so exacerbate the problem.

From a usability perspective though, the pressing problem is that Patterns and Recipes created in this manner will only be valid for as long as the *WORKFLOWS_SESSION_ID* remains valid. This is as only a *WORKFLOWS_SESSION_ID* will ever be registered for a user, and they will be regenerated throughout the lifetime of the MiG. However, creating a new *WORKFLOWS_SESSION_ID* will not update the variables passed in these Patterns and so when the resultant jobs try to send a message to the MiG they will be rejected. This means that the Patterns and Recipes will need to be re-registered each time a new *WORKFLOWS_SESSION_ID* is created. For this reason this solution is not suggested as a final implementation, but merely as a stop-gap demonstration of potential future functionality. A more robust implementation would be additional functionality within `mig_meow` such as each job creating its own *WORKFLOWS_SESSION_ID*, thus allowing Patterns and Recipes interactions to be verified without having to share hard-coded credentials across jobs. A similar system is already in place on the MiG for SSH users within jobs, which allows individual job mount requests to be similarly authenticated, so it is not expected to be a significant challenge to do that same for MEOW interactions.

16.6 CONCLUDING THE SELF-MODIFYING EXAMPLE

Although this was somewhat of a toy example, with a simple configuration file taken as input, this is not the limit of the possibilities. Any input file could be taken in and parsed so as to produce new or modified MEOW constructs. Although only the dynamic creation of Patterns has been shown, it is perfectly possible for new Recipes to be constructed at runtime in the same manner, it would just take considerably more lines of code to create a new Jupyter Notebook from scratch.

It is worth noting that we are not limited to merely adding new constructs, but can modify existing ones if we used some of the functions included in `mig_meow` for reading the current state of the workgroup. Here we could read in definitions, and write modified values back in the same manner as if we were altering them programatically within a Jupyter Notebook. This means that a MEOW system can create, modify and delete itself, or its parts at runtime. It can also make decisions about when to do so within a suitably written Recipe, and so we can conclude that MEOW analysis is Turing complete at runtime. Whilst it would be bold claim to state that this is unique, none of the currently encountered SWMS have come close to this level of self modification.

17

TESTING MEOW

This chapter will briefly explain what testing was carried out during the development of MEOW and `mig_meow`. These are unit tests to ensure basic functionality, user tests to check a real world system, and the specific investigation into `watchdog` and the overheads of using MEOW.

17.1 UNIT TESTS

As is good practice in any software development, a number of unit tests were written for `mig_meow` and the MiG. To go through each would not be especially useful or interesting, but all can be found either as part of `mig_meow` at [62] within *mig_meow/tests/*, or the MiG at [61] at *mig/unittest/test-workflows.py*. Collectively, these test the creation, removal and modification of the various MEOW constructs. Attention is drawn to the tests for the `WorkflowRunner` at *mig_meow/tests/testLocalRunner.py*. These test the ongoing behaviour of a live system, checking that events are identified and jobs are scheduled. Chains of progression are confirmed, so we can be confident that the requirements as set out in Section 6.2 have been correctly implemented.

17.2 USER TESTING

The other common way of testing functionality was by user testing. This could either be by running the `WorkflowRunner`, or using the development server of the MiG to test the performance of a live system. Often times this was carried out on small modifications, or in conjunction with logging to ascertain exactly what happened and what state has been reached. Although a great deal of this testing was carried out by me, additional testing was provided by other MUMMERING ESRs, both during the MEOW workshop presented in Section 20.5 and during their own analysis using MEOW.

Events Made	Events Seen	Duration (s)	Events (per s)
1,000	1,000	0.35	2857.28
10,000	10,000	3.00	3328.19
100,000	100,000	31.69	3155.38
1,000,000	1,000,000	352.94	2833.31

Table 17.1: Results of the `watchdog` test. All results are averages of 20 runs and rounded to 2 decimal places. Run on the Desktop resource, shown in Appendix E.

17.3 INVESTIGATING WATCHDOG

Any implementation of MEOW depends on the ability to detect events. Within the MiG this is already done through the use of the `watchdog`[93] package, which was introduced in Section 9.2. To check if it did indeed catch all file events, a number of specific tests were carried out. To do this a test was created that would start a monitoring process that would increment a count every time it registered an event. A number of writer threads were then spawned that would simultaneously create as many events as they could within the file system. This test was run a number of times in different variations. It was discovered that simply creating an empty file as an event was faster than repeatedly appending to the same file. It was also discovered that on the machine running the test, four writer processes was the quickest configuration. This gives us the final test code shown in Appendix O. Results of this test are shown in Table 17.1, having been run on the Desktop resource shown in Appendix E. This table shows the results for four different tests, with differing numbers of events produced in each. For each test the number of seen events was recorded, along with the time taken for all events to be produced. From these numbers we can calculate how many events were produced per second.

These results show that all events have been identified by `watchdog`. As `watchdog` is capable of identifying thousands of events per second, we can conclude that it is fit for purpose as a means of programatically identifying all system events.

17.4 OVERHEADS WHEN USING MEOW

Whilst the tests presented so far have focused on verifying correct behaviour in MEOW, it is also important to measure the performance of the system. One significant difficulty here is that MEOW is an event based system. As discussed in Section 4.2, there are few comparable event-based scheduling systems to test against, and certainly none that enjoy mass-adoption. However, several systems exists that are widely used and have a similar system for queuing and executing jobs. One such system is

Slurm, which although not a dedicated SWMS, is a tool for the mass scheduling of jobs on distributed clusters of resources and so has a similar use case to the MiG.

Each of the following test were run in a dedicated Docker[25] container. This was as a container system was required in order to run a testable instance of the MiG. In order to make the tests as comparable as possible, Docker was therefore also used in the Slurm and `mig_meow` tests cases. This would hopefully ensure that any overheads from the use of Docker in the MiG tests would also be present in all other tests, and so a more even conclusion can be drawn. All tests were repeated 10 times to get an averaged result, and each test was run from 10 to 500 jobs. These tests were not run evenly however, with the amount of jobs increasing at an accelerated rate at the jobs increase. This was as it was expected that smaller trends would be visible with low amounts of jobs, whilst with large amounts the general trend should already be visible. Each test was also run on both the laptop and threadripper resources outlined in Appendix E. The laptop is a small machine, representative of the lower powered machines most researchers have as a personal workstation. Meanwhile the threadripper is a custom built machine designed for massive processing of scientific problems. As many of the tests are not explicitly parallelised, there will not be a massive performance difference between the two, but the threadripper will be less prone to being swamped by new threads or context switching.

17.4.1 *Overheads in Slurm*

Slurm has two basic methods of scheduling jobs, `srun` and `sbatch`. Both will schedule one or more jobs, though `srun` is a blocking operation, where a user must wait for all jobs to schedule and execute before their script or terminal can progress. In contrast, `sbatch` schedules jobs in the background and so is a non-blocking operation. This means that whatever foreground process is creating jobs can continue on to other things without needing to wait for any jobs to actually complete. This is more akin to how the MiG, `mig_meow`'s `WorkflowRunner` and most SWMS's work and so `sbatch` will be used for the baseline tests as well.

To get a baseline for Slurm's performance, `srun` and `sbatch` were used to schedule large numbers of jobs at once and how long it took the scheduling to complete was timed. Note that this does not include the execution time. As MEOW can also be used for continuous analysis we should also investigate the overhead of an ongoing system. Another tests was therefore developed where jobs were scheduled individually, with each job scheduling a new job. This forms a chain of processing akin to how MEOW is expected to be used. Note that this second test includes execution times as each job needs to be executed to schedule the next. The test code itself is visible within a Github repository at [28]. The results of all three tests on the Threadripper are shown in Figure 17.1. For each test the

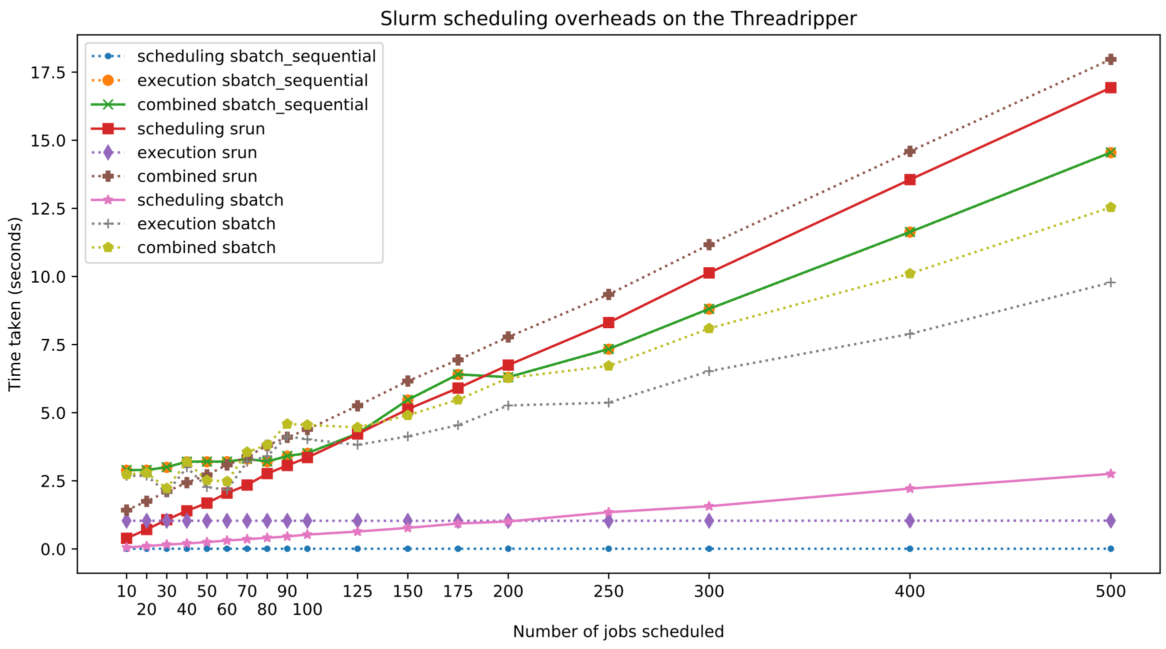


Figure 17.1: Slurm scheduling durations on the Threadripper. Each result is an average of 10 runs.

scheduling time, execution time, and a combination of both are shown. In the case of the `srun` and `sbatch` tests, the scheduling time is the pertinent part to look at and has been highlighted. In the case of the continuous scheduling test it is the combined time, which has also been highlighted. This is as the scheduling is in fact only the timing for scheduling the initial job and so shows an inaccurately fast time, whilst the combined time is the actual time for all jobs to be scheduled.

As Slurm is a relatively bare-bones system, where work to be done is split into a number of separate jobs then enrolled in a queue, there is not much scope for overhead. This is reflected in the demonstrated very quick scheduling by `sbatch`, which was the best performing and so will be taken as the baseline to beat. As the test was run with only a single Slurm worker, `srun` must wait for jobs to complete before scheduling the next, leading to the much larger overhead. Running `sbatch` sequentially runs is effectively doing the same thing, as each jobs needs to execute to schedule the next, but the use of `sbatch` means that some concurrency can happen between the scheduler and the processor, hence the slightly decreased overhead. Significantly, in all three tests the scheduling time per job is increasing linearly, meaning the Slurm will scale very well with larger job submissions. This is true of both the Threadripper and Laptop tests. Despite the larger core count and clock size of the Threadripper over the Laptop, the relative linearity of the test means that no great performance difference is found between the two. Results for both resources, and graphs showing the per-job scheduling duration are shown in Appendix P, with raw results available at [8].

17.4.2 *Overheads in* `mig_meow`

To isolate some of the MiG overheads, we will first time the `WorkflowRunner` within `mig_meow`. This will then be compared against similar tests in the MiG. Some of the same overheads will be present in both, as the structure of the `WorkflowRunner` was designed to mimic the MiG.

Within both the MiG and the the `WorkflowRunner`, the overheads will consist of the sum of the following components: Event identification in `watchdog`, Rule lookup, creating one or more new jobs, as well as any associated overhead caused by running the rest of the MiG or `WorkflowRunner`. To help identify how much each of these components contribute to the total overhead, as well as to identify additional overheads, five different experiments were created. Unless otherwise noted the Recipe in each is some inconsequential execution. The five experiments would are as follows:

- **Single Pattern, Multiple Files (SPMF)**. In this experiment a single Pattern would be created that would trigger on any file event within a directory. *N* files would then be created within the directory, causing the parallel scheduling of *N* jobs. This should allow us to identify the aggregate overheads of MEOW scheduling. This test can be directly compared against the Slurm `sbatch` test.

- **Single Pattern, Single Files, Parallel Scheduling (SPSFP)**. In this experiment a single Pattern would be created that would trigger on any file event within a directory. This Pattern would include an *N* wide parameter sweep over some variable. A single file would then be created within the directory, causing the parallel scheduling of *N* jobs. By comparing this to SPMF we should be able to identify the overhead caused by `watchdog` and event identification, by minimising it within this experiment. This test can be directly compared against the Slurm `sbatch` test.

- **Single Pattern, Single Files, Sequential Scheduling (SPSFS)**. In this experiment a single Pattern would be created that would trigger on any file event within a directory. The Recipe used by this Pattern would create another file in the same directory, so triggering the Pattern again. A variable will also be included so that a file is only created by the first *N*-1 jobs. This will result in sequential scheduling of *N* jobs. This test should illustrate the overhead in continuous looping scheduling, the most anticipated use-case for a MEOW system. This test can be directly compared against the Slurm sequential `sbatch` test.

- **Multiple Patterns, Single File (MPSF)**. In this experiment *N* Patterns would be created that would trigger on any file event within a directory. A single file would then be created within the directory, causing the parallel scheduling of *N* jobs. When compared against the SPMF experiment, this can isolate the overhead in Rule lookup. This test can be directly compared against the Slurm `sbatch` test.

- **Multiple Patterns, Multiple Files (MPMF).** In this experiment *N* Patterns would be created that would each trigger on a different specific file within a directory. *N* files would then be created within the directory, each matching to a single Pattern causing the parallel scheduling of *N* jobs. By comparing this to the other tests we should be able to identify the expected general overhead in a live system, where many differing events may happen at once, as well as the overhead for Pattern lookup in a larger memory construct compared to the SPMF test. This test can be directly compared against the Slurm `sbatch` test.

The source code for each of these tests is available within [26], and the results for the tests run on the Threadripper are shown in Figure 17.2 and again in Figure 17.3 on a logarithmic scale. As with the Slurm tests, complete results and more in depth results are shown in Appendix P. In the SPMF, MPSF and SPSFP tests the `WorkflowRunner` actually comes out ahead of the Slurm tests, with a speedup of roughly 2.5, though this obviously varies with experiment and the amount of jobs. In each case the per-job scheduling time is reasonably constant demonstrating good scalability in these situations. The MPMF does not scale well however, with it being slower than the Slurm past roughly 100 jobs at once. This is down to two parts. Firstly, each file event is identified and processed separately, and it takes time for the `WorkflowRunner` to navigate the stored Patterns. These overheads each occur in the SPMF or MPSF without adding any noticeable slowdown, and so at least for up to 500 jobs we can conclude that they are negligible in isolation, but have a quadratic effect on the per-job time when they both increase.

The performance of the SPSFS test was roughly 100 times slower than the sequential tests. This is down to three key overheads, settling, querying, and executing. Firstly, each event has a ‘settle time’, where to prevent the `WorkflowRunner` getting swamped by multiple events from the same location any events that occur at the same location and very close time are represented as a single event. This means after any single event the *File Monitor* process will wait for one second to catch any subsequent events at the same location. By definition, this will add at least 1 second of overhead to processing each event. This will occur in all tests, but in all others it will occur only once no matter how many jobs will be eventually scheduled, whilst in the SPSFS test it will occur for each sequential job. There is also the additional overhead of waiting for each job to be executed in turn. Aside from the raw time taken to execute the code, which is more complex in the `WorkflowRunner` jobs than in the Slurm testing, there will also be a delay inherent in the *Worker* process requesting a job from the *Queue* process. In these tests the *Worker* was set to query for new jobs every second. From all this we can conclude that in these tests at least the overhead from each of these was roughly 1s, and totalled

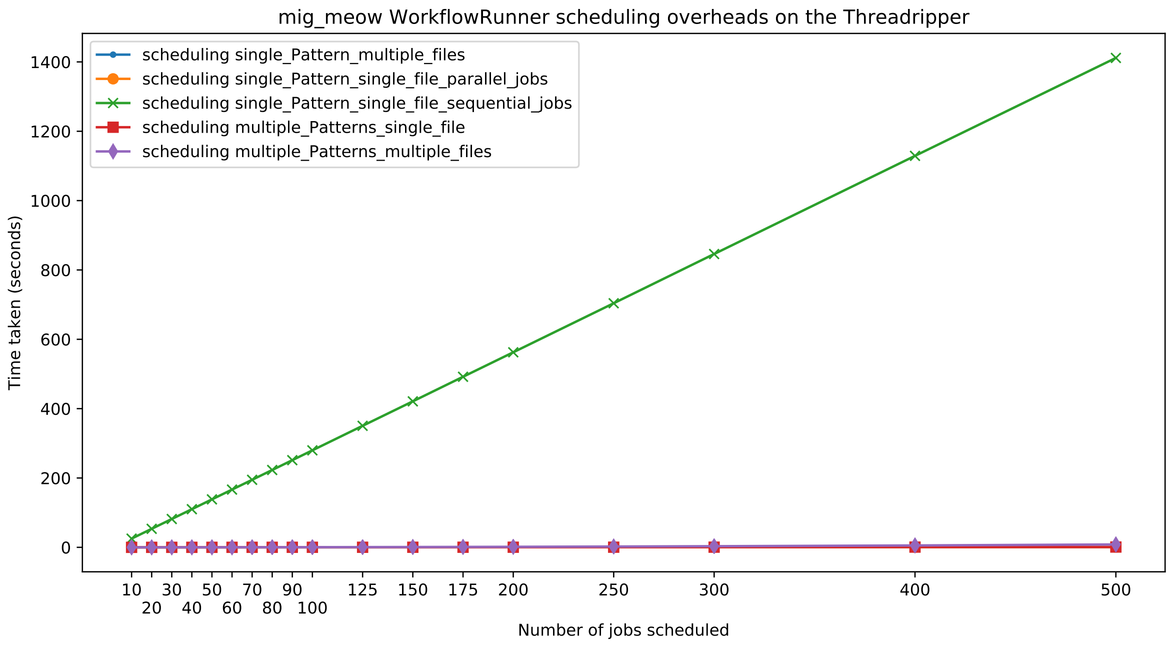


Figure 17.2: MEOW scheduling durations on the Threadripper. Each result is an average of 10 runs. This shows the same information as Figure 17.3, except in a linear scale.

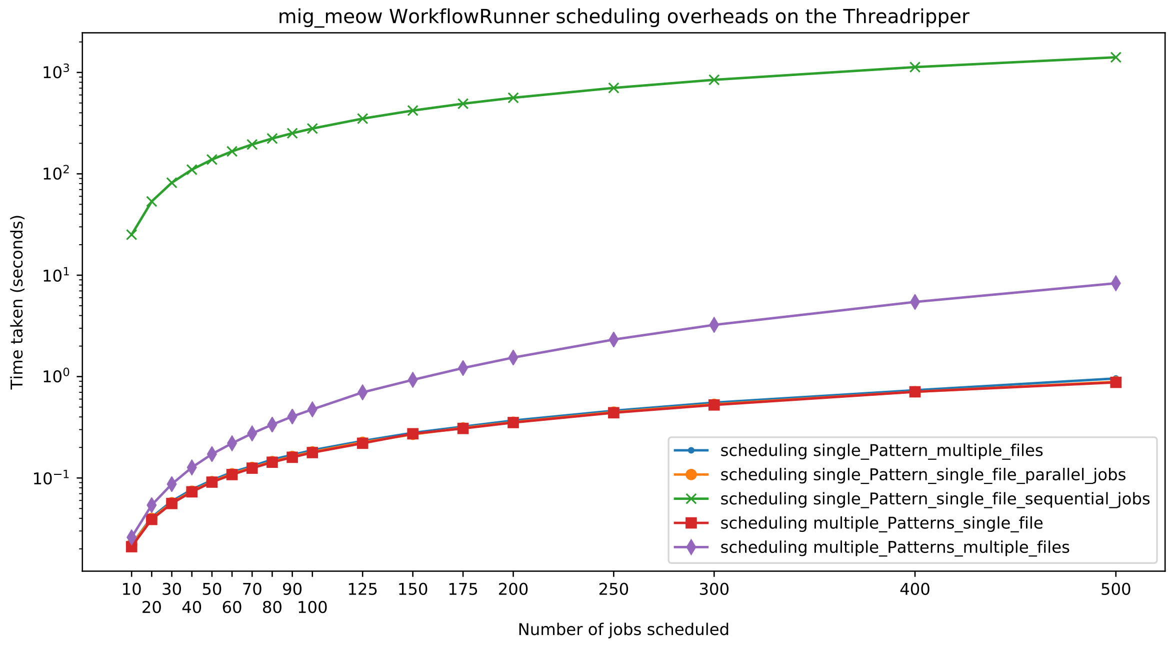


Figure 17.3: Logarithmic MEOW scheduling durations on the Threadripper. Each result is an average of 10 runs. This shows the same information as Figure 17.2, except in a logarithmic scale.

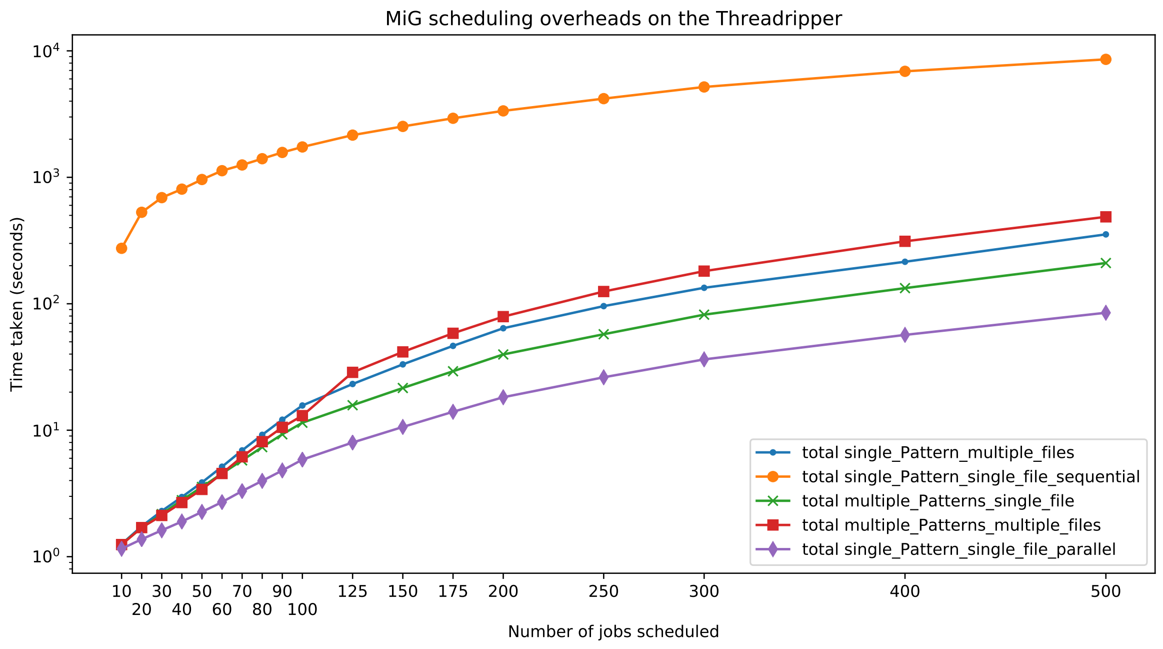


Figure 17.4: Logarithmic MiG scheduling durations on the Threadripper. Each result is an average of 10 runs This shows the same information as Figure 17.5, except in a logarithmic scale.

3s. This is significantly slower that the approximately 0.035s achieved by the sequential `sbatch` test. However, it is worth highlighting that was in the case of the SPMF, MPSF and SPSFP tests, the SPSFS demonstrates very good scalability as it has a constant per-job overhead of approximately 3s. This overhead should remain constant even across a larger system of multiple users operating MEOW analysis at the same time.

17.4.3 *Overheads on the MiG*

Exactly the same tests were run on the MiG as on the `WorkflowRunner`. The test code itself is available at [27], with the complete results shown in Appendix P. As can be seen in Figure 17.4, performance from these tests is worse across the board, but this is to be expected. The MiG is a large grid management system rather than a lightweight scheduling system, so will always run much slower than either Slurm or the `WorkflowRunner`. With this in mind we shouldn't be too surprised that generally the MiG is roughly 25-50 time slower than `sbatch`. The sequential test is even slower at roughly 500 times slower than the sequential `sbatch` test. That being said, it is worth remembering here what this is a timing actually of. It is worth noting that whilst the SPSFS test demonstrates linear scalability, all others were quadratic in nature, as is shown in Figure 17.5. This is demonstrated in Figure 17.6 where we can see that the difference in per-job timings is linear as the total number of scheduled jobs increases, which results in a quadratic increase in total scheduling time.

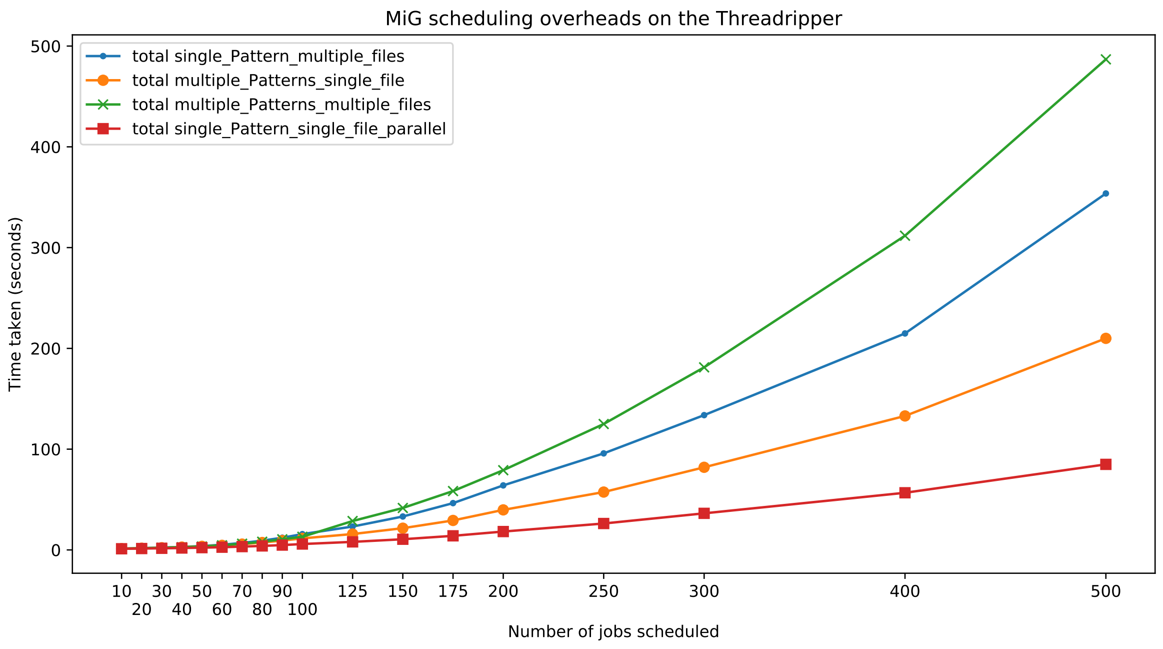


Figure 17.5: MiG scheduling durations on the Threadripper. Each result is an average of 10 runs. Note that the SPSFS result have been excluded as it has a much greater scale that crushes the rest of the results. This shows the same information as Figure 17.4, except in a linear scale.

This increase is due to the nature of event processing on the MiG. As each event occurs the `watchdog` monitor will catch the event by performing some initial processing such as attaching a timestamp to it, and then matching it against the current list of Rules. This 'pre-processing' is kept to a minimum, and so any actions from a Rule match being carried out in a threaded function that must wait for processing resources to be available. This is done to keep the monitor process free to catch further events, but means that if many events happen at once then many different threads will be started. On the MiG this 'pre-processing' is quite extensive, with a number of authentication and robustness checks needed to be carried. Matching has been sped up through the use of regex, but the overheads are unavoidable. As many different matching events occur at the same time in these tests, more and more overheads are added by starting so many different threads at the same time. Note that this does not occur on the `WorkflowRunner` as this uses the multiprocessing architecture described in Section 10.5, which does not require starting a new thread for each event.

Whilst it was expected that the MiG would run slowly, it was not expected that it would run quite as slowly as it has. Nevertheless, when we look at the results for the `WorkflowRunner` we can at least conclude that this is down to underlying issues within the MiG rather than with the MEOW system itself. If MEOW was the issue then the same lack of scalability would be present in the `WorkflowRunner`.

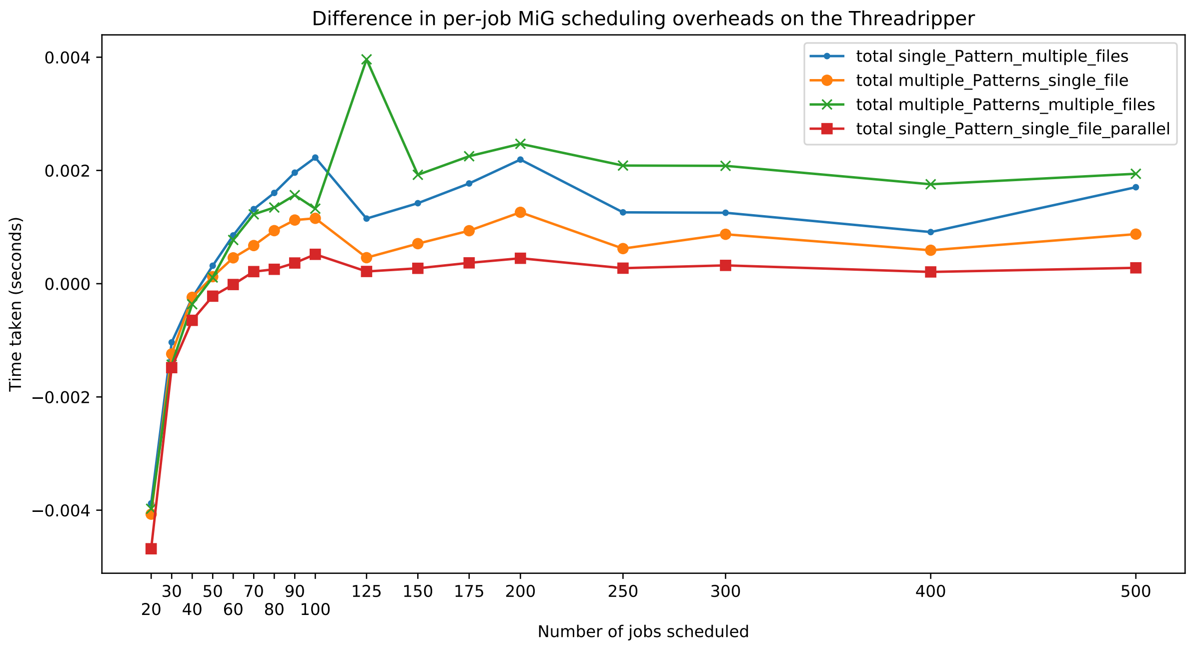


Figure 17.6: Delta in per-job MiG scheduling durations on the Threadripper. Each result is an average of 10 runs. Note that the SPSFS result have been excluded as it has a much greater scale that crushes the rest of the results.

17.4.4 *Evaluating the MEOW overheads*

In light of all these tests we can conclude that MEOW itself is not inherently problematic. Under certain conditions it can be used to schedule large amounts of jobs at once, with a minimum of user definitions. This scaling can be linear in nature when either a large amount of files or Patterns are in use at one moment. However, when both are in use at the same time the system can begin to experience greater and greater overheads. This is especially true in the MiG implementation. Even in the slowest non-sequential test (500 Laptop MPMF jobs) it must be remembered that only 0.97s was spent on scheduling each job and most tests ran considerably quicker than that. It does not seem unreasonable to expect that most scientific analysis of significance would take longer than this to complete, and so the scheduling would fade into the background compared to any execution time.

Although the scaling on sequential jobs is linear, a considerable amount of time can be spent before all required jobs are scheduled, as by definition then all but one of them must be executed before they will all have been scheduled. We also see ever increasing overheads in both the MPMF test on the `WorkflowRunner` and across the board in the MiG, except with the sequential tests. This is concerning, as the MPMF is probably the closest test to how a large grid system would be used, with many unique events occurring and being compared against many unique Patterns. If a sufficiently large enough number of users were using the system to schedule MEOW analysis, such that several hundred events were supposed to be triggering several jobs within the same few second then each user would begin to see increased overheads in line with the results presented in Figure 17.5. However, as will be expanded upon later in Section 26.1, this is seen as unlikely to occur as only a low amount of users are expected for the foreseeable future. As the request for such dynamic workflows is still relatively niche compared to the overwhelmingly linear amount of processing undertaken, it does not currently seem likely that this will occur. However, this will remain an issue until further work is spent on improving the efficiency of the MiG in this regard. MEOW is of course also entirely appropriate for local use such as through the `WorkflowRunner` thanks to its consistently low overheads and efficient scaling.

18 SUMMARY

This part has demonstrated three examples of MEOW in action. This should demonstrate the functionality and ideas outlined in the previous two parts. Each example was run on both the MiG and using the `WorkflowRunner` and can be taken as guides when designing new MEOW analysis systems. Within these examples, it has been shown how we can use MEOW to create branches or loops in our processing. We can also schedule analysis across a wide range of parameters or data files which are free to fail or succeed in isolation. The degree to which this is possible is unique to MEOW, and so analysis structures that make use of that are best suited to it.

With the end of this section, we conclude the parts that are dedicated to MEOW within this thesis. Taken together, they should give a complete view of the reasons for its creation, how it was actually implemented, and what a user could expect to do with it. By rejecting a static DAG paradigm, in favour of a completely dynamic event-based one, we can individually schedule jobs. These are then free to complete in isolation and so we can adopt structures in our analysis that were not previously possible. A fuller evaluation of MEOW and its merits is presented within Part v, although with that of many of the supplementary work presented in Part iv.

Part IV

SUPPORTING WORK

19 INTRODUCTION

In this part, all of the supporting work to MEOW will be presented. These are items that are relevant to the development or future direction of MEOW despite not being part of actual job scheduling. First, various teaching and dissemination works are presented as educating others in the use of MEOW formed an important secondary role through the project. Secondly is a translator between MEOW and the static workflow language, CWL. This work was undertaken as part of a secondment to EuXFEL, but was abandoned fairly early in development due to it lacking both feasibility and utility. Thirdly is a project I helped supervise during my studies. This is for FUR, an automated uploader between scientific instruments and the MiG. Although MEOW does not feature in the project it is may be of direct use in future plans for MEOW and so is briefly presented here. Finally we have work completed in collaboration with Xnovo Technology during secondment there. This is to integrate `corc`, a tool for scheduling processing on cloud resources, and McWeb, a tool for creating tomography analysis. This provided users at Xnovo with a means to easily access commercially available cloud resources for their analysis.

20

TEACHING MEOW

This chapter will examine how MEOW and `mig_meow` were made easier to understand and pick up through a variety of teaching materials. These will be implemented in a variety of fashions, and cover all aspects of MEOW. The hope is that by the presence of these additions, new users are more likely to successfully design their own analysis solutions using MEOW. The work presented here will also form a direct fulfilment of the fourth thesis objective, to *'create training material for the automated, dynamic workflow system'*(see Section 2.2).

20.1 THE NEED TO BE TAUGHT

In projects such as MEOW and `mig_meow`, there is a specific need to teach others how to use them. Without doing this new users will be unwilling, or even unable to learn something that has taken a not insignificant amount of effort to create. This need for supportive teaching materials is especially acute with the event driven system, as it has proven hard for new users to understand. Throughout this project, I have given numerous formal presentations, but also a number of informal talks and introductions to colleagues, researchers and friends. In the majority of these situations I was not left with the impression that the purpose of MEOW had been properly understood, or that the structure of an event driven system could be replicated by them. This was even sometimes the case after talks and presentations specifically about MEOW, and so presented a real problem for expected users of the system.

Now it is always a possibility that both myself and my talks are simply terrible, and the fault lies entirely with them rather than the subject. Although this cannot be outright rejected, it would be one of the first subjects I have encountered that has had such consistent difficulty in getting communicated to others. This leads to the problem being one primarily with the subject of an event-driven system for processing scientific research. So far this thesis, and the papers written throughout this project have focused solely on communicating the theoretical underpinning of how MEOW was designed, what problems it was intended to address, and what the use cases might be going forward. Whilst that is all worthy stuff, dedicated materials will be needed to help people looking to actually use the system as it is clearly not enough. To help new users learn how to use MEOW in their scientific analysis, a number of approaches were adopted. These approaches can be broadly characterised as technical reporting, documentation, workshops and examples. These will now each be examined in turn.

20.2 TECHNICAL REPORTING AND WORD CHOICE

Obviously the first and foremost means of communicating to others about MEOW is through the use of papers such as those shown in Appendices A and B, as well as this very thesis. Within scientific reports, which are read to gain new understanding, word choice is important in a way it is not in other forms of text. We are usually not trying to communicate emotion, or quality, but technical facts or defined concepts. These are not novel observations, but should contextualise that as the project progressed I increasingly found the use of the word workflow to be unhelpful in creating understanding in listeners. Although workflows were a starting point of this project, and MEOW was created in direct response to the requirements for a more dynamic workflow, the concept comes with too many overtly unhelpful connotations to aide understanding of MEOW.

For example, workflows are often first explained as A leads to B leads to C. There is a good deal more to workflows than that, and many domain specific definitions that could be baked in on top, but that is the basic introduction to workflows, and is fundamentally what seems to be understood by listeners grappling with what MEOW means. Put another way, the use of the word workflow seems to make people immediately imagine a rigidly static system, with then any explanation after this having to work against this first assumption. For this reason I have tried to avoid using the word workflow to describe MEOW. Unfortunately this decision was taken after the naming of MEOW and so to some extent we're stuck with this problem. However, where possible alternative descriptions of MEOW will be used, though a standout alternative has yet appeared. For this reason throughout the reports and papers, MEOW is described as an event driven scheduler. This is hopefully more descriptive of what it actually does, or at least, less unhelpful in its connotations.

20.3 DOCUMENTATION

As well as the more technical documentation such as reports, there is also the documentation within `mig_meow` to consider. All code developed through this project, but especially `mig_meow` has been documented through frequent comments and annotations, and each function or class has a descriptive documentation string. An example of this is shown in Listing 20.1. There is not much to say about this, as it is a standard part of any well maintained project but it is worth noting that particular effort has been made to ensure that this aspect of documentation has been addressed. This is as it is the documentation most likely to be used first by actual users of `mig_meow`, as well as by any developers in future who may be expanding or using it themselves.

```python
def generate_id(length=16):
    """
    Generates a random id by using randomly generated alphanumeric strings.
    Uniqueness is not guaranteed, but is a reasonable assumption.
    :param length: (int) [optional] The length of the id to be generated.
    Default is 16
    :return: (str) A random collection of alphanumeric characters.
    """
    charset = CHAR_UPPERCASE + CHAR_LOWERCASE + CHAR_NUMERIC
    return ''.join(SystemRandom().choice(charset) for _ in range(length))
```

Listing 20.1: Function definition and documentation string for `generate_id`, part of `mig_meow`.

20.4 EXAMPLES AND USE CASES

A favoured way of getting new users to understand how MEOW operates is with real-world examples. The trouble with this is that like the use of the word workflow, this can give the wrong impression about the utility and best use case for MEOW. This is as common use cases for chaining together various pieces of scientific processing together are usually best expressed with a static, linear workflow. Therefore, even if a user has understood the dynamic possibilities of MEOW they will not be illustrated by a linear analysis as is often demonstrated by existing use cases.

Another problem with using existing use cases has been that often times the scientific understanding needed to comprehend what is going on has often obscured the structure of the processing that has taken place. This has meant that although they have understood the example, when asked to demonstrate what they have learned they report more often on the algorithms and calculations, and would not be able to apply the MEOW structure to other problems. This is not to say that scientific use cases do not have a place. However, I have found that generally more understanding is gained by using abstract examples such as in Chapters 14.6 and 16.6. In both of these cases, the analysis that is being undertaken is trivial, but it serves to focus the attention onto the makeup of the system and how the structure makes the analysis possible.

20.5 WORKSHOPS

One of the deliverables within the MUMMERING project was the provision of training in the new system developed for scientific analysis. Providing these materials will conclude the fourth thesis Objective from Section 2.2. Specific teaching materials were designed to be used in workshops introducing MEOW. These materials are available at [34], within the *teaching/* directory.

The structure of this workshop is designed as individual exercises for students, where they follow along some worked examples contained in a worksheet, shown in Appendix S. This will take the students from setting up an environment for running MEOW either on their local machine or using the MiG, all the way to running a complete system of analysis. It is intended that students are started on this with minimal introduction so that they can start to explore what it is they are doing. Once 15 minutes or so has elapsed they should have been able to start and begin to setup and run the early examples. At this point a small lecture can be given lasting roughly 30 minutes. This will explain the concepts of MEOW and more of how it can be used in particular situations. Once the lecture is done the students should return to the worksheet.

It is hoped that this structure will act as a better introduction to MEOW than the more rigid structure of a scientific paper, and avoid some of the problems already discussed through this section. For instance, by not mentioning static workflows at the very beginning, or any other SWMS that MEOW contrasts with, it may be that students are better positioned to understand the unique structure of MEOW. Also, by playing around a little bit with the actual MEOW structures before hand, it provides more context for the lectures and so makes them more understandable.

20.6 RESULTS

Unfortunately, no meaningful quantitative evaluation of any of these approaches is possible, due to the very small number of people who have taken part in learning MEOW. This is not just due to only about 14 students attending the workshop, but that most of those who have learned to use MEOW have been doing so on an individual basis for their own personal experiments. Because of this, any assessment of results and the effectiveness is highly subjective. These subjective assessments are mostly what have guided the developments outlined through this section, especially the shift away from the word workflow, and the move to more abstract examples. In either case, I get the impression that they are both positive developments as since these were adopted it has seemed easier to get students to understand what an event based scheduling system really is, and how they could use it in their own research.

21

MEOW TO CWL

This section will present a small piece of work done in collaboration with the Data Analysis group at EuXFEL. It will demonstrate a converter between the dynamic MEOW and the static CWL.

21.1 MOTIVATION

Part of the problem of creating MEOW as a new tool to support the work of researchers, is that it can be difficult to get anyone to use a system they are not already familiar with. The problem is compounded by the fact that MEOW is structured very differently to a more traditionally made SWMS. Therefore, the objective of this secondment was to create a converter between a SWMS, and MEOW. This objective supports the work of Objective 1 to *design a framework to express automated, dynamic workflows*, as presented in Section 2.2. Though any of the static systems presented in Section 4.2 could be used, it was decided to use Common Workflow Language (CWL). This is as CWL aims to be a ubiquitous way of defining a traditional workflow, which can then be interpreted for use in any workflow system.

21.2 A UNIVERSAL LANGUAGE FOR WORKFLOWS

Common Workflow Language (CWL)[3] is an open standard for describing data analysis workflows. In particular, CWL is designed as a unified standard to be used across many different tools and hardware setups, so that users can define their analysis in a manner that is easily written, repeated and shared.

CWL is based on YAML files, which can be used to define either individual steps, or a workflow of one or more steps. For any step file to be run, an accompanying parameters file is also required to define any variables. Within MEOW, these two files in combination are analogous to a Pattern, in that they define the inputs and variables of an individual job. In a similar manner, a workflow is defined within CWL in a single file which will reference a number of individual step files, though will only need a single parameter file for the whole workflow. Due to the independent nature of MEOW there is no equivalent to this level of definition, and so this part of CWL cannot be directly translated.

This fundamental difference of approach was the main problem encountered in translating from CWL to MEOW and vice versa. As MEOW focuses solely on the step level, then a whole workflow layer would need to be constructed in any CWL definitions translated from MEOW. Another problem is that CWL and MEOW do not agree on the importance of outputs. As previously discussed in Section 6.2, outputs are not necessary for a MEOW definition, and in fact actively distract from its dynamic nature. In contrast, CWL makes a strict requirement for output to be declared at the beginning of any analysis. This is not a problem when translating from CWL to MEOW as we can simply ignore the definition in CWL, or at least treat it as any other variable definition. However, converting from MEOW to CWL will be next to impossible without defined outputs. As some non-binding outputs are specified within the `WorkflowWidget` it was not seen as too problematic to say that if you wish to translate between MEOW and CWL then you must declare any MEOW job outputs in a non-optional manner.

In a similar manner, CWL is a rigidly DAG-based system. By this it is meant that it does not overtly support branching, nor does it support looping of processing. This means that many of the analysis structures that are specifically enabled by MEOW will not be easily supported by CWL. A final difficulty will be that CWL has far more overt support for additional job definitions, such as environment requirements and the like. As MEOW was designed primarily with the MiG, it can leave most of this to be handled by the MiG and so makes no definitions itself. This means that if this information is required in the CWL workflow then it will be lost in a direct MEOW translation.

