Scientific Workflow Execution Using a Dynamic Runtime Model

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
“Doctor rerum naturalium”
der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von

Johannes Martin Erbel
aus Koblenz

Göttingen, Juni 2022

Betreuungsausschuss

Prof. Dr. Jens Grabowski,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Ramin Yahyapour,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Philipp Wieder,
Institut für Informatik, Georg-August-Universität Göttingen

Mitglieder der Prüfungskommission

Referent:	Prof. Dr. Jens Grabowski, Institut für Informatik, Georg-August-Universität Göttingen
Korreferent:	Prof. Dr. Ramin Yahyapour, Institut für Informatik, Georg-August-Universität Göttingen
Korreferent:	Prof. Dr. Helmut Neukirchen, Department of Computer Science, University of Iceland

Weitere Mitglieder der Prüfungskommission

Prof. Dr. Dieter Hogrefe,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Marcus Baum,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Julian Kunkel,
Institut für Informatik, Georg-August-Universität Göttingen

Tag der mündlichen Prüfung
01.07.2022

Abstract

Research projects consist of several kinds of steps covering, e.g., individual procedures to gather data, as well as different ways to process and analyze it. Moreover, the individual steps of the project consist of a sequence of tasks that together form a workflow. With the continuous advancements in computer science, scientists have more and more access to different kinds of infrastructures and tools which are suited for different types of experiments. Cloud computing is one of the premier infrastructures to perform experiments on, as it provides flexible, on-demand computing resources that are off-premise. Still, a uniform and platform-independent orchestration of these resources remains challenging, especially when various infrastructures and human interactions are required throughout the execution of a scientific workflow.

This thesis provides an approach to allow scientists to define infrastructural resources for individual tasks within their workflows and dynamically shift them throughout the workflow execution. To reach this goal, we couple recent advancements in cloud orchestration with runtime models and open cloud standards. Combined, we aim for highly tailored workflows while fostering the reuse of already existing methodologies built around the standards. To realize this objective, we build a cloud runtime model orchestration process based on the Open Cloud Computing Interface (OCCI) standard. We extend the OCCI data model with workflow elements and corresponding capabilities to model cloud deployments for individual workflow tasks. This allows forming a runtime workflow model that can be coupled to different systems such as production clouds or simulation environments. To demonstrate the feasibility of the approach, we perform several experiments to validate the standard conform orchestration process and assess the applicability of the runtime workflow model coupled to cloud infrastructures.

Our studies show that runtime models are a suitable knowledge base for adaptive behavior including the modeling and runtime representation of highly tailored workflows. We observe that the orchestration of cloud deployments and the execution of workflows follow a reoccurring pattern which can be described via a sequence of runtime states. Especially the uniform interface of OCCI allows for an automatic management and reuse of existing systems and standards. By monitoring and reflecting operational properties, the runtime model fosters decision-making processes not only for self-adaptive systems but also for human users. We show that the runtime model enables human-in-the-loop activities, allowing, e.g., to influence the control flow or the parallelization of workflow tasks at runtime. Furthermore, the runtime model can be attached to different environments allowing to test adaptation and workflow behavior.

Zusammenfassung

Forschungsprojekte umfassen in der Regel viele einzelne und unterschiedliche Berechnungen, z.B., um Daten zu erfassen oder zu verarbeiten. Diese Schritte bilden zusammen einen Workflow und können automatisiert werden. Mit den kontinuierlichen Fortschritten im Bereich der Informatik steht Wissenschaftlern ein immer größer werdender Pool an verschiedenen Arten von Infrastrukturen und Werkzeugen zur Verfügung. Diese eignen sich jeweils für unterschiedliche und spezifische Aufgaben. Eine der wichtigsten Infrastrukturen in der heutigen Zeit ist die Cloud, welche sich durch die flexible Bereitstellung von Rechenressourcen auszeichnet. Trotz der langen und hohen Industrieakzeptanz von Cloud Ansätzen bleibt eine einheitliche und plattformunabhängige Orchestrierung dieser Ressourcen eine Herausforderung. Zudem wird die Cloud-Orchestrierung komplizierter, wenn Infrastrukturen während eines Workflows gewechselt werden sollen und menschliche Interaktionen zur Laufzeit nötig sind.

Ziel dieser Arbeit ist es, eine vereinheitlichte Cloud-Orchestrierung unter der Nutzung von modellgetriebener Entwicklung und öffentlicher Cloud Standards zu schaffen. So können die nötigen Infrastrukturen von Workflows klar definiert und im Laufzeitmodell dynamisch bereitgestellt werden. Weiterhin wird das Erstellen von maßgeschneiderten Workflows möglich gemacht. Um dieses Ziel zu erreichen, entwickeln wir einen Orchestrierungsprozess, der auf dem Open Cloud Computing Interface (OCCI) Standard basiert. Außerdem erweitern wir den Standard insofern, dass Workflow-Aufgaben abgebildet und mit Infrastrukturen der Cloud direkt verbunden werden können. Somit entsteht ein Workflow-Laufzeitmodell, dass mit verschiedenen Systemen gekoppelt werden kann. Um die Durchführbarkeit unseres Ansatzes zu zeigen, führen wir zwei Fallstudien durch, die den Orchestrierungsprozess untersuchen und drei Fallstudien, die sich mit einem Workflow-Laufzeitmodell befassen, das direkt mit Infrastrukturen der Cloud gekoppelt ist.