21.3 TRANSLATING BETWEEN PARADIGMS

Despite the already discussed difficulties, an attempt was still made at translating CWL into MEOW. This was added as part of `mig_meow`, and incorporated into the `WorkflowWidget`. This would hopefully be a user-friendly way of converting between the two paradigms, and could provide a visualisation of each system to allow for greater understanding. A top level approach to how translation would work is shown in Figure 21.1. In this we can see CWL constructs on the left, with MEOW on the right. The CWL structures of workflows, steps as well as the parameter files which instantiate them are all functionality that is taken care of by Patterns within MEOW. By this it is meant that each CWL step would be expected to become a single MEOW Pattern. These Patterns would overlap in their inputs and outputs according to the CWL workflow definition. Conversely, when translating from MEOW to CWL each Pattern will become a CWL step. The Patterns will

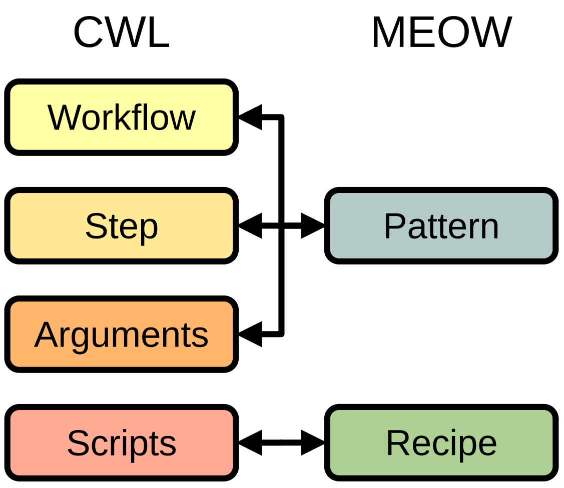


Figure 21.1: The structure of CWL and MEOW, and how they may translate.

also need to be analysed as a complete set and a CWL workflow constructed from their possible interactions. Separately to all of this are the Scripts and Recipes. Within CWL, Scripts are not a defined construct and have been introduced here only as a way of formulating how Recipes fit into the translation. Scripts here refer the code run from the `baseCommand` parameter.

As the `WorkflowWidget` was seen as a good place to base any translations, the ability for `mig_meow` to read in CWL needed to be added. To do this they will simply be kept as dictionaries, in the same manner as Recipes are stored. This is as they are already stored like this within CWL thanks to the YAML format they are saved in. It also means that the helper and validation functions already developed for use with Recipes can easily be adapted to also be of use with CWL constructs. Three constructs will be used in `mig_meow`, `Workflows`, `Steps` and `Arguments`. Each can be constructed either within the `WorkflowWidget` itself, read in from file based definitions, or passed to the `WorkflowWidget` as part of the constructor in the same manner as Patterns and Recipes.

The `WorkflowWidget` was modified so as to work in two distinct modes, either MEOW or CWL. In each mode it is possible to translate any existing definitions present in the other system into the currently selected one. As in MEOW, a visualisation of any CWL definitions is made, as is shown in Figure 21.2. This will show individual steps and how they interact, in the same visual language as the MEOW visualisation.

If we are to translate from MEOW to CWL we need to be much more rigorous in how we are defining our Patterns. Every output should be declared as part of the `outputs` parameter, and all declared outputs will always be present. Any Recipes should only take a single input, that being the triggering input. This obviously limits the potential quite drastically of any convertible MEOW definitions, but it does make it possible to fit them into a static definition.

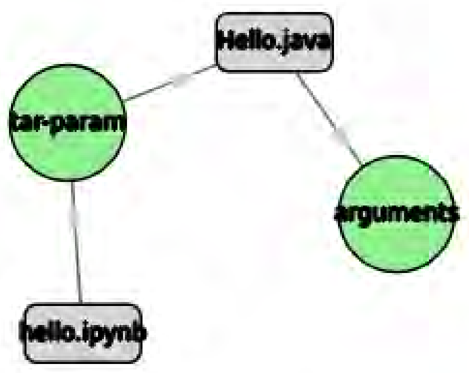


Figure 21.2: A visualisation of a basic CWL workflow. This is the *Writing Workflows* example taken from [44].

21.4 STOPPING DEVELOPMENT

Despite these initial explorations, development on the translator was abandoned before our objective of creating a tool to translate between CWL and MEOW was achieved. Both paradigms would need to be reduced to such a small subset in order to be translated, that any translator would be of no real use at all. Any MEOW system that is to be converted from cannot have any branches, loops, or dynamic structuring at all. Equally, any MEOW generated from CWL will be as linear as the starting CWL, only significantly slower thanks to the overhead of identifying and responding to individual jobs. Any CWL systems are equally limited as many of the definitions supported by them cannot be translated to MEOW as it currently stands.

All of these limitations meant that continued development for the translator became untenable. Although not an impossibility, it would be significantly larger piece of work than was first recognised to create a translator that functioned as expected. Enough compromises needed to be made to both systems that it was only useful to workflows that could not properly utilise the strengths of either system. This secondment was intended to strengthen the result for the MEOW framework in accordance with Objective 1 (Section 2.2), which it has not done due to the simplistic implementation. Nevertheless, development could be theoretically resumed in future as many of the limitations discussed through this section could be solved with enough thought and effort.

22

FUR

This chapter will introduce the Framework for Uploading research data (FUR), developed by Niels Voetmann as part of his Masters[92]. Though this chapter has been written by myself, all of the underlying work presented in this section was carried out by Niels Voetmann under my supervision.

22.1 THE NEED FOR AN UPLOADING FRAMEWORK

FUR was developed to address a need amongst researchers for a tool to automatically upload research data to a centralised storage system. This is as many scientific instruments such as microscopes have limited storage space, and very limited processing capabilities. Therefore, there is an inherent need for any retrieved data to be moved off of the instrument or accompanying system. We should also recall Objective 3 of this thesis, to *integrate the automated dynamics workflows system into the collaborative big data platform* outlined in Section 2.2. Following the work presented in Chapter 11.4, it is possible to create MEOW workflows on the MiG using data already present in the system. With an automated tool for uploading large amounts of experiment data directly from instruments, users could create MEOW workflows directly from their measurements.

22.2 FUR

FUR is designed to work primarily with image based scientific instruments, such as microscopes, where a multitude of images are taken within a single experiment run. The core function of FUR is to automate the uploading of these individual data files to the MiG, and cleanup the local data once it has been uploaded. To do this FUR runs as two processes on whatever storage the scientific instrument outputs to, as is shown in Figure 22.1.

The first process is the *Runner function*, which will establish a connection to the remote storage. Once the connection is established, the local end is sent to a new process, the *Upload function*. This process uploads data to the remote storage. The *Runner function* will now collect data as it is

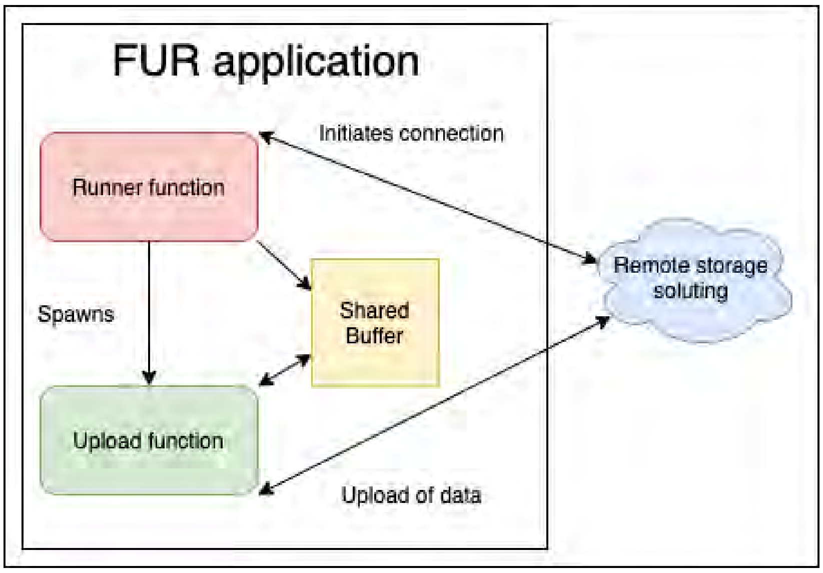


Figure 22.1: Process diagram showing the architecture of FUR. This diagram was drawn by Niels Voetmann.

produced by the scientific instrument and gather it into HDF5[42] collections. Individual data files are sequentially collected into batches of a given size. Once a batch has been filled, a shared buffer is notified. The upload function will pick up any batches in the buffer and upload them. The actual uploading of data can be done in three different ways within FUR. Firstly, HDF5 can be completely ignored and a straight upload of data files could occur. Secondly, each batch can be uploaded as separate HDF5 files. Thirdly, A fresh HDF5 can be created before any uploading, and each batch can be appended to it as they are uploaded. This is the primary intended method for most users to upload their data.

22.3 RELATING TO MEOW

FUR is of direct relevance to this thesis as it could be used to more completely integrates the MEOW scheduler into big data solutions on the MiG, whilst also expanding the core utility of the MiG itself. FUR is a fully working prototype system, which only needs refined into a more usable system than the individual scripts it currently exists as. It directly addresses Objective 3 of the thesis. Recall the scientific example presented in Chapter 15.6. In this example, a scientific instrument generated data which was processed according to MEOW Patterns. For an example like this to be practically possible there needs to be some capacity for instrument data to be directly uploaded automatically to the MiG. This can be done by FUR, which does so in an efficient, ongoing and non-blocking manner.

23

MCSTAS

During this project, work was also carried out as part of a secondment to Xnovo Technology[100]. This work was carried out in collaboration with Rasmus Munk, with parts of it forming the basis for a followup paper[69]. A fuller explanation of `corc` is also available within [68]. Much like the work on FUR in Chapter 22.3, this work will support Objective 3.

23.1 MCSTAS AND MCXTRACE, AND XNOVO

Xnovo Technology is a small company specialising in 3D crystallographic X-ray technology and making access to these specialist techniques easier for companies and research institutions. One piece of software that is commonly used in Xnovo projects is McStas, and its sister McXtrace. These are simulation packages for ray tracing neutrons and X-rays respectively. They function on instrument files, which are used to define scientific detectors through which either neutrons or X-rays are fired. Each simulation can be given a variety of inputs and can be extremely computationally heavy, which means that HPC resources are necessary. As the resources of Xnovo are naturally limited, it is impractical for Xnovo to run their own cluster, which would be extremely costly to setup and run.

Luckily for companies like Xnovo, a range of cloud resources have recently become available such as AWS[4], Microsoft Azure[60], Google Cloud[38] or OCI[73]. These systems allow for relatively easy access to HPC resources which can be deployed when needed and can be wound down when not. Such cloud resources would be ideal for running McStas and McXtrace analysis. As Xnovo does not want to be tied to a particular cloud provider, they require the ability to switch with a minimum of fuss each time. This is an especially pressing need as each provider has its own requirements and tools for interacting with the remote resources which can be taxing for non specialists.

23.2 CORC

To address the main problem would require the development of a tool for scheduling jobs on a given cloud provider. Unlike existing tools for doing this, this new tool could be shared by any supported cloud provider, unlike the bespoke tools that are usually provided. This new tool would be known as Cloud Orchestrator (`corc`) and be written as a Python package by Rasmus Munk. The package was comprised of a number of sub-components, with the overall structure shown in Figure 23.1. Each of the shown parts expose abstract interfaces, which can be implemented for a specific provider. These implementations will be created by developers ahead of time, with users only needing to fill in configuration files for their own services. Currently, the only supported providers are OCI and AEC2. This is as AEC2 already supports a wide range of existing cloud providers, increasing the utility of `corc` with relatively little implementation. Meanwhile, OCI was chosen as it was also a commonly used cloud provider, and Oracle had kindly donated a number of compute hours which could be used to test the system during development.

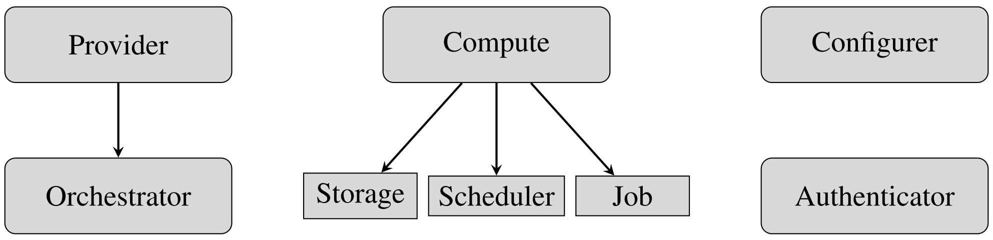


Figure 23.1: Cloud Orchestrator Framework Overview. Taken from [68].

The two most interesting components are the orchestrator and the scheduler, which allow users to create, update, read and delete cluster or job instances. The actual specifics of how this is done depends on the implementation of the of the various sub-components, but the parameters used in any `corc` query are read from a variety of YAML files. The main YAML file is used for global `corc` configurations, though others may also exist for individual services as needed. These configuration files need only be set up once by a user ahead of time. An individual user can then repeatedly create new cluster instances, or schedule new jobs using relatively simple commands such as that shown in Listing 23.1. An example of a `corc` configuration file is shown in Appendix D.

```
corc oci orchestration cluster start
```

Listing 23.1: Starting a cluster with `corc` from the command line.

23.3 MCWEB

As well as developing `corc`, Xnovo also asked for the creation of a GUI to setup their experiments, rather than as a command line tool. Therefore `corc` was integrated into McWeb[59], a web interface used at DTU as part of their own research and teaching using McStas and McXtrace. This was the part of the project I carried out. Most of this work is not directly relevant to the core objectives of this thesis though and so will not be elaborated on here, though we will look at how `corc` was integrated into this system.

Before this project, X-ray analysis within McWeb was scheduled by the backend calling McStas or McXtrace directly. This means that any processing is carried out locally to wherever Not a true acronym, but a web UI for running McStas and McXtrace (McWeb) is hosted, and will probably not be terribly efficient. To integrate `corc` into McWeb was a simple matter of altering the job execution code. This runs in a similar fashion to the already presented MiG, with a job being written to a queue by the frontend, and then picked up by a worker process which then executes it. Jobs are written to the queue as a copy of the appropriate instrument file, which is then executed by the worker by invoking either McStas and McXtrace as a system call. This was simply replaced with `corc` system call instead. The actual parameters for the `corc` scheduling are kept in the YAML configuration files, so relatively little needs to be added to the McWeb GUI to make this possible. A simple checkbox was added so that users could toggle if the job was to be scheduled locally on a remote resource, as well as a box for entering the paths of any additional files that would need to be copied to the remote resource as part of a job.

23.4 RELATING TO MEOW

Although `corc` was developed for Xnovo, it is not tied to any one programme or problem, and could be used on any currently application. This was demonstrated by integrating it into McWeb, allowing for McStas and McXtrace analysis to be run on disparate cloud resources. Much like FUR, `corc` could be used in future to expand the utility of the MiG.

Another potential use for `corc` is to integrate it into the `mig_meow` `WorkflowRunner` to radically increase the utility of running a non-MiG based analysis. This is especially true in light of the overheads identified in Section 17.4. The MiG may be a complete grid solution, but if users just wish to schedule some MEOW analysis quickly and efficiently then the `WorkflowRunner` would be better suited. Currently though, the `WorkflowRunner` is limited by only using local resources to process and scheduled jobs. Some means of easily accessing remote resources would mean that the runner is able to process much larger problems without having to be physically run on a supercomputer. This began to make the `WorkflowRunner` itself a simple big data analysis platform for those without MiG access.

24

SUMMARY

This part has presented all of the ancillary material that does not fit into the main work presented within this thesis. Some of this work is supplementary to MEOW. For instance, FUR is an framework for automating the uploading of data from scientific instruments to the MiG. Although this does not feature MEOW directly, this may be of use in future use cases such as that presented in Chapter 15.6. Similarly, `corc` was integrated into McWeb to improve its performance by accessing cloud resources. In future, this could be directly integrated into MEOW systems, either via the `WorkflowRunner` or the MiG itself.

Another related work piece was that of the CWL-MEOW translator. This was intended as a way of translating between static and dynamic systems, though was quickly abandoned as it proved to be rather a large amount of work for a very small amount of use. The reason this was first considered as worthwhile endeavour was that MEOW was proving hard for new users to understand, and so a translator from something familiar could be of assistance. As the translator did not work out, other methods of teaching were considered. The resultant learning aids and documentation were presented.

All together in combination with the previous 3 parts, this should now give a complete view of all of the relevant research, development, testing, teaching and documentation that has taken place during the development of MEOW.

Part V

DISCUSSION

25

INTRODUCTION

In this part, we will discuss the work that has already been presented. We will evaluate its successes and its failures, and tie it all together into a unified project. In order to evaluate the success of the work, we will consider the objectives presented back in Chapter 2.3. MEOW and `mig_meow` in particular will be considered, in reference to the requirements for a SWMS set out in Section 4.1. This is where we will expand on some of the problems that may already be apparent in MEOW, and consider how we might fix them with further development.

In light of this, we will also consider use cases for MEOW, and where I think it is most likely to be applicable. We will consider a whole range of use cases, from the original motivation ones, to where it could be used as it currently stands, to where it may be used in the future with further work. By the end of this part, readers should understand my own evaluations of the work carried out in this project, as well as my recommendations for what work is worth carrying on in subsequent projects.

26

ASSESSING THE PROJECT

In this chapter we will judge the successes and failures of the work presented in this thesis by considering it against the core research question. As a refresher, this was:

> ***Is it feasible to create a tool for the automatic creation of dynamic scientific workflows, available within a big data capable platform?***

26.1 RATING FEASIBILITY

The main goal of a dynamic workflow system is in direct contrast to the static fashion outlined in Section 4.4, with scientific processing traditionally conducted as part of a linear, unchanging workflow, managed by a SWMS. To address this, MEOW has been introduced as a framework for scheduling jobs in response to events. Rather than adopting a top-down approach such as is typified by a workflow constructed from a DAG, MEOW only allows for a bottom-up identification of individual jobs. These may causally 'link' together, creating a workflow as an emergent property whilst still allowing complete independence of each individual job. This allows for dynamic structures to emerge such as loops or branching analysis that can progress completely automatically. Of particular note are the implementation requirements for such a system described in Section 6.2. These outline exactly what needs to be implemented in a event driven system so that up to date jobs will always be scheduled. This was then demonstrated through Part ii and demonstrated in Part iii using both the MiG and `WorkflowRunner` included within `mig_meow`.

Of course designing the system is only half of the problem, with the other being tying it into a big data platform. Here we have used used the MiG, which was a suitable system as an existing platform for facilitating scientific processing. Extensive work has taken place over prior years to ensure that this system can manage large amounts of concurrent and parallel job processing on a collection of heterogeneous resources. By integrating MEOW, the most significant improvement to the MiG is probably that it is so much easier for users to schedule repeated processing on remotely accessible data using the MiGs resources. This is in keeping with scientific use cases, which often require repeated processing of the similar data sets. At the same time, access to resources and setting up inputs and outputs no longer need to be laboriously done on a per job basis, as it can be taken care of automatically as part of MEOW scheduling. This could increase the usability of the MiG. As was noted earlier, with few users it is difficult to get a definitive answer, though early users have managed to schedule large numbers of job with relative ease.

Many of the tools presented within this thesis support collaborative working practices. The MiG is already a platform designed to store and present data in a collaborative manner, and the MEOW integration only increases this. MEOW systems are designed to be a collaborative effort, as one of the early ideas for a use case is that different members of an eclectic team, such as the ESRs within MUMMERING, could each create and maintain their own Patterns and Recipes. These would overlap and so a complete chain of analysis could be created by individual specialists, with data flowing naturally from one step to the next. To help foster this, Patterns and Recipes on the MiG have been designed to be shareable, unlike most constructs on the MiG which are strictly on a per-user basis.

Avenues for further work exist within this area however. By implementing FUR within the MiG, and using that as a starting point for analysis, it would be possible to directly connect scientific instruments to the MiG. These could then be used as continuous input for all manner of automatic analysis, which would be stored, processed and viewable all through the MiG. Additional development could also be focused on increased interaction with the MiG from within a Python script or Jupyter Notebook that is not spawned by the MiG. This would extend the utility of MEOW by allowing users to utilise the MiG to upload their data and trigger processing without having to somehow upload it separately to the MEOW definitions.

However, despite the success of this investigation, I am not convinced that this is yet a pressing research question. As will be expanded upon in the following Section 27.1, I am unconvinced that MEOW solves a problem that currently exists. It is a significant departure from static systems and there is a degree to which it may be throwing a baby out with the bath water. As demonstrated by Section 4.2, there are already a large amount of SWMSs available, and that section is by no means an exhaustive list. Adding a further tool to an already crowded market may not have been a wise move. That being said, MEOW does offer something unique, and is can definitely construct analysis structures that would be difficult or even impossible to replicate in a static workflow. I anticipate that in the coming years there will only be more demand for humain-in-the-loop interaction, and scientific facilities will need to develop higher throughput systems for storing, managing and processing their data. This highlights that I am not saying that MEOW is useless. Although I do not expect it to be commonly used tomorrow, I do think that it may be worthwhile research for future systems development.

26.2 MEETING SWMS REQUIREMENTS WITH MEOW

As a system for running scientific workflows, MEOW should be capable of meeting the requirements set out in full in Section 4.1. These will now be considered in turn with regards to MEOW, `mig_meow`, and the MiG.

26.2.1 *Automatic Optimisation*

This requirement relates the need for a SWMS to automatically make use of parallel hardware, and run the workflow in an efficient manner without requiring in depth HPC knowledge. Through the independent nature of jobs, MEOW is well suited to automatic optimisation both when run with the `mig_meow` `WorkflowRunner`, and also when run on the MiG. In both cases, it is trivial for multiple jobs to be executed in parallel, with the number of processing nodes automatically pulling single jobs from the queue and executing them. However, two significant limitations occur. These are that there exists the possibility of a race condition within the current MEOW implementations, and that they also currently lack any limitations on the number of scheduled jobs.

First let us consider the possibility of a race conditions by looking to Figure 26.1. This diagram shows the varying state of three components within a MEOW system across five different points in time. At the first point in time, shown in the horizontal band **A**, a data file shown in yellow is created, which schedules a job, also shown in yellow. At point **B**, this job is executed on the resource, which will expect the original triggering file as an input. This is the system working as expected. Point **C** progresses much like the first, with the data file being modified to now be in a new state, shown in red. This will trigger a new job. However before the execution resources can request a new job from the queue, the data file is updated again in point **D** to a new blue state. This will as expected, schedule a new job, with two jobs now in the queue each expecting a file at the same place to act as their input, but expecting that file to be at a different state. At point **E** when the red job is finally executed, it will now execute on the blue state, rather than the red state that it was first scheduled off of.

Such a scenario could occur in a few other ways, and it is not clear how exactly this could be remedied. Some care should be taken so that such a situation does not occur, such as by avoiding overwriting triggering files when it is still likely that a job is still to be processed from the old one. Regardless, this is still a potential issue that a casual user may not expect, though it would be unlikely that such a user would design a MEOW system that could cause such a problem. Therefore this is determined to be unlikely to occur in a real system, but a rare possibility does not mean it has been correctly managed. Future work will be needed to properly address this problem.

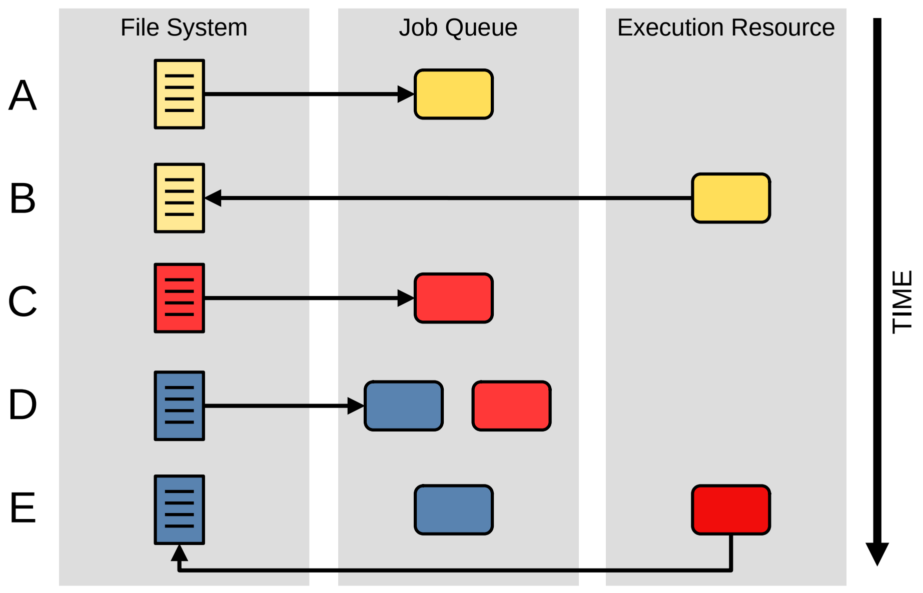


Figure 26.1: The possibility of a race condition within MEOW.

Now we should consider the potential problem of infinite processing. For an example of this we can see the scientific example presented in Chapter 15.6. In this example a loop exists between the two Patterns `initial_check` and `regenerate_check`. Although statistically unlikely, there is nothing here that says that this will not go on forever in an infinite loop. There is no current limiter in the `WorkflowRunner` or on the MiG that will limit the number of jobs scheduled here. Even in non-looping code there is always the possibility that users can create Patterns that swamp a system by scheduling massive amounts of jobs at once.

Within the MiG this is somewhat mitigated as resource access is managed in a variety of ways to help prevent a single user claiming an entire grid system, with there being limits on how long jobs can last, what resources a users jobs can access, and how much of any single resource each job can claim. Within both the `WorkflowRunner` and the MiG job scheduling is also limited to the lifetime of a Rule, in accordance with the requirements set out in Section 6.2. Whenever a Pattern or Recipe is deleted the corresponding Rule will automatically be removed, which will lead to the cancelling of any jobs from that Rule still in the queue. This means that if a user sees an unnecessary amount of processing has been scheduled they can remove the appropriate Pattern to automatically clear the queue whilst still maintaining the live system. This is aided by the `JobMonitor` widget, the MiGs existing job monitoring tools, or the `WorkflowRunner` feedback options.

No inbuilt tools for throttling jobs scheduling currently exist. As the use cases for a MEOW based system are varied in scope and nature, the solution to this problem will probably not be a one-size fits all solution. Many different user configurable options would be desirable, with possibilities including setting an expected number of jobs not to exceed, or a limit on how many new jobs to be scheduled per second. Regardless, exactly what all these limits may be, where to implement them, and how to ensure they effectively limit unwanted behaviour without hindering unpredicted potential may be a substantial piece of work in its own right. For example, if we placed a simplistic limit such as no single Pattern could schedule more than 100 total jobs, then we would make it impossible to create a loop more than 100 iterations long. Therefore this is perhaps the largest avenue for future work.

Recall that one of the main motivations for MEOW was to be used to explore unknown workflow structures, and so setting hard edge limits was consciously avoided.

26.2.2 *Clarity*

Creating individual Patterns and Recipes is certainly easy with `mig_meow`, and checking if they will link together can also be simply checked through the use of the `WorkflowWidget` visualisation. By providing the variable keywords outlined in 10.1 it is possible to create varying overlapping paths that are still predictable without being overly limiting. This utility to extended to the overall structure of any analysis chains, which can be as varied as a user can design, thanks to the independent nature of jobs.

One shortcoming in terms of clarity is that there is an inconsistency in the current parameter definitions. As it stands, the *trigger_path* is defined relative to the base workgroup directory, whilst any variable path for an output file must contain the workgroup in the path. This is highlighted in the Pattern shown in Table 26.1. It shows a Pattern that is triggered by a file called *file1.txt*, located within the base workgroup directory. A variable, *outfile* is given which is to be used as an output path for a new file, which will be placed next to *file1.txt*. However, to get *file2.txt* to be placed next to *file1.txt* we need to prepend the {VGRID} keyword, or use the specific name of whatever workgroup this Pattern is opperating within.

```
input_file: some_input_file
input_paths:
- file1.txt
output:
  outfile: '{VGRID}/file2.txt'
parameterize_over: {}
recipes:
- some_recipe
variables: {}
```

Listing 26.1: An example Pattern showing the inconsistent path levels in different variables.

This is a relatively minor issue however, and could be solved by asking users to prepend a workgroup to the triggering path as well. As this is the only significant issue affecting the clarity of constructing a long chain of analysis, we can confidently claim that the need for clarity has been met by `mig_meow`.

26.2.3 *Predictability*

A MEOW system is eminently predictable at each stage thanks to the explicit *trigger_path* variable which makes it clear exactly what each step will be triggered by. Even with a wildcard character in a, *trigger_path*, it is not hard to predict what files will match to what Patterns. It is therefore easy to predict what processing will take place as a Pattern also explicitly ties a *trigger_path* to a Recipe. The visualisation provided as a part of the `WorkflowWidget` also makes it very easy to predict how different Patterns and Recipes will link together, and the overall structure of analysis that will be undertaken.

26.2.4 *Recordability*

The need for recordability is not well met by MEOW systems, run either locally through `mig_meow`, or through the MiG. In both cases it is possible for a user to individually inspect job records for an exact record of what processing took place, with what definitions. However there is no easy way of accessing these records in a collective manner, so as to get an overview or some such. In addition, although extensive logs and records are kept as to what processing is undertaken, no specific logs are kept as to what hardware resources were used to process these jobs.

Some headway has been made on showing how individual jobs relate to one another, through the provision of the `ReportWidget`. This is currently only available for the MEOW on the MiG. It manages to effectively show how jobs fit together, which along with the visualisation in the `WorkflowWidget` can make for a clear report on the the overall structure. Obviously this is somewhat limited, as it is not currently part of the `WorkflowRunner`.

To address these shortcomings more provision should be made for the reporting of sets of jobs, rather than just as individual elements. In the same manner as the `ReportWidget` visualisation, some report detailing issues encountered throughout the analysis could be provided, or some way of highlighting individual jobs that a user may need to closer inspect. Finally, it would be of great utility if the `ReportWidget` could be made to work with the `WorkflowRunner`.

26.2.5 *Reportability*

This is somewhat met or not in the same manner as recordability. Tracing what processing was done on what data is possible on a per job basis, but it is difficult to access, and come be improved upon. There is currently no inbuilt support for checksumming or the like to validate that data has not been modified since processing was completed. This could be addressed with relatively ease in future updates both on the MiG and with the `WorkflowRunner`

26.2.6 *Responsiveness*

This requirement is directly met by MEOW in a way that is unmatched by existing SWMSs. Obviously, a MEOW system will operate in response to changes in a file structure so as to schedule new jobs as new files are created. However, as demonstrated by the example in Chapter 16.6, MEOW systems are also capable of responding to changes in their own structure. This is a feature that is as far I am aware, unique to MEOW as no pause or shutdown is required for changes to be made.

Some functionality from the MiG can also be leveraged here, as if jobs are scheduled using that then it can alert the user as jobs complete. This email contains little information itself, but does have a direct link to the logs on the MiG so that a user can view the results and any error logs if a problem was encountered. Error handling within MEOW is inherently well managed due to the independent nature of jobs. Any job can fail without affecting others in any way., making for a very resilient system that can effectively manage a mix of valid and invalid inputs as has been shown in the examples in Part iii. For these reasons, the requirement for responsiveness has easily been met by MEOW.

26.2.7 *Reusability*

Reusability is another requirement that is easily met by MEOW. Most obviously, as MEOW systems are live systems, they will automatically apply to any new data that appears. For instance, the example shown in Chapter 15.6 could be left running indefinitely so that data cleanup is applied to any data produced by an instrument automatically with only one setup ever required. This obviously makes the analysis extremely re-usable.

Aside from this, Patterns and Recipes are reusable between systems. Each Recipe is really just a Jupyter Notebook, and Jupyter Notebooks already come with a whole host of features allowing for easily shared and understandable code. Similarly, Patterns can also be understood and adapted with relative ease, or just reused directly in a different system entirely if need be.

26.2.8 *Scientific*

MEOW and `mig_meow` have actively tried to support existing scientific practices through many of the development choices made. Most prominently is the use of Jupyter Notebooks as the basis for processing, as these are commonly used within science already, as a means both of defining processing and of presenting findings[95].

One potential short coming that was discovered, is that the common scientific data storage format, HDF5 is unsuited to event handling in the manner used by MEOW. This has not been fully investigated, but it appears that the act of reading a HDF5 file creates a modification event, and so could trigger processing of a MEOW Pattern. This could result in an undesired cascade of processing, even if no data has ever actually been changed. It is currently unclear how this would be managed, and so for the time being it is recommended that HDF5 files are not used as a the basis for processing. This obviously somewhat limits their functional use within MEOW. To address this, work will be needed within the event listener itself, both in the MiG and the `WorkflowRunner`. Specific implementation will be needed to respond to events with HDF5 files, and to somehow either mask or ignore the event as appropriate. This would especially be an issue if FUR is to be implemented with MEOW in some manner, as its primary output is HDF5 files.

26.2.9 *Well-formedness*

This requirement overlaps a fair bit with clarity and predictability, and as such inherits the strengths and weaknesses listed for them. A number of helper functions and points of feedback are provided within `mig_meow` and the MiG to assist users in creating MEOW constructs. Patterns are probably the most complex single part of MEOW and so a specific `integrity_check` function was created specifically to check the definition for common mistakes or inconsistencies. Individual functions consistently try to provide useful feedback as to what has gone wrong, if something has. This applies to both `mig_meow` and the MiG.

27

USE CASES

Within this chapter we will consider the main identified use cases that the work presented in this thesis could be used for. Some of these will require further work, in which case it is highlighted, but there are also examples of whatMEOW could be used for right now.

27.1 EXPLORATORY WORKFLOWS

One of the originally motivating examples for developing MEOW was the exploratory workflow. This means that the analysis is undertaken to gain new understanding of the experiment space, and so by definition it is not completely understood at the beginning. For example, consider a use case within oceanography. Various simulations have been created over the years, such as Veros[39]. This can simulate various ocean states, with a user defining a large range of input parameters and the simulation progressing from there. If some state is observed in an actual ocean, one technique to investigate it is to try various input parameters in Veros and see which ones produce similar outputs to those observed. In such a use case, numerous simulations will need to be run across a range of input parameters. Scheduling all of the analysis in such a situation demands automation, as to do so for each individual parameter value would be extremely time consuming. Because this experiment space is not completely understood, there is always the chance that some of these inputs are invalid, or produce unexpected results.

MEOW offers a clear solution to these problems as the event driven nature, in combination with parameter sweeping, allows for the mass scheduling of jobs from only a few inputs. In addition, even if some values produce incorrect results or encounter an error, the rest of the processing can continue without interruption. With some clever Recipes writing, and modifying the Patterns as was shown in the example in Chapter 16.6, it would be possible to create analysis that would discard those results that were further from the expected output. You could setup a system that through repeated iterations could zero in on appropriate starting parameters in manner similar to a machine learning algorithm. Such a system is outside of the scope of this project, as it would require considerable more development for it to be feasible within MEOW.

Another potential use case for MEOW is long running, many staged analysis, such as . Even with relatively simple workflows, such as that presented way back in Figure 3.2, if the analysis is sufficiently complex then computing each individual step can take hours or even days. In cases such as this it may be that some analysis changes, either by some parameters being changed, or a whole algorithm being swapped out for another. In a static system this would usually mean running the whole analysis again from scratch, even if the changes only occurred half way through the complete workflow. As each job is independent, in MEOW you can change an individual Pattern or Recipe and only the jobs from that step and later will be scheduled, completely saving re-running the earlier processing that has not changed.

It is worth noting that currently these use cases do not seem to be that common. By this I mean that during my PhD most researchers I have interacted with have simple, linear workflows on reduced data sets. They test their workflows in this manner until they have a working implementation which then then run the complete data set on. Until now, setting up exploratory workflows in the manner described above would be time consuming using only static systems. Therefore it might seem oblivious to say that no-one currently works in a dynamic manner, whilst also stating that the tools to do so haven't existed till now. Nevertheless, the concern remains that due to the tools not being present, researchers are already used to working in a static fashion, and so some small amount of work may be needed by others to convert their analysis workflows into ones that better take advantage of the dynamic possibilities of MEOW.

27.2 TRIVIALLY REPEATABLE JOBS

A use case that I expect is more likely to occur is that of the trivially repeatable job. By this it is meant the sort of processing that has to be done time and again on different data sets. It is not hard to predict the outcome, nor is it especially taxing to set up individually, but the difficulty comes from the sheer repetitiveness of the task. Such tasks can be easily automated through the use of MEOW, so as to be taken care of with minimal human input required. For an example let us consider scientific instruments, and collecting data from them.

Many scientific instruments produce data that needs some sort of cleaning, formatting or calibrating. This is not especially complicated. It is however repeated every time the instrument is run. Within large facilities such as EuXFEL, external researchers come from all manner of other organisations to use the highly specialised instruments available there. These external researchers require support from internal specialists in the particular instruments, who will assist in the running of the experiment and the retrieval of the data. Some of the tasks these internal specialists have to perform are complex, requiring a good deal of time and effort to resolve. Others meanwhile are simple, repetitive tasks that require a good deal of local knowledge to set up but no real technical expertise. For instance, when on secondment as EuXFEL I observed that specialists were required to clean up the data and get it usable before external researchers could being to use it within their analysis. This cleanup was trivial for the internal specialists, as it was mostly a matter of applying scripts with parameters they could easily retrieve. However, in a 24 hour facility such as EuXFEL, they were often not around and so delays were inevitable as external researchers needed to wait for the internal specialists to be on hand to trigger something trivial to them, but unknowable to anyone else.

Such small jobs that are easily created with local knowledge would be ideal candidates for automation through MEOW. They could be performed immediately, reducing delays which is is important as experiment time is a valuable commodity at such large facilities. Additionally, it will free up the time of internal specialists who can now concentrate on the complex, bespoke problems that emerge, rather than having to set up simple, repetitive tasks. Although we have mostly talked about large facilities here, even small instruments could benefit from MEOW used in this way.

27.3 CONTINUOUS MONITORING SYSTEMS

Another use case that MEOW would be well suited to is in continuous systems, such as typified by WIFIRE[2]. This is as MEOW is designed to be run as part of a live systems, and can easily cope with infinite loops of processing. WIFIRE is a system for monitoring wild fire risks in the USA. It does this by running a continuous loop of smaller workflows, which create a predicted state of fire risk. This state is monitored, but also used as input for subsequent runs. As well as just simulated data, large amounts of real world data are used as input. The range of data that is available as inputs is not always the same, with sensors being added, moved, or removed on a continuous basis. Additional sources such as twitter can also be brought in at times, to get a more complete picture of whatever situation is developing on the ground, an important feature in what can be an important disaster response tool.

Although WIFIRE itself is a large, and complex enough system that I would not recommend that it itself get retooled into using MEOW, it does demonstrate a type of system that could benefit from MEOW. A live system such as MEOW, which is continuously monitoring for changes in a file structure can respond immediately to any changes in the file state, so as new data comes in or old data is overwritten new analysis can be conducted. The looping nature of WIFIRE is also easily replicated within MEOW. Another aspect that would suit this well is the ease with which errors are handled within MEOW. As jobs can fail individually without crashing the whole system, it would be a robust foundation to build a continuous system onto.

One shortcoming in the current setup however is that within MEOW as it currently exists, it is impossible to respond where no event has occurred. By this it is meant that there is nothing like a timer, or such like to trigger a new job if not new data has been produced in a certain amount of time, or to respond to a failed job that did not produce any data. These would be important features in a monitoring system as responding to what has not occurred but should have, can be as essential as responding to what has. That being said, if this functionality was needed, it would be possible to create Recipes which replicated them. However, specific implementations of these would still be desirable within MEOW.

27.4 HETEROGENEOUS SYSTEMS

Finally, MEOW systems are especially suited to heterogeneous systems that are disparate in type, timing and space. Often times within computing heterogeneous systems refers to a collection of hardware of differing sizes, such as a collection of processors operating at different clock speeds or with differing amounts of memory. Here we are talking about systems that are even more disparate than that, such as the difference between a CPU running a Python script, an FPGA processing sensor data, and a user sorting files. These are all run in completely different ways but all all produce or respond to file events.

For example, consider the analysis presented in Figure 27.1. It has six different resources of varying types witihn it. Our first resource is the scientific instrument itself, which is tightly coupled with the second resource, the instruments own storage. The instrument itself will be used to produce raw data, which will need to be cleaned. This could be done according to MEOW definitions, as described in the earlier Section 27.2. Space will always be limited in this local storage, and so data will be moved to a more permanent home. This may be an online grid solution such as the MiG, but could also be a facilities central storage server such as is used in beamline facilities.

Once data is sent to the centralised storage, there is not the same urgent need for instant processing of data. This is where a researchers analysis will start to be conducted. In this example the first stage is some simple analysis like creating an image from the data, along with some basic statistical analysis so as to inform the next stage. This initial analysis would only require a relatively small amount of processing, and would take only a few minutes to complete on a small HPC resource such as a single-digit multi-core processor. The output of this initial analysis would then be assessed by a human in some way. This could be through manual inspection of image files, with the user

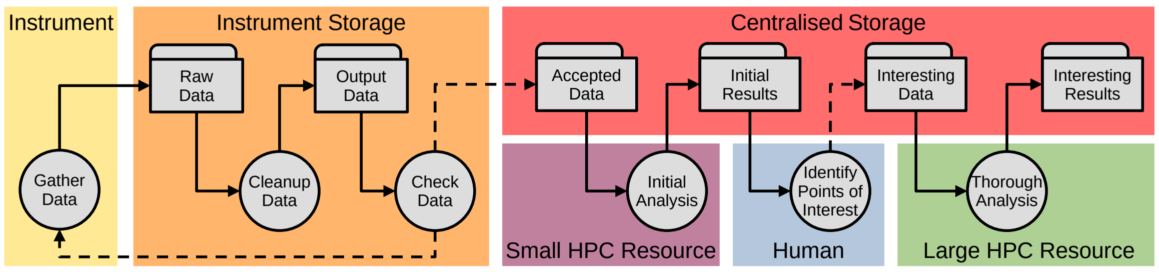


Figure 27.1: A heterogeneous analysis. Note that solid lines show where all data inputs will produce output, whilst dotted lines show where only some inputs will produce outputs.

looking for particular results of interest that will require further investigation. Unlike the other steps in this analysis, this on is not conducted automatically as a human needs to interact with the system. Therefore it will almost certainly be much slower than the other parts as humans go home for the night, or get distracted by other important issues. It could be that this identification takes minutes, or potentially weeks for the human to get round to it. When they do identify the regions of interest, further detailed analysis is done on the data. This is extremely complex processing, that requires a large amount of computing resources to complete in a reasonable time, such as a cluster of GPUs, or a grid of cloud compute resources. Even with these resources the processing is expected to take several days to complete before the final, presentable results are produced.

Throughout this analysis, a variety of jobs have been scheduled or set up, which have each been processed on different resources, and will each take a radically different amount of time. MEOW is obviously well suited to this style of system as it can automatically respond to each job and data input as they emerge, with each stage able to be completed in parallel on their respective resources. Here, each resource can complete its processing in its own fashion, without central management, and so can operate as differently as it likes. The only condition is that it can request jobs from the MEOW system running on the central storage, and that it can produce output back into the system. Even this is not a formal requirement however, and it would be perfectly possible to have a series of completely independent storage locations each with their own MEOW system, such as is shown between the instrument and centralised storage. This would mean that even more differences could be accommodated such as FPGAs, or other dedicated hardware. However, such a system would become so decentralised that managing it and tracking both results and errors would become very arduous. Nevertheless, this style of heterogeneous analysis is very well suited to MEOW systems, and strikes me as the most apparent use case for future implementations.

The beginnings of such a system are already in place, as demonstrated by the scientific example in Chapter 15.6. However, proper support for communication between instruments and the MiG storage would assist greatly if a user was to create such a system. FUR goes some way to address this by creating a managed way of uploading large amounts of data, but it relies on appending to HDF5 files to be most efficient, which is currently unsuited to `watchdog` event management as every opening, read or write operation is read as a modification of the file and so triggers an unwanted cascade of job scheduling. Modifications to the event listener on the MiG would therefore be required for FUR to be considered integrate-able into such a system. We are also currently limited in what jobs can be scheduled using MEOW, with all Recipes being Jupyter Notebooks. This is currently sufficient for scientific analysis, but in a heterogeneous system such as the one presented here, a broader range of processing and job definition would be of great use. For these reasons, I conclude that though MEOW makes possible heterogeneous analysis, further work is still possible to increase the utility of such a system.

28

SUMMARY

This part has evaluated the work presented within this thesis. This was done by comparing it against the core research question. It has been demonstrated that it is indeed feasible to create a system for automatic dynamic workflow construction. This enables a whole host of new analysis structures that were either not possible before, or have been made considerably easier with this innovation. Most notably is the potential for more exploratory workflows, repeatable analysis or continuous system. Therefore we can state that this project is a success and has demonstrated the feasibility of such a system.

As a new SWMS, it was only appropriate that MEOW was rated against the stated requirements for such as system. This was as these are the direct competition for MEOW and so users will also be using these requirements in their judgements. In this regard MEOW was less successful. Although it met many of the goals, it had two shortcomings in there exists the possibility of race conditions within jobs as well as few limits on runaway job creation. Together these mean that MEOW is not yet fully complete as a SWMS. It is still capable of running scientific analysis in conjunction with the MiG or on its own in the `WorkflowRunner` in `mig_meow`, but could do with more development before it is considered robust enough for widespread adoption.

Part VI

CONCLUSION AND FUTURE WORK

29

FUTURE WORK

In this chapter we will consider some of the potential future work that could be carried out to continue the project. As well as the specific examples that will be considered here, there of course also exists the myriad of bugs and polishing that could take place on a large project such as this. These will not be considered as they are not seen as overly interesting.

29.1 EXPANDING MEOW DEFINITIONS

Perhaps the most obviously point of future work is to expand the definitions possible within MEOW, both for use within `mig_meow` and the MiG. Some way of defining environment requirements or the like would be of help. This should be added to the Recipe definitions as it is there that actual software requirements will be most applicable. For example, having some new property of a Recipe that listed the packages required by the Jupyter Notebook it is based off of would be of use.

Additionally, there are currently a number of job parameters used by the MiG that are not possible to edit for MEOW jobs. For example, in Appendix H we can see that properties such as CPUTIME, NODECOUNT and MAXFILL are always set to the same values when part of a MEOW job. Expanding the definitions of either Patterns or Recipes to allow for the additional definition of values such as these would be of use. This would allow a user to better tailor their jobs, so the MiG can better manage them.

This is perhaps the avenue for future development that would provide the most utility. It would not be overly difficult to achieve as many of the definitions stated already exist in various parts of the system, they just need some way of being declared in the front-end and matched to the back-end. Therefore, if any work is to continue on MEOW it is recommended first and foremost that at least some of these additions are made.

29.2 LIMITING INFINITE SCHEDULING

Perhaps the biggest area of future work is regarding the limitation of infinite scheduling. Currently there is no limit on how many jobs may be scheduled on a MEOW system, either at once or in aggregate. This can lead to swamping a system or unwanted infinite chains of processing. As outlined in Section 26.2.1 this was not totally unintentional, but unfortunately there was not the time to properly investigate a real solution to the problem. For MEOW to be deployed at scale this would need to be solved however. It is suspected that this may require considerable investigation to see which of the dynamic possibilities enabled by MEOW are unwanted behaviours to be closed off and what should be encouraged, but managed carefully.

Some possibilities for the sorts of limits to be put in place might include placing a lifetime on Patterns, so that they automatically delete after X hours or days. Even simple limits like being able to say that you do not expect a Pattern to be triggered more than Y times may be of considerable use here. More extensive options might also be adding more attributes to the Pattern themselves. For instance it may be that we could want to be more selective on what events are responded to such as by ignoring events at a path we have already responded to, or only scheduling jobs whilst some flag somewhere else on the system is set. These possibilities deserve to be investigated as potential scalable solutions to this infinite scheduling problem.

29.3 IMPROVING THE `WORKFLOWRUNNER`

Another avenue for fruitful future development is to improve the `WorkflowRunner`. In its current inception it was designed as a learning tool, and an illustration of how MEOW would work for those without access to the MiG. This means that it is quite simplistic in some regards, with no environment management or support for anything other than local execution. However, there is much potential for this to be expanded in future work to allow for a much more robust platform for scientific analysis. This would be of especial use in a heterogeneous system such as that described in Section 27.4. An improved `WorkflowRunner` would be able to run as a central scheduler for any sort of MEOW analysis in such a system. Therefore, this is another high priority for future development.

One particular avenue for development would be to enable provenance reporting within the `WorkflowRunner`. The solution outlined in Section 11.2 won't work as no SSH communication is taking place. It may be possible to adapt the `WorkflowRunner` so as to use SSH mounts of local workgroup directories, thereby enabling logging with `paramiko`. However, this is not anticipated to work as expected. This is due to some work carried out by me on the MiG that is was not otherwise relevant to this thesis. It was discovered that `watchdog` does not pick up events through symlinks as the underlying file system only generates events at their absolute paths. When locally mounting directories, it is expected that a similar issue may arise, and so some new solution will probably be needed.

29.4 INVESTIGATING AND SUPPORTING HDF5 AND FUR

During testing of `watchdog` and `h5py`, it was noted that merely the act of reading a HDF5 file would be interpreted by `watchdog` as a modification event. This is not desirable as any job scheduled on some data may then trigger further jobs just from reading in its own input. This matter should be addressed, especially as it would be of worth to be able to use FUR in conjunction with MEOW to properly support scientific analysis. Although the matter has not been fully investigated yet, it is suspected that a programming solution can be found within the current framework. For instance, currently both the MiG and the `WorkflowRunner` bundle together events at the same path that occur at a very similar time, to prevent multiple jobs being scheduled off of minute edits. A similar system could potentially be developed, so that the monitor itself distinguishes between read and write operations within HDF5 files. In addition, other popular meta-formats should also be investigated to see if this problem is universal or unique to HDF5.

29.5 UNIFYING LANGUAGE

Mostly, the language of MEOW is unified with words like Pattern, Recipe and Rule used consistently to refer to individual constructs within MEOW. The glaring exception to this is the use of the word workflow. As mentioned in Sections 4.4 and 20.3, the use of the word workflow was inherited from the systems that MEOW was designed in response to, but was found to be increasingly unhelpful for new learners.

Later materials deliberately avoid using workflow completely, if possible though this is somewhat undercut by its use in constructs such as the `WorkflowWidget` and `WorkflowRunner`. Most prominently, workflow also features in the acronym, MEOW. It would be of considerable help to new users to avoid confusion by removing the word workflow from MEOW. This is perhaps a much simpler task that the others presented here, but with much more indeterminate results. Therefore this would not be a task worth conducting on its own, but should be done in conjunction with something else.

29.6 A LIBRARY FOR TOMOGRAPHY IN MEOW

The main goal of MUMMERING was to make a unified tool for tomography. What we have provided is a platform for designing tomography analysis, with MEOW and the MiG. This does meet the needs of such a tool, but may not be the specific implementation that the project planners first envisioned.

If a specific tool for tomography was needed then one suggested format for this would be to collect a variety of tomography Jupyter Notebooks from ESRs and package them into a python library. These Jupyter Notebooks could be formatted so as to be directly importable as Recipes for analysis, and appropriate Patterns could also be included alongside them. New tomography users could then use this library of tomography analysis Patterns and Recipes to construct a MEOW system to process their own data. A mock-up of how this might look from a user perspective is presented in Listing 29.1.

```python
import mig_meow as meow
import tomo_meow as tomo

# Read in the Pattern definitions
p_segment = tomo.patterns.Segmentation(
    name='segment', input_dir='raw', output_dir='segmented')
p_analysis = tomo.patterns.Analysis(
    name='analysis', input_dir='segmented', output_dir='final')

# Setup dict of all Patterns
patterns = {
    'segment': p_segment,
    'analysis': p_analysis,}

# Setup dict of all Recipes
recipes = {
    'segmentation': tomo.recipes.Segmentation(name='segmentation'),
    'analysis': tomo.recipes.Analysis(name='analysis')}

# Start the local runner
runner = meow.WorkflowRunner('example', 1, patterns=patterns, recipes=recipes)
```

Listing 29.1: A mock-up of what a python library for tomography constructs in MEOW might look like.

In a real example, more specific implementations would be made than simply ‘analysis’, with each Recipe being an implementation of some specific algorithm or technique. A standardised format of data would need to be established between Recipes, to allow for such plug-and-play possibilities, though this is not an impossible task. By creating such a library, MUMMERING would finally have a specific tool for the analysis of tomography. It would also perhaps make it easier for new users, who could immediately plug-and-play different Patterns and Recipes as a way of learning either MEOW or tomography. This work would also be relatively easy to achieve, though perhaps time consuming to gather a wide range of suitable processing for Recipes. Each would also potentially have to be converted to an appropriate format, and some effort made to make them cross-compatible with one another. Therefore it is concluded that this work is feasible, but would not be quick.

30

CONCLUSION

This thesis has presented a body of work undertaken by myself as part of the MUMMERING project. The main goal was to create a dynamic way of processing tomography data in a shareable manner. To meet this goal, I developed MEOW as a framework for dynamic job scheduling. This would use file events as a catalyst in a live system, in contrast to static workflows which will often use a DAG or some other form of *a priori* definitions to identify jobs. The key advantage of doing this is that each job is completely independent of every other job, and so can be scheduled concurrently with any other, making good use of any available resources. In addition, it is trivial for jobs to fail or succeed and so we can schedule mass amounts of jobs without being concerned about one error scuppering the whole analysis. This also means that the output for each job can be completely optional, which only increases the dynamic nature of the system.

In order for users to create such a system the MEOW framework was outlined, with users creating Patterns and Recipes. These match events to processing, and define the processing itself respectively. As the inputs and outputs for Patterns overlap, they link together with a workflow becoming an emergent property of the system rather than a specific user definition. In order to actually define these constructs, a python package `mig_meow` was created. This can be used in isolation to run a local analysis system, but is primarily designed to interact with the MiG. A number of functions are provided within `mig_meow` to allow for the easy creation, reading, updating and deleting of MEOW constructs on the MiG. This can be somewhat complex for a user to understand and debug, and so a number of helper Jupyter Notebook widgets have been created. These provide features such as visualisations of the expected analysis from given Patterns and Recipes, as well as reporting on what jobs have been scheduled, and showing how those jobs link together in a provenance report.

In support of MEOW, a number of learning materials and documentation have been presented, along with some exploratory examples of how it might be used in a scientific context. Most prominently, MEOW would be of use as a system where continuous scheduling is necessary such as in a live monitoring system, or those with often repeated analysis. It is also well suited to exploratory workflow with unpredictable results. Finally I also think it would be a of use in heterogeneous analysis systems, with processing spread over many resources disparate in both architecture and timing. This is especially true as there is more demand for human-in-the-loop interactions in traditional SWMS. As it currently stands, MEOW is already suited to many of these applications, especially if used in conjunction with the MiG. However, there is still plenty of capacity for further work. There is much scope for the MEOW definitions to be expanded to make specific demands of the environment each job is to be run in. This could easily be integrated with the existing systems for doing this on the MiG, but would require a more bespoke solution for the `WorkflowRunner`. It would also be of significant help if the `WorkflowRunner` was expanded to be a robust analysis tool in its own right, perhaps by integrating `corc` or FUR into its use.

In conclusion this project has successfully demonstrated the feasibility of a tool for the automatic creation of dynamic scientific workflows. This is true both for tomography specifically as well as science more generally. It has prompted enough novel research to be published and presented at conferences, and has provided fruitful collaboration with a number of partners. Additional supporting work has also been completed, such as efficient uploading of research data to the MiG as well as demonstrating integration of cloud scheduling into existing applications. Several avenues of future work remain however, most notably in fully integrating these supporting works into either `mig_meow` or the MiG. In addition, time should be taken to address the potential shortcomings of few job scheduling controls and a potential race condition in MEOW. Despite these gaps, we can conclude that MEOW was successful as it can currently be used to automate the scheduling of scientific analysis in a novel, dynamic structure. This enables workflow structures that would be difficult to implement in an old, static system such as exploratory workflows, repeatable analysis, or heterogeneous systems.

Thank you for reading.

Part VII

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An Extensible System for Design and Execution of Scientific Workflows," in *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004*, IEEE, 2014.

[2] I. Altintas, J. Block, R. de Callafon, D. Crawl, C. Cowart, A. Gupta, M. Nguyen, H.-W. Braun, J. Schulze, M. Gollner, A. Trouve, and L. Smarr, "Towards an Integrated Cyberinfrastructure for Scalable Data-driven Monitoring, Dynamic Prediction and Resilience of Wildfires," *Procedia Computer Science*, vol. 51, pp. 1633–1642, 2015, International Conference On Computational Science, ICCS 2015, ISSN: 1877-0509.

[3] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, and et al., *Common workflow language, v1.0*, Jul. 2016. DOI: 10.6084/m9.figshare.3115156.v2. [Online]. Available: https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156/2.

[4] *Amazon Web Services*, https://aws.amazon.com/, 2021.

[5] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 25–36, ISBN: 9781450366700. DOI: 10.1145/3307681.3325400. [Online]. Available: https://doi.org/10.1145/3307681.3325400.

[6] R. M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, "Comp superscalar, an interoperable programming framework," *SoftwareX*, vol. 3-4, pp. 32–36, 2015, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2015.10.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711015000151.

[7] J. Bardino, M. Rehr, and B. Vinter, "Event-driven, collaborative and adaptive scientific workflows on the grid," English, in *Communicating Process Architectures 2017 and 2018, WoTUG-39 and WoTUG-40 - Proceedings of CPA 2017 (WoTUG-39) and Proceedings of CPA 2018 (WoTUG-40)*, J. Pedersen, K. Chalmers, J. Broenink, B. Vinter, K. Vella, P. Welch, M. Smith, and K. Skovhede, Eds., ser. Concurrent Systems Engineering Series, 39th WoTUG Conference on Communicating Process Architectures, CPA 2017 and 40th WoTUG Conference on Communicating Process Architectures, CPA 2018 ; Conference date: 19-08-2018 Through 22-08-2018, IMIA and IOS Press, 2019, pp. 61–80. DOI: 10.3233/978-1-61499-949-2-61.

[8] *Benchmarking raw results*, https://sid.idmc.dk/sharelink/a4K7ZbzOyy, 2022.

[9] K. Benedyczak, B. T. Schuller, M. Sayed, J. Rybicki, and R. Grunzke, "Unicore 7 — middleware services for distributed and federated computing," Jul. 2016, pp. 613–620. DOI: 10.1109/HPCSim.2016.7568392.

[10] J. Berthold, J. Bardino, and B. Vinter, "A principled approach to grid middleware: Status report on the minimum intrusion grid," in *Algorithms and Architectures for Parallel Processing*, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds., Springer, 2011, pp. 409–418.

[11] J. Bjørndalen, B. Vinter, and O. Anshus, "Pycsp - communicating sequential processes for python.," vol. 65, Jan. 2007, pp. 229–248.

[12] S. Bowers and B. Ludäscher, "Actor-oriented design of scientific workflows," vol. 3716, Oct. 2005, pp. 369–384, ISBN: 978-3-540-29389-7. DOI: 10.1007/11568322_24.

[13] *CERN*, https://home.cern, 2021.

[14] *CERN Data preservation*, https://home.cern/science/computing/data-preservation, 2021.

[15] *CERN Storage*, https://home.cern/science/computing/storage, 2021.

[16] *Cloud Technologies in Education*, https://cte.ccjournals.eu/cte2020/, 2020.

[17] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in condor," *Workflows for e-Science*, pp. 357–375, Jan. 2007. DOI: 10.1007/978-1-84628-757-2_22.

[18] *Creating an IPython Notebook programatically*, https://gist.github.com/fperez/9716279, 2016.

[19] *crontab*, https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html, 2018.

[20] *cwltool*, https://github.com/common-workflow-language/cwltool, 2021.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04, San Francisco, CA: USENIX Association, 2004, p. 10.

[22] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Pegasus: Planning for execution in grids," GriPhyN, Tech. Rep. Technical Report 2002-20, 2002. [Online]. Available: http://pegasus.isi.edu/publications/ewa/pegasus%5C_overview.pdf.

[23] J. Dias, G. Guerra, F. Rochinha, A. L. Coutinho, P. Valduriez, and M. Mattoso, "Data-centric iteration in dynamic workflows," *Future Generation Computer Systems*, vol. 46, pp. 114–126, 2015.

[24] H. Digabel and C. Lantuéjoul, "Iterative algorithms," in *Proc. 2nd European Symp. Quantitative Analysis of Microstructures in Material Science, Biology and Medicine*, Stuttgart, West Germany: Riederer Verlag, vol. 19, 1978, p. 8.

[25] *Docker*, https://www.docker.com/, 2021.

[26] D. Marchant, *MIG_MEOW DOCKER CONTAINER*, https://github.com/PatchOfScotland/docker-meow, 2022.

[27] D. Marchant, *MiG docker container*, https://github.com/PatchOfScotland/docker-migrid, 2022.

[28] D. Marchant, *Slurm docker container*, https://github.com/PatchOfScotland/docker-slurm, 2022.

[29] M. Dreher and T. Peterka, "Decaf: Decoupled dataflows for in situ high-performance workflows," Argonne National Laboratory, Lemont, IL, Tech. Rep. ANL/MCS-TM-371, Jul. 2017.

[30] *EduHPC-19*, https://tcpp.cs.gsu.edu/curriculum/?q=eduhpc19, 2019.

[31] *ESRF*, https://www.esrf.eu/, 2021.

[32] *Euro-Par 2021*, https://2021.euro-par.org/, 2021.

[33] *EuXFEL*, https://www.xfel.eu, 2021.

[34] *Example file data*, https://sid.idmc.dk/cgi-sid/ls.py?share_id=FocgdzkyBf, 2021.

[35] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Junior, and H.-L. Truong, "Askalon: A tool set for cluster and grid computing," *Concurrency and Computation: Practice and Experience*, vol. 17, Feb. 2005. DOI: 10.1002/cpe.929.

[36] J. E. Ferreira, Q. Wu, S. Malkowski, and C. Pu, "Towards flexible event-handling in workflows through data states," in *2010 6th World Congress on Services*, 2010, pp. 344–351. DOI: 10.1109/SERVICES.2010.60.

[37] I. Foster, “Globus Toolkit Version 4: Software for Service-Oriented Systems,” in *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag, 2006, pp. 2–13.

[38] *Google Cloud*, https://cloud.google.com/, 2021.

[39] D. Häfner, R. L. Jacobsen, C. Eden, M. R. B. Kristensen, M. Jochum, R. Nuterman, and B. Vinter, “Veros v0.1 – a fast and versatile ocean simulator in pure python,” *Geoscientific Model Development*, vol. 11, no. 8, pp. 3299–3312, 2018. DOI: 10.5194/gmd-11-3299-2018. [Online]. Available: https://gmd.copernicus.org/articles/11/3299/2018/.

[40] P. B. Hansen, *Operating system principles*. Prentice-Hall, Inc., 1973.

[41] B. P. Harenslak and J. R. de Ruiter, *Data Pipelines with Apache Airflow*, 1st ed. Manning, 2021, ISBN: 978-16-172-9690-1.

[42] *HDF5*, https://www.hdfgroup.org/solutions/hdf5/, 2021.

[43] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[44] T. Hodges and M. R. Crusoe, “Common workflow language user guide,” Aug. 2017, Based off of the excellent Software/Data Carpentry templates: https://github.com/swcarpentry/styles/. DOI: 10.5281/zenodo.840663.

[45] *ImageFilter*, https://pillow.readthedocs.io/en/5.1.x/reference/ImageFilter.html, 2022.

[46] *inotify(7) Linux manual page*, https://man7.org/linux/man-pages/man7/inotify.7.html, 2020.

[47] *ipywidgets documentation*, https://ipywidgets.readthedocs.io/en/latest/, 2021.

[48] *Project Jupyter*, https://jupyter.org/, 2021.

[49] H. Karlsen and B. Vinter, “Minimum intrusion grid - the simple model,” Jul. 2005, pp. 305–310, ISBN: 0-7695-2362-5. DOI: 10.1109/WETICE.2005.46.

[50] J. Kerridge, *Using Concurrency and Parallelism Effectively*, 2nd ed. Bookboon, 2014, ISBN: 978-87-403-1038-2.

[51] *Kubernetes*, https://kubernetes.io/, 2021.

[52] D. Marchant, C.-J. Johnsen, B. Vinter, and K. Skovhede, “Teaching concurrent and distributed programming with concepts over mathematical proofs,” English, in *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, IEEE, 2019. DOI: 10.1109/EduHPC49559.2019.00012.

[53] D. Marchant, R. Munk, and B. Vinter, “Developments in event-oriented, emergent workflows.”

[54] D. Marchant, R. Munk, B. Vinter, and E. Brenne, “Managing event oriented workflows,” English, Dec. 2020, pp. 23–28. DOI: 10.1109/XLOOP51963.2020.00009.

[55] J. M. Martin and P. H. Welch, “A design strategy for deadlock-free concurrent systems,” *Transputer Communications*, vol. 3, no. 4, pp. 215–232, 1997.

[56] M. Mattoso, J. Dias, K. A.C.S.Ocaña, E. Ogasawara, F. Costa, F. Horta, V. Silva, and D. de Oliviera, “Dynamic steering of HPC scientific workflows: A survey,” *Future Generation Computer Systems*, vol. 46, pp. 100–113, 2015.

[57] *MAX IV*, https://www.maxiv.lu.se, 2021.

[58] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher, “Scientific workflow design for mere mortals,” *Future Generation Computer Systems*, vol. 25, pp. 541–551, 2008.

[59] P. K. Willendrup, *McWeb*, https://github.com/McStasMcXtrace/McWeb, 2020.

[60] *Microsoft Azure*, https://azure.microsoft.com/en-us/, 2021.

[61] J. Bardino, M. Rehr, R. Munk, and D. Marchant, *MiG on SourceForge*, https://sourceforge.net/projects/migrid/, 2021.

[62] D. Marchant, *mig_meow on GitHub*, https://github.com/PatchOfScotland/mig_meow, 2022.

[63] D. Marchant, *mig_meow on PyPi*, https://pypi.org/project/mig-meow, 2021.

[64] R. Montella, D. Di Luccio, and S. Kosta, "DagOn*: Executing Direct Acyclic Graphs as Parallel Jobs on Anything," in *Proceedings of WORKS 2018: 13th Workshop on Workflows in Support of Large-Scale Science, Held in conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 64–73, ISBN: 9781728101965. DOI: 10.1109/WORKS.2018.00012.

[65] *multiprocessing*, https://docs.python.org/3/library/multiprocessing.html, 2021.

[66] *Mummering Project website*, http://www.mummering.eu, 2021.

[67] R. Munk and B. Vinter, "Mummering platform idea's & ubiquitous data analysis," 2018, pp. 323–333. DOI: 10.3233/978-1-61499-949-2-323.

[68] R. Munk, "Grid of clouds: A model for how resources can be shared amongst organisations," Ph.D. dissertation, Niels Bohr Institute, University of Copenhagen, Feb. 2021.

[69] R. Munk, D. Marchant, and B. Vinter, "Cloud enabling educational platforms with corc," in *Proceedings of the 8th Workshop on Cloud Technologies in Education (CTE 2020)*, 2020, ISBN: 0000000239479.

[70] *nbformat*, https://github.com/jupyter/nbformat, 2021.

[71] *notebook-parameterizer*, https://pypi.org/project/notebook-parameterizer/, 2021.

[72] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[73] *Oracle Cloud Infrastructure*, https://www.oracle.com/cloud/, 2021.

[74] *Oracle Grid Engine*, https://www.oracle.com/technetwork/oem/host-server-mgmt/twp-gridengine-overview-167117.pdf, 2010.

[75] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE transactions on systems, man, and cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[76] *papermill*, https://github.com/nteract/papermill, 2021.

[77] *paramiko*, http://www.paramiko.org/, 2020.

[78] D. M. Pelt, K. J. Batenburg, and J. A. Sethian, "Improving tomographic reconstruction from limited data using mixed-scale dense convolutional neural networks," *Journal of Imaging*, vol. 4, no. 11, p. 128, 2018.

[79] *Pillow*, https://pillow.readthedocs.io/en/stable/, 2021.

[80] C. Ramon-Cortes, F. Lordan, J. Ejarque, and R. M. Badia, "A programming model for hybrid workflows: Combining task-based workflows and dataflows all-in-one," *Future Generation Computer Systems*, vol. 113, pp. 281–297, Dec. 2020, ISSN: 0167-739X. DOI: 10.1016/j.future.2020.07.007. [Online]. Available: http://dx.doi.org/10.1016/j.future.2020.07.007.

[81] S. Rinderle, M. Reichert, and P. Dadam, "Correctness criteria for dynamic changes in workflow systems—a survey," *Data & Knowledge Engineering*, vol. 50, no. 1, pp. 9–34, 2004, Advances in business process management, ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2004.01.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169023X04000035.

[82] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, Jan. 2015, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013.

[83] R. Rosca and H. Fangohr. (). "Jupyter for reproducible science at photon and neutron facilities," PaNOSC - Photon and Neutron Open Science Cloud, [Online]. Available: https://www.panosc.eu/wp-content/uploads/2019/07/PaNOSC_20190611-13_ROSCA_Berkeley_JupiterForScience.pdf.

[84] *SC19*, https://sc19.supercomputing.org/, 2019.

[85] *SC20*, https://sc20.supercomputing.org/, 2020.

[86] F. Simancik, J. Jerz, J. Kovacik, and P. Minar, "Aluminium foam-a new light-weight structural material," *METALLIC MATERIALS*, vol. 35, pp. 187–194, 1997.

[87] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "PyCOMPSs: Parallel computational workflows in Python," *International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017, ISSN: 17412846. DOI: 10.1177/1094342015594678.

[88] C. The MPI Forum, "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93, Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883, ISBN: 0818643404. DOI: 10.1145/169627.169855. [Online]. Available: https://doi.org/10.1145/169627.169855.

[89] R. Tolosana-Calasanz, J. Bañares, P. Álvarez, and J. Ezpeleta, "On interlinking of grids: A proposal for improving the flexibility of grid service interactions," Jul. 2008, pp. 714–720, ISBN: 978-0-7695-3163-2. DOI: 10.1109/ICIW.2008.39.

[90] *Torque Resource Manager*, https://adaptivecomputing.com/cherry-services/torque-resource-manager/.

[91] W. Van Aarle, W. J. Palenstijn, J. De Beenhouwer, T. Altantzis, S. Bals, K. J. Batenburg, and J. Sijbers, "The astra toolbox: A platform for advanced algorithm development in electron tomography," *Ultramicroscopy*, vol. 157, pp. 35–47, 2015.

[92] N. A. T. Voetmann, "Framework for Uploading Research data (FUR)," M.S. thesis, Niels Bohr Institute, University of Copenhagen, Mar. 2021. [Online]. Available: https://sid.idmc.dk/sharelink/hZjJ5NSvlb.

[93] *watchdog*, https://pypi.org/project/watchdog/, 2021.

[94] P. H. Welch, G. R. Justo, and C. J. Willcock, "Higher-level paradigms for deadlock-free high-performance systems," in *Transputer Applications and Systems"93, Proceedings of the 1993 World Transputer Congress*, IOS Press, Netherlands, vol. 2, 1993, pp. 981–1004.

[95] J. M. Perkel, *Why Jupyter is data scientists' computational notebook of choice*, https://www.nature.com/articles/d41586-018-07196-1, Oct. 2018.

[96] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?" *Commun. ACM*, vol. 20, pp. 822–823, 1977.

[97] *Workload Manager*, https://www.ibm.com/docs/en/aix/7.2?topic=management-workload-manager, 2021.

[98] I. T. Union, *X.509 technical specification*, https://www.itu.int/rec/T-REC-X.509, 2019.

[99] *XLOOP 20*, https://wordpress.cels.anl.gov/xloop-2020/about-xloop/, 2020.

[100] *Xnovotech*, https://xnovotech.com/, 2021.

[101] *The Official YAML Web Site*, https://yaml.org/, 2011.

[102] O. Yildiz, J. Ejarque, H. Chan, S. Sankaranarayanan, R. M. Badia, and T. Peterka, "Heterogeneous hierarchical workflow composition," *Computing in Science Engineering*, vol. 21, no. 4, pp. 76–86, 2019. DOI: 10.1109/MCSE.2019.2918766.

[103] A. Yoo, M. Jetter, and M. Grondona, "Slurm: Simple linux utility for resource management," *Lecture Notes in Computer Science*, vol. 2862, pp. 44–60, 2015.

[104] Y. Zhao, I. Raicu, and I. Foster, "Scientific workflow systems for the 21st century, new bottle or new wine?" In *Proceedings of the 2008 IEEE Congress on Services - Part I, SERVICES '08*, Washington, DC, USA, 2008, pp. 467–471.

Part VIII

APPENDICES

A

MANAGING EVENT ORIENTED WORKFLOWS

This paper, and a presentation describing it were originally presented at the 2nd Annual Workshop on Extreme-Scale Experiment-in-the-Loop Computing[99], part of SC20: The International Conference for High Performance Computing, Networking, Storage and Analysis[85]. It was first published as part of the proceedings[54].

Managing Event Oriented Workflows[⋆]

David Marchant[1][0000−0003−4262−7138], Rasmus Munk[1][0000−0003−0333−4295], Elise O. Brenne[2][0000−0003−2424−4117], and Brian Vinter[1][0000−0002−3947−9878]

[1] Niels Bohr Institute, University of Copenhagen, Blegdamsvej 17, 2100 Copenhagen, Denmark {david.marchant, rasmus.munk, vinter}@nbi.ku.dk

[2] Department of Energy Conversion and Storage, Technical University of Denmark, Fysikvej, 2800 Kgs. Lyngby, Denmark elbre@dtu.dk

Abstract. This paper introduces an event-driven solution for modern scientific workflows. This approach enables truly dynamic workflows by splitting workflows into their constituent parts and expressing them using combinations of Patterns and Recipes. This system is implemented on the MiG and examples are presented. Different users use cases are considered to asses the feasibility of the design, and it is found to be sufficient. Finally, further developments are considered with particular attention put towards more user friendly providence reporting and parameter sweeping.

Keywords: workflow, event-driven, adaptive, dynamic, Jupyter, MiG

1 Introduction

Scientific workflows systems are an essential part of modern research. They are used to process large datasets on numerous heterogeneous resources. Traditional scientific workflow systems adopt a so-called static structure, where all processing, data and outcomes are set at the beginning of the workflow. This is sufficient for a number of use cases, but some have identified a need for a dynamic structure to their workflows, where the outcomes of any processing are unknown at the start of the workflow [17].

To address this, this paper proposes MEOW, a system for defining event-driven workflows. This is done by splitting workflows into separate Patterns and Recipes. These can be defined by users, with the workflow emerging from several overlapping Patterns. The paper also describes a prototype implementation of a MEOW based event-driven workflow scheduler on the Minimum Intrusion Grid (MiG) [7, 8]. This dynamic workflow environment enables new workflow structures, impossible in a static system such as optional outputs or cyclic execution of processes. Furthermore, this system is suitable for a whole range of users, right down to HPC amateurs and so could be applied to a wide range of use cases.

[⋆] This project has received funding from the European Unions Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No 765604.

This project is developed as part of the MUMMERING[3] research project with the aim of providing researchers with a generic tool of the management and processing of scientific sets of data.

2 Background

2.1 Scientific Workflows

To explore the growing collection of scientific data, the use of automated analysis techniques is required. To meet the needs of scientists, workflows as a concept have been adapted from the more 'traditional' models used within business. The specific implementations of workflow systems designed for scientists are referred to as 'scientific workflows' with examples such as Kepler [4], Pegasus [10], Taverna [19], and Globus Workflow [15], among others. These efforts usually employ a top-down model when defining how the set of tasks should be processed, which in turn produces workflows that defines an entire chain of steps through a simple scripting language, a textual based graph, or a mathematical based definition [11]. The resulting jobs are then scheduled automatically. However, for tasks such as exploring large datasets [12], it is unlikely that the initial defined assumptions will produce a successful result without having first explored several workflow permutations. In this situation, the workflow execution would benefit from being able to make adaptions during runtime and make task scheduling decisions based on the intermediate results of the experiment at hand while also allowing human-in-the-loop interactions during the initial analysis and iterations of the workflow. [17]

Once a worklfow is activated, typically it will take the defined tasks and chain them together for subsequent scheduling. The activation itself is normally conducted in a data, control or hybrid driven manner. The choice among these depends on the implementation of the engine itself. In the area of scientific workflow systems, the approach has most often been to schedule workflows in a data-driven/data-flow manner, the reason being that in most scientific experiments suitable for workflow execution the task often involves some complex chain of data processing and transformation tasks which naturally maps themselves to a data-driven model.

Beyond the ability to process workflows, the historically successful scientific workflow systems have also provided a wide range of additional features to complement the workflow. These are often data management of the input and output generated from the workflows, task failure management, integration with grid and cloud resources for distributed task scheduling, and enabling collaboration between users across data management and workflow features [17, 27].

[3] *MU*ltiscale, *M*ultimodal, and *M*ultidimensional imaging for *E*nginee*RING* (MUMMERING)

2.2 Static and Dynamic Workflows

Scientific workflows tend to be data oriented and exploratory in nature [9]. This is as scientists are often running workflows as part of their experiments, and so may be running the same workflow repeatedly to fully explore an experiment space. The nature of these repeated runs may be unknown at the start of a scientists investigation, unlike in more traditional business workflows where it is expected that a user will already have a full understanding of the workflow and the work to be carried out. This means that scientific workflows have a specific need to be dynamic, as identified by [9,12,16,17], amongst others. Note that for a workflow to be dynamic it must be alterable at runtime. This could mean that parameters within a job can be changed, jobs can be cancelled, or any other user interaction should be possible without having to restart the whole workflow.

Many recent scientific workflow developments such as Dask [22], PyCOMPs [24], and DagOn [18] employ a data-driven model for their workflows. This has led them to a common way of representing the overall workflow, namely as a static Directed Acyclic Graph (DAG) [14]. Static in this regard means that, once the workflow is defined it is set and not subject to change during runtime execution, which obviously does not address the need for scientific workflows to be dynamic. As highlighted by [9, 12], this possesses several constraints when the experiment at hand involves the exploration of large datasets. Several workflow systems currently exist that allow for varying degrees of runtime adaption. For example DVega [25], which allows for exception handling in individual workflow steps. These exceptions are caught by the workflow system, which can recover by replacing the step with a new one. Other solutions in other workflow systems also use a similar method for workflow alteration at runtime, with steps replaced or rescheduled on different resources. However, these adaptions require the user to define the possible changes ahead of time and so are limited to expected outcomes. It is suspected that this limitation comes from workflows still being defined in a static paradigm, with adaptions applied later within a model that does not sufficiently support such changes. Consider the work of workflow standards such as CWL [6], which has no inbuilt functionality for defining adaptability within a workflow, even though its ambition is to be the universal standard for expressing scientific workflows. A different approach would be to design workflows from the ground up to be dynamic. The proposed MEOW system is intended as such a system.

2.3 Scientific Workflow Requirements

Whatever type of workflow manager is implemented, it must adhere to the requirements of any scientific workflow system (SWFS) as outlined in [28]. Zhao, Raicu and Foster describe a SWFS as having four defining characteristics. Firstly it should be possible within the system to implement complicated data analysis according to the requirements of a given researcher. Secondly, the system will make possible the automation of data processing tasks. Thirdly, the system will utilize parallel computation to improve the workflows performance regarding throughput and execution time. Lastly, the system should provide a capability to trace data through the system, with metadata being maintained about any given result to show exactly where it came from and what processing has taken place upon it. These four requirements sufficiently describe all SWFS's and the functionality expected from them. These shall be used as the base requirements for the proposed MEOW system presented within this paper.

3 The Proposed Solution

3.1 Designing MEOW

To create an event-driven workflow we will adopt the bottom up approach first demonstrated in [7]. This means that users define individual components and a workflow becomes an emergent property of the system, rather than being explicitly defined by the user. This bottom up, event-driven approach shall be referred to as Managing Event-Oriented Workflows, or MEOW for short.

To define what MEOW means it is necessary to break a workflow down into its constituent parts, which can then individually be defined by a user. Workflows are made up of several steps, with each step being a self contained block of processing. An example workflow is shown in Figure 1. This workflow is typical in that it has several steps, shown as the light grey boxes, each representing some processing. Data is passed from step to step, as is shown by the linking arrows.

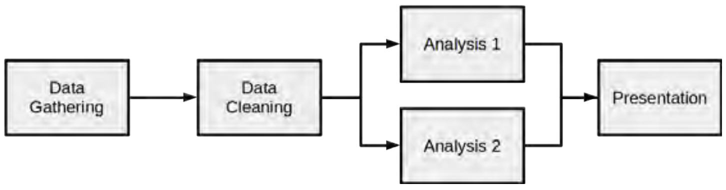


Fig. 1: A sample workflow showing various processing steps some data would undergo. In this diagram the focus is on the processing, with the data being passed from step to step.

Workflows are usually shown in this manner, but could also be expressed with each step representing the data, and the arrows showing the processing happening on said data. This can be seen in Figure 2 and is usually referred to as a dataflow, due to its focus on how the data is used between steps. Both Figure 1 and 2 show the same workflow, but the data-centric method on display in Figure 2 is better suited to an event-driven model. This is as the driving events for such as system will be data being *created* or *modified* rather than processes starting or finishing.

This leads us to what it actually is that user will define, namely *Patterns* and *Recipes*. Patterns are, at their most basic, a description of what events should

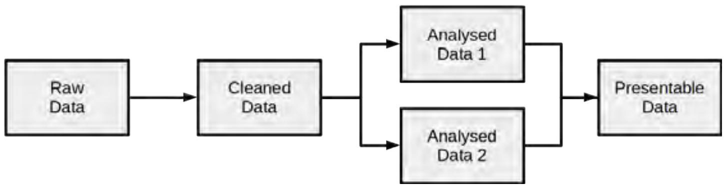


Fig. 2: A sample dataflow showing various states some data would exist in as it undergoes various processing. In this diagram the focus is on the data, with it being processed by different functions in turn.

result in the processing of a Recipe. This event description can be as broad or specific as the user requires, so may define a specific file path, or may be broad enough to cover any instance of a certain file type. The processing that is triggered by a Pattern match is defined in a Recipe. A Recipe should (but is not required to) take some input data, perform some set of instructions, and then should (but is not required to) produce some output. As well as the event description, a Pattern must also declare a Recipe as the processing that shall be triggered in the event of a match. Taken together, a Pattern and Recipe define one 'step' of a workflow. As a user defines multiple Patterns and Recipes, a workflow emerges from the collection of individual steps. Note that Patterns are unique, in that each one will correspond to one step in a workflow, but that Recipes may be shared by multiple Patterns.

The advantage of breaking down the workflows into these Patterns and Recipes is that each step of the workflow is now completely independent. This contrasts with traditional workflows, where the entire workflow is processed together[4]. As a result, individual steps could be said to lack meaningful interdependencies and are scheduled and completed in isolation. This isolation is what enables the emergent workflow to be completely dynamic, as any Pattern or Recipe may be changed at any point, and so any job at any stage can be changed, cancelled or added. As each step is completely independent this can be done without limitation, and regardless of what other jobs have already been scheduled, completed or ignored.

3.2 MEOW Requirements

To properly define a MEOW system, the following definitions for Patterns and Recipes have been constructed. These would be the minimum that would need to be defined in a Pattern or Recipe by a user for them to create a functioning system as described. For a Recipe the requirements are:

– **Name**: This is the identifier of the Recipe. It is used by Patterns to identify the linked Recipe, and by the implementation to keep track of changes to an already registered Recipe. The name must be unique.

[4] This may in practice be done in several batches of processing running in parallel, or sequentially. It is nevertheless defined as one holistic system

- **Recipe**: The processing code itself. This is the core job code that contains, for example, a user's analysis algorithm. It may rely on input data or variables, provided by a Pattern.

The Recipe requirements are currently fairly light and are open to being expanded in future. For instance, it may make more sense for a Recipe to define what variables or input it needs, rather than simply hoping that a Pattern provides the expected variables. This is not currently a requirement as it is not necessary within the current implementation. The requirements for a Pattern are:

- **Name**: This is the identifier of the Pattern. Must be unique.
- **Triggering Event**: This is an event description, used to match against system events. In case of a match then a job should be scheduled according to the definition of the Pattern. This job will take the file creating the triggering event as input, along with other defined variables.
- **Outputs**: A list of any and all data to be retrieved from the job as output. These outputs may not always be produced, but all possible outputs should be listed.
- **Recipe**: The name of a Recipe, used to define the processing taking place in a job.
- **Variables**: A set of variables to be passed to the Recipe by this Pattern at job creation. These could be any data structure understood by the Recipe and may include additional input files or possible output locations.

As in the case of the Recipe, these requirements may expand as the implementation is further explored. Further requirements must be met for a MEOW based scheduler to be truly event-driven. These are:

- **New Recipes must seek existing Patterns.** When the system registers a Recipe, the system must check to see if any already registered Patterns have stated this Recipe as their target. If any Patterns have stated the Recipe, then they are linked to the Recipe.
- **New Patterns must seek existing Recipes.** When the system registers a new Pattern, the system must check to see if the Patterns stated Recipe is already registered in the system. If the Recipe is present, the Pattern is linked to it.
- **Patterns and Recipes that are linked, must create a trigger.** If any links from Patterns to Recipes are established, then the system must create an appropriate event trigger.
- **New triggers must be able to apply retroactively.** If a new trigger is created, it must be able to check within the system, would any already existing files activate the trigger, were they created now. If any appropriate files are present they must be treated as though they were just created and so activate the trigger.
- **Deleting a Pattern or Recipe must be able to remove any associated trigger.** If a Pattern or Recipe is deleted, then any triggers created from it must be immediately deleted.

– **Deleting a trigger must be able to cancel any associated jobs.** If a trigger is ever deleted then any jobs that were scheduled as a result of activating that trigger should be cancelled. This is not strictly necessary but without it the system may quickly bloat with jobs that could now have been superseded.
– **Updating a Pattern or Recipe will delete the old version and create a new one.** If a Pattern/Recipe is ever updated then it should be processed in the system by the existing Pattern/Recipe being deleted, and a new one being registered as though it was created for the first time.

The presented requirements, when taken together ensure that the resulting workflow is adaptive to changes in Patterns, Recipes, and the underlying data files. It ensures that jobs will be rescheduled if a Pattern or Recipe is updated, and that no data is missed. If each requirement is implemented correctly then a workflow will, by necessity, be an emergent property of Pattern and Recipe definitions.

4 Implementing MEOW within MiG

4.1 MiG

To both test and demonstrate a MEOW system in action, an implementation was developed on the MiG [8]. This is a take on providing a middleware system for a set of distributed compute resources. It does this while requiring only a minimum amount of integration or configuration of the committed grid resources to join the overall organisation, thereby seeking to eliminate much of the failures of previous grid developments due to their complex and high administrative dependent architectures. Because the MiG is already a mature system it will act as a good basis for the MEOW workflow implementation. A distinct attribute to the MiG system is that the committed grid resources are mostly manged independently of the managing grid server, for instance when a job is submitted by a user to the managing grid server, the grid resources themselves are responsible for de-queuing and accepting a job for processing. A managing grid server will therefore never dictate a resource to process a job or conduct a similar task, it simply makes them available for a suitable resource to process.

The heart of the MiG system from a user's perspective, is the concept of Virtual Grids (VGrids). These are a super-set of a typical grid organisation. The grid organisation and its internal permission constraints are defined by the creators themselves. This is an invaluable tool across many research projects at the University of Copenhagen, both for sharing data but also to provide cross organisational collaboration at scale.

The ability to provide runtime adaptable workflows was recently also introduced in [7]. It introduced this as a part of a generic event-driven workflow system that defines an alternative to the DAG workflow model. It does this by not defining the complete set of workflow tasks in its entirety as a graph, but instead adopts a bottom up perspective on the individual tasks. The workflow tasks are instead defined as independent and disjointed pieces of work that can be adapted and managed at runtime.

While this notion is suitable for providing runtime adaptability, the initial implementation suffers from being highly complex in terms of being able to define individual trigger rules correctly and organising these independent rules to produce fully fledged workflows. For examples, and a fuller explanation of how event management works on the MiG please consult [7].

4.2 mig_meow: A Package for Defining MEOW Workflows

A Python package was developed allowing users to define Patterns and Recipes, called mig_meow [1]. This package contains the definition of a Pattern object, along with a number of helper functions for defining event paths, Recipes, outputs[5], and variables. It is primarily intended to be used within a Jupyter Notebook [2]. This is as they are already commonly used in scientific computing, and hopefully can offer a more user-friendly and maintainable interface than using a web based or custom made application.

Although it is possible to construct Patterns programmatically, mig_meow provides a Jupyter Notebook widget which also allows for Pattern and Recipe construction. This widget is intended as the primary method for users to do so, and also provides a visualisation of the emergent workflow from the defined Patterns and Recipes. This is especially important due to the separated nature of the individual workflow steps, as they are not definitionally linked like in a traditional workflow. Having a method of checking that the outputs of one Pattern leads into the inputs of another Pattern is therefore helpful. Each defined input and output location is attached to a Pattern via arrows, showing the path along which data is processed. Where these inputs and outputs overlap they will point to the same location, such as in Figure 3.

4.3 mig_meow on the MiG

Recipes are defined by Jupyter Notebooks. All they require according to the specification is a name and some code, which Notebooks already have with their filename, and the code cells within them. Parameters can be passed to a Recipe by a Pattern allowing the same Recipe to be used by multiple jobs/Patterns with different results. Patterns are modelled as objects using mig_meow, as it is hoped that objects should be at least passingly familiar to anyone using this system. Objects also make it easier to define a number of functions that could correctly configure the Pattern variables so as to comply with the requirements set out in Section 3.2, and to accommodate different input types without appearing too confusing to the user. When Patterns and Recipes are created using mig_meow,

[5] Note that a Patterns outputs can be defined in mig_meow, despite this not being necessary according to MEOW. These do not have any effect other than aiding workflow visualisation, and actual outputs are not limited to those defined, nor are the defined outputs expected.

they can be exported to a VGrid, whereupon the necessary triggers are created, updated or deleted as appropriate and job scheduling can begin.

5 A Fully Worked Example

To illustrate MEOW in use, an example workflow is presented here. This example examines the size and distribution of the pores within an artificial dataset representing 3D X-ray computed tomography (CT) data of 100 samples of aluminium foam [23]. The goal is to analyze the pore radius distribution in all samples. Some samples have very few pores, or very large pores. We want to discard these samples and exclude them from the final analysis. This can be done effectively in a dynamic system, meaning we can setup the whole workflow at once and not need to pre-sort our data sets as would often be required in a static system.

The main steps in the image processing workflow are as follows. Firstly, we need to segment the data, i.e. label the image voxels according to the two phases present; aluminium and air. Secondly, we identify the individual pores and estimate their radii. This is a time consuming process, so before we attempt these two steps we can perform an initial check to ensure that the porosity is within the desired range. This will exclude defect samples from the time consuming analysis.

The Recipes: The following items of processing were required before any workflow could be constructed. Each was written as a Jupyter Notebook and registered under the given name. Recipe code is available in [3].

- **Recipe 'porosity_check' linked to Pattern 'initial_porosity_check':** A two-component Gaussian Mixture Model is fitted to a small sample (around 1 %) of the intensity data, providing a rough idea of the air-to-aluminium ratio through the model component weights.
- **Recipe 'segmentation' linked to Pattern 'segment_foam_data':** In the first step of the segmentation process, noise is reduced using a Gaussian filter. The filter kernel size is defined as a variable whose value is set in the Pattern. Thereafter, the image is segmented using Otsu thresholding [20]. Finally, a morphological closing operation is performed to remove possible remaining single-voxel noise.
- **Recipe 'pore_analysis' linked to Pattern 'foam_pore_analysis':** To investigate the pore size distribution, the individual pores are identified using the watershed algorithm [13] with local peaks in a distance transform of the segmented data as seeds.

The Foam Analysis Workflow: The final implementation of the workflow is illustrated in Figure 3. Each of the three created Patterns show in green circles. Each Pattern has as an input path a file type in a directory, as shown by the white rounded rectangles with arrows pointing to the Pattern. Any optional output locations are shown with the arrows leading out of the Pattern. Each Pattern specifies the corresponding Recipe, as stated in the previous paragraph.

The 100 artificial CT datasets to be analysed were generated using the Python package foam_ct_phantom [21] and the ASTRA toolbox [26]. 20 phantoms were generated with insufficient porosity compared to the remaining 80 phantoms. To start processing using the workflow, these datasets were uploaded to the 'foam_ct_data' directory in the '.npy' NumPy array format. This triggers the first Pattern, 'initial_porosity_check'. The Recipe linked to this Pattern classifies each dataset as either accepted or discarded depending on some predefined porosity threshold, and accordingly, a text file with the dataset filename is created in one of the directories 'foam_ct_data_accepted' or 'foam_ct_data_discarded'. This was done to avoid copying the whole dataset needlessly, as this would result in gigabytes of additional space being used up. The creation of each text file triggers the next Pattern, 'segment_foam_data'. The segmentation method described in the linked Recipe is applied to the accepted datasets and the result stored in the directory 'foam_ct_data_segmented'. Finally, the pore analysis is performed on the segmented data, producing the final plots stored in the directory 'foam_ct_data_pore_statistics' as determined by the 'foam_pore_analysis' Pattern.

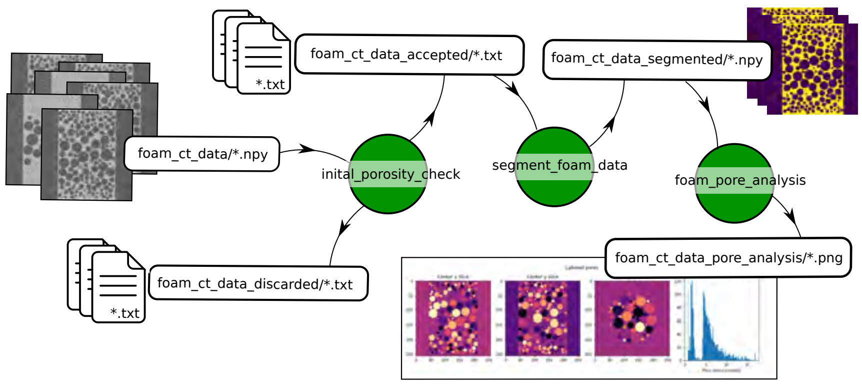

*.txt
foam_ct_data_accepted/*.txt
foam_ct_data_segmented/*.npy
foam_ct_data/*.npy
inital_porosity_check
segment_foam_data
foam_pore_analysis
foam_ct_data_pore_analysis/*.png
*.txt
foam_ct_data_discarded/*.txt

Fig. 3: The foam analysis workflow. Note that this image is based on the visualisation described in Section 4.2, but additional file images have been added to make the data state clearer at the different stages.

Suitability: The toy workflow presented here demonstrates how an event-driven workflow may be constructed, and a scenario in which it would be advantageous to do so. This is as we can setup one continuous workflow without any prior sorting of the data. In addition, individual job scheduling is completely separated so one dataset may be sorted, segmented and analysed before all datasets have been sorted. This could lead to more efficient use of concurrent hardware setups, as we don't need to wait for a group of jobs to finish before starting the next ones.

6 Users and Use Cases

When using MEOW to define a dynamic workflow rather than a static workflow system, different design paradigms are possible. Some possibilities and use cases are presented here as inspiration. These are not intended as improvements per se over any other methodology, but merely as possibilities within MEOW that are either not possible in other systems or very difficult to achieve.

An event-driven, dynamic workflow system such as MEOW is ideally suited to repeatable jobs, especially if they have unpredictable outputs. This could be extremely useful for facilities such as CERN, MAX IV or EuXFEL, where external users can book experiment time. These users are often an eclectic mix of specialists in their own field but may have little to no technical understanding of HPC. Some of these user's only concern is with the final produced data on which they shall perform their analysis, and they are largely unconcerned with any data cleanup or such like that must be performed on the raw data produced by the experiment. This could be scheduled automatically by a dynamic system, so that even 'novice' users can use a MEOW workflow if a more experienced user has set up the appropriate Patterns.

Similarly, a dynamic workflow would be suited to continuous monitoring systems such as WIFIRE [5], a system for simulating, monitoring and predicting wildfires in southern California. WIFIRE uses Kepler [4], a static workflow system to create workflows where heterogeneous data from cameras, satellites, weather stations and previous data sets is used to predict/simulate fire risks. Notably, the output state of one run can be used as input for the next run so as to generate an updating and continuous fire risk simulation. These data events could be used as trigger in a dynamic, continuous workflow where the same data is processed repeatedly by the same Pattern(s). These cycles of processing could theoretically go on forever if a continuous monitoring system was desired, or could continue until a certain threshold was reached. In either case, it would be much more difficult to set up a system like this using static workflows. In addition, if additional sensors are brought online, or existing ones removed, the workflow can be modified at runtime without the system needing extensive downtime for modification.

Most notably, a dynamic structure allows for optional branching within the workflow, such as in presented in Section 5. As each step is meaningfully independent of each other, there is no requirement for any particular step to produce any particular output, which is not easily possible in a static workflow system where step dependencies are often explicit. This means that adaptive workflows are particularly suited to a dynamic model.

All of these use cases are nothing new, as automation in workflows already exists in plenty of static systems. What is unique in a dynamic system is that it is possible to automate even when the outcome is unknown. For instance, it would be possible to create some automatic system that checks all raw data from a machine and could calibrate, sort or discard data based on some analysis. This data can then trigger further analysis based on the outcome to the initial processing.

7 Future Work

The most pressing point for further improvement in the MiG implementation of MEOW is proper providence support. This is currently possible as each job scheduled on the MiG produces individual feedback at job completion, but there is currently no way to view this easily, nor is there a way to get an overview of several jobs at once. Some sort of final report would be ideal, where a user can get an overview of what processing has been scheduled according to the defined Patterns and Recipes, and what output has been produced. Due to the nature of an event-driven system this would not be a 'final' report produced at the end of the workflow, as there is no end to an event-driven system where new events can occur at any time. Instead having a button, or way of triggering a report of all processing up to that point would suffice.

Parameter sweeping within Patterns is a feature scheduled for further development. By this it is meant that from a given input it should be possible to schedule a number of jobs with each given a different value for some parameter within a range. This would be ideal for initial exploration of datasets/analysis techniques, especially in a dynamic system were it is not necessary that each job successfully completes for the workflow to do so.

As it currently stands, the MEOW definition is robust, but limited. It does not mandate any sort of environment requirements and so could be expanded in future to do so. It also acts on the assumption that one file event will lead to one job, or many jobs once parameter sweeping is implemented. The converse of this may also be desirable, having multi-input jobs that only trigger once several files are triggered. This could be especially helpful for final analysis or the reduce segment of a map-reduce structured workflow. It is possible to achieve this currently through cunning use of Patterns and Recipes, but an explicit and simple way of doing so would be a benefit. There are of course also innumerable usability tweaks and fixes to make as is expected in a prototype system.

8 Conclusions

This paper has proposed one possible solution to some of the problems of modern scientific workflows. Unlike static workflows that require all steps to be defined at the very beginning, MEOW is proposed as a bottom up approach which breaks workflows down into Patterns and Recipes. This allows for an event-driven workflow system, which is fully dynamic at runtime. This enables a whole host of new ways of thinking about scientific workflows and how they are structured.

A demonstration system was implemented and described working in conjunction with the MiG. An example dynamic workflow using mig_meow was presented. MEOW was then considered in light of the needs of four hypothetical users and whether the proposed implementation is capable of meeting their needs for an event-driven, collaborative workflow system. The design repercussion of MEOW were also considered, with it being of particular note that the event-driven nature enables workflow structures for which traditional static workflow systems are unsuited. For example, optional branching and cyclic workflows are possible within this system.

Further work is expected on increasing the usability of mig_meow, most notably with better providence reporting. This work is worth continuing as it presents scientists with a dynamic paradigm for workflow construction, enabling new ways of interacting with and exploring even extremely large datasets.

References

1. mig_meow. https://pypi.org/project/mig-meow (2019)
2. Project Jupyter. http://jupyter.org (2019)
3. mig_meow code examples. https://sid.idmc.dk/sharelink/CJv3sw1fp2 (2020)
4. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: An Extensible System for Design and Execution of Scientific Workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. IEEE (2014)
5. Altintas, I., Block, J., de Callafon, R., Crawl, D., Cowart, C., Gupta, A., Nguyen, M., Braun, H.W., Schulze, J., Gollner, M., Trouve, A., Smarr, L.: Towards an Integrated Cyberinfrastructure for Scalable Data-driven Monitoring, Dynamic Prediction and Resilience of Wildfires. Procedia Computer Science **51**, 1633 – 1642 (2015). https://doi.org/https://doi.org/10.1016/j.procs.2015.05.296, http://www.sciencedirect.com/science/article/pii/S1877050915011047, international Conference On Computational Science, ICCS 2015
6. Amstutz, P., Crusoe, M.R., Tijani, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Mnager, H., Nedeljkovich, M., et al.: Common workflow language, v1.0 (Jul 2016). https://doi.org/10.6084/m9.figshare.3115156.v2, https://tinyurl.com/sqhzosy
7. Bardino, J., Rehr, M., Vinter, B.: Event-driven, collaborative and adaptive scientific workflows on the grid. Communicating Process Architecture pp. 59–78 (2017)
8. Berthold, J., Bardino, J., Vinter, B.: A principled approach to grid middleware: Status report on the minimum intrusion grid. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) Algorithms and Architectures for Parallel Processing, pp. 409–418. Springer (2011)
9. Caeiro Rodriguez, M., Priol, T., Nemeth, Z.: Dynamicity in scientific workflows. Institute on Grid Information, Resource and Workflow Monitoring Services, CoreGRID-Network of Excellence, Tech. Rep. TR-0162, August (01 2008)
10. Deelman, E., Blythe, J., Gil, Y., Kesselman, C.: Pegasus: Planning for execution in grids. Tech. Rep. Technical Report 2002-20, GriPhyN (2002), http://pegasus.isi.edu/publications/ewa/pegasus_overview.pdf
11. Deelman, E., Gannon, D., Shields, M.S.: Workflows for e-Science. Workflows for e-Science (2007). https://doi.org/10.1007/978-1-84628-757-2
12. Dias, J., Guerra, G., Rochinha, F., Coutinho, A.L., Valduriez, P., Mattoso, M.: Data-centric iteration in dynamic workflows. Future Generation Computer Systems **46**, 114–126 (2015)

13. Digabel, H., Lantuéjoul, C.: Iterative algorithms. In: Proc. 2nd European Symp. Quantitative Analysis of Microstructures in Material Science, Biology and Medicine. vol. 19, p. 8. Stuttgart, West Germany: Riederer Verlag (1978)
14. Fakhfakh, F., Kacem, H.H., Kacem, A.H.: Workflow scheduling in cloud computing: A survey. In: 2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations. pp. 372–378 (Sep 2014). https://doi.org/10.1109/EDOCW.2014.61
15. Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: IFIP International Conference on Network and Parallel Computing. pp. 2–13. Springer-Verlag (2006)
16. Gil, Y., Deelman, E., Ellismand, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., Myers, J.: Examining the challenges of scientific workflows. IEEE Computer **40**(12), 24–32 (2007)
17. Mattoso, M., Dias, J., A.C.S.Ocaa, K., Ogasawara, E., Costa, F., Horta, F., Silva, V., de Oliviera, D.: Dynamic steering of HPC scientific workflows: A survey. Future Generation Computer Systems **46**, 100–113 (2015)
18. Montella, R., Di Luccio, D., Kosta, S.: DagOn: Executing Direct Acyclic Graphs as Parallel Jobs on Anything. In: Proceedings of WORKS 2018: 13th Workshop on Workflows in Support of Large-Scale Science, Held in conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 64–73 (2019). https://doi.org/10.1109/WORKS.2018.00012
19. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004)
20. Otsu, N.: A threshold selection method from gray-level histograms. IEEE transactions on systems, man, and cybernetics **9**(1), 62–66 (1979)
21. Pelt, D.M., Batenburg, K.J., Sethian, J.A.: Improving tomographic reconstruction from limited data using mixed-scale dense convolutional neural networks. Journal of Imaging **4**(11), 128 (2018)
22. Rocklin, M.: Dask: Parallel Computation with Blocked algorithms and Task Scheduling. Proceedings of the 14th Python in Science Conference (SCIPY), 126–132 (2018). https://doi.org/10.25080/majora-7b98e3ed-013
23. Simancik, F., Jerz, J., Kovacik, J., Minar, P.: Aluminium foam-a new light-weight structural material. METALLIC MATERIALS **35**, 187–194 (1997)
24. Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R.M., Torres, J., Cortes, T., Labarta, J.: PyCOMPSs: Parallel computational workflows in Python. International Journal of High Performance Computing Applications **31**(1), 66–82 (2017). https://doi.org/10.1177/1094342015594678
25. Tolosana-Calasanz, R., Baares, J.A., Rana, O.F., lvarez, P., Ezpeleta, J., Hoheisel, A.: Adaptive exception handling for scientific workflows. Concurrency and Computation: Practice and Experience **22**(5), 617–642 (2010). https://doi.org/10.1002/cpe.1487, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1487
26. Van Aarle, W., Palenstijn, W.J., De Beenhouwer, J., Altantzis, T., Bals, S., Batenburg, K.J., Sijbers, J.: The astra toolbox: A platform for advanced algorithm development in electron tomography. Ultramicroscopy **157**, 35–47 (2015)
27. Zhang, J., Kuc, D., Lu, S.: Confucius: A Tool Supporting Collaborative Scientific Workflow Composition. IEEE Transactions on Services Computing **7**(1) (2014)
28. Zhao, Y., Raicu, I., Foster, I.: Scientific workflow systems for the 21st century, new bottle or new wine? In: Proceedings of the 2008 IEEE Congress on Services - Part I, SERVICES '08. pp. 467–471. Washington, DC, USA (2008)

B

DEVELOPMENTS IN EVENT-ORIENTED, EMERGENT WORKFLOWS

This paper was submitted to Euro-Par 2021[32], but was not accepted, on the grounds that it focused too much on technical implementation details, rather than genuine novel research. Before being re-submitted further research would have to take place, perhaps focusing on direct comparisons between MEOW and more static systems once the local runner is more robust. Results could then be added to this paper with direct measurements of overheads and other comparison metrics.

Developments in Event Oriented, Emergent Workflows[⋆]

David Marchant[1][0000−0003−4262−7138], Rasmus Munk[1][0000−0003−0333−4295], and Brian Vinter[2][0000−0002−3947−9878]

[1] Niels Bohr Institues, University of Copenhagen, Denmark
d.marchant@ed-alumni.net, rasmus.munk@nbi.ku.dk
[2] Aarhus University, Faculty of Technical Sciences, Aarhus, Denmark vinter@au.dk

Abstract. The recently introduced framework for constructing dynamic workflow systems, Managing Event Oriented Workflows (MEOW), and its implementation `mig_meow`, both require further development before they can be considered complete. This paper describes three major improvements that should ensure MEOW and `mig_meow` meet the needs of any scientific workflow system. Firstly, the ability has been added for users to create their own workflow runner, independent of any previously required grid hardware. Secondly, a system for trivially scheduling large numbers of jobs across a range of parameters has been added. Finally, the ability for MEOW workflows to dynamically identify job outputs and construct a provenance report will also be outlined. This sets up MEOW as a promising way for researchers to schedule large volumes of jobs, and to experiment with fully dynamic workflows across both large and small datasets.

Keywords: MEOW · Workflows · Emergent · Dynamic · CSP · MiG.

1 Introduction

Scientific workflows have a need to be dynamic and exploratory in nature[6]. To meet this need, the framework Managing Event Oriented Workflows (MEOW) was recently introduced [11] as an event-driven system, which used emergent workflows that can easily be adapted at runtime. An implementation of this, `mig_meow` was also presented, however, it lacked some features before it could be considered a mature system. This paper outlines three such developments within `mig_meow`. Namely, the addition of a local workflow runner, parameter sweeping, and provenance reporting should all help complete `mig_meow` as a Scientific WorkFlow System (SWFS).

[⋆] This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604.

1.1 The state of mig_meow

An in-depth introduction to mig_meow[10] can be found in [11], though a brief introduction shall be provided here. The mig_meow package is based upon MEOW, a theoretical framework for describing event-driven systems that use the Patterns and Recipes to create emergent workflows. MEOW Recipes define a jobs processing, while Patterns define the conditions under which a new job should be scheduled. When the outputs of one job act as the inputs of another, a workflow will emerge. This event-driven system has been implemented within the Minimum intrusion Grid (MiG)[2], using file system events as its foundation, as will be outlined in section 2.1. To both interact with the MiG, and to define Patterns and Recipes, mig_meow was implemented and deployed as part of the Jupyter Notebook environment provided by the MiG[14].

1.2 Areas For Development

In works such as [22] and [12], a SWFS is described as needing to possess clarity, well-formedness, predictability, recordability, provenance, portability, and performance optimisations. Although these are not formal requirements, a well regarded SWFS should pay heed to as many as possible, and currently mig_meow offers no significant features to address provenance, that being the ability to track data through the workflow and easily determine what processing has taken place on it. This shortcoming will be addressed by the addition of logging of workflow job outputs, and the ability to compile a report displaying all jobs and their connections in Section 3.1.

Another motivating feature request from users is the ability to schedule several jobs at once from the same input file[17], each with a different value for one or more parameters. This style of scheduling multiple otherwise identical jobs shall be referred to throughout this paper as parameter sweeping, and is seen as especially desirable due to the exploratory nature of scientific workflows [6]. A solution for this is provided in section 3.2.

Another common request is the ability to run MEOW workflows without having to use the MiG[2]. This would be very helpful, both as a method of testing workflows before deploying them on the MiG and potentially scheduling time on expensive resources, but also to run smaller workflows on a users own machine. The workflow runner outlined in section 3.3

2 Background

Before we get onto looking at how these feature requests were met, it is worth considering some wider context. For instance, it is worth expanding on the MiG, and looking at the state of other SWFS's. As mig_meow has already been briefly introduced in section 1.1 it will not be examined again here.

2.1 MiG

The MiG[2] is a feature rich, cloud storage and processing solution developed at the University of Copenhagen. It allows researchers at the University as well as external collaborators to upload, store, and share data amongst user-defined Workgroups. Compute resources are available on which users can schedule processing. The MiG also provides services such as JupyterLab, granting users easy access an interactive compute environment, which can also be used to define processing. It is intended as the primary system for `mig_meow` workflows, with the MiG Workgroup file system being the basis for triggering events, MiG resources being used to process MEOW jobs, and Jupyter widgets being a primary means of interacting with MEOW workflows.

Processing happens on the MiG using a queue system. Resources are enrolled to the MiG, and once ready, will poll the MiG which will pull an appropriate job from the queue. A job script is generated by the MiG and sent to the resource to be run. In a workflow job, the appropriate Workgroup from MiG storage is mounted into the job resource using SSHFS[19]. This allows for easy access to Workgroup data, and a way to write output data back into the Workgroup.

2.2 Other Workflow Systems

All of the motivating features exist to some degree in many current SWFS's. For example, many existing workflow systems such as the commonly used Apache Airflow[1] or Snakemake[18] are able to perform parameter sweeps. This is qualified as they lack explicit features to enable parameter sweeping, but a user is able to replicate this functionality through the use of scripting, or clever nesting of workflows. This could be seen as a limitation as scientific workflows are often exploratory in nature [3][6], and so having explicit functionality to ease parameter sweeping would be advantageous.

It is hard to generalise about provenance reporting provided by SWFS's, as the type, amount, completeness and usability of different workflow reporting varies massively. Many of the more commonly used systems have GUI's to create workflows, display jobs, and provide extensive reporting. For instance, Airflow provides a web interface that can report on a workflow. Individual jobs are listed along with their inputs, outputs, parameters and the run code/scripts. However, this is not a universal standard. Many workflow systems have more simple reporting, such as Dask[5] or cwltool[4] which can produce a Directed Acyclic Graph (DAG) showing the various workflow steps and how they lead into each other. This can be useful to demonstrate the workflow structure, but is more limited than the previously mentioned GUI, which tend to include far more additional information.

Whatever the structure of a workflow report, the most common format is a DAG. Assembling such a DAG is an easy task for a SWFS as any workflow steps will have defined inputs and outputs, with the outputs of one step usually acting as the input for another. This allows for easy understanding of the resultant workflow structure, and identification of inter-jobs dependencies. This is more difficult to achieve in MEOW workflows, as a user has no requirement to identify outputs ahead of processing.

Almost every SWFS has some ability to run workflows locally. These local runners enable users to learn, test, and explore the possibilities of a workflow system. Testing workflows in advance can be especially useful, as the ever increasing size of scientific processing drives more and more computation to cloud and heterogeneous resources. These resources will have attached costs both in terms of time and money, and so users need the ability to check in advance that their workflow structure is valid.

3 Developments in `mig_meow`

Now that we have outlined the various problems facing `mig_meow`, we can start to address them. These will each be addressed in their own section, with all demonstrated through examples later in 4.

3.1 Provenance reporting without defining outputs

When used as first designed, `mig_meow` job identification and processing will be done on the MiG. Therefore, it is the responsibility of the MiG to record what jobs are scheduled as part of any MEOW workflows. The MiG already keeps a thorough record, but only on a per-job basis, and there is no distinction between MEOW jobs and regular jobs.

Compiling a list of MEOW jobs is not challenging, but linking jobs together into a holistic report is. This is because as part of MEOW, a user does not need to specify a jobs output. Within `mig_meow`, a user can define output, but this is only used for the visualisation and is not meaningful to the job execution. This is a deliberate choice within MEOW, as it keeps to the dynamic aims of the system, where no particular output is demanded or even expected. In order to start linking jobs, we need to identify at runtime what output is produced.

As jobs are processed on external resources with the relevant MiG storage location mounted via SSHFS, communications via this mount can be monitored. Any write operations that occur through the mount are the produced output. The MiG SSH daemon uses `paramiko[16]` to manage SSH connections, which provides interfaces for `write` operations. By writing a decorator for `write`, each write operation through the SSHFS connection can be intercepted and logged. Each SSHFS connection created as part of a MiG job has associated unique session credentials, allowing any logged writes to be easily paired with the workflow job that initiated them. This allows job outputs to be successfully identified and attributed, without requiring a user to identify them ahead of time, maintaining the dynamic nature of MEOW.

Inputs are significantly easier to identify, as all MEOW jobs will take a triggering file as input. Once each job has both input and output, a DAG can be constructed showing the path through the workflow that data has taken. This is semantically different to the DAGs of other SWFS as it can only be constructed in retrospect, due to the emergent nature of the workflow. To display a MEOW workflow report another widget was created within `mig_meow`, called the `ReportWidget`. It retrieves the job logging of a given Workgroup from the MiG. The `ReportWidget` allows users to filter down the report according to job names, Patterns, Recipes, or triggering paths. A sample of one report is shown in Figure 1. For brevity only a single job is shown, though already we can see how individual jobs are triggering each other and creating a workflow.

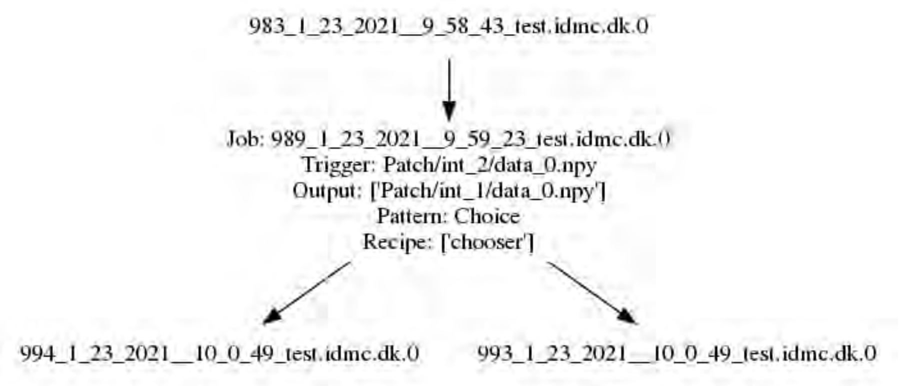

983_1_23_2021__9_58_43_test.idmc.dk.0
Job: 989_1_23_2021__9_59_23_test.idmc.dk.0
Trigger: Patch/int_2/data_0.npy
Output: ['Patch/int_1/data_0.npy']
Pattern: Choice
Recipe: ['chooser']
994_1_23_2021__10_0_49_test.idmc.dk.0
993_1_23_2021__10_0_49_test.idmc.dk.0

Fig. 1. Provenance report filtered to only show the job '989_1_23_2021__9_59_23.test.idmc.dk.0'. This shows the previous job, whose output triggered this job, along with the two jobs triggered by its own output. It also shows the most important characteristics of the job itself.

3.2 Parameter sweeping in MEOW

To avoid users having to use clever tricks to schedule a parameter sweep from a single input file, specific functionality was added to `mig_meow`. This was done by adding a new attribute to the Pattern object with four properties. Each `parameter_sweep` has a *name*, that being the name of the variable to assign a value. The value to replace is then defined by a *starting value*, *ending value*, and *step value*. These form a list of values in the same manner as a traditional 'for loop' within C-like programming. There is therefore a requirement that parameter sweeps can only express numerical values, though can be either integers or floats. This was judged to be acceptable as numeric values have always been used in all the parameter sweep use cases the authors are aware of.

On the MiG, when a Pattern is triggered with a defined parameter sweep, a list of values is assembled according to the start, end, and step values. A new job is then scheduled for each value. If multiple parameter sweeps are defined then a new job is scheduled for every combination of values. Much like any other workflow job running on the MiG, each job is scheduled in isolation so it can be completed, modified or fail. This ability to fail can be replicated in other SWFS's, either through a users own careful error management within their code, or through a workflows inbuilt functionality to replace failed jobs such as in DVega[20]. Within MEOW however, no additional management is needed on the part of the user and any jobs with inappropriate parameters will fail without affecting the results of any valid jobs. This is especially useful if the sweep is being used to explore values in an algorithm and the user does not yet know yet even an approximate value yet.

3.3 Running MEOW workflows without the MiG

To allow users to be able to run MEOW workflows without depending on the MiG, a local `WorkflowRunner` was developed. This would serve two purposes, firstly it would allow users to run their own workflows, but also it would allow the testing of Patterns and Recipes before they are uploaded to the MiG. For this reason the `WorkflowRunner` should behave in a similar manner to the MiG where possible. As the MiG is a complete grid management solution we cannot, and should not, model it in full, but we can borrow the job scheduling and processing structure.

As described in section 2.1, within the MiG jobs are put into a central queue of all jobs, held on the MiG server itself. Resources poll the server for jobs and one is taken from the queue and sent to the resource to be executed. This structure will be adopted by the `WorkflowRunner`. As the MiG server is run with multiple processes, a multi-processed structure is necessary in the runner if it is to behave in the same way. An effective way of designing multi-process programmes is to use the design model, Communicating Sequential Processes(CSP)[8]. A Python implementation of CSP does exist, called `PyCSP`. It is unfortunately not maintained, and so was not used for the workflow runner. Instead, the multiprocessing package was used as it could achieve similar results, and as part of the python standard library, can be expected to be kept up to date.

The CSP model is still of use to the runner, as it can provide some of the core design principles used to assess its correctness. For instance, according to CSP, no processes should share access to data. This ensures that race conditions cannot happen, as no two processes will ever be reading or writing at the same location. A very useful property of CSP is that if process interactions are setup appropriately, then the system is guaranteed to be free of deadlock. Although we are not using CSP directly, we can apply the principles of CSP to achieve the same result. The key way in CSP to avoid deadlock, is to avoid a circular loop of dependent communication[9]. As can be seen in Figure 2, this has been done as there is a linear hierarchy of primary processes communication. For this reason we can be confident in the claim that this system will not suffer from deadlock.

The process structure for the `WorkflowRunner` is shown in Figure 2, with communication between processes through `Pipes`. Each individual process type shall now be considered in depth.

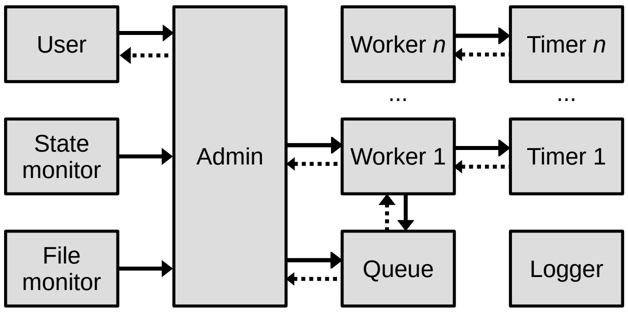


Fig. 2. Process structure of the `WorkflowRunner`, showing individual processes and their interactions. Note that in addition, the admin, state monitor, file monitor, queue and worker processes also can send messages to the logger process, though these connections have not been shown for brevity. Secondary connections used only for replies are shown in dotted lines. Zero to n workers are created based on user input.

The USER process: The *User* process is the base process in which the constructor for the`WorkflowRunner` is called, and from which the `WorkflowRunner` object is returned. Within the constructor, all other processes are setup and started. The `WorkflowRunner` object is then used as the entry point for any user interaction, with each sending an appropriate message to the `Admin` process. A response is always expected from the `Admin`.

The STATE MONITOR process: Both the *State Monitor* and *File Monitor* inherit from the `PatternMatchingEventHandler`, part of the `watchdog` API[21]. The `PatternMatchingEventHandler` responds to system events, according to given sub-paths from a watched directory. In the case of the *State Monitor*, this is the hidden '.workflow_runner_data/' directory, with the sub directories 'patterns/' and 'recipes/'. These locations are used to store files defining Patterns and Recipes, in a manner similar to how they are stored on the MiG. These files can be altered and updated at any time either by direct user interaction, or through using functions from the *User* process. In either case, this monitor will catch any changes and send any updates to the `Admin` process. No response is ever expected from the `Admin`, so the *State Monitor* process should never be blocked, and is therefore always able to process new events.

The FILE MONITOR process: The *File Monitor* is very similar to the *State Monitor* process, though its monitors base data directory. The base directory is equivalent to a Workgroup on the MiG. As the *File Monitor* does not know what Patterns and Recipes have been established, it can do relatively little processing of events itself. All it can do is filter out irrelevant events, such as 'delete' events, or bunch together repeated events at the same file location so as to not spam the `Admin` process. Whenever an appropriate event is identified, it is sent to the `Admin` to be checked against the registered Patterns and Recipes.

The ADMIN process: By far the most complex process is the *Admin*. It maintains the in-memory state of the runner, in which all currently registered Patterns and Recipes are stored. Updates to this state are provided by the *State Monitor* process, ensuring that the in-memory state is up to date with the saved state expressed in the Pattern and Recipe files. Patterns and Recipes can also be added, removed, or modified via user interaction from the *User* process. Any changes will result in the appropriate update to the file state, with files being added, removed, or updated. This will in turn generate more updates from the *State Monitor*. To prevent a circular loop of events creating file writes which are interpreted as events, files are strictly only written by the *Admin* process if a change has occurred to its in-memory state.

The *Admin* process will also receive input from the *File Monitor*. These events will be compared against the currently registered Patterns and Recipes. If the event path matches the 'trigger_path' attribute of a Pattern, then the *Admin* will create a new job, and send its ID to the *Queue*, so that it may be processed. Creating a new job consists of creating a unique job ID, with a corresponding job directory created to store job files. These files are a new notebook file, created as a copy of the appropriate Recipe, along with two yaml files. These contains the variables defined by the appropriate Pattern, such as the triggering path, and the jobs meta information, such as when it was scheduled.

As well as this core functionality, the *Admin* deals with requests from the the *User* process. Aside from the previously mentioned adding or modifying Patterns and Recipes, users may also query the current state, or running status of the `WorkflowRunner`. Some requests, such as to query the current queue composition require further messages to be sent to the *Queue* before a response can be generated, but a response is inevitable and provided as soon as possible. The *Admin* process utilises a `wait` statement to stand by until receiving input from either the *State monitor*, *User*, or *File Monitor* processes. These three inputs are prioritised in the order given, so that if multiple are available at the same time, only the first is read and processed.

One limitation of this, is that it can lead to starvation. For instance, if the *State monitor* produced continuous messages faster than they could be processed by the *Admin*, both the *User*'s, and *File Monitor*'s messages would never be read. This should not be a problem here though, as the `WorkflowRunner` is only intended for use on a local machine. Events from any of the monitors or the *User* can all be responded to relatively quickly, so it is seen as unlikely that the runner will get truly swamped with events to process. Messages from the *State Monitor* are always of the highest priority as a fresh state will always be needed by the Admin. Changes in the state file system will also be finite in nature, as a user is incredibly unlikely to make so many changes to Patterns and Recipes as to swamp the runner whilst it is running. Secondly, are messages from the *User* process. These are secondary as they will be requests from a user, and so will be conducted on a human time-frame. This means that they do not need to be responded to within nanoseconds and so can wait behind any *State Monitor* updates.

Lastly, this leaves the *File Monitor*. This may produce a theoretically infinite number of messages as there is no limit on the number of files created or updated by jobs. Despite this being an unlikely use case, it is nevertheless a possibility and should be accounted for, therefore it must be the lowest priority as anything behind it could be eternally starved in this scenario.

The WORKER process: Jobs are executed within the *Worker* processes. The amount of these to be spawned is determined by the user, and at least one is needed if the workflow runner is to process jobs. Each *Worker* has its own `Pipes` from the *Admin* and to the *Queue*. By default a *Worker* starts in a stopped state, and will only start when told to do so by the *Admin*. Once a worker is started, it will request a job from the *Queue* process. If a job is available, the ID will be sent to the worker, and its definition files are read from the job directory created by the admin.

The job itself is processed by first parameterizing the input notebook using the python module `notebook_parameterizer`[13]. This is then run using `papermill`[15] in the same manner as is done on the MiG. Once execution has been completed, the job files are copied into a separate job output directory where they can be individually inspected. Jobs may produce output directly into the data directory, monitored by the *File Monitor*, in the same manner as can be done within the MiG. As no SSHFS mounting is used in the `WorkflowRunner`, no workflow logging can be carried out using the same technique as was presented in section 3.1. For this reason, it is not currently possible for a provenance report to be generated from a locally run MEOW workflow.

If no job was available in the *Queue*, the *Worker* sends a notification to its *Timer* process to start sleeping. If a job completes, or the *Worker* is notified by the *Timer*, it will poll the *Queue* for another job. This polling of the *Queue* will loop until the *Worker* is manually stopped by user input.

The TIMER process: To prevent spamming the *Queue* process with requests for new jobs, each *Worker* has its own *Timer* process. This process will wait for a start signal from their *Worker* and then sleep. Once the sleep is over, it will send a signal to the *Worker* as a prompt to request a job again from the *Queue*. By having the timer in a separate process rather than internal to the *Worker*, the *Worker* is still free to receive messages from the *Admin*, which would not be the case if it itself were sleeping.

The QUEUE process: The *Queue* process acts as a buffer for all jobs that have not yet been processed by a worker. It accepts messages either from the *Admin* or any of the *Worker* processes. From the *Admin*, the *Queue* will either receive the identity of a new job to be added to the queue, or a request for the current composition of the queue. Alternatively, any of the *Workers* may request the identity of a new job to execute. In any case, a response is always immediately generated and sent. It was necessary to separate the queue into its own process, rather than having it stored within the *Admin*, as otherwise there would be a risk of deadlock between *Workers* and the *Admin*.

The LOGGER process: Every non-*Logger* process except the *Timer* processes have a `Pipe` to send messages to the *Logger*. These are messages to be written to a log file for debugging, and/or printed to the console if the appropriate flags are set during runner creation.

4 Example Workflows

To illustrate the utility of MEOW, two workflows will be presented here. The first is a scientific workflow already demonstrated on the MiG[11]. It will be run using the `WorkflowRunner`, to demonstrate that it is capable of running the same workflows as the MiG. Secondly, a demonstration workflow with a complex structure shall be shown both on the MiG and the `WorkflowRunner` to demonstrate some of the more exotic possibilities of a MEOW workflow.

4.1 Tomographic Analysis

This workflow was first presented in the paper 'Managing Event Oriented Workflows'[11]. As it is a direct repeat of the analysis carried out in that paper, it will only be briefly introduced here.

Problem Outline This workflow is designed to analyse 3D X-ray computed tomography datasets. The input data is artificially generated, and we want to analyse the distribution of pore radii in each sample. However, the analysis is time consuming and not all generated samples may have a sufficient amount of pores. These insufficient samples will be discounted from the final analysis. Those samples which are deemed acceptable will be segmented into different materials, and finally the segmented data will be analysed.

Setup Three Patterns and Three Recipes will be set up, with all being identical to those used in the previous running of this example. The only difference is that this time the Patterns and Recipes have been set up programmatically rather than through the `WorkflowWidget`. This makes for easier reuse. The code to run the experiment can be found in [7]. The same input data was used as before, with 100 artificial datasets, of which 80 were valid and 20 were not. A significant difference was in the hardware used to run the workflow. To run the workflow, a small laptop with a 4 core, Intel Core i7-8550U CPU @ 1.80GHz and 8GB of RAM. As it was a 4 core machine, it seemed appropriate to use 3 worker processes, so that hopefully they could always be run in parallel, whilst leaving an extra core to run the runner itself.

Workflow Results The workflow produced 260 jobs, and 80 analysis graphs were produced as final output. To complete all 260 jobs took 2 hours and 4 minutes, roughly half the time taken to do the same analysis as on the MiG. This is probably due to this actually being a very small use case, with the total input dataset only being 6.7GB in size. When coupled with the relatively simple processing carried out, it means it is still possible to run in a reasonable time on a users personal machine. This is especially true when comparing it to the MiG, which will have a great deal of overhead and delays from the grid-based nature of the system. For this reason we should not take from this that the `WorkflowRunner` is necessarily a better option than the MiG, only that the `WorkflowRunner` can run MEOW workflows suitable for the MiG.

4.2 Abstract Analysis

To better illustrate some of the utility of MEOW workflows, a more abstract example has been developed. This workflow has no scientific purpose, so we can focus purely on the structure and MEOW interactions.

Problem Outline This workflow will demonstrate the utility of a MEOW workflow by making specific examples of a branching, looping, and failing workflow. It will also show both the provenance reporting and parameter sweeping in action. It will take some nonsense data, and does some simple maths on it so we can say at least some processing has taken place.

Setup The workflow structure is shown in Figure 3, and the Patterns and Recipes are explained below. Input data for the workflow is any number of 2D `numpy` 5x5 arrays, containing random integers between 0 and 100. The input data starts in a directory called 'initial_data'.

- **Pattern *Add_5* with Recipe *addition*:** This is the first Pattern to trigger as it takes any file in 'initial_data/' as input. The Recipe will read in the data and add 5 to each index. This is then saved to a given location, 'int_1/'.
- **Pattern *Doubler* with Recipe *multiplier*:** This Pattern takes files within 'int_1/' as input. The Recipe *multiplier* will multiply all values in the data by the given factor, 2. Output is saved to 'int_2/'.
- **Pattern *Choice* with Recipe *chooser*:** The *Choice* Pattern takes as input, files in 'int_2/'. Depending on if any value in the input data is larger than a given threshold it will output to one of two locations. If the threshold is met, then the data is written to 'final/'. Otherwise, it is incremented and is written to 'int_1/'. This will form a loop between the *Doubler* and *Choice* Patterns. Due to the threshold, the loop will not be infinite, but it is unclear at the start of the workflow how many times it will be run. For this workflow, the threshold was set to 10000.
- **Pattern *Division* with Recipe *divider*:** Like the *Doubler* Pattern, this is triggered by any files in 'int_1/'. It will read in the data and sum it, then

take the modulus of the sum and a given number, in this case 8. If modulus is 0 then the data will be written to 'div_by_8/'. Otherwise, an `Exception` will be raised causing the job to fail.

– **Pattern *Add_Range* with Recipe *addition*:** This Pattern functions very similarly to *Add_5*, though it takes files in 'div_by_8/' as input. Rather than defining a single value for the 'extra' variable, but uses a parameter sweep to use the values 1, 2, and 3. Each will output to 'add_to_div/'.

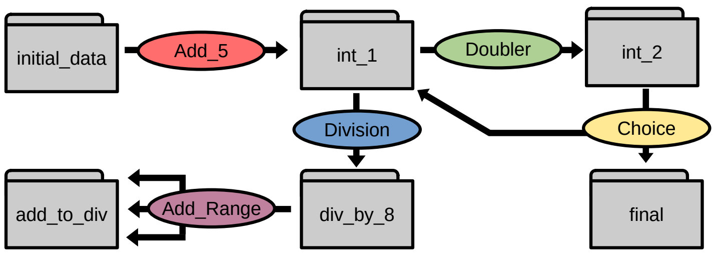


Fig. 3. Overview of the abstract workflow. Patterns are shown in bubbles, File locations are shown as grey folders.

Workflow Results The described Patterns and Recipes were registered on the MiG. Only three input files were used, as this would be sufficient to display three routes through the same MEOW workflow, without overly cluttering the output.

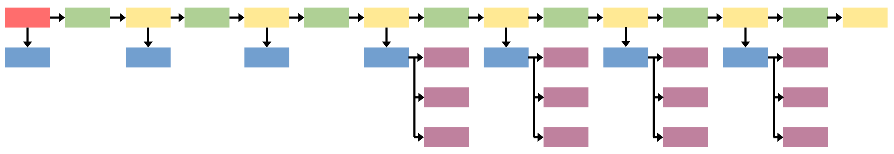


Fig. 4. Provenance graph, with each rectangle representing a job as first triggered by 'initial_data/data_0.npy'. Note the colours have been added to match those of the Patterns in Figure 3.

In total 108 different jobs were created. All jobs were completed in roughly 25 minutes, and a provenance report was compiled using the `ReportWidget`. All results are visible in [7]. As the complete report is rather large, a representation of it is shown in Figure 4. It can be seen that the data is processed once according to the *Add_5* Pattern, before looping repeatedly through *Doubler* and *Choice*. Eventually the threshold is met and the loop stops, as expected. While this loop is ongoing we can see that the *Division* is also triggered several times, but does not complete the first three times it runs. When it does complete, it triggers the subsequent *Add_Range* Pattern which produces the expected three jobs each time. From all this we can conclude the the emergent MEOW workflow is performing as expected, with the dynamic route through the Patterns producing a changing workflow with relative ease. Parameter sweeps have allowed for the easy scheduling of jobs over a range of values. Finally, the provenance report offers a marked improvement over relying on the MiGs job reporting, as it clearly demonstrates how different jobs link together and the workflow emerges.

These Patterns and Recipes were also run on the `WorkflowRunner`. 108 jobs were created and completed in roughly 10 minutes. As in section 4.1, this should not be taken that the `WorkflowRunner` is faster, only that it has less overhead.

5 Future Work

While the presented additions help build up MEOW as a complete SWFS, there are still many avenues for future development. The limitation of a lack of shared memory identified first in [11] has still not been addressed, as this is a far more substantial undertaking than can be addressed in this paper. In addition to this, identifying a way of constructing a provenance report for the `WorkflowRunner` would improve its usability significantly. It may be that write operations could be decorated in a similar matter to those on the MiG. Though this has not been investigated, it is expected that this would significantly reduce performance. A better approach may be to keep track of what system processes are writing data into the monitored directories, and match worker processes id's to the jobs they are currently processing.

6 Conclusion

This paper has described three significant improvements in the MEOW workflow system as presented in [11]. By enabling parameter sweeping, a user can schedule a whole range of jobs from a single data input quickly and easily. This is especially ideal within MEOW as it was already well suited to exploratory workflows, so an ability to test ranges of parameters is an ideal compliment. A way of demonstrating the provenance of the produced data was also provided when MEOW workflows are run using the MiG. This makes it significantly easier to see what each job produced, from what input, and how it triggered further jobs. Now, the emergent workflow is explicitly visible to the user. Finally, a `WorkflowRunner` was developed allowing users to run MEOW workflows on their own machine. This has the dual uses of letting users test their workflows before activating anything on the MiG, or letting them run workflows without using the MiG at all. These improvements have each been successful, and expand the feature-set of mig_meow so that it can act as a more complete SWFS.

References

1. Apache Airflow. https://airflow.apache.org (2020)
2. Berthold, J., Bardino, J., Vinter, B.: A principled approach to grid middleware: Status report on the minimum intrusion grid. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) Algorithms and Architectures for Parallel Processing, pp. 409–418. Springer (2011)
3. Caeiro Rodriguez, M., Priol, T., Nemeth, Z.: Dynamicity in scientific workflows. Institute on Grid Information, Resource and Workflow Monitoring Services, CoreGRID-Network of Excellence, Tech. Rep. TR-0162, August (01 2008)
4. cwltool. https://github.com/common-workflow-language/cwltool (2020)
5. Dask. https://docs.dask.org/en/latest/ (2020)
6. Dias, J., Guerra, G., Rochinha, F., Coutinho, A.L., Valduriez, P., Mattoso, M.: Data-centric iteration in dynamic workflows. Future Generation Computer Systems **46**, 114–126 (2015)
7. Experiments:. https://sid.idmc.dk/wsgi-bin/ls.py?share_id=D1KHIT7Nij (2021)
8. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
9. Kerridge, J.: Using Concurrency and Parallelism Effectively. Bookboon, 2 edn. (2014)
10. Marchant, D.: mig_meow. https://pypi.org/project/mig-meow (2020)
11. Marchant, D., Munk, R., Brenne, E.O., Vinter, B.: Managing event oriented workflows. In: 2020 IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP). pp. 23–28 (2020). https://doi.org/10.1109/XLOOP51963.2020.00009
12. McPhillips, T., Bowers, S., Zinn, D., Ludäscher, B.: Scientific workflow design for mere mortals. Future Generation Computer Systems **25**(5), 541 – 551 (2009). https://doi.org/https://doi.org/10.1016/j.future.2008.06.013, http://www.sciencedirect.com/science/article/pii/S0167739X08000873
13. Munk, R.: notebook_parameterizer. https://github.com/rasmunk/notebook_parameterizer (2019)
14. Munk, R., Marchant, D., Vinter, B.: Cloud enabling educational platforms with corc (2020), CEUR-WS.org, will be published as part of proceedings of CTE 2020: 8th Workshop on Cloud Technologies in Education
15. Papermill. https://github.com/nteract/papermill (2020)
16. Paramiko. http://www.paramiko.org (2020)
17. Parkinson, D.: Interactive parallel workflows for synchrotron tomography (2020), https://wordpress.cels.anl.gov/xloop-2020, will be published as part of proceedings from XLOOP 2020 : 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing, held as part of SC20
18. Snakemake. https://snakemake.github.io (2020)
19. SSHFS. https://github.com/libfuse/sshfs (2021)
20. Tolosana-Calasanz, R., Bañares, J.A., Rana, O.F., Álvarez, P., Ezpeleta, J., Hoheisel, A.: Adaptive exception handling for scientific workflows. Concurrency and Computation: Practice and Experience **22**(5), 617–642 (2010). https://doi.org/10.1002/cpe.1487, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1487
21. watchdog. https://pypi.org/project/watchdog/ (2020)
22. Zhao, Y., Raicu, I., Foster, I.: Scientific workflow systems for the 21st century, new bottle or new wine? In: Proceedings of the 2008 IEEE Congress on Services - Part I, SERVICES '08. pp. 467–471. Washington, DC, USA (2008)

C

TEACHING CONCURRENT AND DISTRIBUTED PROGRAMMING

This paper, and a presentation describing it were originally presented at EduHPC-19: Workshop on Education for High Performance Computing[30], part of SC19: The International Conference for High Performance Computing, Networking, Storage and Analysis[84]. It was first published as part of the proceedings[52].

Teaching Concurrent and Distributed Programming With Concepts Over Mathematical Proofs

1st David Marchant
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
david.marchant@nbi.ku.dk

2nd Carl-Johannes Johnsen
Neils Bohr Institute
University of Copenhagen
Copenhagen, Denmark
carl.johnsen@nbi.ku.dk

3rd Brian Vinter
Neils Bohr Institute
University of Copenhagen
Copenhagen, Denmark
brian.vinter@nbi.ku.dk

4th Kenneth Skovhede
Neils Bohr Institute
University of Copenhagen
Copenhagen, Denmark
kenneth.skovhede@nbi.ku.dk

Abstract—**This paper describes how a concept-based approach to teaching was used to update how concurrent and distributed systems were taught at the University of Copenhagen. This approach focuses on discussion to drive student engagement whilst fostering a deeper understanding of the presented topics compared to more traditional displays of crude facts. The course is split into three sections: local concurrency, networked concurrency, and concurrency in hardware. This allows for an easier student journey through the course, as they are introduced to all core concepts in the first section, then have them reinforced in greater detail in the subsequent sections. Finally, the experience gained in updating this course is presented so others attempting to do similar may learn from it.**

Index Terms—**Concurrent, Parallel, CSP, SME, ZeroMQ, Teaching, Concepts**

I. Introduction

Scientific data processing is a considerable computing task that necessitates the use of High Performance Computing. Despite the presence of various libraries[1] to manage all aspects of parallel programming, knowledge of how a distributed system works is still essential to make full use of parallel hardware. This can present a problem as parallelisation is not a trivial topic, and scientists running experiments may not have an extensive background in computer science or programming.

At The University of Copenhagen, distributed computing courses are taught by the eScience group, part of the Niels Bohr Institute. These courses are mostly taught to physics students, who need this knowledge so that they can set up experiments to make use of parallel computing. A new *Concurrent and Distributed Systems* course has been introduced to replace an older course. The previous course had a theoretical focus, with rigid facts rather than engaging concepts. This would turn students off and give only a surface level of understanding.

This paper proposes replacing a fact-based approach to teaching with a concept-based approach. This will be broken down into 3 linked stages, local concurrent programming, distributed concurrent programming, and concurrent programming at a hardware level. In this paper these areas are demonstrated using PyCSP, PyZMQ and SME, but any similar library could be used in their place as it is the shared concepts beneath them that are important. These underlying concepts, such as deadlock, race conditions, and distributed states are the core of concurrent programming and so are the true learning goals.

Ultimately this was partially successful, with students responding well, though at great time investment on the part of the lecturers. Despite this, the techniques demonstrated their validity and could be applied to other similar courses going forward.

II. Objectives

This paper is an account of, and reflection on, teaching carried out within the *Concurrent and Distributed Systems* course. This is done with 3 goals in mind. In no particular order these are:

1) To record how a new approach to teaching concurrent systems design was put into practice.
2) To evaluate and reflect on how this new methodology worked.
3) To recommend for similar future courses what could be carried forward and what should be changed.

All of these stated objectives will be considered in the context of teaching parallel programming to non-computer scientists. Within this paper 'computer scientists' are those students whose primary area of study falls within computing. Any other students are 'non computer scientists' and are assumed to have no more than a passing familiarity with programming.

III. Background

The Niels Bohr Institute is a research institute specialising in astronomy, biophysics, condensed matter physics, geophysics, quantum physics, and particle physics. In addition to this, the eScience department conducts its own research into scientific methods using computers. As well as maintaining physical hardware and software to support the other departments, it is also responsible for teaching High Performance Computing. A new Masters level course in concurrent and distributed systems was designed to be run in the academic year 2018-2019 with the authors as the teaching team. This course was

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604.

[1]As illustrated by, but not limited to: CUDA [1], OpenMP [2], CSP [3], ZeroMQ [4], and MPI [5]

titled *Concurrent and Distributed Systems* and was intended as a refreshed version of older courses in similar areas that were now deemed insufficient.

The *Concurrent and Distributed Systems* course used as a basis for this paper was taught at the University of Copenhagen, from the 19th of November 2018, until the 27th of January 2019. It was taught with 2 lecture slots a week, each 1.45 hours in duration. There was a weekly practical session also 1.45 hours in duration. 3 assignments were given out over the course, each lasting roughly 2 weeks and a final examination was given in the form of a take home exam with students having 11 days to complete it.

IV. Facts VS Concepts

The base assumption for the new course was that focusing on the *concepts* of distributed computing was preferable to focusing on *facts*. By *facts*, *mathematical proofs*, *information*, or *technical details* we refer to pieces of knowledge that are (probably[2]) true. It is often fundamental to the subject area, can be easily rote memorised [7], and can easily be expressed in a book, lecture or other one way communication. The syntax of a built in function, the clock speed of different machines, particular communication protocols, would all be examples of *data*, *information* or *technical details*. For the rest of this paper these shall all be referred to as *facts*.

Ideas or *concepts* refer to pieces of knowledge that are not necessarily verifiable. These are the grand approaches that can emerge from multiple facts and used to explain or guide systems. For example, consider system architectures, design approaches, or algorithm structures. All of these demand engagement from someone to understand and cannot be meaningfully rote memorised. These depend on many facts to be understood, and our understanding of them is constantly morphing and being updated. They can also be used to extrapolate new areas of knowledge and understanding [7]. For the rest of this paper these shall be referred to as *concepts*.

Concepts such as object-orientation are already widely taught and understood within computing, demonstrating their utility within computing education. This is especially important as the difference between sequential and parallel programming is not one of technical details, but one of thought. How you approach a parallel problem is fundamentally different to a sequential one, with a completely different structure and thought process behind it. Put another way, the problems of parallel are concepts, rather than facts, and so they require teaching focused on those concepts rather than on facts [8]. Therefore, even though *Concurrent and Distributed Systems* is aimed at potentially novice computer scientists, the teaching should be concept-based.

We could say that facts are more basic than concepts in a learning context, as facts are specific and cannot broadly be applied. However, facts are still required to act as a base for understanding concepts, and so both concepts and facts should be taught together. This means that when this paper refers to teaching concepts rather than facts, it is meant that concepts should be emphasised over facts, but not that facts should be ignored entirely as they form a necessary part of students learning.

In response to all this it was decided to start from scratch with a new series of lectures. The university format meant that we had to keep to 2 lectures a week with a practical. As concepts are much more difficult to explain than facts, as they tend to require a back and forth discussion in order to teach [8], the lecture format may have presented a problem. However, the class was expected to be small enough that the necessary discussions could take place. Note that despite all that has been said, facts are still extremely important. Concepts without facts become meaningless as they cannot be applied to the external world. It was hoped then, that the resulting slides could communicate the necessary concepts of concurrent and parallel systems, with enough supporting facts so as to be understandable.

V. Course Goal

Concurrent and Distributed Systems was aimed primarily at non computer scientists with limited programming experience. This is justified by the increasing requirement for parallel processing by scientists in all areas of physics studied at the Niels Bohr Institute [9]. In addition, students may be involved in designing scientific instruments or experiments which contain concurrent systems. Although there exist libraries that purport to take care of all parallelisation for the user, such as CUDA [1], MPI [10] or OpenMP [2], these still need a good base of knowledge from the user before they can be fully utilised. It would be possible to run a course that only related the facts of how these systems work, by highlighting specific commands and their expected outcome. However, this would leave students with a very narrow pool of knowledge, specific only to the exact software and problems described in the course. By adopting a concept-based approach, where instead the base concepts of distributed programming are explored and understood, non computer scientists can claim a theoretical understanding of parallel programming. This should suffice for them to effectively use any of these preexisting systems, and potentially even start designing their own custom implementations.

As most undergraduate physicists are not expected to be running big enough experiments to justify the use of high performance systems, *Concurrent and Distributed Systems* was set at a Masters level with classes expected to have between 6 and 12 students enrolled. The sought after learning objective would be some measurable understanding of asynchronous concurrent and distributed systems. That is, a system comprised of multiple processes, where the order of processing is not and cannot be determined at the start of processing. By the end of the course students should be able to design and implement concurrent and parallel systems in both hardware and software. These systems should be robust

[2]At the very least it is expected to be true, even if it is up for debate. Within the field of Epistemology it could be said that these statements are justified beliefs, that are true as far as can currently be determined. Consider this in the context of the works of Edmund Gettier and others. [6]

to common design problems such as deadlock, livelock and race conditions. Students should also be able to demonstrate their systems correctness using diagrams and descriptions.

VI. Selecting Course Content

To introduce and reinforce universal concepts of parallel computing, the decision was taken to break the subject down into three smaller sections. These could then slowly introduce concepts, and illustrate their universality within distributed programming. As these concepts would occur repeatedly through the three sections in increasing depth, it was hoped that they would be further reinforced. Starting with local parallelisation would be logical, as it meant that problems such as networking could be ignored. This allows for the introduction of the base concepts such as determinism, race conditions, deadlock, livelock, compartmentalisation, as well as identifying what sort of tasks are suitable for parallel or not.

All scientific programming within the Niels Bohr Institute is taught in Python[3], where possible. This meant that the underlying language was already set, and that some familiarity with the language could be assumed. To illustrate parallel processing concepts in the first course section, PyCSP[4] [13] was selected as all members of the teaching team were already familiar with it. It is worth noting that PyCSP has been taught in related courses previously, and has been found to be a very good introduction to concurrent and distributed concepts, even for novices [14]. Sticking with what the teaching team were familiar with was seen as important as it meant all members had already built up a body of knowledge designing systems using PyCSP. It was hoped that this would mean that the teaching team could adequately answer questions without having to rely on slides or textbooks to do the heavy lifting. This would be essential if we were to avoid dry lectures of reading technical information to students, but were instead to encourage conversation and interaction.

From a localised system the next logical step was a distributed one. These would still be using the same concepts from before, but now with added challenges such as the impossibility of global memory. This could be done using PyZMQ[5] [15] as again, it is Python based, simple to learn, and the teaching team were already familiar with it. Finally it was felt that students should have some introduction to physical devices that could be used to run a distributed system, such as in an *Internet of Things* device. This was as the problems that would be introduced in the first two sections are just as much problems in hardware as in software. To illustrate this, FPGAs[6] were used, with SME[7] [16] as the code base.

It was decided that it might help students to keep motivated and interested in the material if they could relate it to their own interests or research [7]. As at this point it was known that the FPGA boards were to be used, and that the students would build a system on the board, it followed that this system could be something scientific. A simple sound locator system was decided upon. This could be used in all course sections, so that all the assessments are tied together by a common thread. In the first section the students design a PyCSP system to process multiple microphones listening for sounds to determine the direction of a sounds source. In the second section they design a networked system, where they each link together their individual systems. In the third section they then program an individual microphone.

The hope is that these three sections will cover all essential areas of knowledge for the students, and assumes relatively little background knowledge. By starting on local systems and working up to more and more decoupled examples it is also intended that students are slowly introduced to the topic without them being hit with incomprehensible topics all at once. The students journey through the course should be simplified greatly as in the first section performance and efficiency are secondary concerns to robustness and ease of understanding. As the students continue through the course they are introduced more and more to requirements of performance and working within the already introduced concepts to get more processing done in less time.

VII. Software and Hardware

Software and hardware infrastructure was needed to run the course effectively. Students would need Python, PyCSP, ZeroMQ, and SME. These libraries and their dependencies could be time consuming to set up per individual, setting them up would not be particularly informative to the students. To get around this, JupyterLab Notebooks were used as a learning environment. JupyterLab Notebooks are documents accessible online, capable of displaying and running live code. They are centrally stored and so can have all dependencies pre-loaded onto them, meaning all students can easily start from the same point, with a complete system. For hardware, the PyNQ [17] board was selected as they contained a FPGA chip, had the necessary hardware to run Python scripts, and were reasonably priced.

VIII. Previous Teaching Material

For most of the course there already existed relevant lecture slides from previous similar courses. Naturally, these needed slight editing to fit with the new course, but in the case of section one, on PyCSP, a complete rework was required.

[3]This is due to the utility of Python for scientific analysis [11] and research, as well as its wide adoption throughout the scientific community. [12]

[4]PyCSP is a Python specific implementation of Communicating Sequential Processes (CSP) [3], a formal definition of how a system could be split up into several independent sub-sections and how those sub-sections would communicate. Other implementations exist in other languages, in varying states of completeness. The underlying principles between each are shared however.

[5]PyZMQ is a python specific implementation of ZeroMQ (also known as ØMQ) [4]. It is a library for easy asynchronous communication over a network.

[6]Field Programmable Gate Arrays (FPGA) are essentially programmable circuit boards, allowing for the implementation of many different hardware circuits using only one device, as opposed to expensive, custom made chips.

[7]Synchronous Message Exchange (SME) is a CSP derived language for programming FPGA boards. It compiles into VHDL and is designed to be more user friendly and quicker to program.

Plenty of teaching material was available [14], in the form of slides, books and workbooks. Each of these were considered in turn but rejected for a variety of reasons. These materials were often for a different length of course, meaning serious cutting or padding would be needed. As the different resources available were also from disparate sources they were all designed inconsistently, so even more editing would be needed to bring together any slides into a common visual language.

Aside from these small, practical considerations, it was also felt that the available material relied far too much on reading complex information off of slides as a method of teaching. Slides would be either extensive blocks of code, complex diagrams, or paragraphs of text. Often times, mathematical proofs of correctness were included and run through, proving the validity of a certain approach. This may be correct, but that level of detailed understanding is unnecessary for most students, especially non computer scientists. The material did not support interaction beyond asking simple memory recall questions rather than discussion and could be said to be entirely fact-based, as discuss in section IV.

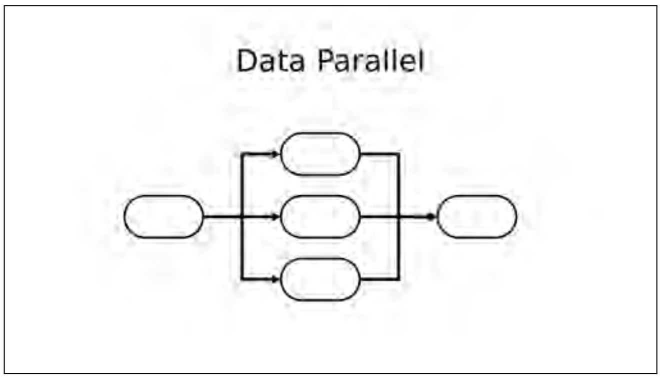
It was felt that a better approach would be a more discussion based one [18], with a focus on the ideas and concepts behind CSP rather than on the technical details [19]. This should be especially possible given that the course was set at a Masters level and so should have students who can engage in a topic more in terms of ideas rather just simple facts. The expected small enrolment also meant that facilitating informal discussions rather than a strict lecture should be possible. Finally, a conceptual understanding would be preferable for non computer scientists as numerous libraries and systems exist to automate the generation of parallel code. It is the theoretical understanding behind these libraries that the non-computer scientists need.

IX. Goals for the New Material

To design the new material, objectives had to be set against which it could be designed, and success judged. These goals were devised with the aim of using an inductive approach to teaching [20]. These are presented below, with higher priority goals being at the top of the table. The material should:

	Goal
G1	Facilitate the teaching of concurrent and parallel concepts.
G2	Support the presented concepts with facts.
G3	Encourage student engagement in the class through exercises and discussion.
G4	Provoke questions and discussion from the students.
G5	Enable the teacher to explain in their own words, rather than relying on technical definitions.
G6	Be clear and easy to understand.
G7	Be reusable in subsequent courses, even by others not on the current teaching team.

Most of these goals should be self explanatory and so will not explained at length. G3 and G4 may require clarification



Data Parallel

Fig. 1. Lesson 1, slide 23. This demonstrates the simple, clean design for the slides with the bare minimum of information. This diagram is intended as a conversation aid, rather than an explanation on its own.

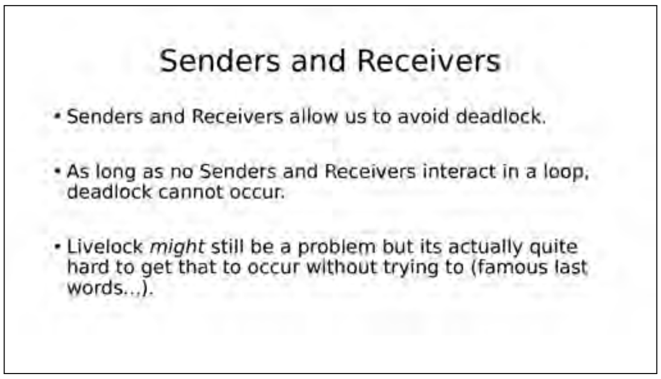
though. Ultimately they both are the same idea, to focus more on conversation about a topic, rather than a direct lecture on it. It has been split into two goals to show that this is a two part process. In G4 we need to make sure that the teacher is accepting of this style in their teaching, whilst in G3 we should encourage the students to be interacting with the lesson. After all, if only one person is trying to start a conversation, it will not happen easily. Note the use of the words ‘teacher’ and ‘lesson’ rather than ‘lecturer’ and ‘lecture’. This is done to suggest that the person standing at the front is not merely talking *at* the students, but *with* them [19], and does not denote any further difference.

X. Designing New Material

With these goals in mind, eight sub-topics were selected for section one of *Concurrent and Distributed Systems*[8]. These could then roughly align to the 4 lectures given in this section, with each lecture split by a small break, forming 8 half-lecture slots of roughly 45 minutes. In that time new concepts should be introduced, the facts to support them presented, and the resulting discussion engaged in. This is a lot to do, so presentations were kept to around 25 slides. Sentences on slides were kept short and well spaced. Diagrams were plain and presented without accompanying text on the slide. For examples, see figures 1 and 2.

Both of these slides are typical of the newly made slides for this course, as both of them provide one or two key facts, and very little else. This was done deliberately with the aim of fostering conversation. By having so little information on the slides the lesson could not turn into a session of just reading information from slides, as there simply is not enough information to fill the time by doing so. This approach would also be coupled with regular questions from the teacher so as to foster more dialogue than in a more traditional lecture.

[8]These were *parallel design problems*, *an introduction to PyCSP*, *deadlock and livelock*, *parallel system design principles*, *determinism and race conditions*, *compartmentalisation*, *additional CSP concepts*, and *network communication*. For a complete course description and to see the contents of each section, all course materials are available at [21]



Senders and Receivers

- Senders and Receivers allow us to avoid deadlock.
- As long as no Senders and Receivers interact in a loop, deadlock cannot occur.
- Livelock *might* still be a problem but its actually quite hard to get that to occur without trying to (famous last words...).

Fig. 2. Lesson 3, slide 15. Where text must be used it is kept to a minimum. As before, these sentences are intended as aides to what is currently being discussed rather than a lesson on their own. Even the written text is written conversationally.

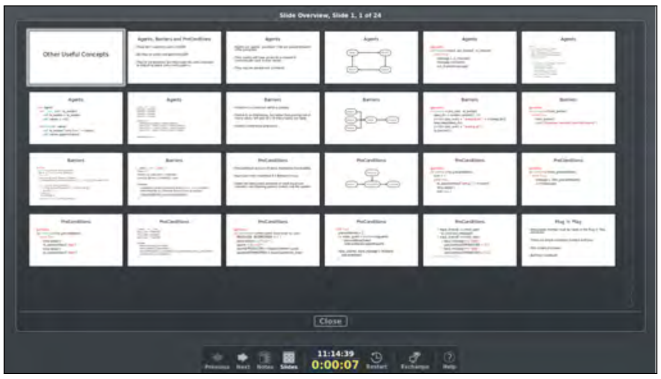


Fig. 3. Lesson 7, tiled view. This is demonstrated in LibreOffice Impress, but many other slideshow programs have similar features

The design of the slideshow itself also changed, with lessons being divided into sub-categories and where possible, not depending on a specific ordering to make sense. When displaying the slides in a lecture a tiled slide selector could be used to jump from slide to slide in a non-deterministic manner, and so follow the current direction of discussion. This is illustrated in figure 3. The simple design of the slides also helped here as it meant the correct slide is still readable on a laptop screen when in tiled view and so can be selected without difficulty.

To achieve the goal of more concepts and more discussion, regular exercises were introduced [19]. This is perhaps best exemplified by lecture 4, on designing a concurrent and parallel system that is only 5 slides long. The first slide is a title slide, while the second sets the exercise of designing a system. The third illustrates some discussion points that might occur as the students solve the problem and acts as an initial guide to students if they don't know how to start. The fourth introduces more exercise as it expands the initial problem. The final slide acts as a reinforcement to the core concept of this exercise, explicitly stating some supporting facts to ideas hopefully encountered. These slides would last no more than a few minutes, and exemplify every lectures role more as support

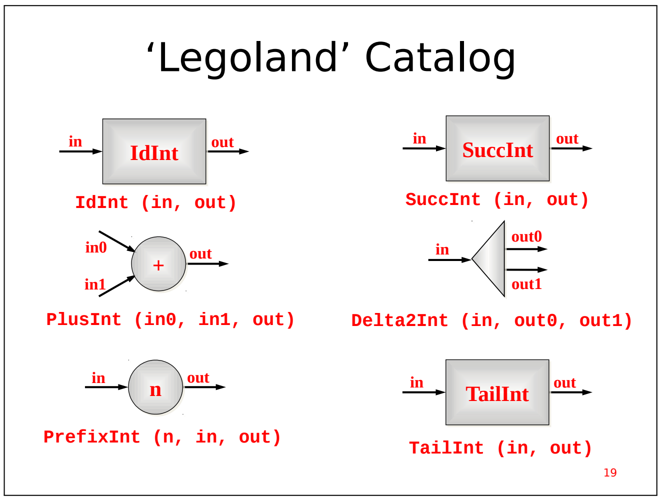


Fig. 4. An example slide from the previously used material for similar courses. This example has been picked as it is superficially similar to the newly designed slides, yet would foster a different style of teaching.

to a discussion rather than as the lesson itself.

The overall design of the slides compares favourably with the previous materials as the new ones are clearer, more condensed displays of relevant information. They act as effective notes for students as the minimalist design helps ensure that the information that is left makes effective, if brief notes as to the key supporting facts for the lessons concepts. As a comparison a sample slide from the older material is shown in figure 4. This slide appears superficially similar the a new ones, but it only displays a variety of available inbuilt CSP cookie-cutter processes[9]. These are rarely used in actual practice and so would be an example of teaching facts for facts sake as they do not lead to any wider conceptual understanding. As a result of this, the slide fosters little discussion beyond a description of the displayed information

One final brief note on the design of the new slides is that they each use the same visual language from the very beginning to the very end. Diagrams were expressed in the same way consistently, meaning that students only needed to decipher one way of reading diagrams. As there is no formal UML definition for network diagrams, previous materials visual languages could change dramatically between diagrams. Focusing on concepts over ideas may be hard enough for some students to follow, so these additional complications should be minimised by using the same style throughout. The designed slides, along with all other course material is available at a public Git repository [21].

XI. Lectures and Practicals

When teaching the lectures, to foster an environment of discussion, affairs were kept fairly informal. For instance, a

[9]These are provided processes that each perform some very basic functionality such as adding together two input numbers. They can be combined together to form more advanced functionality. In practice this is not done as the overhead from having so many processes makes for a bloated and slow system.

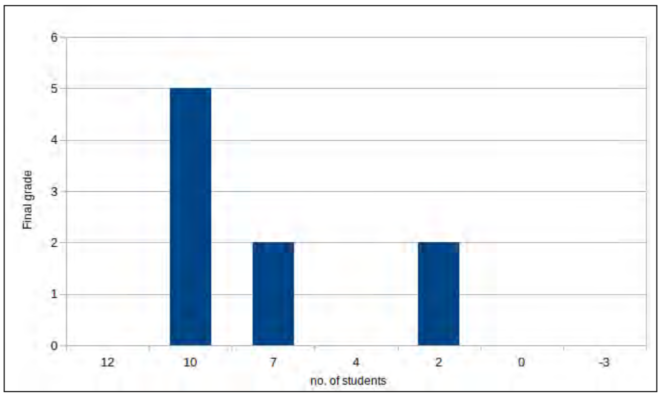


Fig. 5. Final results. Note that only nine students are shown as one is currently in the process of re-sitting the final assignment and so has not yet received a final grade. Initially they received the grade '0'.

conversational tone would be adopted throughout the lesson, with regular comments and observations. Questions from the students should be allowed as soon as they occurred to the students, rather than waiting for the end or some gap in explanations. It is also important that the teacher is open to admitting what they don't know as part of encouraging a conversation [19].

Practicals were relatively unplanned compared to lectures as again they were expected to be more conversational by their very nature. As students should already expect this from a practical rather than a lecture, less effort was needed to foster this specific atmosphere. There was no specific plan set for individual practicals and they were intended mostly as support sessions for help with assignments and troubleshooting any technical issues that emerged.

XII. Results

The course ran as expected with no major issues. Ten students enrolled on the course, though unexpectedly, eight were computer science students. This meant that a higher standard of background computing knowledge could be assumed. There was a reasonable spread of grades after the final exam, as shown in figure 5. These results are certainly higher than the expected bell curve, but with a small sample size and most students having a higher than expected familiarity with the topic, is not surprising or concerning.

The PyNQ boards caused minor software trouble throughout the course. These technical difficulties were never sufficient to derail the course and have now been ironed out so should not provide difficulty in future courses. However, the true problem with them was that they served as a distraction in students reports on their system. When asked to comment on the shortcomings of their designs, most would default to listing simple technical problems, rather than engaging critically with the conceptual problems. Most were able to engage with concepts once verbally prompted. The inability to do so in their report is potentially down to a failure to set expectations at the beginning of the assignment, rather than a fundamental problem of approach. Greater care should be taken in future to communicate what is expected from assignments, with explicit instructions in the assignment handouts.

Students responded positively in their end of course surveys [21]. Most seemed happy with the quality of teaching, but thought that the workload was too light. This may be due to the students being more familiar with the course contents than was anticipated. Students also felt that better use could have been made of the practical sessions. They were intended as informal help-sessions, and so were only lightly attended. This was expected, but a more defined structure with some set exercises may have helped give non-computer scientists more hands on time to get familiar with programming. It also would have filled out the workload. Students in related courses have also reported that having mandatory, short, defined workshop exercises every week helped them learn the topics. Workshops would also particularly suit the presented style of teaching as concepts are developed through practice and playing around with the presented facts. The workshop materials and any work produced in them would also inherently produce good revision and reference material for students if they needed to revise.

Preparing the teaching materials took roughly two weeks of work time spent on just 4 hours of lessons. The sheer length of time taken may be a function of the teachers relative inexperience, but it nevertheless illustrates the extensive planning and preparation required for such an involved teaching style. Naturally, this will reduce with practice but should be kept in mind by any new teaching teams attempting to replicate this approach.

The teaching team also feels that too much material was covered, particularly in the second section. Several types of communication protocols were discussed at length but never used or recommended. More time spent on FPGAs would have better served the students. This would help with the final point of criticism for the course. Throughout the course, but particularly in relation to hardware, the teaching team overestimated the students familiarity with programming concepts. During hardware setup they could be asked to define their board's IP address but no specific instructions on how to do this were provided at first as it was simply assumed that anyone could do this. Again, this can easily be addressed in future iterations of the course through increased support in practical sessions and more explicit instructions/guides in assignment hand outs.

XIII. Reflections by the Teaching Team

It is reasonable to conclude that the course went well. An expected number of students passed, the core topics have been taught, and students responded positively in their end of course surveys. To further evaluate the success of the course, the goals presented in section IX should be considered. These evaluations will rely heavily on the personal reflection by the primary author, who taught this section of the course. Further from student questionnaires and colleagues has also been added where possible and appropriate. Therefore, this section should not be considered scientific and unbiased. However, it was felt that the personal experience of an honest attempt at trying to implement this style of teaching may be illuminating to others considering doing the same, especially as scientific guidance does not, and cannot, meaningfully exist for this [20].

A. G1: Facilitate the teaching of concurrent and parallel concepts

As the most important objective, G1 was kept in mind through the whole process, and was the reason that most previous teaching materials were replaced. Most sections worked extremely well, such as lesson 5 on Determinism and Race Conditions. Almost all of the slides in this lesson are prompts to student interaction, with students guessing at the outcome of some simple programs. These examples illustrate what determinism is so that hopefully by the time a quick slide explaining it through facts is shown, they already have formed the core concept, that can then be reinforced by the necessary facts.

However, several slides through the course turned out more full of explicit information than was first expected, and so became the primary teaching tool within their lessons. This is suspected to have led to less conceptual understanding from the students as the concepts communicated in those lessons did not appear in the final assessment submissions. Greater discipline is needed in limiting facts on slides, and a rule of thumb such as 'only 3 facts per slide' would help keep slides short and force discussion to the fore.

The failure of some slides illustrates the success of others. As mentioned in sections IV and X, concepts require facts to support them and so all facts on the slide should support a concept, rather than being a fact in and of itself. As most slides fostered conversation and with them conceptual understanding (as evidenced by the concepts being well understood in assignments) it is demonstrated that the presented teaching techniques are suitable for this goal.

B. G2: Support the Presented Concepts with Facts

Where the presented concepts fell down however was in the application to parallel programming as a whole. Once the course moved to topics other than CSP it seemed that several of these core concepts were forgotten, or it was not realised by the students that these were broadly applicable concepts rather than just relating to CSP. This is perhaps similar to the shape of the earth problem encountered by Vosniadou and Brewer [22]. The key similarity here is that the children and students appeared to have a complete understanding of the topic when first asked, but actually did not upon closer inspection. Vosniadou and Brewer suggest that the children did not have enough supporting facts for them to form an accurate conception of what the earth looks like when it is said to be round. It could be that a similar problem has occurred here, with insufficient examples provided illustrating the broader applications of the core concepts. Surprisingly, this issue was not limited to the physicists and even appeared with the computer scientists, who were expected to have an existing broad understanding of computing and so be able to apply concepts more accurately. Care should be taken in future to use a wider range of examples to demonstrate that these concepts are bigger than they might otherwise appear.

C. G3: Encourage student engagement in the class through exercises and discussion

Student engagement was fostered throughout the course through the use of exercises, questions, and prompts, rather than as a defined stage within the teaching cycle [7][10]. At certain points this was hard to keep up, and the teaching fell back on explaining things at the students. lesson 7 in particular suffers from this, as it is just an explanation of some common methodologies that exist in other CSP implementations. This was definitely the least successful lesson with very little seemingly being learned from it, judging on the taught material not being present in any of the students submissions. These topics are not notably more difficult than other lectures, nor were they explained worse than other topics. The lack of interaction was the only notable difference, leading to the conclusion that this is why it stuck less in the student's memory than other lessons.

D. G4: Provoke questions and discussion from the students

Students engaged willingly and consistently in discussion of the presented topics, with engagement from the whole cohort. A good way to foster conversation between the students was to set exercises and put them in groups of 2 or 3. This meant that to complete the exercise students were forced to exchange ideas, especially as the tasks where conceptual in nature, such as to design a system.

Better use could have been made of the practical sessions. As they were mainly conceived as trouble shooting sessions, attendance was low and those that did attend mostly did not interact with each other beyond to socialise. Something more structured could have given more of the students a reason to talk about the subject together, and allow for more time to reinforce how to apply the learnt concepts as discussed in section XIII-B.

It is worth noting that the exercise for the second section required the students to come to a mutual agreement on a communication protocol, so that each of their systems could communicate with each others. This section failed as no common agreement was made by the students despite repeated prompting by the teaching team. It may be that this failure was due to insufficient background being presented, and so students did not feel they had an understanding of where to begin. It may also have been that no student wanted to be the one to suggest a protocol that everyone else would have to follow. This vagueness of problem means it is hard to meaningfully reflect on the issue, and so perhaps this style of assignment is simply best avoided in future.

[10]It is worth noting that the models that present very separate, defined stages do not necessarily intend for them to be implemented as such, and often will explicitly state as such.

E. G5: Enable the teacher to explain in their own words, rather than relying on technical definitions

Similar to section XIII-D, this goal was mostly achieved. The slides were bare-bones so that they could not simply be read out, and most topics were explained ahead of displaying the relevant slide. This meant that a personal explanation, usually delivered in plain English acted as the primary introduction to a topic, with the defined points of a slide only introduced at the end to reinforce what was already said. The text that was on the slides also acted as memory prompts so that once an effectively ad-libbed explanation was complete, the slides could be checked to confirm that all essential points had been hit.

F. G6: Be clear and easy to understand

This informal approach meant that explanations or slides could be rather opaque. However, students seemed to follow along at the expected rate, and understood what was being said. This may be down to most student being computer scientists however, and so would potentially more familiar with this subject matter. I would expect that this problem could mostly be addressed by further practice at explaining the subject, and is affected mostly by practice at teaching.

G. G7: Be reusable in subsequent courses, even potentially by others not on the current teaching team

Re-usability was mostly forgotten through material creation, and in hindsight would not have been included as a goal. Much of the ambition of this style of teaching was in improvisation and discussion, with pre-made slides potentially discouraging that. The slides were only ever intended as a visual support to what was expected to be said, which may differ considerably from others lecturers. These slides may be a useful guide or starting point for another lecturers slides, but are not expected to be entirely usable by another lecturer without work.

This leads into the major downside of this approach. That being the length of time taken to produce these lectures. Whilst this process should speed up with experience, it will still need to be repeated for each course, making this a very time intensive form of teaching. The ideas presented within this paper about a concept focus are not new[11], yet it is perhaps this time commitment that limits their wider adoption.

XIV. FUTURE RECOMMENDATIONS

It is recommended that *Concurrent and Distributed Systems* continues in future, and that the ideas put forward in its teaching style are iterated upon. The conceptual basis of the course worked well, and PyCSP, PyZMQ, and SME acted as good illustrators of these concepts. The use of established libraries meant that relatively little time could be spent on simple facts and allowed for discussions both broad and deep about the theoretical underpinnings of distributed systems. One of the primary goals was to make a course for physicists who needed to understand parallel programming, and yet only 20% of the eventual enrolment were physicists. The course description for students should better reflect who it is intended for. Additionally, it may be worth advertising the course directly to physics students, perhaps by making sure supervisors within the various departments at the Niels Bohr Institute are aware of its existence, and are mentioning it to those who may benefit from it.

Masters students can still need considerable prompting to engage critically with material rather than just rote learning. Being very clear about this from the beginning is essential. Even greater emphasis should also be taken on student engagement in lectures. This can be done with exercises and should lead to better learning outcomes. In addition it will foster more conversations by their very nature. In particular, small group exercises during class are extremely good at this, especially when the work is conceptual in nature.

In contrast to the success of the small group exercises, the only assignment that required agreement from the whole cohort did not demonstrate any. Each individual solved the problem in their own way. Although some guesses were made as to what caused this failure, no definitive problem could be found. This might demonstrate that long form group work may not be as effective as the short class exercises. It could also demonstrate that a group of 10 is too big to solve a small problem, and so should have been broken up.

Greater use should be made of the practical sessions by including short programming exercises. For example, after a lesson on deadlock, a short exercise to purposefully implement a deadlocking system would be good. These small exercises could be done in the practical sessions further away from assignment hand-ins to keep the students engaged through the quieter parts of the course. It would also allow them to build up their practical experience with facts they themselves have discovered, and to reinforce the concepts they have just been exposed to.

The PyNQ boards proved to be something of a distraction. They had constant minor technical difficulties, even with the JupyterLab Notebooks. With more experience these issues could be ironed out. It may also be that a more abstract series of assignments that did not have to run on a particular board would be better, as any technical problem proved too tempting students to focus on when reflecting on their own solution, rather than reflections on the concepts within their system. Eliminating the physical element may help address this.

All of this leads to following key recommendations for others seeking to introduce an inductive, concept-based approach to teaching parallel systems. In no particular order:

- The student journey of local parallelisation, distributed parallelisation and finally parallel in hardware works well. This is exemplified by PyCSP, to PyZMQ and then SME, and is even an effective introduction for non-computer scientists.
- Student engagement can be fostered through bare-bones teaching materials which force both the lecturer and the students to actively participate throughout the lecture.

[11]Consider that several references on this paper are over a decade old at this point

- Care must be taken when designing materials that they support active discussion. A non-linear slide show that can be adapted to the flow of conversation is a good example of this.
- Group work and practical assignments are essential for allowing concepts to develop and cement in students mind and should be utilised as much as possible.
- Student may be unused to this style of course and assessment, so the expectations on them should be repeated during course descriptions, introduction lectures and assignment handouts.

None of the techniques suggested in this paper (group activities, discussion, bare-bones slides) are unique to parallel computing and could in theory be applied to any subject matter. However, it does take considerable setup time. Despite this, it is hoped that with practice this preparation time will dramatically reduce, making it a more feasible teaching method.

XV. Conclusions

This paper has presented an account of how the *Concurrent and Distributed Systems* course at the Niels Bohr Institute eScience department has modernised the teaching of distributed systems to physicists. This was done because previous courses were insufficient. They were dry courses, full of facts, and with very little student interaction. By focusing instead on concepts supported by facts, discussion and student engagement it was hoped that learning outcomes could be better achieved, with non-computer scientists achieving a deeper level of understanding in the area of parallel systems. Students were introduced to local concurrent programming, followed by networked concurrent programming, and finally concurrent programming on hardware. This allowed for the core concepts to be introduced early, and then applied to different situations, demonstrating their universality.

This course was judged to be a success and so should be refined and repeated in future years. Lessons learned from it could also be applied to other similar courses. In particular, the bare-bones slides and the regular group activities were helpful in fostering an atmosphere of conversation, and elevated the education beyond a discussion of mere facts. However, there was a failure in setting a correct expectation amongst students at the beginning, and the time taken to prepare these lessons was extensive. Regardless, it is still recommended that the experience gained in teaching this course is carried forward, both in this course as it continues next and in other similar courses.

References

[1] "CUDA," https://developer.nvidia.com/cuda-zone, 2019.
[2] "OpenMP," https://www.openmp.org, 2019.
[3] C. A. R. Hoare, *Communicating Sequential Processes*. http://usingcsp.com/cspbook.pdf: Prentice Hall International, 2015.
[4] "ZeroMq," http://zeromq.org, 2019.
[5] "MPI for Python," https://mpi4py.readthedocs.io/en/stable/, 2019.
[6] E. L. Gettier, "Is Justified True Belief Knowledge?" *Analysis*, vol. 23, no. 6, pp. 121–123, 1963.
[7] N. Entwistle, *Teaching for Understanding at University*. Basingstoke: Palgrave Macmillan, 2009.
[8] J. Biggs and C. Tang, *Teaching for Quality Learning at University*. Maidenhead: Open University Press, 2007.
[9] R. Munk, "Teaching Parallel and Distributed Techniques at UCPH through JupyterLab," 7 2019.
[10] B. Barney, "Message Passing Interface (MPI)," https://computing.llnl.gov/tutorials/mpi/, 2019.
[11] J. M. Perkel, "Programming: Pick up Python," https://www.nature.com/news/programming-pick-up-python-1.16833, 2015.
[12] D. Robinson, "Why is Python Growing So Quickly?" https://stackoverflow.blog/2017/09/14/python-growing-quickly/, 2017.
[13] "PyCSP," https://pypi.org/project/pycsp/, 2016.
[14] B. Vinter and M. O. Larsen, "Teaching Concurrency: 10 years of Programming Projects at UCPH," in *Communicating Process Architecture 2017*, K. Chalmers and J. B. Pedersen, Eds. IOS Press, 2017, pp. 135–156.
[15] "PyZMQ," https://github.com/zeromq/pyzmq, 2019.
[16] B. Vinter and K. Skovhede, "Synchronous Message Exchange for Hardware Designs," in *Communicating Process Architectures 2014*, P. H. W. et al, Ed. Open Channel Publishing, 8 2014, pp. 169–179.
[17] "PYNQ: Python Productivity for ZYNQ," http://www.pynq.io, 2019.
[18] M. Prince, "Does Active Learning Work? A Review of the Research," *The Research Journal for Engineering Education*, vol. 93, no. 3, pp. 223–231, 2004.
[19] P. H. Scott, E. F. Mortimer, and O. G. Aguiar, "The Tension Between Authoritative and Dialogic Discourse: A Fundamental Characteristic of Meaning Making Interactions in High School Science Lessons," *The Research Journal for Engineering Education*, vol. 95, no. 2, pp. 123–138, 2006.
[20] M. J. Prince and R. M. Felder, "Inductive Teaching and Learning Methods: Definitions, Comparisons, and Research Bases," *The Research Journal for Engineering Education*, vol. 95, no. 2, pp. 123–138, 2006.
[21] "Concurrent and Distributed Systems course material," https://github.com/PatchOfScotland/ConcurrentAndDistributedCourseMaterial, 2019.
[22] S. Vosniadou and W. F. Brewer, "Mental Models of the Earth: A Study of Conceptual Change in Childhood," *Cognitive Psychology*, vol. 24, pp. 535–585, 1992.

D

CLOUD ENABLING EDUCATIONAL PLATFORMS WITH CORC

This paper, and a presentation describing it were originally presented at the 8th Workshop on Cloud Technologies in Education (CTE 2020)[16]. It was first published as part of the proceedings[69].

Cloud enabling educational platforms with corc

Rasmus Munk[a], David Marchant[b] and Brian Vinter[c]

[a]*Niels Bohr Institute, Blegdamsvej 17, Copenhagen, 2100, Denmark*
[b]*Niels Bohr Institute, Blegdamsvej 17, Copenhagen, 2100, Denmark*
[c]*Aarhus University, Ny Munkegade 120, Aarhus C, 8000, Denmark*

Abstract
In this paper, it is shown how teaching platforms at educational institutions can utilize cloud platforms to scale a particular service, or gain access to compute instances with accelerator capability such as GPUs.

Specifically at the University of Copenhagen (UCPH), it is demonstrated how the internal JupyterHub service, named Data Analysis Gateway (DAG), could utilize compute resources in the Oracle Cloud Infrastructure (OCI). This is achieved by utilizing the introduced Cloud Orchestrator (corc) framework, in conjunction with the novel JupyterHub spawner named MultipleSpawner. Through this combination, we are able to dynamically orchestrate, authenticate, configure, and access interactive Jupyter Notebooks in the OCI with user defined hardware capabilities. These capabilities include settings such as the minimum amount of CPU cores, memory and GPUs the particular orchestrated resources must have. This enables teachers and students at educational institutions such as UCPH to gain easy access to the required capabilities for a particular course. In addition, we lay out how this groundwork, will enable us to establish a Grid of Clouds between multiple trusted institutions. This enables the exchange of surplus computational resources that could be employed across their organisational boundaries.

Keywords
Teaching, Cloud Computing, Grid of Clouds, Jupyter Notebook

1. Introduction

The availability of required computational resources in organisations, such as scientific or educational institutions, is a crucial aspect of delivering the best scientific research and teaching. When teaching courses involving data analysis techniques it can be beneficial to have access to specialized platforms, such as GPU accelerated architectures.

At higher educational institutions, such as the University of Copenhagen (UCPH) or Lund University (LU), these centers are substantial investments, that are continuously maintained and upgraded. However, the usage of these resources often varies wildly between being fully utilized to sitting idly by.

We therefore propose, that these institutional resources be made available (with varying priority) across trusted educational and scientific organisations. Foremost, this is to enable the voluntary sharing of underused resources to other institutions, thereby potential establishing greater scalability than can be found within each individual institution.

CTE 2020: 8th Workshop on Cloud Technologies in Education, December 18, 2020, Kryvyi Rih, Ukraine
EMAIL: rasmus.munk@nbi.ku.dk (R. Munk); d.marchant@ed-alumni.net (D. Marchant); vinter@au.dk (B. Vinter)
URL:
https://research.ku.dk/search/result/?pure=en%2Fpersons%2Frasmus-munk(a72145a7-0203-4791-bb8f-6073bc23fe2f).html (R. Munk); https://research.ku.dk/search/result/?pure=en%2Fpersons%2Fdavid-gray-marchant(ff6af890-33df-4414-9a9d-c3d33258ad1f).html (D. Marchant); https://pure.au.dk/portal/en/persons/brian-vinter(a4fe861f-5a04-4e93-a5b3-c633045eb82e).html (B. Vinter)
ORCID: 0000-0003-0333-4295 (R. Munk); 0000-0003-4262-7138 (D. Marchant); 0000-0002-3947-9878 (B. Vinter)

1.1. Basic IT

Within institutions such as UCPH, there is a mixture of services that each provides. At the very basic level, there are infrastructure services such as networking, account management, email, video conferencing, payroll management, license management, as well OS and software provisioning. In this paper, we define these as Basic IT services. At educational institutions, additional services can be added to this list, these include services for handling student enrollment, submissions, grading, course management, and forum discussions. As with the initial Basic IT services, these are typically off the shelf products that needs to be procured, installed, configured and maintained on a continuous basis.

A distinguishing trait of Basic IT services, in an education context, is that they are very predictable in terms of the load they will exhibit, both in times of high and low demand. For instance, there will be busy junctions, such as assignment hand in days, release of grades, student enrollment, and so on. In contrast, holiday and inter-semester periods will likely experience minor to no usage. Given this, these services are classic examples of what cloud computing was developed to provide. Efficient utilization of on-demand resources, with high availability and scalability to handle fluctuating usage in a cost effective manner.

1.2. Science IT

Science IT services, in contrast, revolve around the institutions scientific activities whether by researchers or students. They include services such as management, sharing, transferring, archiving, publishing, and processing of data, in order to facilitate the scientific process. In addition, these facilities also enable lecturers to utilize their research material in courses, giving students access to the same platform and resources.

What distinguishes these services, is that they impose different constraints compared to Basic IT services. These typically involve areas such as, computational load, security, budgetary, scientific, and legal requirements, among others. For example, it is often too inefficient, or costly to utilize public cloud resources for the storing and processing of large scientific datasets at the petabyte scale. In this case, a more traditional approach such as institutional compute resources is required. [1].

Research fields such as climate science [2], oceanography [3], and astronomy [4], often employ experimental simulations as a common scientific tool. These simulations produce output up to petabytes in size, that still need to be stored for subsequent postprocessing and analysis. Upon a scientific discovery from this process, the resulting datasets needs to be archived in accordance with regulatory requirements, which in the case of UCPH is 5 years [5] (only available in Danish).

1.3. Institutional Resources

High Performance Computing (HPC) and regular compute centers are often established at higher educational institutions to provide Science IT services. The UCPH [6], University of Antwerp [7], and LU [8] compute centers are examples of this. In addition, institutions can also gain access to similar resources through joint facilities like the Vienna Scientific Cluster [9], which supports 19 institutions, 10 of which are higher educational institutions. Finally there are national and pan-national resources such as ARCHER (UK) [10] or the EuroHPC [11] that review applications before access is granted.

These established centers are very expensive to build and have a limited lifespan before they need to be replaced. Even smaller educational compute platforms follow a similar life-cycle. For instance, at the UCPH a typical machine has a lifetime of 5 years before it needs to be replaced. This is whether the machine has been heavily utilized or not. Therefore, it is important that these systems across institutions are utilized, not only efficiently, but at maximum capacity throughout their lifetime.

For organising the sharing of resources across trusted educational and scientific organisations, inspiration is drawn from the way traditional computational Grids have been established [12]. The difference is, that instead of establishing a Grid where individual resources are attached, this model will instead be based on each institution establishing a Cloud of resources that are shared via a Grid. This means that the Grid is responsible for interconnecting disjointed clouds, whether they be institutional or public cloud platforms. The result being an established model for sharing cloud resources across educational institutions in support of cloud services for bachelor and master courses, general workshops, seminars and scientific research.

In this paper, we present how an existing teaching and research service at UCPH could be enabled with access to a cloud framework, which is the first step towards a Grid of Clouds resources. We accomplish this by using the Cloud Orchestrator (corc) framework [13]. Through this, we are able to empower the DAG service with previously inaccessible compute resources across every course at UCPH. This was previously not feasible with internal resources alone. Since we do not have access to other institutional resources at this point in time, we utilized a public cloud provider to scale the service with external resources.

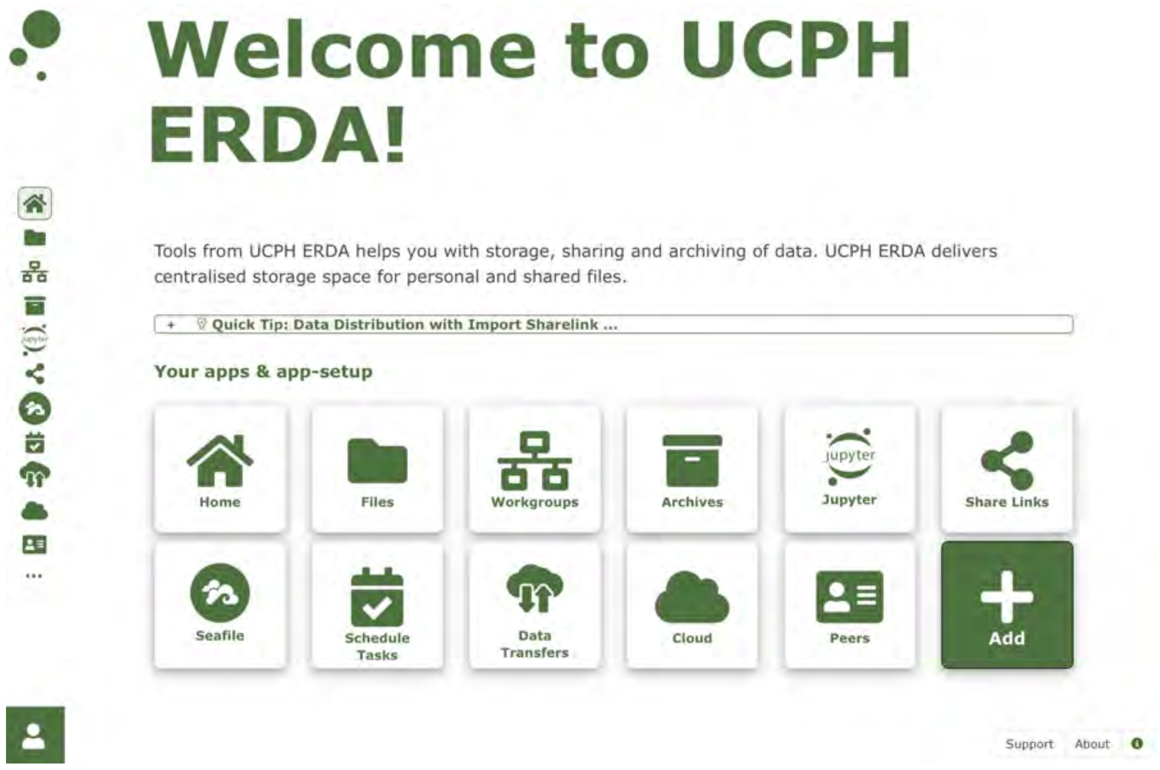
2. Background

At the Niels Bohr Institute (NBI), part of UCPH, we host a number of Science IT services that are part of providing a holistic educational platform for researchers, teachers, students, and general staff. A subset of these Science IT services have been especially beneficial across all levels of teaching. Namely, services such as the University Learning Management System (LMS), called Absalon, which is based on Canvas [14] for submissions and grading. The Electronic Research Data Archive (ERDA) [15] for data management and sharing tasks. In addition to the Data Analysis Gateway (DAG) [16], which is a JupyterHub powered platform for interactive programming and data processing in preconfigured environments.

2.1. Teaching Platforms

The combination of these subset services, in particular the combination of ERDA and DAG, has been especially successful. Teachers have used these to distribute course material through ERDA, which made the materials available for students to work on at the outset of the course. This ensures that students can get on with the actual learning outcomes from the get go, and not spend time on tedious tasks such as installing prerequisite software for a particular course. Due to budgetary limitations, we have only been able to host the DAG service with standard servers, that don’t give access to any accelerated architectures.

Across education institutions, courses in general have varying requirements in terms of computing resources, environments, and data management, as defined by the learning outcomes of the course. The requirements from computer science, data analysis, and physics oriented courses are many, and often involve specialized compute platforms. For example, novel data analysis techniques, such as Machine Learning or Deep Learning have been employed across a wide range of scientific fields. What is distinct about these techniques is the importance of the underlying compute platform on which it is being executed. Parallel architectures such as GPUs in particular are beneficial in this regard, specifically since the amount of independent linear systems that typically needs to be calculated to


Welcome to UCPH ERDA!

Tools from UCPH ERDA helps you with storage, sharing and archiving of data. UCPH ERDA delivers centralised storage space for personal and shared files.

+ Quick Tip: Data Distribution with Import Sharelink ...

Your apps & app-setup

Figure 1: ERDA Interface

give adequate and reliably answers are immense. The inherent independence of these calculations, makes them suitable for being performed in parallel, making it hugely beneficial to utilize GPUs. [17].

Given that the DAG service was an established service at UCPH for data analysing and programming in teaching bachelor and master students, it seemed the ideal candidate to enable with access to cloud resources with accelerator technology. For instance, courses such as Introduction to Computing for Physicists (abbreviated to DATF in Danish) [18], Applied Statistics: From Data to Results (APP-STAT) [19], and High Performance Parallel Computing (HPPC) [20], all would benefit from having access to GPU accelerators to solve several of the practical exercises and hand-in assignments.

2.2. ERDA

ERDA provides a web based data management platform across UCPH with a primary focus on the Faculty of Science. Its primary role is to be a data repository for all employees and students across UCPH. Through a simple web UI powered by a combination of an Apache webserver and a Python based backend, users are able to either interact with the different services through its navigation menu, or a user's individual files and folders via its file manager. An example of the interface can be seen in Figure 1. The platform itself is a UCPH-specific version of the open source Minimum Intrusion Grid (MiG) [21], that provides multiple data management functionalities. These functionalities includes easy and secure upload of datasets, simple access mechanisms through a web file manager, and the ability to establish collaboration and data sharing between users through Workgroups.

2.3. Jupyter

Project Jupyter [22] develops a variety of open source tools. These tools aim at supporting interactive data science, and scientific computing in general. The foundation of these is the IPython Notebook (.ipynb) format (evolved out of the IPython Project [23]). This format is based on interpreting special segments of a JSON document as source code, which can be executed by a custom programming language runtime environment, also known as a kernel. The JupyterLab [24] interface (as shown in Figure 2) is the standard web interface for interacting with the underlying notebooks. JupyterHub [25] is the de-facto standard to enable multiple users to utilize the same compute resources for individual Jupyter Notebook/Lab sessions. It does this through its own web interface gateway and backend database, to segment and register individual users before allowing them to start a Jupyter session.

In addition, JupyterHub allows for the extension of both custom Spawners and Authenticators, enabling 3rd party implementations. The Authenticator is in charge of validating that a particular request is from an authentic user. The responsibility of the Spawner is how a Jupyter session is to be scheduled on a resource. Currently there exist only static Spawners that utilize either preconfigured resources that have been deployed via Batch, or Container Spawners, or at selective cloud providers such as AWS [26]. As an exception to this, the WrapSpawner [27] allows for dynamic user selections through predefined provides. However, these profiles cannot be changed after the JupyterHub service is launched, making it impossible to dynamically change the set of supported resources and providers. Therefore it would be of benefit if a Spawner extended the WrapSpawner's existing capabilities with the ability to dynamically add or remove providers and resources.

3. Related Work

As presented in [28], Web-based learning by utilizing cloud services and platforms as part of the curriculum is not only feasible, but advisable. In particular, when it comes to courses with programming activities for students, educational institutions should enable access to innovative Web-based technologies that supports their learning. These include interactive programming, version control and automated programming assessments to ensure instant feedback.

3.1. Interactive Programming Portals

Research in cloud computing for education typically revolves around using Web-enabled Software as a Service (SaaS) applications. Examples of such include platforms such as GitHub [29], Google Docs [30], Google Colaboratory [31], Kaggle [32], and Binder [33]. Each of these can fill a particular niche in a course at the teacher's or student's discretion. Nevertheless, the provided capability often does come with its own burdens, in that the administration of the service is often left to the teaching team responsible for the course. This responsibility typically includes establishing student access, course material distribution to the specific platform, guides on how to get started with the service and solving eventual problems related to the service throughout the course. In addition, many of the external cloud services that offer free usage, often have certain limitations, such as how much instance utilisation a given user can consume in a given time span. Instead, providing such functionalities as Science IT services, could reduce these overheads and enable seamless integration into the courses. Furthermore, existing resources could be used to serve the service by scaling through an established Grid of Clouds.

In terms of existing public cloud platforms that can provide Jupyter Notebook experiences, DAG is similar to Google Colaboratory, Binder, Kaggle, Azure Notebooks [34], CoCalc [35], and Datalore

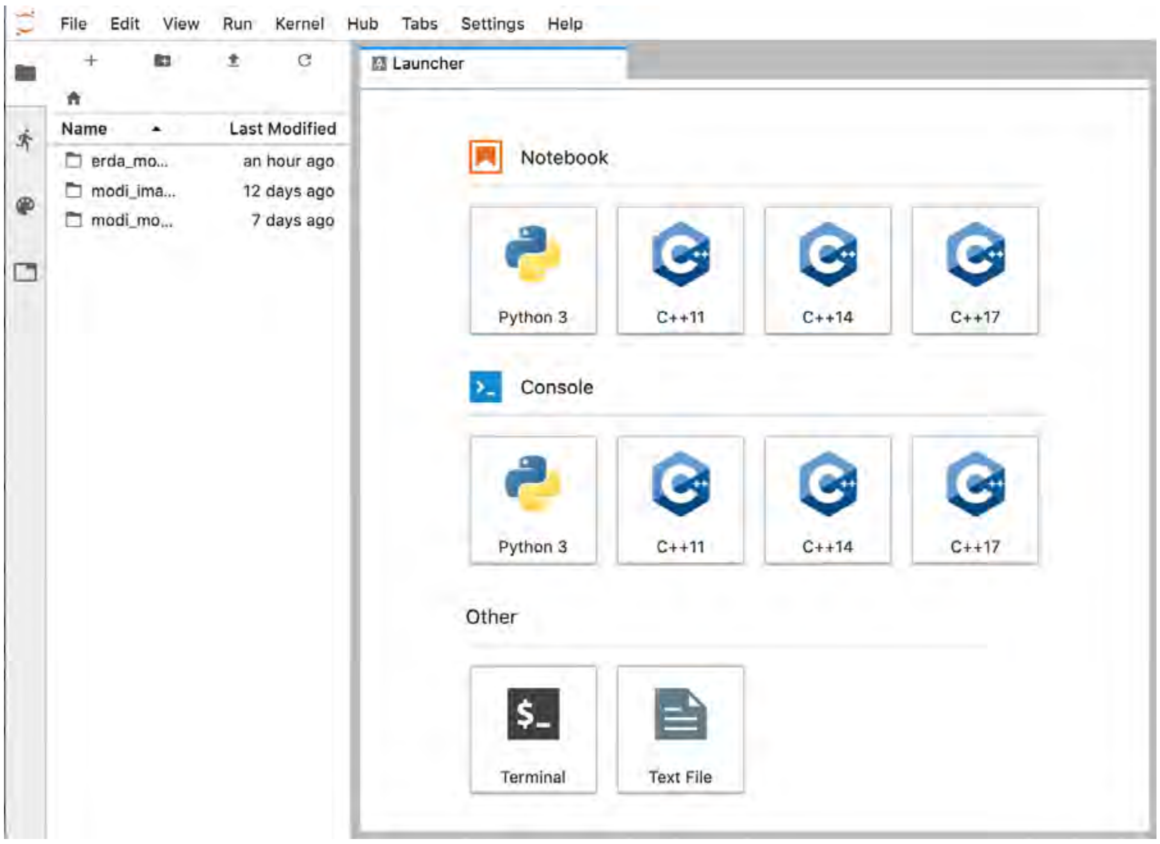


Figure 2: JupyterLab Interface

[36]. All of these online options, have the following in common. They all have free tier plans available with certain hardware and usage limitations. All are run entirely in the web browser and don't require anything to be installed locally. At most they require a valid account to get started. Each of them present a Jupyter Notebook or Notebook like interface, which allows for both export and import of Notebooks in the standard format. An overview of a subset of the supported features and usage limits across these platforms can be seen in Table 1, and their hardware capabilities in Table 2. From looking at the features, each provider is fairly similar in terms of enabling Languages, Collaborating, and Native Persistence (i.e. the ability to keep data after the session has ended). However, there is a noticeable difference, in the maximum time (MaxTime) that each provider allows a given session to be inactive before it is stopped. With CoCalc being the most generous, allowing 24 hours of activity before termination. In contrast, internal hosted services such as DAG allow for the institution to define this policy. At UCPH, we have defined this to be 2 hours of inactivity, and an unlimited amount of active time for an individual session. However, as Table 2 shows, we currently don't provide any GPU capability, which is something that could be changed through the utilisation of an external cloud with GPU powered compute resources.

Given this, the DAG service seemed as the ideal candidate to empower with external cloud resources. Both because it provides similar features as the public cloud providers in terms of Languages and Collaborate ability, but also since it is integrated directly with UCPHs data management service.

Table 1
Subset of Jupyter Cloud Platforms Features

Provider	Native Persistence	Languages	Collaborate	MaxTime (inactive,max)
Binder[37]	None	User specified [1]	Git	10m, 12h[2]
Kaggle [38]	Kaggle Datasets	Python3,R	Yes	60m, 9h
Google Colab [39]	GDrive, GCloud Storage	Python3,R	Yes	60m,12h* [3]
Azure Notebooks [40] [41]	Azure Libraries	Python{2,3},R,F#	NA	60m,8h* [4]
CoCalc [42]	CoCalc Project	Python{2,3},R,Julia,etc	Yes*	30m, 24h
Datalore [43]	Per Workbook	Python3	Yes	60m, 120h [5]
DAG [44]	ERDA	Python2,3,R,C++,etc	Yes	2h, unlimited [6]

Table 2
Hardware available on Jupyter Cloud Platforms

Provider	CPU	Memory (GB)	Disk Size (GB)	Accelerators
Binder	NA	1 Min, 2 MAX	No specified limit*	None
Kaggle1	4 cores	17	5	None
Kaggle2	2 cores	14	5	GPU [7] or TPU [8] [45]
Google Colab Free	NA	NA	GDrive 15	GPU or TPU (thresholded access)
Azure Notebooks (per project)	NA	4	1	GPU (Pay)
Cocalc (per project)	1 shared core	1 shared	3	None
Datalore	2 cores	4	10	None
DAG	8 cores	8	unlimited [9]	None

3.2. Cloud Orchestration

Cloud resources are typically provided by the infrastructure service through some form of orchestration. Orchestration is a term for providing an automated method to configure, manage and coordinate computer systems [46]. Through orchestration, an organisation or individual is able to establish a complex infrastructure through a well defined workflow. For instance, the successful creation of a compute node involves the processing of a series of complex tasks that all must succeed. An example of such a workflow can be seen in Figure 3. Here a valid Image, Shape, Location and Network has to be discovered, selected, and successfully utilized together in order for the cloud compute node to be established. An Image is the target operating system and distribution, for instance Ubuntu 20.04 LTS. A Shape is the physical configuration of the node, typically involving the amount of CPU cores, memory and potential accelerators. Location is typically the physical location of where the resource is to be created. Cloud providers often use the term Availability Zone instead but it generally defines which datacenter to utilize for the given task. Network encompasses the entirety of the underlying network configuration, including which Subnet, Gateway, and IP address the compute node should utilize. In the context of a federated network like a Grid, the orchestration would ideally involve the automated provisioning of the computational resource, the configuration of said resource, and ensure that the resource is correctly reachable through a network infrastructure.

Multiple projects have been developed that automate development and system administration tasks such as maintenance, testing, upgrading, and configuration. These includes packages such as TerraForm [47], Puppet [48], Chef [49], and Ansible [50], all of which open source projects that can be utilized across a range of supported cloud providers. Nevertheless, in terms of enabling workflows that can provide orchestration capabilities, these tools are limited in that they typically only focuses on a

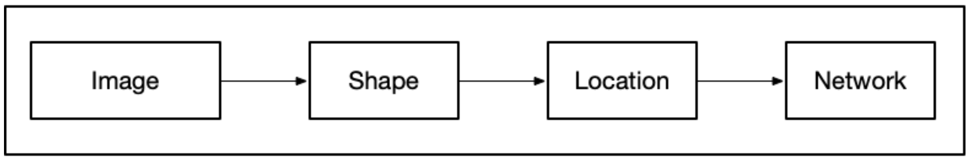


Figure 3: Workflow for orchestrating a compute node

subset of the orchestration functionalities such as provisioning and deployment or configuration and maintenance. For instance TerraFrom is a tool that focuses on infrastructure deployment whereas Puppet, Chef and Ansible are primarily concerned with configuration and maintenance of existing systems. In contrast commercial cloud providers typically also provide their own orchestration-like tools and Software Development Kits (SDK)s, enabling the ability to interact with their respective cloud system. For instance, Oracle provides the Oracle Cloud Infrastructure CLI [51] tool that can interact with their infrastructure. The same applies to the Amazon AWS CLI [52], in addition to a vast complement of tool-kits [53] that provide many different AWS functionalities including orchestration. In contrast, commercial cloud provided tools are often limited to only support the publishing cloud vendor and do not offer cross-cloud compatibility, or the ability to utilize multiple cloud providers interchangeably.

Cloud orchestration developments for the scientific community, especially those aiming to provide cross-cloud deployments, have mostly been based on utilizing on premise cloud IaaS platforms such as OpenStack [54] and OpenNebula [55]. Developments have focused on providing higher layers of abstraction to expose a common APIs that allow for the interchangeable usage of the underlying supported IaaS platforms. The infrastructure is typically defined in these frameworks through a Domain Specific Language (DSL) that describes how the infrastructure should look when orchestrated. Examples of this include cloud projects such as INDIGO-cloud [56] [57], AgroDAT [58] and Occupus [58]. These frameworks, nonetheless do not allow for the utilization of commercial or public cloud platforms, since they rely on the utilization of organisationally defined clouds that are traditionally deployed, managed, and hosted by the organisation itself. Although required, if as stated, we are to establish a Grid of Clouds which should allow for the inclusion of public and commercial cloud platforms. The corc framework was developed and designed to eventually support the scheduling of cloud resources across both organisations and public cloud providers.

4. The first cloud enabled service

To establish a Grid of Cloud resources, we started with enabling the usage of a single public cloud provider to schedule DAG Notebooks on. Through this we created the foundations for the eventual Grid structure that would allow the resources to be scheduled across multiple clouds and organisations.

4.1. Corc

The corc framework was implemented as a Python package. The package establishes the foundations for essential functions such as orchestration, computation, configuration, and authentication against supported cloud providers and cloud resources. Overall, corc is a combination of an Infrastructure as a Service (IaaS) management library, and a computation oriented scheduler. This enables the ability

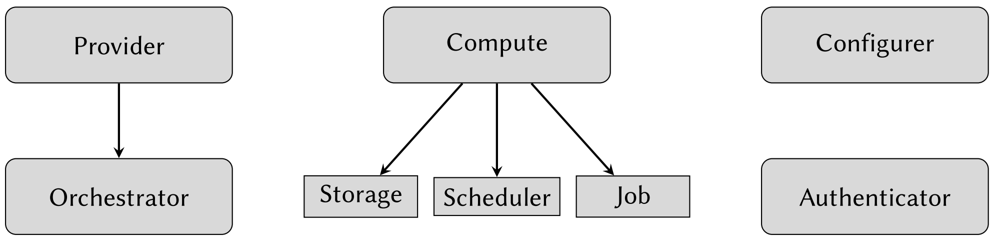


Figure 4: Cloud Orchestrator Framework Overview

to schedule services on a given orchestrated resource. An overview of the architecture can be seen in Figure 4.1.

The first provider to be integrated into the framework was the OCI IaaS. This was chosen, because the UCPH had a preexisting collaboration with Oracle, that enabled the usage of donated cloud resources for testing and development. As also highlighted, this does not limit the integration of other cloud providers into the framework, which the framework was designed for. Furthermore, as explored in section 2.3. A new Spawner, named MultipleSpawner was introduced, to provide the necessary dynamic selection of cloud providers.

As Figure 4.1 indicates, for each provider that corc supports, an orchestrator for that provider needs to be defined within corc. In addition, the framework defines three other top level components, namely Compute, Configurer, and Authenticator. All three are abstract definitions allowing for specific implementations to support the targeted resources which they apply to. A service can therefore be enabled with the ability to utilize cloud resources by integrating the corc components into the service itself. This method is limited to services that are developed in Python. In addition, corc also defines a Command Line Interface (CLI), that can be used to interact with the cloud provided resources directly. Details about how the framework and CLI can be used will not be presented in this paper, but can be found in [13].

```
{
    "virtual_machine": [
        {
            "name": "oracle_linux_7_8",
            "provider": "oci",
            "image": "Oracle Linux 7.8"
        }
    ]
}
```

Listing 1: Spawner Deployment configuration

4.2. MultipleSpawner

MultipleSpawner [59] is a Python package allowing for the selection of dynamic Spawners and resources. Structurally, it is inspired by the WrapSpawner [27], through the MultipleSpawner integrates corc into the Spawner ifself. This enables the JupyterHub service to manage and utilize cloud resources on a dynamic set of providers. In order to enable the MultipleSpawner to support these dynamic resources providers, two JSON configuration files needs to be defined. One of these is shown in Listing 1, and defines the specific resource type that should be deployed on the provider. Currently the MultipleSpawner supports deploying, 'virtual_machine', 'container', and 'bare_metal' resources. The other configuration file is shown in Listing 2. It defines the template configuration settings that specify which Spawner, Configurer, and Authenticator the MultipleSpawner should use to spawn, configure and connect to the deployed resource.

```
[
    {
        "name": "VirtualMachine Spawner",
        "resource_type": "virtual_machine",
        "providers": ["oci"],
        "spawner": {
            "class": "sshspawner.sshspawner.SSHSpawner",
            "kwargs": {
                "remote_hosts": ["{endpoint}"],
                "remote_port": "22",
                "ssh_keyfile": "~/.corc/ssh/id_rsa",
                "remote_port_command": "/usr/bin/python3
                /usr/local/bin/get_port.py"
            }
        },
        "configurer": {
            "class": "corc.configurer.AnsibleConfigurer",
            "options": {
                "host_variables": {
                    "ansible_user": "opc",
                    "ansible_become": "yes",
                    "ansible_become_method": "sudo",
                    "new_username": "{JUPYTERHUB_USER}"
                },
                "host_settings": {
                    "group": "compute",
                    "port": "22"
                },
                "apply_kwargs": {
                    "playbook_path": "setup_ssh_spawner.yml"
                }
            }
        },
        "authenticator": {
            "class": "corc.authenticator.SSHAuthenticator",
            "kwargs": {"create_certificate": "True"}
        }
    },
]
```

Listing 2: Spawner Template configuration

5. Results

By integrating corc into the MultipleSpawner, we enabled the architecture shown in Figure 5, where the DAG service is able to dynamically schedule Jupyter Notebooks across the two resource providers. As is indicated by Figure 5, the UCPH and OCI providers are defined to orchestrate resources, in this case cloud compute instances, in preparation for scheduling a requested Notebook. In order to validate that the architecture worked as expected, we setup a test environment on a separate machine. This machine was configured with a corc and JupyterHub environment, where OCI was defined as a corc provider and the MultipleSpawner as the designated JupyterHub Spawner. With this in order, the JupyterHub service was ready to be launched on the machine.

The MultipleSpawner was configured to use the template and deployment settings defined in Listing 1 and 2. This enables the MultipleSpawner to create Virtual Machine cloud resources at the OCI. Subsequently, the MultipleSpawner uses the SSHSpawner [60] created by the National Energy Research Scientific Computing (NERSC) Center to connect and launch the Notebook on the orchestrated resource. Prior to this, it uses the corc defined SSHAuthenticator and AnsibleConfigurer to ensure that the MultipleSpawner can connect to a particular spawned resource and subsequently configure it with the necessary dependencies.

An example of a such a spawn with the specified requirements can be seen in Figure 6. To validate that this resource had been correctly orchestrated, the corc CLI was utilized to fetch the current allocated resources on OCI. Listing 3 shows that an instance with 12 oracle CPUs, 72 GB of memory and one NVIDIA P100 GPU had been orchestrated. This reflects the minimum shape that could be found in the EU-FRANKFURT-1-AD-2 availability domain that met the GPU requirement.

```
rasmusmunk$ corc oci orchestration instance list
{
    "instances": [
    {
        ...
        "availability_domain": "lfcb:EU-FRANKFURT-1-AD-2",
        "display_name": "instance20201018103638",
        "image_id": "ocid1.image.oc1.eu-frankfurt....",
        "shape": "VM.GPU2.1",
        "shape_config": {
            ...
            "gpus": 1,
            "max_vnic_attachments": 12,
            "memory_in_gbs": 72.0,
            "ocpus": 12.0,
        },
    }
    ],
    "status": "success"
}
```

Listing 3: Running OCI Notebook Instance

As shown in Figure 7, the JupyterHub spawn action redirected the Web interface to the hosted Notebook on the cloud resources. Relating this to the mentioned courses at UCPH, this then enabled the students with access to an interactive programming environment via the JupyterLab interface.

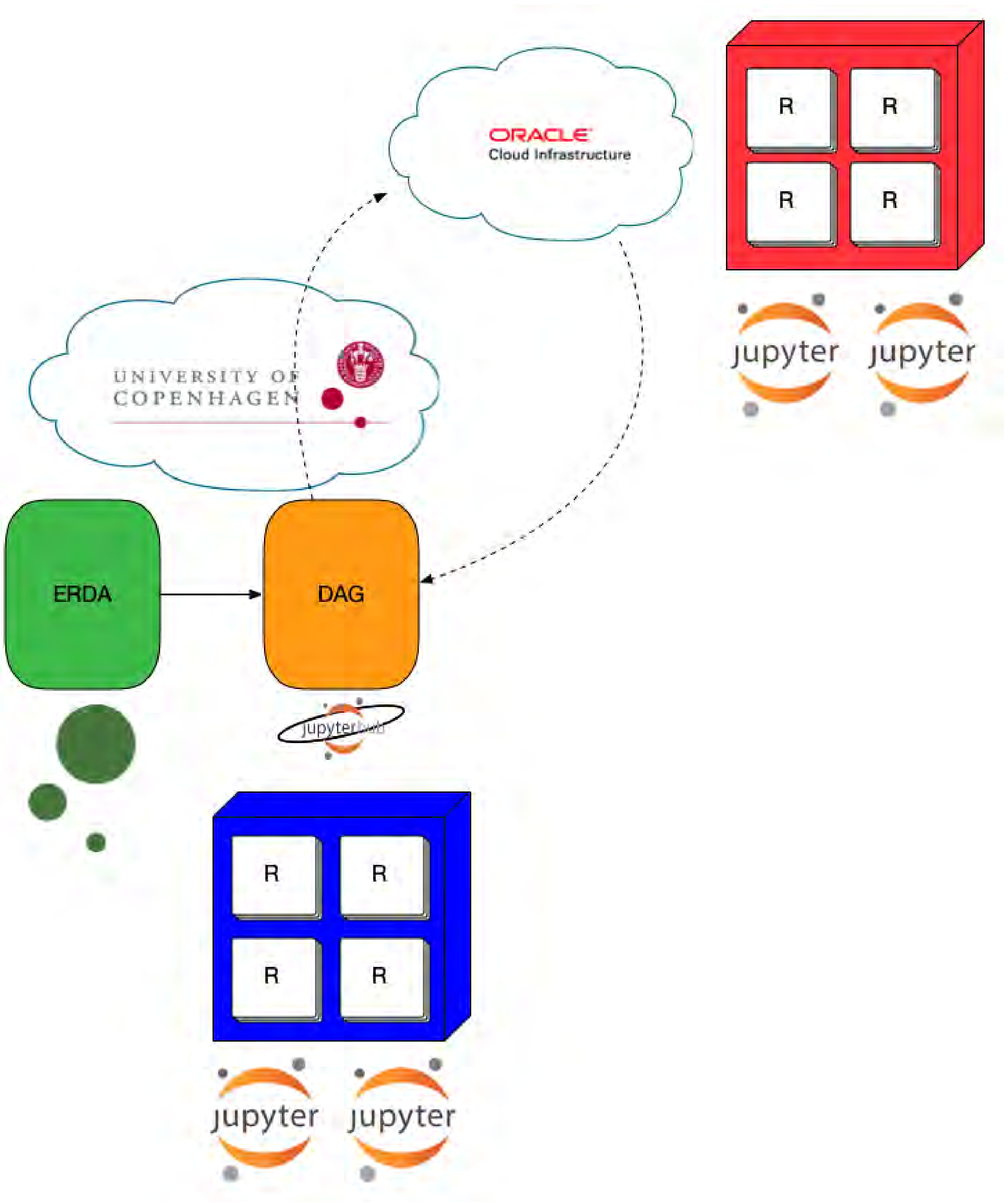
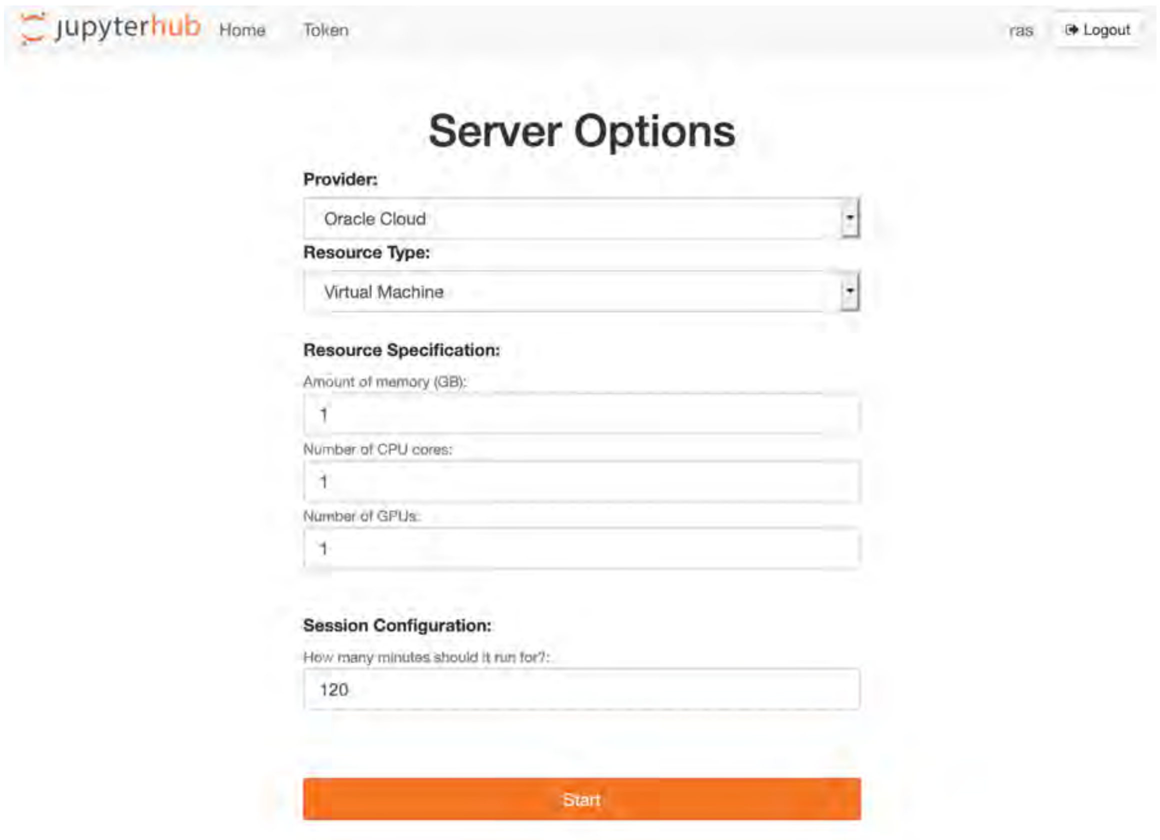


Figure 5: DAG MultipleSpawner Architecture, R = Resource

Building upon this, a simple benchmark was made to evaluate the gain in getting access to a compute resource with a NVIDIA P100 GPU. A Notebook with the Tensorflow and Keras quick start application [61] was used to get a rough estimate of how much time would be saved in building a simple neural network that classifies images. Listing 5, shows the results of running the notebook on the GPU powered compute resource for ten times in a row, and Listing 4 shows the results of running



Server Options

Provider: Oracle Cloud

Resource Type: Virtual Machine

Resource Specification:

Amount of memory (GB): 1

Number of CPU cores: 1

Number of GPUs: 1

Session Configuration:

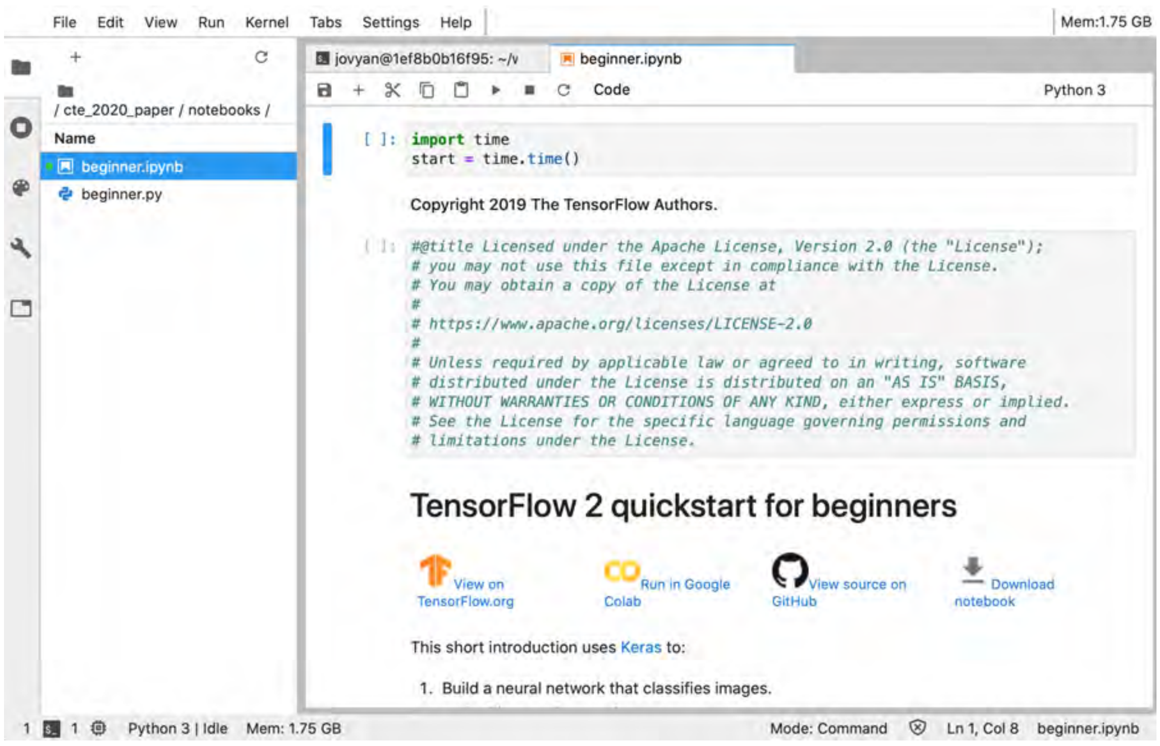
How many minutes should it run for?: 120

Figure 6: MultipleSpawner Interface

the same benchmark on an existing DAG resource. As this shows, the GPU version was on average 24,7 seconds faster or in other words gained on average a 2,8 speedup compared to the DAG resource without a GPU.

```
(python3) jovyan@d203812f76e8:~/work/cte_2020_paper/notebooks$ \
> python3 beginner.py
Took: 38.107945919036865
Took: 36.123350381851196
Took: 37.37455701828003
Took: 37.69051790237427
Took: 41.16242790222168
Took: 37.24052095413208
Took: 38.685391902923584
Took: 40.02782320976257
Took: 38.40936994552612
Took: 39.34704780578613
Average: 38.41689529418945
```

Listing 4: DAG compute resource Tensorflow times



```
[ ]: import time
     start = time.time()
```

Copyright 2019 The TensorFlow Authors.

```
[ ]: #@title Licensed under the Apache License, Version 2.0 (the "License");
     # you may not use this file except in compliance with the License.
     # You may obtain a copy of the License at
     #
     # https://www.apache.org/licenses/LICENSE-2.0
     #
     # Unless required by applicable law or agreed to in writing, software
     # distributed under the License is distributed on an "AS IS" BASIS,
     # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     # See the License for the specific language governing permissions and
     # limitations under the License.
```

TensorFlow 2 quickstart for beginners

This short introduction uses Keras to:

1. Build a neural network that classifies images.

Figure 7: A Tensorflow + Keras Notebook on an OCI resource

```
(python3) jovyan@56e3c30c2af6:~/work/cte_2020_paper/notebooks$ \
> python3 beginner.py
Took: 19.479900360107422
Took: 12.859123706817627
Took: 13.047293186187744
Took: 13.296776056289673
Took: 13.002363204956055
Took: 13.118329048156738
Took: 13.067508935928345
Took: 13.089284658432007
Took: 13.160099506378174
Took: 13.032178401947021
Average: 13.715285706520081
```

Listing 5: OCI GPU compute resource Tensorflow times

From this simple benchmarking example, we can see that by utilizing the MultipleSpawner in combination with corc, users are able to get access through a simple gateway to the expected performance gains of accelerators like a GPU. Expanding on this, the teachers and students at UCPH will now be able to request a compute resource with a GPU on demand, thereby gaining simple access to achieving similar faster runtimes in their exercises and assignments.

6. Conclusions and Future Work

In this paper, we presented our work towards establishing a Grid of Clouds that enables organisations, such as educational institutions to share computational resources amongst themselves and external collaborators. To accomplish this, we introduced corc as a basic building block enables the ability to orchestrate, authenticate, configure, and schedule computation on a set of resources by a supported provider.

OCI was the first provider we chose to support in corc, foremost because of the existing collaboration with UCPH and the associated credits that got donated to this project. This enabled us to utilize said provider to cloud enable part of the DAG service at UCPH. This was made possible through the introduction of the MultipleSpawner package that utilized corc to dynamically chose between supported cloud providers. We demonstrated that the MultipleSpawner was capable of scheduling and stopping orchestrated and configured resources at OCI via a local researcher's machine.

In terms of future work, the next step involves the establishment of a Grid layer on top of the UCPH and OCI clouds. This Grid layer is planned to enable the establishment of a federated pool of participating organisations to share their resources. By doing so, we will be able to dynamically utilize cross organisation resources for services such as DAG, allowing us for instance to spawn Notebooks across multiple institutions such as other universities. Enabling the sharing of underused resources across the Grid participants. To accomplish this, corc also needs to be expanded to support additional providers, foremost through the integration of the Apache libcloud [62] library which natively supports more than 30 providers, we will allow corc and subsequently the MultipleSpawner to be utilized across a wide range of cloud providers.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604. Furthermore, many thanks is given to Oracle for donating the cloud resources that made this project possible.

References

[1] A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B. S. Lee, P. Faraboschi, R. Kaufmann, D. Milojicic, The who, what, why, and how of high performance computing in the cloud, Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom 1 (2013) 306–314. doi:10.1109/CloudCom.2013.47.

[2] B. Vinter, J. Bardino, M. Rehr, K. Birkelund, M. O. Larsen, Imaging Data Management System, in: Cloud NG:17 Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures, Belgrade, Serbia, 2017.

[3] D. Häfner, R. L. Jacobsen, C. Eden, M. R. B. Kristensen, M. Jochum, R. Nuterman, B. Vinter, Veros v0.1 &amp;ndash; a Fast and Versatile Ocean Simulator in Pure Python, Geoscientific Model Development Discussions (2018) 1–22. doi:10.5194/gmd-2018-3.

[4] P. Padoan, L. Pan, M. Juvela, T. Haugbølle, Nordlund, The Origin of Massive Stars: The Inertial-inflow Model, The Astrophysical Journal 900 (2020) 82. doi:10.3847/1538-4357/abaa47.

[5] University of Copenhagen, University of Copenhagen policy for scientific data, Technical Report, Copenhagen, 2014. URL: https://kunet.ku.dk/arbejdsomraader/forskning/data/forskningsdata/Documents/Underskrevetogendeligversionafpolitikforopbevaringafforskningsdata.pdf.

[6] University of Copenhagen, SCIENCE AI Centre, 2020. URL: https://ai.ku.dk/research/.
[7] University of Antwerp, High Performance Computing CalcUA, 2020. URL: https://www.uantwerp.be/en/core-facilities/calcua/.
[8] Lund University, LUNARC, 2020. URL: https://www.maxiv.lu.se/users/lunarc/.
[9] Vienna Scientific Cluster, Vienna Scientific Cluster, 2009. URL: https://vsc.ac.at//access/.
[10] ARCHER, 2019. URL: https://www.epcc.ed.ac.uk/facilities/archer.
[11] European Union, European Commission, EuroHPC JOINT UNDERTAKING, Technical Report, 2020. URL: https://op.europa.eu/en/publication-detail/-/publication/dff20041-f247-11ea-991b-01aa75ed71a1/language-en. doi:10.2759/26995.
[12] I. Foster, C. Kesselman, The history of the grid, Advances in Parallel Computing 20 (2011) 3–30. doi:10.3233/978-1-60750-803-8-3.
[13] R. Munk, Cloud Orchestrator, 2020. URL: https://github.com/rasmunk/corc.
[14] Instructure, Canvas, 2020. URL: https://www.instructure.com/canvas/about.
[15] J. Bardino, M. Rehr, B. Vinter, R. Munk, ERDA, 2019. URL: https://www.erda.dk.
[16] R. Munk, Jupyter Service, 2019.
[17] G. Zaccone, R. Karim, A. Menshawy, Chapter 7: GPU Computing, in: Deep Learning with TensorFlow, 1 ed., Packt Publishing, Limited, 2017, p. 316.
[18] University of Copenhagen, Introduction to Computing for Physicists, 2019. URL: https://kurser.ku.dk/course/nfya06018u/.
[19] University of Copenhagen, Applied Statistics, 2019. URL: https://kurser.ku.dk/course/nfyk13011u.
[20] High Performance Parallel Computing, 2020. URL: https://kurser.ku.dk/course/nfyk18001u/.
[21] J. Berthold, J. Bardino, B. Vinter, A Principled Approach to Grid Middleware, in: Algorithms and Architectures for Parallel Processing, volume 7016, Springer, 2011, pp. 409–418. doi:https://doi.org/10.1007/978-3-642-24650-0{_}35.
[22] Project Jupyter, Project Jupyter, 2019. URL: https://jupyter.org/about.
[23] F. Perez, B. E. Granger, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering, Computing in Science and Engineering 9 (2007) 21–29. URL: http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.53. doi:10.1109/MCSE.2007.53.
[24] Project Jupyter, JupyterLab, 2018. URL: http://jupyterlab.readthedocs.io/en/stable/.
[25] Project Jupyter, JupyterHub, 2015. URL: https://pypi.org/project/jupyterhub/.
[26] Project Jupyter, JupyterHub Spawners, 2020. URL: https://github.com/jupyterhub/jupyterhub/wiki/Spawners.
[27] Project Jupyter, WrapSpawner, ???? URL: https://github.com/jupyterhub/wrapspawner.
[28] S. L. Proskura, S. H. Lytvynova, The approaches to Web-based education of computer science bachelors in higher education institutions, CEUR Workshop Proceedings 2643 (2020) 609–625.
[29] GitHub, GitHub, 2020. URL: https://www.github.com.
[30] Google, Google Docs, 2020. URL: https://docs.google.com.
[31] Google Colab, 2020. URL: https://colab.research.google.com.
[32] Kaggle Inc, Kaggle, 2018. URL: https://www.kaggle.com.
[33] Project Jupyter, Binder, 2017. URL: https://mybinder.org.
[34] Microsoft, Azure Notebooks, 2020. URL: https://notebooks.azure.com.
[35] CoCalc, CoCalc, 2020. URL: https://cocalc.com.
[36] JetBrains, Datalore, 2020. URL: https://datalore.jetbrains.com.
[37] BinderFAQ, 2017. URL: https://mybinder.readthedocs.io/en/latest/faq.html.
[38] Kaggle Inc, Kaggle Notebooks Documentation, 2020. URL: https://www.kaggle.com/docs/

notebooks.
[39] Google, Google Colab FAQ, 2020. URL: https://research.google.com/colaboratory/faq.html.
[40] Microsoft, Azure Notebooks Overview, 2020. URL: https://docs.microsoft.com/en-us/azure/notebooks/azure-notebooks-overview.
[41] Microsoft, Azure Notebooks manage and configure projects, 2020. URL: https://docs.microsoft.com/en-us/azure/notebooks/azure-notebooks-overview.
[42] CoCalc, Cocalc Docs, 2020. URL: https://doc.cocalc.com/index.html.
[43] JetBrains, Datalore Documentation, 2020. URL: https://datalore.jetbrains.com/documentation.
[44] R. Munk, DAG, 2019. URL: https://github.com/ucphhpc/jupyter_service.
[45] Kaggle Inc, Kaggle GPU Tips and Tricks, 2020. URL: https://www.kaggle.com/page/GPU-tips-and-tricks.
[46] RedHat, What is Orchestration, 2020. URL: https://www.redhat.com/en/topics/automation/what-is-orchestration.
[47] TerraForm, TerraForm, 2020. URL: https://www.terraform.io/docs/index.html.
[48] Puppet, Puppet, 2020. URL: https://puppet.com.
[49] Chef, Chef, 2020. URL: https://www.chef.io/products/chef-infra.
[50] Ansible, Ansible, 2020. URL: https://www.ansible.com.
[51] Oracle Corporation, Oracle Cloud Infrastructure CLI, 2020. URL: https://github.com/oracle/oci-cli.
[52] Amazon, AWS Command Line Interface, 2020. URL: https://aws.amazon.com/cli/.
[53] Amazon, Tools to build on AWS, 2020. URL: https://aws.amazon.com/tools/.
[54] OpenStack, OpenStack, 2020. URL: https://www.openstack.org.
[55] OpenNebula Systems, OpenNebula, 2020. URL: https://opennebula.io.
[56] INDIGO-DataCloud, INDIGO-DataCloud, 2020. URL: https://www.indigo-datacloud.eu.
[57] M. Caballer, S. Zala, López García, G. Moltó, P. O. Fernández, M. Velten, Orchestrating complex application architectures in heterogeneous clouds, Journal of Grid Computing 16 (2017) 3–18. doi:10.1007/s10723-017-9418-y.
[58] J. Kovács, P. Kacsuk, Occopus: A multi-cloud orchestrator to deploy and manage complex scientific infrastructures, Journal of Grid Computing 16 (2018) 19–37. doi:10.1007/s10723-017-9421-3.
[59] R. Munk, MultipleSpawner, 2020. URL: https://github.com/ucphhpc/multiplespawner.
[60] NERSC, SSHSpawner, 2016. URL: https://github.com/NERSC/sshspawner.
[61] NVIDIA, TensorFlow 2 Quickstart Notebook, 2020. URL: https://www.tensorflow.org/tutorials/quickstart/beginner.
[62] The Apache Software Foundation, libcloud, 2020. URL: https://libcloud.apache.org.

E

RESOURCES USED WITHIN THIS THESIS

Table E.1 outlines all resources used in processing throughout this thesis. Unless otherwise noted all processing was carried out using the Laptop. This table is only applicable to the work directly reported in this thesis, and not the papers contained in the Appendices, which will have their own resources stated.

Resource	Nodes	CPU	Cores	Clock Speed (Ghz)	Memory (GB)
Laptop	1	i7-8550U	4/8	1.8	8
Desktop	1				
Threadripper	1	Threadripper 1950X	16/32	2.2	110
Small OCI Cluster	1	EPYC 7551	4	2.0	60
Large OCI Cluster	20	EPYC 7551	24	2.0	320

Table E.1: Resources used throughout for testing, timing, and benchmarking. Note that the laptop, Desktop and Threadripper are Hyperthreaded and so although they can simulate as many cores as shown on the right, the only have the number of the left many physical cores.

F

MUMMERING GRANT AGREEMENT EXTRACTS

Though the full MUMMERING Grant Agreement cannot be shown, certain relevant extracts from it are presented here in full.

```
Task 2.1 Automating data analysis through workflows. Data acquisition is
    improving not only is resolution and speed, but also in robustness, which
    means that still more experiments are performed. The combination of more and
    larger data sets is stressed further by the fact that new analysis
    approaches have still more components in them. Task 2.1 will develop tools
    to help automate the analysis through workflows. Analysis workflows are well
    known from other scientific fields, most notably bioinformatics and high
    energy physics, both of which are defined by very large datasets that
    undergo exactly the same analysis. For X-ray data the analysis is rarely
    identical, but with the increased robustness of data acquisition more data
    sets will require identical analysis. A workflow that analyse data may be
    run because new data arrives, or because the workflow itself has changed,
    including individual analysis components. Thus 2.1 will build a system where
    new results, be it original data, outputs from one analysis step, or changes
    in the analysis, can automatically trigger all causally dependent analysis,
    and thus improve overall analysis speed and reduce human errors.
```

Listing F.1: MUMMERING WP2 Task 2.1 full description. Note this task was primarily my own responsibility.

```
Task 2.2 A Big Data System for data analysis. The increased size of individual
    datasets, where individual tomograms easily grows to more than 64GiB and
    soon will require 180GiB, which essentially excludes individual
    workstations, and implicitly requires parallel processing. While
    conventional Big Data systems, such as Hadoop, have been examined for image
    analysis, it has been shown in literature that the model is not well suited
    for data such as tomograms. Task 2.2 will build an alternative Big Data
    platform that is tailor made for large images and volumes, and for
    processing these. The platform will optimize data access from stable storage
    and provide users with an interface that directly supports image and volume
    analysis, without exposing the programmer to the underlying parallelism. The
    successful Big Data platform will provide an improved performance that
    scales with the number of nodes that are allocated for analysis.
```

Listing F.2: MUMMERING WP2 Task 2.2 full description. Note this task was primarily the responsibility of Rasmus Munk.

```
Task 2.3 Total data management. The workflow tool from task 2.1, the analysis
    platform from task 2.2, experiment scheduling and storage scheduling must be
    coordinated overall. One of the big challenges in managing data from large
    X-ray facilities is the sheer size of the data, we will soon see 4D datasets
    that grows to several hundred TB. These sizes are not compatible with
    conventional storage systems, and we will see a migration to slower media
    that are both cheaper, use less energy and are more reliable, these include
    variable rotation speed hard drives and conventional magnetic tape systems.
    Such archive storage technology has many advantages, but the primary
    downside is that they are too slow to service conventional operations. To
    remedy the decrease in speed the data transfers must be scheduled.
    Experiments reserve bandwidth for storing data, the same data can be
    scheduled for immediate analysis, and the results from those analyses may
    force scheduling another level of analysis, and so forth. To ensure that
    there is sufficient resources to store and process all data the overall data
    management system must be integrated and supported by a scheduler.
```

Listing F.3: MUMMERING WP2 Task 2.3 full description. Note responsibility for this task was shared between Rasmus Munk and I.

G

CODE CONTENTS OF *ADD.IPYNB*

```python
# Default parameters values
# Amount to add to data
extra = 10
# Data input file location
infile = 'example_data/data_0.npy'
# Output file location
outfile = 'standard_output/data_0.npy'

import numpy as np
import os

# load in dataset. Should be numpy array
data = np.load(infile)

# Add an amount to all the values in the array
added = data + int(float(extra))

# Create output directory if it doesn't exist
output_dir_path = os.path.dirname(outfile)

if output_dir_path:
    os.makedirs(output_dir_path, exist_ok=True)

# Save added array as new dataset
np.save(outfile, added)
```

Listing G.1: Code contents of *add.ipynb* Jupyter Notebook.

H

MIG BASED MEOW JOB TEMPLATE

The following is a somewhat abridged section of the code within the MiG, showing the template that is used to define a MEOW job. Note that this code is an amalgamation of two different functions, with numerous small pieces of processing in between. Therefore this should not be taken as a technical reference for what is going on, merely as a guide for what the template looks like and how it may be filled.

```
# Prepare for output notebook
task_output = "%s_" + recipe['name'] + "_output.ipynb"
task_output = task_output % "+JOBID+"

# Prepare execution lines
executes = []
executes.append("${NOTEBOOK_PARAMETERIZER} %s %s -o %s -e" % (task_path,
    parameter_path, param_task_path))
executes.append("${PAPERMILL} %s %s" % (param_task_path, task_output))

template = {'execute': '\n'.join(executes), 'output_files': task_output}

# Insert variables into template
template_mrsl = """
::EXECUTE::
%(execute)s

::OUTPUTFILES::
%(output_files)s%(sep)sjob_output/+JOBID+/%(output_files)s

::MAXFILL::
CPUCOUNT
CPUTIME
DISK
MEMORY
NODECOUNT

::RETRIES::
0

::MEMORY::
64

::DISK::
1

::CPUTIME::
60

::CPUCOUNT::
1

::NODECOUNT::
1
```

```

::MOUNT::
+TRIGGERVGRIDNAME+ +TRIGGERVGRIDNAME+

::VGRID::
+TRIGGERVGRIDNAME+

::ENVIRONMENT::
LC_ALL=en_US.utf8
PYTHONPATH=+TRIGGERVGRIDNAME+
WORKFLOW_INPUT_PATH=+TRIGGERPATH+

::NOTIFY::
email: SETTINGS

::RUNTIMEENVIRONMENT::
NOTEBOOK_PARAMETERIZER
PAPERMILL
""" % template
```

Listing H.1: The job template used within the MiG.

I

CODE CONTENTS OF *INITIAL_POROSITY_CHECK.IPYNB*

```python
# Variables that will be overridden according to Pattern
input_filename = 'foam_ct_data/foam_016_ideal_CT.npy'
output_filedir_accepted = 'foam_ct_data_accepted'
output_filedir_discarded = 'foam_ct_data_discarded'
porosity_lower_threshold = 0.8
utils_path = 'idmc_utils_module.py'

import numpy as np
import importlib
import matplotlib.pyplot as plt
import os
import importlib.util

spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(spec)
spec.loader.exec_module(utils)
n_samples, n_components = 10000, 2

#Load data
ct_data = np.load(input_filename)
utils.plot_center_slices(ct_data)

#Perform GMM fitting on samples from dataset
sample_inds=np.random.randint(0, len(ct_data.ravel()), n_samples)

means, stds, weights = utils.perform_GMM_np(
    ct_data.ravel()[sample_inds],
    n_components,
    plot=True,
    title='GMM fitted to '+str(n_samples)+' of '
    +str(len(ct_data.ravel()))+' datapoints')
print('weights: ', weights)

# Classify data as accepted or discarded and write output
filename_withouth_npy=input_filename.split('/')[-1].split('.')[0]

if np.max(weights)>porosity_lower_threshold:
    os.makedirs(output_filedir_accepted, exist_ok=True)
    acc_path = os.path.join(output_filedir_accepted,
                            filename_withouth_npy+'.txt')
    with open(acc_path, 'w') as file:
        file.write(str(np.max(weights))+' '+str(np.min(weights)))
else:
    os.makedirs(output_filedir_discarded, exist_ok=True)
    dis_path = os.path.join(output_filedir_discarded,
                            filename_withouth_npy+'.txt')
    with open(dis_path, 'w') as file:
        file.write(str(np.max(weights))+' '+str(np.min(weights)))
```

Listing I.1: Code contents of *initial_porosity_check.ipynb* Jupyter Notebook.

J

CODE CONTENTS OF *SEGMENT_FOAM_DATA.IPYNB*

```python
# Variables that will be overridden by Patterns
input_filename = 'foam_ct_data_accepted/foam_016_ideal_CT.txt'
input_filedir = 'foam_ct_data'
output_filedir = 'foam_ct_data_segmented'
utils_path = 'idmc_utils_module.py'

import numpy as np
import importlib
import importlib.util
import matplotlib.pyplot as plt
import os
import scipy.ndimage as snd
import skimage

spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(spec)
spec.loader.exec_module(utils)
median_filter_kernel_size = 2

# Load data
filename_withouth_txt = input_filename.split(os.path.sep)[-1].split('.')[0]
input_data = os.path.join(input_filedir, filename_withouth_txt+'.npy')
ct_data = np.load(input_data)
utils.plot_center_slices(ct_data, titl =filename_withouth_txt)

# Median filtering
data_filtered = snd.median_filter(ct_data, median_filter_kernel_size)
utils.plot_center_slices(
    data_filtered,
    title=filename_withouth_txt+' median filtered')

# Otsu thresholding
threshold = skimage.filters.threshold_otsu(data_filtered)
data_thresholded = (data_filtered>threshold)*1
utils.plot_center_slices(
    data_thresholded,
    title=filename_withouth_txt+' Otsu thresholded')

# Morphological closing
data_segmented = (skimage.morphology.binary_closing((data_thresholded==0))==0)
utils.plot_center_slices(
    data_segmented,
    title=filename_withouth_txt+' Otsu thresholded')

# Save data
filename_save = filename_withouth_txt+'_segmented.npy'
os.makedirs(output_filedir, exist_ok=True)
np.save(os.path.join(output_filedir, filename_save), data_segmented)
```

Listing J.1: Code contents of *segment_foam_data.ipynb* Jupyter Notebook.

K

CODE CONTENTS OF *FOAM_PORE_ANALYSIS.IPYNB*

```
# Variables that will be overwritten by the Pattern
input_filename = 'foam_ct_data_segmented/foam_016_ideal_CT_segmented.npy'
output_filedir = 'foam_ct_data_pore_analysis'
utils_path = 'idmc_utils_module.py'

# Imports
import numpy as np
import importlib
import importlib.util
import matplotlib.pyplot as plt
import os
import scipy.ndimage as snd

from skimage.morphology import watershed
from skimage.feature import peak_local_max
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap

spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(spec)
spec.loader.exec_module(utils)

# Load data
data = np.load(input_filename)
utils.plot_center_slices(data, title = input_filename)

#distance map
distance = snd.distance_transform_edt((data==0))

#get watershed seeds
local_maxi = peak_local_max(
    distance,
    indices=False,
    footprint=np.ones((3, 3, 3)),
    labels=(data==0))
markers = snd.label(local_maxi)[0]

#perform watershed pore separation
labels = watershed(-distance, markers, mask=(data==0))

## Pore colour map
somecmap = cm.get_cmap('magma', 256)
cvals=np.random.uniform(0, 1, len(np.unique(labels)))
newcmp = ListedColormap(somecmap(cvals))

utils.plot_center_slices(
    -distance,
    cmap=plt.cm.gray,
    title='Distances')
utils.plot_center_slices(
    labels,
    cmap=newcmp,
    title='Separated pores')

# Plot statitistics: Pore radii
volumes = np.array([np.sum(labels==label) for label in np.unique(labels)])
volumes.sort()

#ignore two largest labels (background and matrix)
radii = (volumes[:-2]*3/(4*np.pi))**(1/3)
_=plt.hist(radii, bins=200)

# Save plot
filename_withouth_npy = input_filename.split(os.path.sep)[-1].split('.')[0]
filename_save = filename_withouth_npy+'_statistics.png'

fig, ax = plt.subplots(1,3, figsize=(15,4))
ax[0].imshow(labels[:,:,np.shape(labels)[2]//2], cmap=newcmp)
ax[1].imshow(labels[:,np.shape(labels)[2]//2,:], cmap=newcmp)
_ = ax[2].hist(radii, bins=200)
ax[2].set_title('Foam pore radii')

os.makedirs(output_filedir, exist_ok=True)
plt.savefig(os.path.join(output_filedir, filename_save))
```

Listing K.1: Code contents of *foam_pore_analysis.ipynb* Jupyter Notebook.

L

CODE CONTENTS OF *GENERATOR.IPYNB*

```python
# importing the necessary modules
import numpy as np
import random
import os
import importlib.util

# Variables to be overridden
dest_dir = 'foam_ct_data'
discarded = 'discarded/foam_data_0-big-.npy'
utils_path = 'idmc_utils_module.py'
gen_path = 'generate_foam_module.py'

# import modules dynamically from local files
u_spec = importlib.util.spec_from_file_location("utils", utils_path)
utils = importlib.util.module_from_spec(u_spec)
u_spec.loader.exec_module(utils)

g_spec = importlib.util.spec_from_file_location("gen", gen_path)
gen = importlib.util.module_from_spec(g_spec)
g_spec.loader.exec_module(gen)

# Other variables, will be kept constant
vx, vy, vz = 256, 256, 256
res=3/vz
nspheres_per_unit_few=100
nspheres_per_unit_ideal=1000

# Randomly determine if dataset is sufficient or too small.
def get_dataset_type(name):
    num = random.randint(1, 3)
    if num == 1:
        name = name.replace('--', 'X-few-')
        return (gen.generate_foam, nspheres_per_unit_few, name)
    else:
        name = name.replace('--', 'X-ok-')
        return (gen.generate_foam, nspheres_per_unit_ideal, name)

# Create a dataset for a given filename
def create_random_dataset(name):
    generator, spheres, filename = get_dataset_type(name)
    dataset = generator(spheres, vx, vy, vz, res)
    os.makedirs(dest_dir, exist_ok=True)
    np.save(os.path.join(dest_dir, filename+'.npy'), dataset)

# Determine base filename and generate replacement dataset
filename = os.path.basename(discarded)
filename = filename[:filename.index('-')] + '--'
create_random_dataset(filename)
```

Listing L.1: Code contents of *generator.ipynb* Jupyter Notebook.

M

PYTHON SCRIPT *INITIAL_GENERATION.PY*

```python
# importing the necessary modules
import numpy as np
import random
import os
import generate_foam_module as gen
import idmc_utils_module as utils

# Variables definitions
dest_dir = 'Patch/foam_ct_data'
default_filename = 'foam_data_'
to_generate = 20

vx, vy, vz = 256, 256, 256
res=3/vz

nspheres_per_unit_few=100
nspheres_per_unit_ideal=1000

# Function to randomly determine dataset composition.
# 1 in 3 chance of defective dataset.
def get_dataset_type(name):
    num = random.randint(1, 3)
    if num == 1:
        name = name.replace('--', '-few-')
        return (gen.generate_foam, nspheres_per_unit_few, name)
    else:
        name = name.replace('--', '-ok-')
        return (gen.generate_foam, nspheres_per_unit_ideal, name)

# Create a dataset for a given filename
def create_random_dataset(name):
    generator, spheres, filename = get_dataset_type(name)
    dataset = generator(spheres, vx, vy, vz, res)
    os.makedirs(dest_dir, exist_ok=True)
    np.save(os.path.join(dest_dir, filename+'.npy'), dataset)

if __name__ == "__main__":
    # Generate all datasets
    for x in range(to_generate):
        print('%d/%d' % (x+1, to_generate))
        name = default_filename + str(x) + '--'
        create_random_dataset(name)
```

Listing M.1: Python script *initial_generation.py*

N

CODE CONTENTS OF *PATTERN_MAKER_RECIPE_MIG.IPYNB*

```
# Variables to be overridden
meow_dir = 'meow_directory'
filter_recipe = 'recipe_filter'
input_yaml = 'input.yml'
workgroup = '{VGRID}'
workflows_url =
    'https://test-sid.idmc.dk/cgi-sid/jsoninterface.py?output_format=json'
workflows_session_id = *redacted*

# Names of the variables in filter_recipe.ipynb
recipe_input_image = 'input_image'
recipe_output_image = 'output_image'
recipe_args = 'args'
recipe_method = 'method'

# Imports
import yaml
import mig_meow as meow
import os

# Setup environment variables for meow to workgroup communication
os.environ['WORKFLOWS_URL'] = workflows_url
os.environ['WORKFLOWS_SESSION_ID'] = workflows_session_id

# Read in configuration data
with open(input_yaml, 'r') as yaml_file:
    y = yaml.full_load(yaml_file)

# Assemble a name for the new Pattern
name_str = '%s_%s' % (
    y['filter'],
    '_'.join("{!s}_{!r}".format(k,v) for (k,v) in y['args'].items()))

# Create the new Pattern
new_pattern = meow.Pattern(name_str)
new_pattern.add_recipe(filter_recipe)
new_pattern.add_single_input(recipe_input_image, y['input_path'])
new_pattern.add_output(recipe_output_image, y['output_path'])
new_pattern.add_variable(recipe_method, y['filter'])
new_pattern.add_variable(recipe_args, y['args'])

# Register the new Pattern with the system.
meow.export_pattern_to_vgrid(workgroup, new_pattern)
```

Listing N.1: Code contents of *pattern_maker_recipe_mig.ipynb* Jupyter Notebook.

O

TESTING WATCHDOG

```python
import csv
import gc
import os
import shutil
import threading
import time
from watchdog.observers import Observer
from watchdog.events import PatternMatchingEventHandler

TEST_DIR = 'test_dir'
RESULTS_FILE = 'results.csv'
WRITERS = 4
FILES = [1000, 10000, 100000, 1000000]
REPEATS = 20


class TestingEventHandler(PatternMatchingEventHandler):
    def __init__(
        self,
        q,
        patterns=None,
        ignore_patterns=None,
        ignore_directories=False,
        case_sensitive=False
    ):
        """Constructor"""
        PatternMatchingEventHandler.__init__(self, patterns,
                                             ignore_patterns,
                                             ignore_directories,
                                             case_sensitive)
        self.q = q
        self.count = 0
        self.start = None
        self.end = None

    def on_created(self, event):
        self.count += 1
        if self.count == 1:
            self.start = time.time()
        self.end = time.time()
        duration = self.end - self.start
        self.q.append([self.count, self.end, duration])


def monitor(q):
    path = TEST_DIR
    patterns = os.path.join(path, '*')
    event_handler = TestingEventHandler(q, patterns=patterns)

    observer = Observer()
    observer.schedule(event_handler, os.path.realpath(path), recursive=True)
    observer.start()


def writer(w, r):
    for i in range(r):
        p = '%s/%d-%d' % (TEST_DIR, w, i)
        with open(p, 'w') as f:
            pass


if __name__ == "__main__":
    if os.path.exists(RESULTS_FILE):
        os.remove(RESULTS_FILE)

    for file_count in FILES:
        results = []
        ranges = [int(file_count / WRITERS)] * WRITERS
        m = file_count % WRITERS
        for i in range(m):
            ranges[i] += 1
        print(ranges)

        for x in range(REPEATS):
            if os.path.exists(TEST_DIR):
                shutil.rmtree(TEST_DIR, ignore_errors=False, onerror=None)
                print('finished cleanup of writing dir')
            os.mkdir(TEST_DIR)

            print('starting run %d for %d writers' % (x, WRITERS))
            q = []

            monitor_thread = threading.Thread(
                target=monitor,
                args=[q])
            monitor_thread.daemon = True
            monitor_thread.start()

            time.sleep(3)

            writer_threads = []
            for w in range(WRITERS):
                writer_thread = threading.Thread(
                    target=writer,
                    args=[w, ranges[w]])
                writer_thread.daemon = True
                writer_threads.append(writer_thread)
            print('writing created')
            for writer_thread in writer_threads:
                writer_thread.start()
            print('writing started')
            for writer_thread in writer_threads:
                writer_thread.join()

            print('writing complete')

            time.sleep(3)

            waiting = True
            last = -1
            settle_count = 0
            while waiting:
                result = q[-1]

                current = result[1]

                if current == last:
                    settle_count += 1
                else:
                    settle_count = 0

                if settle_count == 5:
                    waiting = False
                else:
                    time.sleep(3)
                    last = current

            monitor_thread.join()
            results.append(result)

            del monitor_thread
            for writer_thread in writer_threads:
                del writer_thread
            del q
            gc.collect()

        with open(RESULTS_FILE, 'a', newline='') as csv_file:
            csv_writer = csv.writer(csv_file)
            csv_writer.writerow(['%s files by %s writer(s)' % (file_count,
WRITERS), ''])
            csv_writer.writerow(['Events', 'Duration'])
            for r in results:
                csv_writer.writerow([r[0], r[2]])
```

Listing O.1: Testing the event identification rate of `watchdog`.

P

RESULTS FROM OVERHEAD INVESTIGATION

The following graphs and table are the complete results of the experiments carried out in Section 17.4. All experiments were carried out on either the Laptop or Threadripper resources described in Appendix E. Source code for each test can be found at [28], [26] and [27].

To fit on a single page Table P.1 only shows timings for selected job counts. All speedups are shown relative to the Slurm `sbatch` or sequential `sbatch` tests run on the same machine. All results have been rounded to 2 decimal places or 1 significant figure for display only. Raw results can be viewed at [8]. Note that SPMF etc. refers to the experiment types outlined in Section 17.4.2.

	Total duration			Per-job duration			Speedup vs `sbatch`			Speedup vs sequential `sbatch`		
	10	100	500	10	100	500	10	100	500	10	100	500
Laptop `srun`	0.47s	4.16s	20.54s	0.05s	0.04s	0.04s	0.14	0.17	0.15	6.65	0.93	0.78
Threadripper `srun`	0.39s	3.35s	16.93s	0.04s	0.03s	0.03s	0.13	0.16	0.16	7.4	1.05	0.86
Laptop `sbatch`	0.06s	0.71s	3.07s	0.006s	0.007s	0.006s	-	-	-	48.74	5.43	5.2
Threadripper `sbatch`	0.05s	0.53s	2.76s	0.005s	0.005s	0.006s	-	-	-	54.93	6.67	5.28
Laptop sequential `sbatch`	3.14s	3.87s	15.96s	0.31s	0.04s	0.03s	0.02	0.18	0.19	-	-	-
Threadripper sequential `sbatch`	2.9s	3.52s	14.55s	0.29s	0.04s	0.03s	0.02	0.15	0.19	-	-	-
Laptop MEOW SPMF	0.04s	0.3s	1.44s	0.004s	0.003s	0.003s	1.5	2.41	2.14	72.93	13.08	11.12
Threadripper MEOW SPMF	0.02s	0.19s	0.96s	0.002s	0.002s	0.002s	2.4	2.8	2.87	131.62	18.7	15.18
Laptop MEOW MPSF	0.05s	0.28s	1.37s	0.005s	0.003s	0.003s	1.29	2.59	2.25	62.72	14.08	11.68
Threadripper MEOW MPSF	0.02s	0.18s	0.88s	0.002s	0.002s	0.002s	2.51	2.96	3.14	137.89	19.75	16.59
Laptop MEOW MPMF	0.06s	0.71s	14.35s	0.005s	0.007s	0.03s	1.17	1.01	0.21	57.02	5.46	1.11
Threadripper MEOW MPMF	0.03s	0.47s	8.33s	0.003s	0.005s	0.02s	2.03	1.11	0.33	111.37	7.4	1.75
Laptop MEOW SPSFP	0.04s	0.28s	1.37s	0.004s	0.003s	0.003s	1.5	2.52	2.25	72.93	13.69	11.69
Threadripper MEOW SPSFP	0.02s	0.18s	0.89s	0.002s	0.002s	0.002s	2.4	2.91	3.1	131.62	19.42	16.37
Laptop MEOW SPSFS	28.57s	317.13s	1601.91s	2.86s	3.17s	3.2s	0.002	0.002	0.002	0.11	0.01	0.01
Threadripper MEOW SPSFS	25.12s	279.75s	1411.54s	2.51s	2.8s	2.82s	0.002	0.002	0.002	0.12	0.01	0.01
Laptop MiG SPMF	1.23s	10.65s	241.52s	0.12s	0.11s	0.48s	0.05	0.07	0.01	2.56	0.36	0.07
Threadripper MiG SPMF	1.26s	15.69s	353.72s	0.13s	0.16s	0.71s	0.04	0.03	0.008	2.3	0.22	0.04
Laptop MiG MPSF	1.22s	8.84s	180.64s	0.12s	0.09s	0.36s	0.05	0.08	0.02	2.57	0.44	0.09
Threadripper MiG MPSF	1.25s	11.46s	209.95s	0.13s	0.11s	0.42s	0.04	0.05	0.01	2.31	0.31	0.07
Laptop MiG MPMF	1.22s	10.19s	423.65s	0.12s	0.1s	0.85s	0.05	0.07	0.007	2.58	0.38	0.04
Threadripper MiG MPMF	1.25s	13.04s	486.81s	0.12s	0.13s	0.97s	0.04	0.04	0.006	2.32	0.27	0.03
Laptop MiG SPSFP	2.83s	31.17s	392.19s	0.28s	0.31s	0.78s	0.02	0.02	0.008	1.11	0.12	0.04
Threadripper MiG SPSFP	1.16s	5.86s	84.91s	0.12s	0.06s	0.17s	0.05	0.09	0.03	2.51	0.6	0.17
Laptop MiG SPSFS	284.37s	1788.73s	8988.12s	28.44s	17.89s	17.98s	0.0002	0.0004	0.0003	0.01	0.002	0.002
Threadripper MiG SPSFS	274.47s	1737.65s	8566.8s	27.45s	17.38s	17.13s	0.0002	0.0003	0.0003	0.01	0.002	0.002

Table P.1: Selected timings for scheduling durations tests. To fit on this page, only certain values for jobs are shown. All speedups are shown relative to the Slurm `sbatch` test run on the same machine. All results have been rounded to 2 decimal places, or 1 significant figure if it is smaller that 0.01.

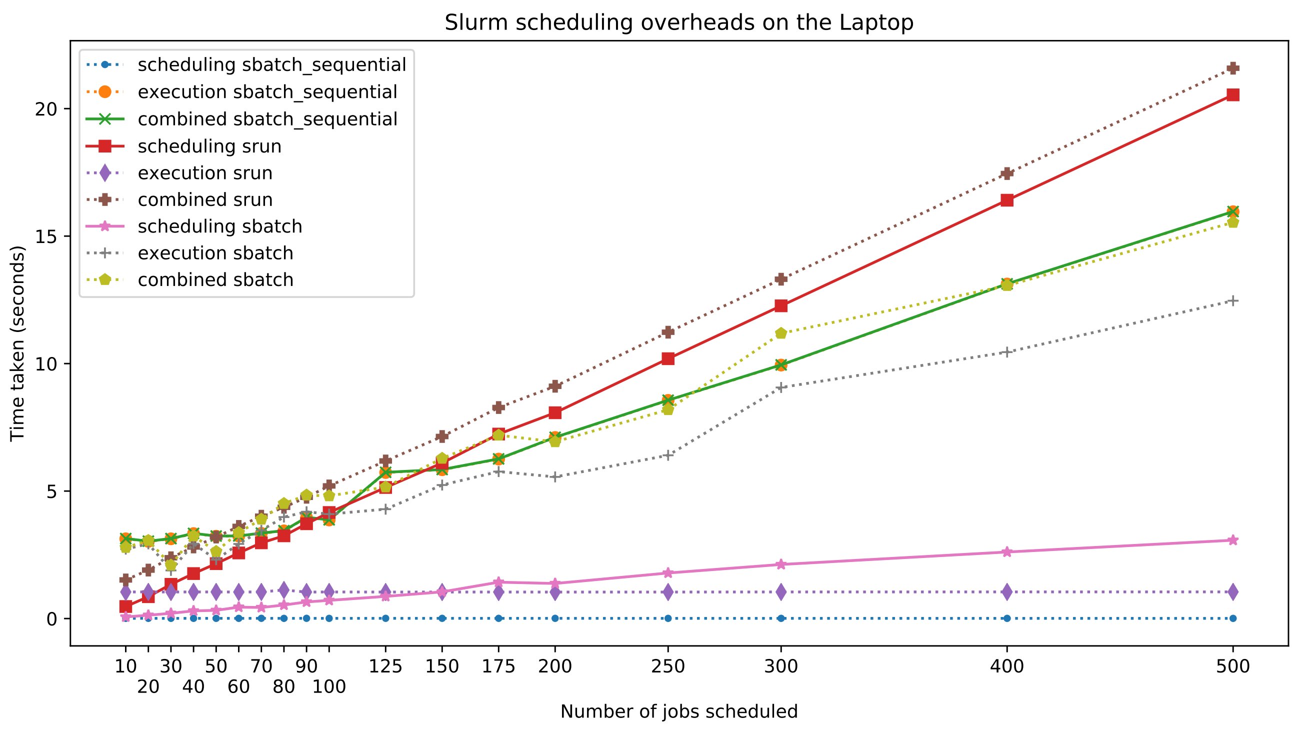


Figure P.1: Slurm scheduling durations on the Laptop. Each result is an average of 10 runs.

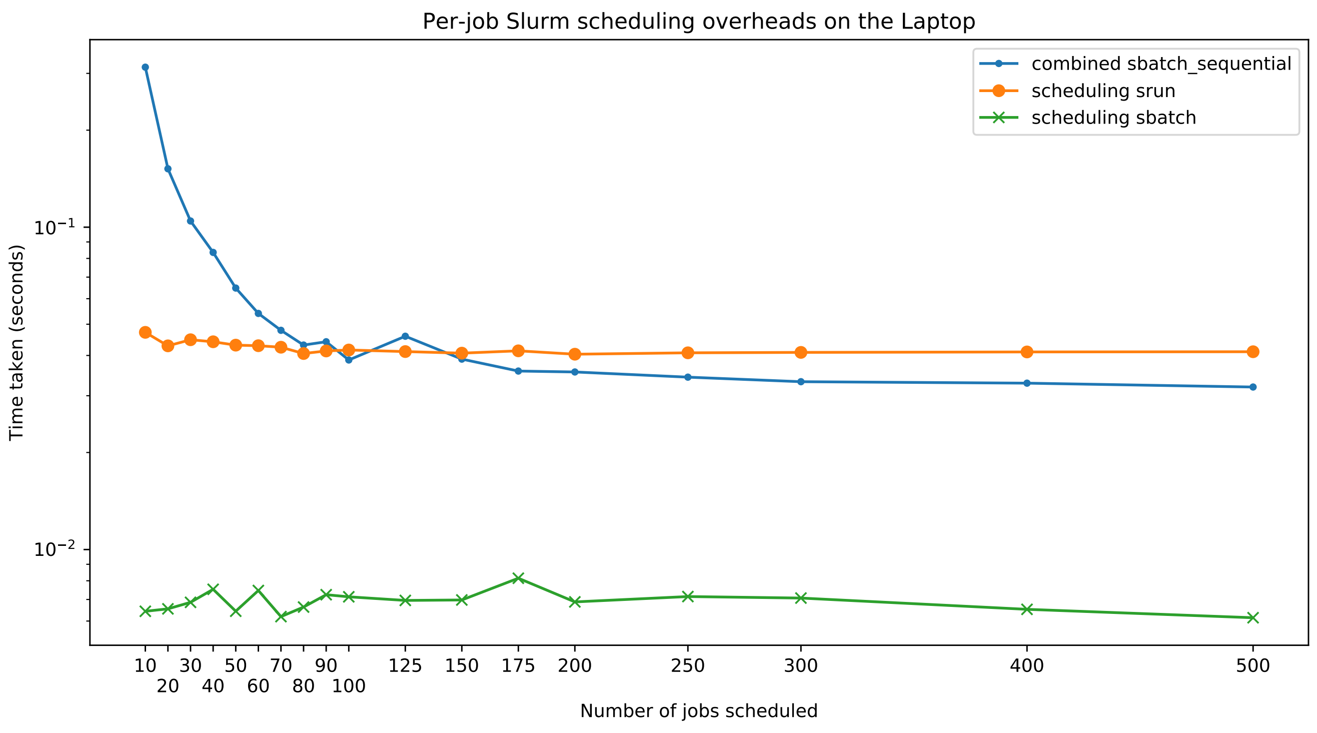


Figure P.2: Per-job Slurm scheduling durations on the Laptop. Calculated by dividing the total duration by the number of scheduled jobs. Note the X axis uses a logarithmic scale.

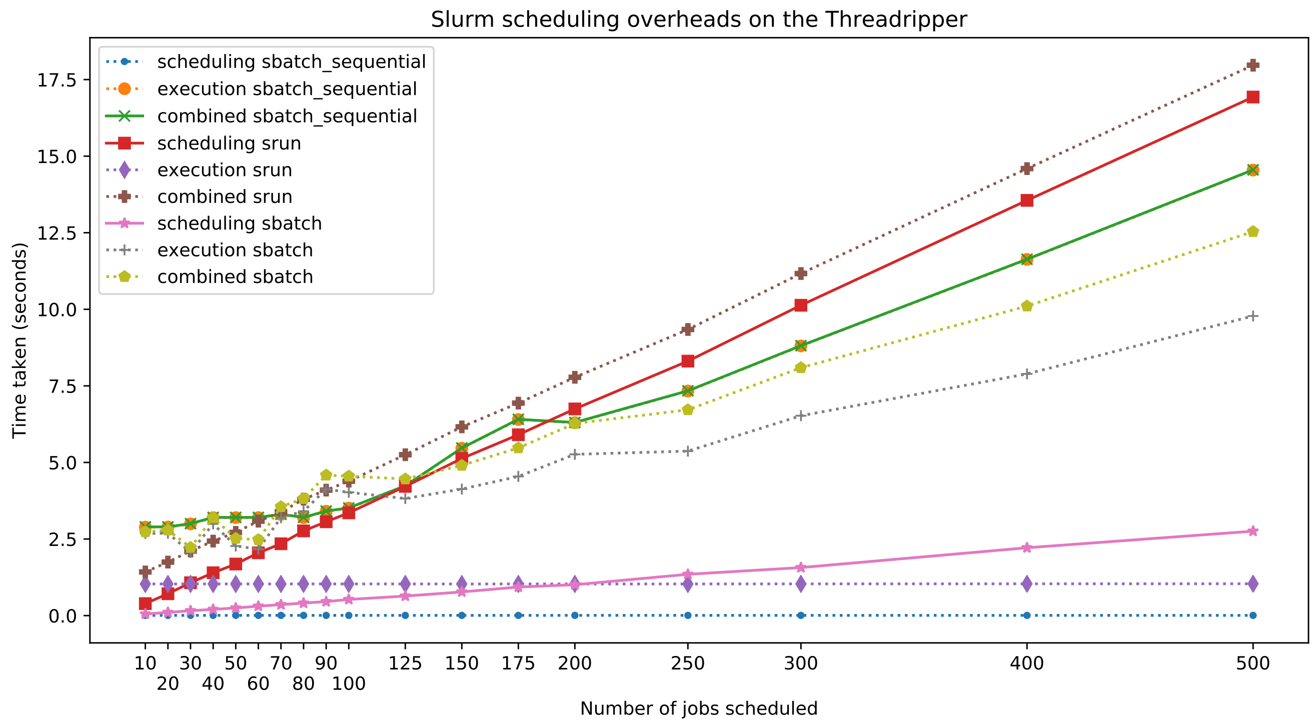


Figure P.3: Slurm scheduling durations on the Threadripper. Each result is an average of 10 runs.

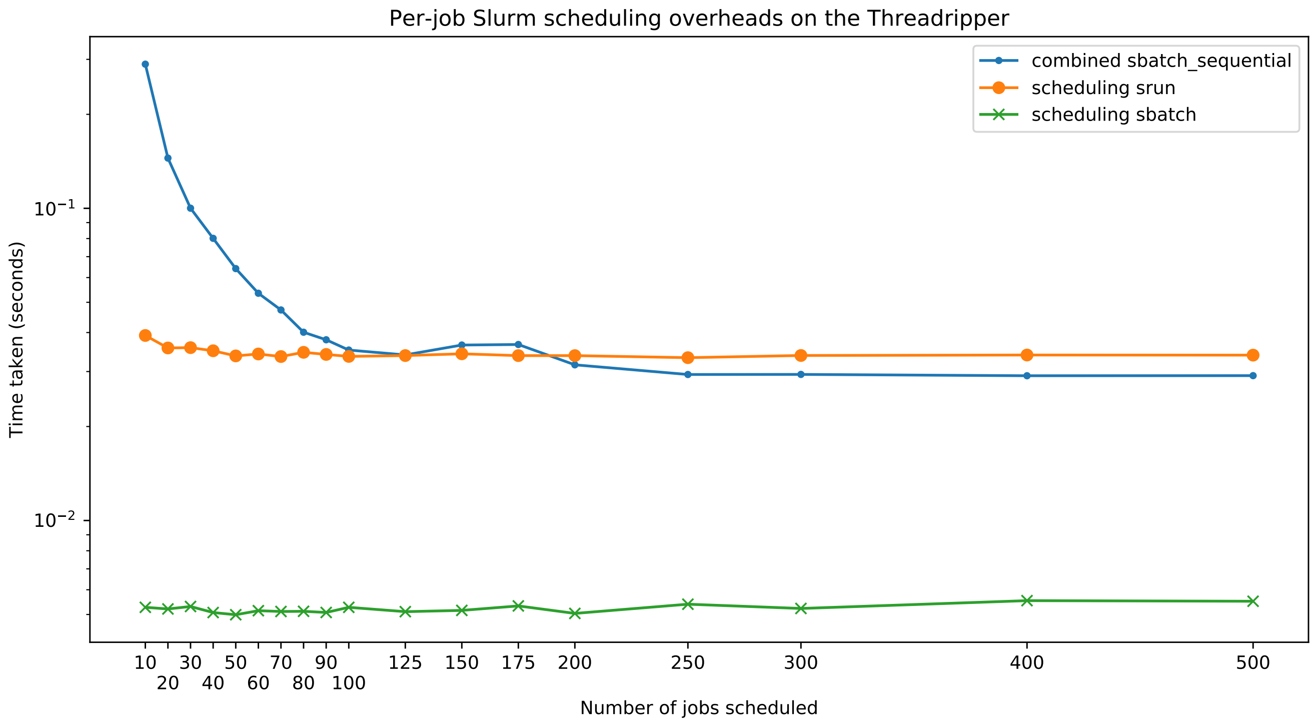


Figure P.4: Per-job Slurm scheduling durations on the Threadripper. Calculated by dividing the total duration by the number of scheduled jobs. Note the X axis uses a logarithmic scale.

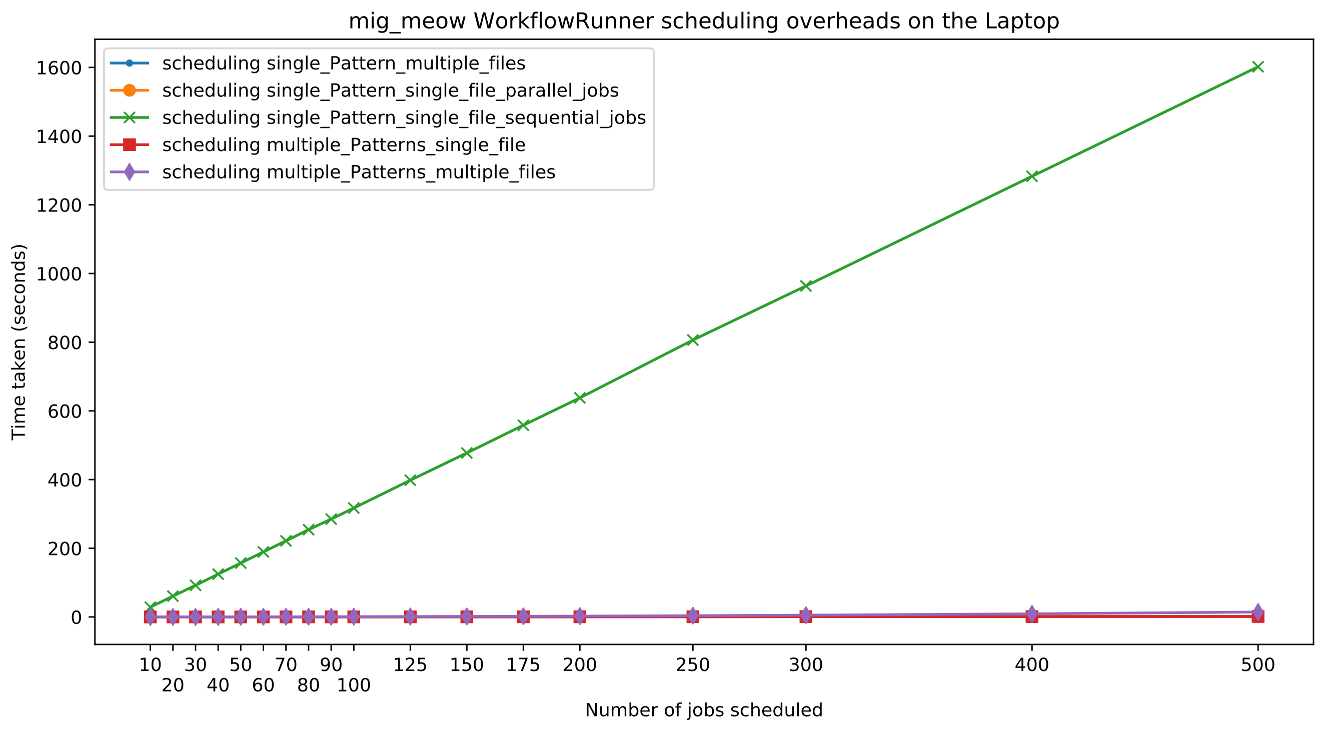


Figure P.5: `mig_meow WorkflowRunner` scheduling durations on the Laptop. Each result is an average of 10 runs.

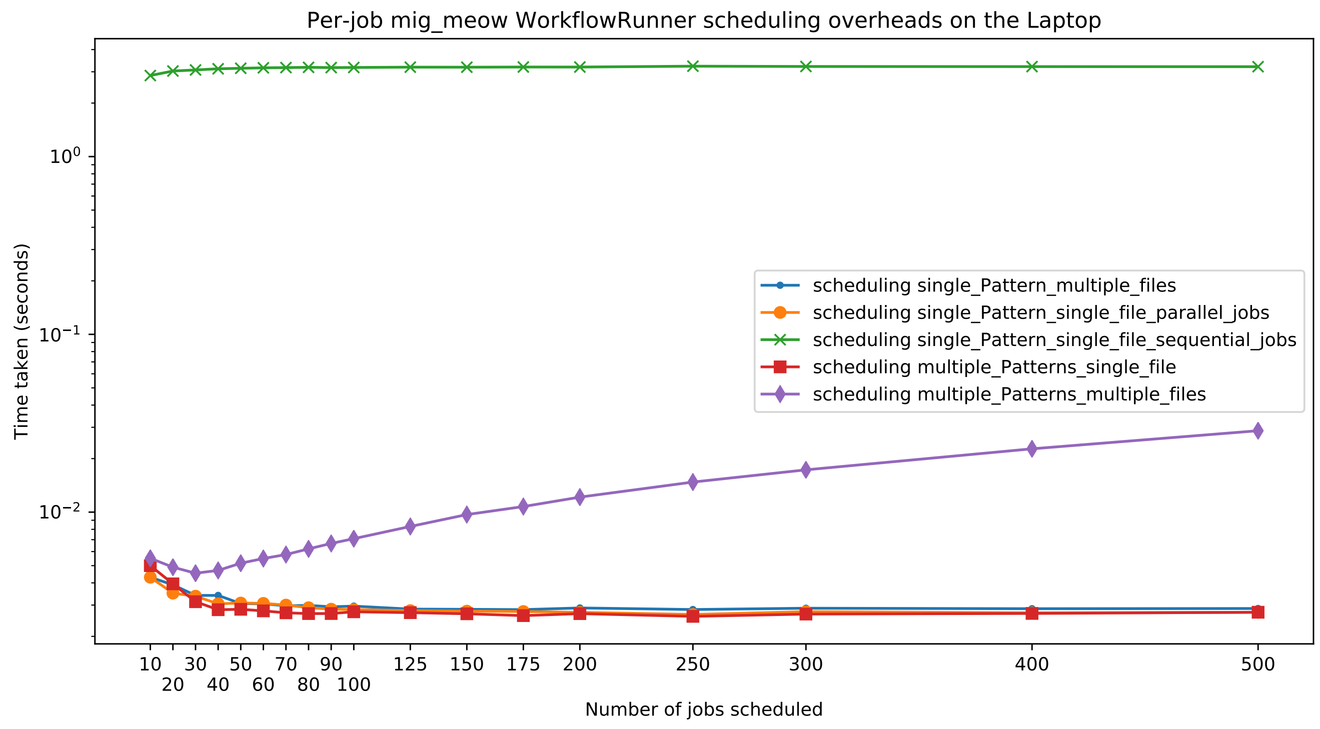


Figure P.6: Per-job `mig_meow WorkflowRunner` scheduling durations on the Laptop. Calculated by dividing the total duration by the number of scheduled jobs. Note the X axis uses a logarithmic scale.

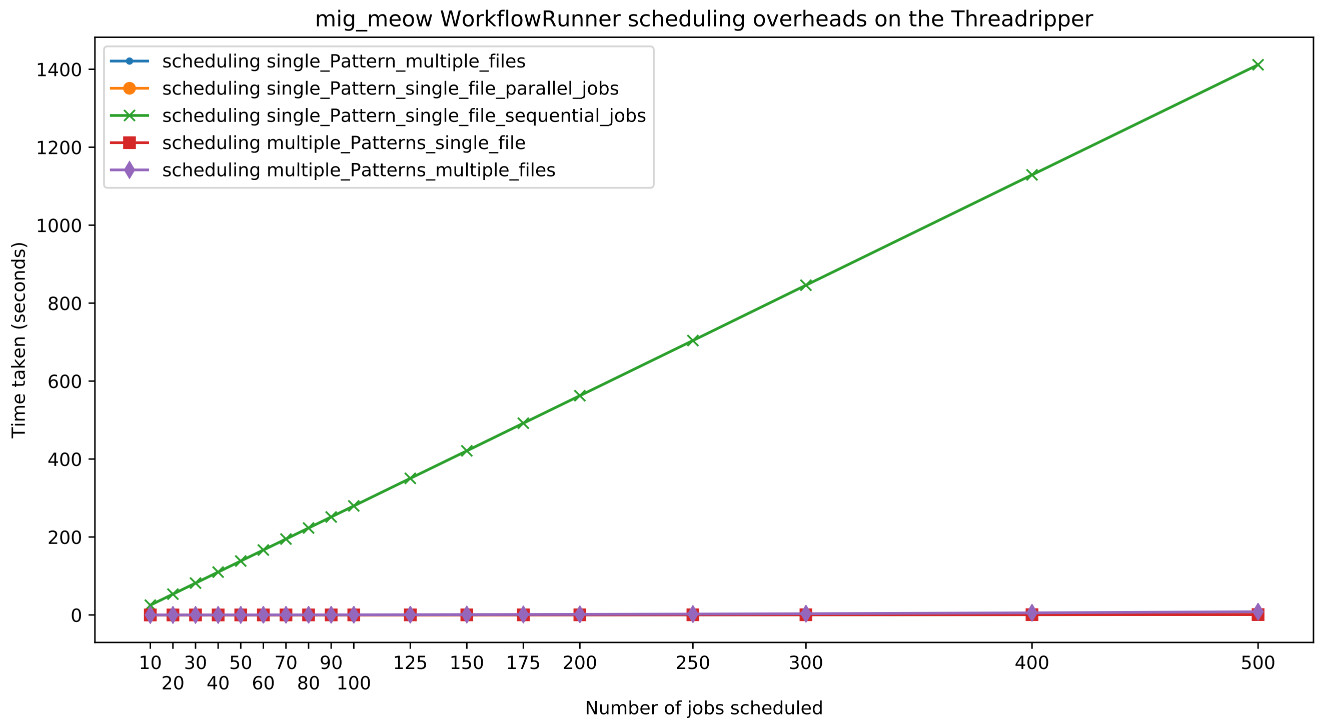


Figure P.7: `mig_meow WorkflowRunner` scheduling durations on the Threadripper. Each result is an average of 10 runs.

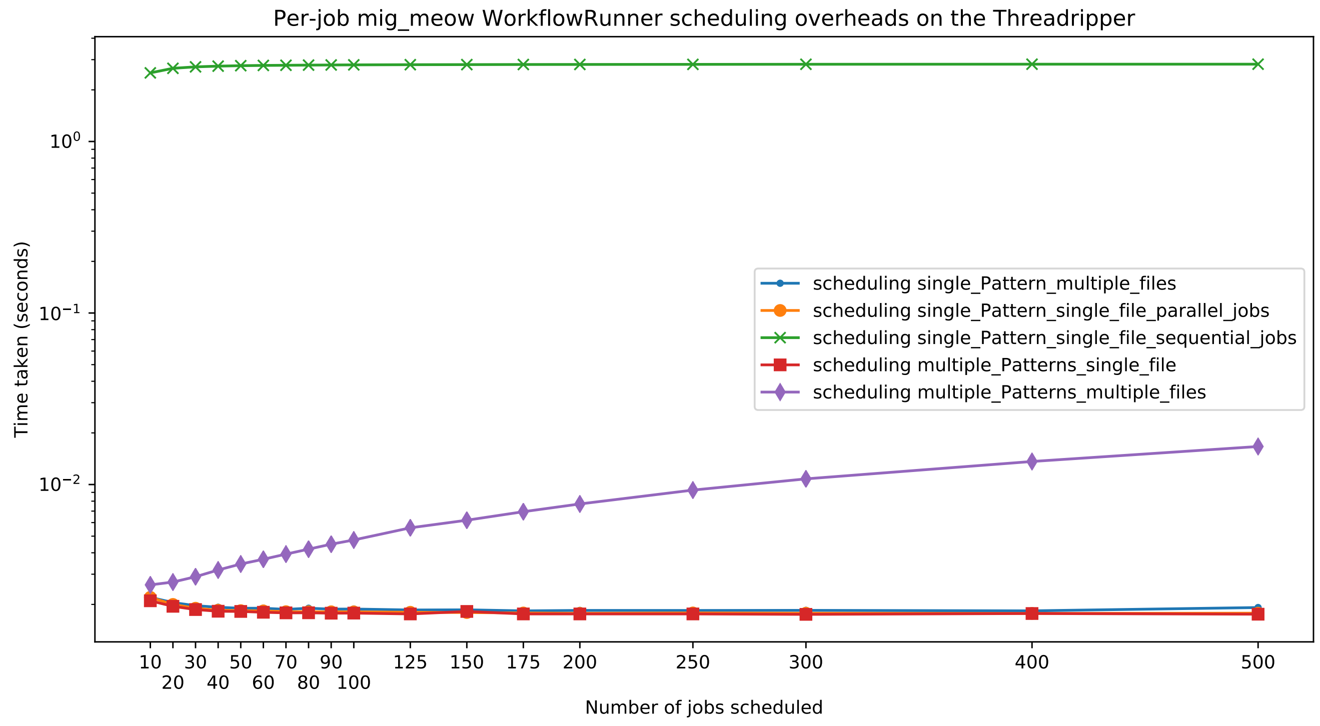


Figure P.8: Per-job `mig_meow WorkflowRunner` scheduling durations on the Threadripper. Calculated by dividing the total duration by the number of scheduled jobs. Note the X axis uses a logarithmic scale.

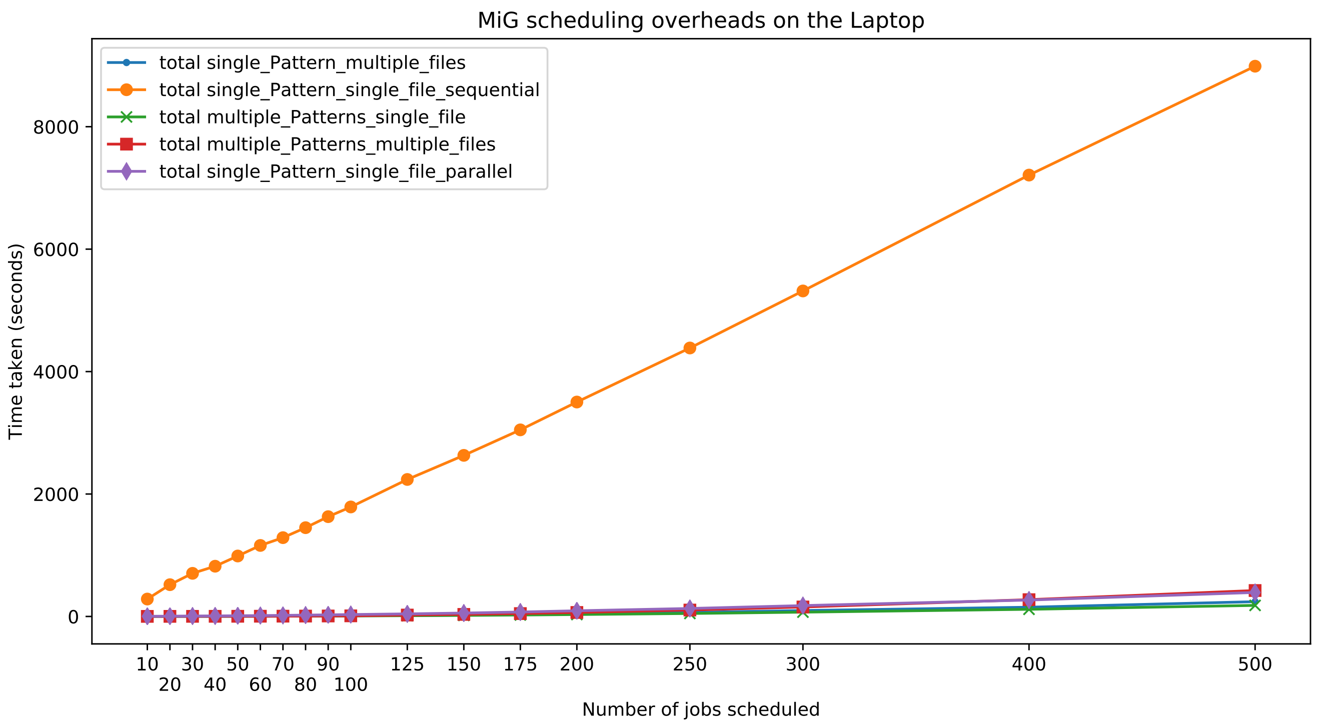


Figure P.9: MiG with MEOW scheduling durations on the Laptop. Each result is an average of 10 runs.

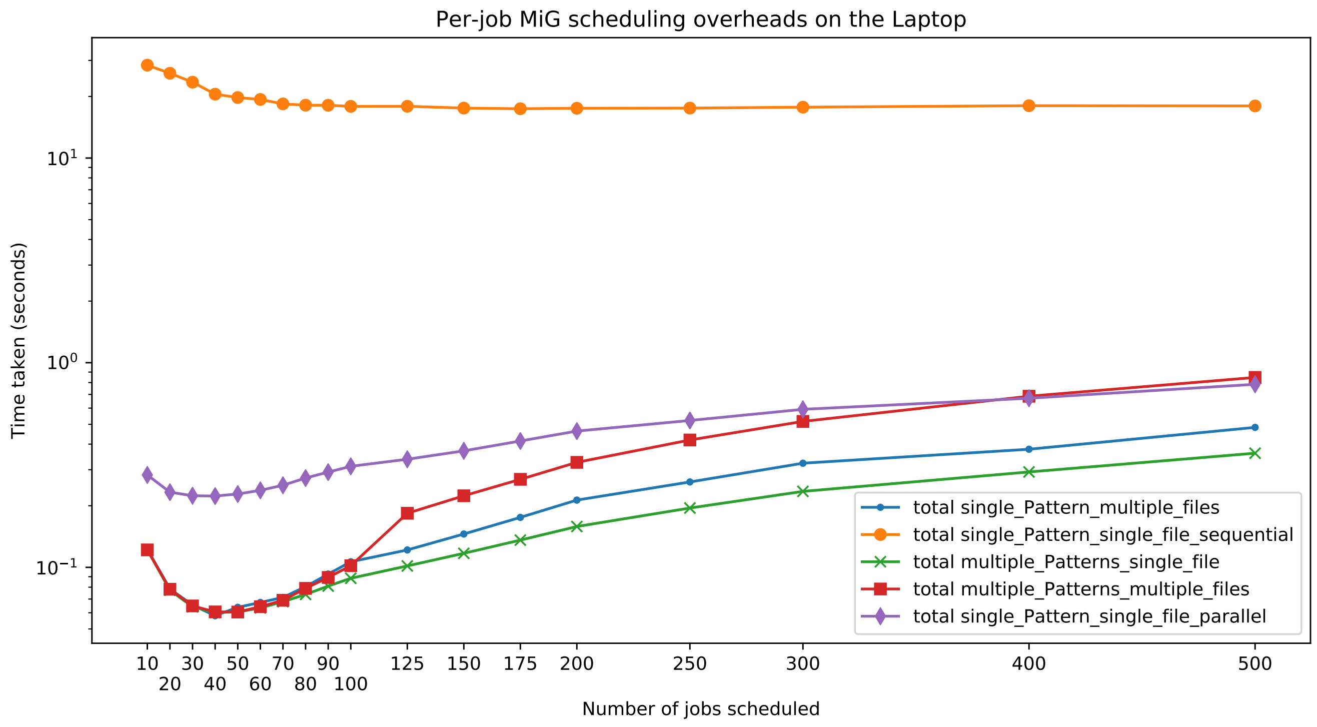


Figure P.10: Per-job MiG with MEOW scheduling durations on the Laptop. Calculated by dividing the total duration by the number of scheduled jobs. Note the X axis uses a logarithmic scale.

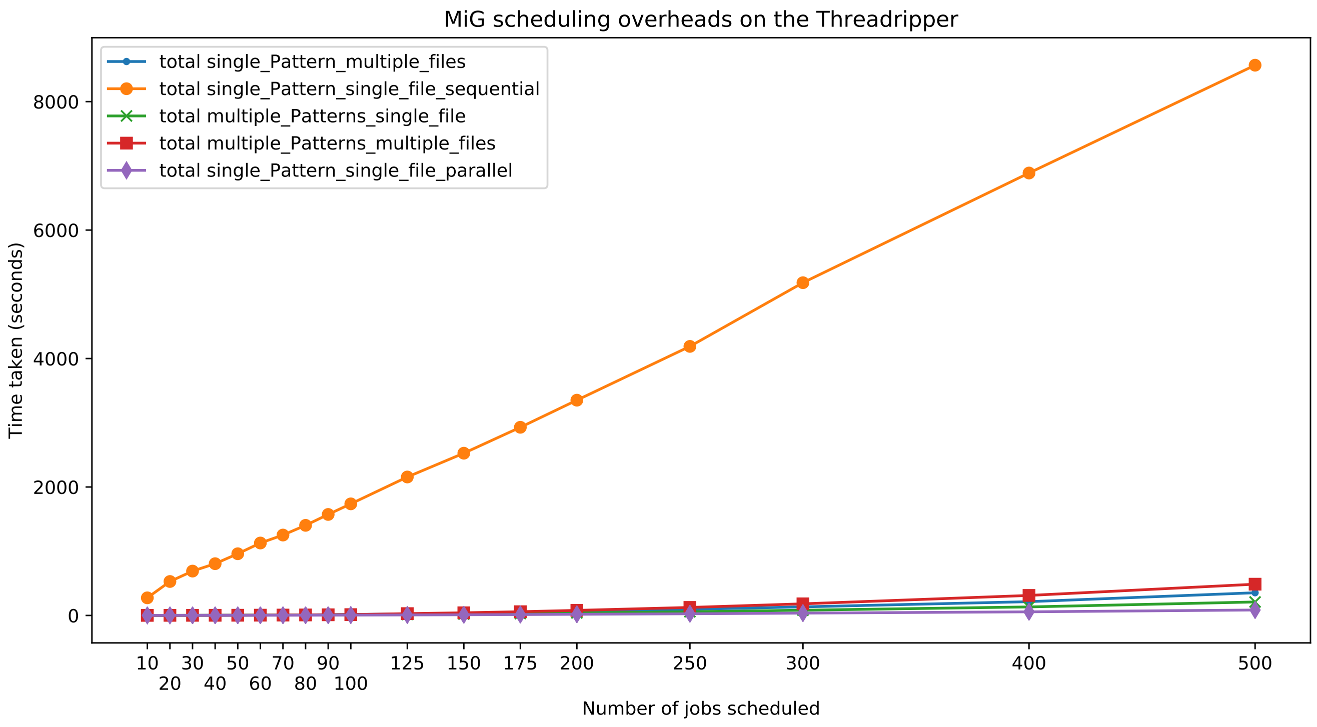


Figure P.11: MiG with MEOW scheduling durations on the Threadripper. Each result is an average of 10 runs.

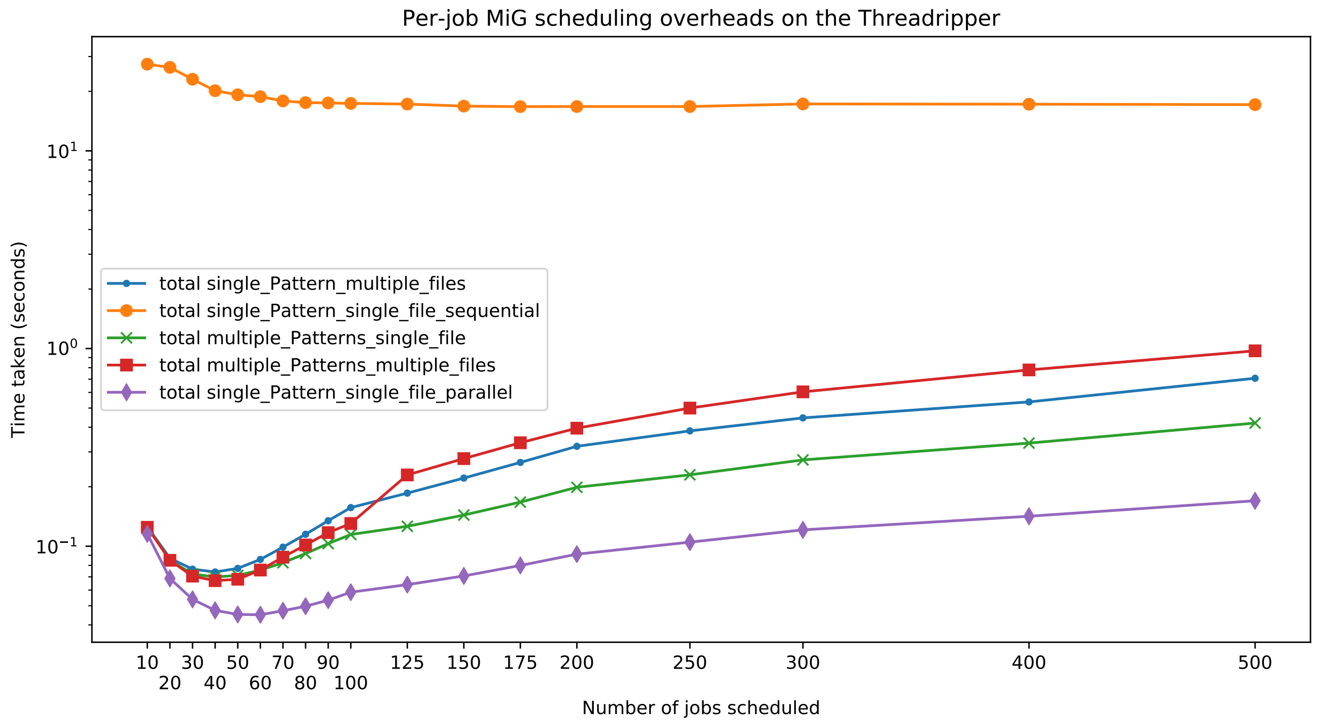


Figure P.12: Per-job MiG with MEOW scheduling durations on the Threadripper. Calculated by dividing the total duration by the number of scheduled jobs. Note the X axis uses a logarithmic scale.

Q

REQUIRED FILES FOR CWL WORKFLOW EXAMPLE

The following commands and files are all necessary parts of the first workflow example presented as part of the Common Workflow Language User Guide[44]. These are all designed to be used with cwltool[20], used with the commands shown in Q.5.

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0
class: Workflow
inputs:
  tarball: File
  name_of_file_to_extract: string

outputs:
  compiled_class:
    type: File
    outputSource: compile/classfile

steps:
  untar:
    run: tar-param.cwl
    in:
      tarfile: tarball
      extractfile: name_of_file_to_extract
    out: [extracted_file]

  compile:
    run: arguments.cwl
    in:
      src: untar/extracted_file
    out: [classfile]
```

Listing Q.1: *1st-workflow.cwl.*

```
tarball:
  class: File
  path: hello.tar
name_of_file_to_extract: Hello.java
```

Listing Q.2: *1st-workflow-job.yml.*

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0
class: CommandLineTool
baseCommand: [tar, --extract]
inputs:
  tarfile:
    type: File
    inputBinding:
      prefix: --file
  extractfile:
    type: string
    inputBinding:
      position: 1
outputs:
  extracted_file:
    type: File
    outputBinding:
      glob: $(inputs.extractfile)
```

Listing Q.3: *tar-param.cwl*.

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0
class: CommandLineTool
label: Example trivial wrapper for Java 9 compiler
hints:
  DockerRequirement:
    dockerPull: openjdk:9.0.1-11-slim
baseCommand: javac
arguments: ["-d", $(runtime.outdir)]
inputs:
  src:
    type: File
    inputBinding:
      position: 1
outputs:
  classfile:
    type: File
    outputBinding:
      glob: "*.class"
```

Listing Q.4: *arguments.cwl*.

```
echo "public class Hello {}" > Hello.java && tar -cvf hello.tar Hello.java
cwl-runner 1st-workflow.cwl 1st-workflow-job.yml
```

Listing Q.5: CWL workflow example commands.

R

CONTENTS OF A corc CONFIGURATION FILE

The following is a sample configuration file for an installation of `corc`, which can be used to interact with a cloud resources solution. Note that additional configurations will be needed specific to the implementation, as well as additional credential files.

```
corc:
  configurers:
    ANSIBLE: {}
  job:
    capture: true
    meta:
      debug: false
      env_override: true
      name: ''
      num_jobs: 1
      num_parallel: 1
    output_path: /tmp/output
    working_dir: ''
  providers:
    aws: {}
    oci:
      cluster:
        domain: ''
        image: nielsbohr/mccode-job-runner:latest
        kubernetes_version: ''
        name: cluster
        node:
          availability_domain: lfcb:EU-FRANKFURT-1-AD-1
          id: ''
          image: Oracle-Linux-7.8-2020.09.23-0
          name: NodePool
          node_shape: VM.Standard2.4
          size: 1
      instance:
        availability_domain: ''
        operating_system: CentOS
        operating_system_version: '7'
        shape: VM.Standard2.1
        ssh_authorized_keys: []
      profile:
        compartment_id: ''
        name: DEFAULT
      vcn:
        cidr_block: 10.0.0.0/16
        display_name: VCN Patch Network
        dns_label: vcn
        id: ''
        internetgateway:
          display_name: default_gateway
          id: ''
          is_enabled: true
        routetable:
          display_name: default_route_table
          id: ''
          routerules:
          - cidr_block: null
            destination: 0.0.0.0/0
            destination_type: CIDR_BLOCK
            id: ''
        subnet:
          cidr_block: 10.0.1.0/24
          display_name: worker_subnet
          dns_label: workers
          id: ''
storage:
  credentials_path: /mnt/creds
  download_path: ''
  enable: false
  endpoint: 'https://ku.compat.objectstorage.eu-frankfurt-1.oraclecloud.com'
  input_path: /tmp/input
  output_path: /tmp/output
  s3:
    bucket_id: ''
    bucket_input_prefix: input
    bucket_name: ''
    bucket_output_prefix: output
    config_file: ~/.aws/config
    credentials_file: ~/.aws/credentials
    name: default
  upload_path: ''
```

Listing R.1: Contents of a `corc` configuration file.

S

MEOW WORKSHEET

This worksheet is part of the teaching material into MEOW, `mig_meow`, and how to use them to conduct scientific analysis. The complete materials are available at [34], under the *meow-workshop/* directory.

An Introduction to mig_meow

This document is intended as an introduction to mig_meow, a package for implementing MEOW workflows. This tutorial is designed to be run either on IDMC, or your own local machine, and requires only that you have python, and a text editor to work through.

We will be using standard Python files throughout this tutorial, though many of the files could be replaced with Jupyter notebooks if you are more comfortable with them. Some modifications may be needed in this case, so it is encouraged that you stick with python files for now.

Note that you only need to 2 one of the two following 'Getting started' sections.

IDMC – Getting started

If you wish to run MEOW workflows on IDMC then you will need to copy the source files using the sharelink: hRRLoSvSyK

Using the sharelink you can import the files to your own IDMC file system, or can download them to your own system.

To make sure we have all the software running we need to use the correct environment. This will happen automatically in a Notebook, but if we run something on the terminal you will need to start with by running:

```
conda activate python3
```

Make sure to place the source files in a new directory. All commands and paths presented in this tutorial assume you are operating in this directory. By default terminals on IDMC start in a home directory. To get to your IDMC files you should first use the following, replacing 'your_source_files_dir' with the directory you have stored the files in:

```
cd work/your_source_files_dir
```

You may also need check you have the most up to date version of mig_meow. Within a terminal run:

```
pip3 show mig_meow
```

If you do not get version 0.20 you should update using :

```
pip3 install mig_meow –upgrade --user
```

Local machine – Getting started

If you wish to run MEOW workflows on your own machine then you will need to copy the source files using the sharelink: hRRLoSvSyK

Using the sharelink you can import the files to your own IDMC file system, or can download them to your own system.

Make sure to place the source files in a new directory. All commands and paths presented in this tutorial assume you are operating in this directory.

Although not strictly necessary, it is a good idea to set up a virtual environment before you install any software. This helps keep it more stable, and will isolate other installs you have on your machine. To do this you will need virtualenv installed. If you don't have it you can find it here: https://virtualenv.pypa.io/en/latest/installation.html

To set up a new virtual environment you need to use the command line / terminal. You should run:

```
virtualenv venv
```

followed by

```
source venv/bin/activate
```

Once you're in the 'venv' virtual environment make sure to install mig_meow using:

```
pip3 install mig_meow
```

MEOW basics

Before we go further its worth mentioning what MEOW is. This should be explained more fully in an accompanying talk, but for now its sufficient to say that its a way of repeatedly scheduling processing on files. It does this by defining two parts. Firstly, Recipes are the processing that takes place. Secondly, Patterns are the conditions under which this processing is started. In mig_meow, these conditions are always file events, e.g. a file getting created or modified.

Recipes are Jupyter notebooks. 'addition.ipynb' is an example Recipe notebook. Although it is a trivial amount of processing, we can see that it takes some input, alters that input and produces some output. As long as your algorithm can be expressed in a Jupyter notebook, it can be a MEOW Recipe.

Patterns are objects, and will define an input path for some processing. This is a path, where if a file at that path is created or modified it will trigger some Recipe processing. The triggering file, is given to the defined Recipe and is processed.

Several of these Patterns and Recipes can be chained together to form a workflow, with processing able to start at any stage.

Defining a Recipes

To start our MEOW processing we are going to need to create a Python script. This can be done in any text editor, such as notepad, notepad++, or using an IDE such as PyCharm. Within JupyterLab this can be done using 'Text File' option on the Launcher screen. Lets create a new Python file called 'defining_a_recipe.py'. In it we should type:

```
import mig_meow as meow

# We need to read in a notebook as a Recipe.
my_recipe = meow.register_recipe('addition.ipynb')

# Check the name of our recipe.
print(my_recipe['name'])
```

That all we need to do to register a recipe. Note that the name will be the filename of the notebook by default. You can set a new name using ‘name=’, such as:

```
my_recipe = meow.register_recipe('addition.ipynb', name=’something_different’)
```

Note that all the real process definitions have taken place in the ‘addition.ipynb’ notebook. You should have a look at this before proceeding to check what it is it actually does. If you don’t have Jupyter installed, remember that it is available on ERDA/IDMC.

Once you’ve written ‘defining_a_recipe.py’ you can run it from the command line / terminal using

```
python3 defining_a_recipe.py
```

Defining a Pattern

Patterns a slightly more involved. Lets create a new file called ‘defining_a_pattern.py’.

```
import mig_meow as meow

# We must start by declaring a new Pattern object.
my_pattern = meow.Pattern('my_first_pattern')

# We need to add a Recipe. This is the Recipe that will run when a relevant file
# event happens.
my_pattern.add_recipe('addition')

# This defines our input file. Note that it will replace the 'infile' variable in
# the 'addition' Recipe. This will be replaced with the triggering file path.
# It also defines that path against which any file events are
# tested. Here we will match any events within the directory 'initial_data;.
my_pattern.add_single_input('infile', 'initial_data/*')

# This define our output varibles. Note that it will replace the 'outfile' variable
# in the 'addition' Recipe. This value will be replaced with the given output path.
# Note that the {VGRID} and {FILENAME} keywords will be replaced at runtime. We'll
# explain them more fully later.
my_pattern.add_output('outfile', '{VGRID}/my_output_1/{FILENAME}')

# This defines another variable. This will replace the 'extra' variable in the
# 'addition' notebook with the value 15.
my_pattern.add_variable('extra', 15)

# This will tell us if we've made some obvious mistakes. If it returns (True, '').
# Then we're good to go.
print(my_pattern.integrity_check())

# Save the pattern so we don't need to redefine all this next time
meow.write_dir_pattern(my_pattern)
```

This time we've also written our Pattern definition to disk using the 'write_dir_pattern' function. If you navigate to meow_directory/patterns' you should find a file called 'my_first_pattern' that shows the arguments we've given it. Recipes can be saved in the same way as this using the 'write_dir_recipe' function as well.

Starting a basic MEOW workflow

To get an actual workflow going we're going to need to define our Pattern and Recipe, and have a system to listen out for file events. This is where the LocalRunner comes in. This is a small system that takes definitions of Patterns and Recipes, and uses them to schedule processing when file events happen. Its designed to mimic functionality on IDMC, but can be used on your own machine and is a good way to test your workflow before you export it to IDMC.

Lets start another file called 'defining_a_workflow.py'

```
import mig_meow as meow

# Lets load up our pattern and recipe again.
my_recipe = meow.register_recipe('addition.ipynb')
my_pattern = meow.read_dir_pattern('my_first_pattern')

# We need to put them in a dictionary structure for the local runner
recipes = {
	'addition': my_recipe
}

patterns = {
	'my_first_pattern': my_pattern
}

# Here we start the runner and give it the Patterns and Recipes. the 'first_meow_workflow'
# directory is the base directory we will be operating in. The number 2 refers to how many
# workers we will start. These process jobs at the same time so more will make the runner go
# faster, but only whilst we have the hardware to support it.
meow.start_local_workflow('first_meow_workflow', patterns, recipes, 2)
```

When you run this you should get a small wall of text informing you that Patterns and Recipes have been added, and that Rules have been created. Nothing else will happen though, and if you look at the directory you started it in, you should have a new folder 'first_meow_workflow', which is currently empty. This is the base for our new workflow, and the runner is listening to events inside it. Leave the runner running for now, and lets add a file 'test_file.txt' inside the 'first_meow_workflow' directory.

Once you add the file, you should see a notification in the runner terminal, and ongoing notifications that there are no jobs in the queue. This is still good, and what we'd expect as we've created a file that doesn't match the Patterns input_path, which was 'initial_data/*'. We can add a folder 'initial_data' inside 'first_meow_workflow' though and start triggering some processing. We can add some data to our 'first_meow_workflow/initial_data' directory using the data pre-generated in example_data. If you give it a few second, you should see some feedback from some job processing, and a new directory as ‘my_first_workflow/my_output_1’.

If you want a quick way to read the numpy data, you can run the provided script ‘reader.py’. This expects a directory to be given to it, where it will print all numpy data files inside. You can use the commands:

```
python3 reader.py first_meow_workflow/my_output_1
```

and

```
python3 reader.py first_meow_workflow/initial_data/
```

to check that processing has taken place as we’d expect.

That’s cool and all, but we just set up a lot of extra steps to run one small bit of processing. The fun part is that this we run again for every file we add to ‘first_meow_workflow/initial_data’. If we add more data as it gets generated, or replace the data with more up to date results the processing will automatically take place again. Try adding more numpy files and see what happens.

Note that if we added non-numpy data (e.g. a txt file) our job will break as addition.ipynb expects a numpy file as its input. How might we fix this?

Chaining Patterns

The true power of MEOW is in chaining together different Patterns. Lets stop the previous runner if you haven’t already by hitting Ctrl+C a few times. Now we can start a new file called ‘defining_a_chain.py’

```
import mig_meow as meow

# Lets load up our pattern and recipe again.
my_recipe = meow.register_recipe('addition.ipynb')
my_first_pattern = meow.read_dir_pattern('my_first_pattern')

# Lets define new Pattern
my_second_pattern = meow.Pattern('my_second_pattern')

# We can reuse recipes in different patterns
my_second_pattern.add_recipe('addition')

# Here our input path matches up to the output path for my_first_pattern
my_second_pattern.add_single_input('infile', 'my_output_1/*')

# We should make sure to output somewhere new
my_second_pattern.add_output('outfile', '{VGRID}/my_output_2/{FILENAME}')

# We can use different values for variables, so the same recipe can give
# different results according to different patterns.
my_second_pattern.add_variable('extra', 1)

# This will tell us if we've made some obvious mistakes. If it returns (True, '').
# Then we're good to go.
print(my_second_pattern.integrity_check())
```

```
# Save the pattern so we don't need to redefine all this next time
meow.write_dir_pattern(my_second_pattern)

# We need to put them in a dictionary structure for the local runner
recipes = {
	'addition': my_recipe
}

patterns = {
	'my_first_pattern': my_first_pattern,
	'my_second_pattern': my_second_pattern
}

# Make a new base directory, just so we can compare results.
meow.start_local_workflow('second_meow_workflow', patterns, recipes, 2)
```

Once this is written and started add some inputs to ‘second_meow_workflow/initial_data’ to test that its functioning correctly. Obviously this is still a very small example but you can link together as many different Patterns and Recipes to form very complex chains of processing. Multiple Patterns can match the same inputs, and can produce zero, one, or many outputs allowing for all sorts of odd workflows that are difficult to define in other systems.

A slightly more complex example

As a final example lets create ‘defining_a_loop.py’:

```
import mig_meow as meow

# Lets get a load of predefined patterns and recipes
patterns = {}
for pattern in ['first_mult', 'second_mult', 'third_choo']:
	patterns[pattern] = meow.read_dir_pattern(pattern)

recipes = {}
for recipe in ['mult', 'choo']:
	recipes[recipe] = meow.read_dir_recipe(recipe)

# Make a new base directory, just so we can compare results.
meow.start_local_workflow('looping_meow_workflow', patterns, recipes, 2)
```

This uses some pre-defined Patterns and Recipes. Have a look at their definitions in ‘meow_directory’ to see if you can work out what they’re doing. The source notebooks have also been included as ‘mult.ipynb’ and ‘choo.ipynb’ for a clearer view of them. Add some data to ‘looping_meow_workflow/initial_data’ and see what happens.

You should get a loop of processing, with the key part being the ‘third_choo’ Pattern and ‘choo’ Recipe, which has 2 outputs. This will either act as input to the earlier ‘second_mult’ Patterns, or will output to a new ‘final’ directory based on if the data that has been read in is big enough or not. This loop is very unusual in a workflow system as it has indeterminate length, and shows off some of the unusual possibilities for MEOW.