Unsere Studien zeigen, dass Laufzeitmodelle eine geeignete Grundlage für adaptives Verhalten darstellen, inklusive der Laufzeitrepräsentation von Workflows. Die einheitliche Schnittstelle von OCCI ermöglicht eine automatische Verwaltung und Wiederverwendung bestehender Systeme und Standards. Die Abbildung von Attributen in dem Laufzeitmodell ermöglicht es, den Kontrollfluss oder die Parallelisierung von Workflow-Aufgaben zur Laufzeit manuell zu beeinflussen. Das Laufzeitmodell kann an verschiedene Umgebungen angebunden werden, sodass Anpassungen und Workflow-Verhalten einfach getestet werden können.

Acknowledgements

Now that my time as a PhD student comes to an end, I want to thank the people that supported me throughout my studies.

I especially want to thank Prof. Dr. Jens Grabowski who took me into his group and allowed me to study the wonders of distributed computing. He always provided me with feedback on how to improve and guided me through all aspects of my PhD.

I additionally, want to thank Prof. Dr. Ramin Yahyapour and Prof. Dr. Philipp Wieder who agreed to be part of my thesis advisory committee. After my presentations, they always provided me with valuable hints on how to pursue my goal. Furthermore, I would also like to thank Prof. Dr. Helmut Neukirchen, Prof. Dr. Marcus Baum, Prof. Dr. Dieter Hogrefe, and Prof. Dr. Julian Kunkel for providing their time for me.

Another person, I want to thank is Fabian Korte who supervised me during my Bachelor and Master studies. He is the person who taught me the beauty of distributed systems, which encouraged me to pursue my PhD in the first place.

Many thanks also belong to all of my former and current colleagues who accompanied me during my PhD and made sure that it was a great time overall. I will especially remember the lively discussions about current research at the lunch table, including the terrible jokes. I want to especially thank Alexander Trautsch, Fabian Trautsch, Patrick Harms, Ella Albrecht, Philip Makedonski, Steffen Herbold, Kolja Thormann, Emmanuel Dapaah and Zaheed Ahmed for proofreading this dissertation and being open to all of my questions.

Finally, I want to thank my mother Doris, my father Norbert, and my brother Markus who supported me not only during my PhD studies, but my whole life. They constantly encouraged me in the pursue of being a computer scientist. Even when the times were rough, they always had an open ear for my concerns. Last but not least, I want to dedicate all the thanks in the world to my wife Khyra. She was the calming influence during my studies and was always open for discussion about my research ideas. Even in the late hours of the night.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

MAPE-K *Monitor-Analyze-Plan-Execute-Knowledge*. 18, 19, 38, 43, 64, 93

ARGON *An infRastructure modelinG tool for clOud provisioniNg*. 30

ATL *ATL Transformation Language*. 16

AWS *Amazon Web Services*. 20

BPEL *Business Process Execution Language*. 33

BPMN *Business Process Model and Notation*. 11, 12, 33, 35

CAMEL *Cloud Application Modelling and Execution Language*. 30, 134

CAMF *Cloud Application Management Framework*. 32

CAMP *Cloud Application Management for Platforms*. 23, 32

CDMI *Cloud Data Management Interface*. 23

CI *Continuous Integration*. 72, 96, 135

CIM *Computation Independent Model*. 16, 17, 38

CloudML *Cloud Modelling Language*. 30, 134

COCCI *Comparing OCCI*. 129

CPU *Central Processing Unit*. 20, 50, 90, 93, 98, 112, 119, 129

CRUD *Create, Read, Update, Delete*. 24

CWG *Cloud Working Group*. 23

DAG *Directed Acyclic Graph*. 10, 11, 23, 26, 45, 138

DCG *Directed Cyclic Graph*. 10, 11, 14, 58

DOCCI *Deployment of OCCI*. 82

DSL *Domain-Specific Language*. 13, 30

EC2 *Elastic Compute Cloud*. 20

EDMM *Essential Deployment Metamodel*. 32

EGL *Epsilon Generation Language*. 16

EMF *Eclipse Modeling Framework*. 15, 30, 33, 71, 140

EOL *Epsilon Object Language*. 15, 71

Epsilon *Extensible Platform for Specification of Integrated Languages for mOdel maNagement*. 15

ETL *Epsilon Transformation Language*. 16

ETSI *European Telecommunications Standards Institute*. 23

FSM *Finite State Machine*. 23, 26, 50, 55, 76, 139

GENTL *GEneralized Topology Language*. 32

GME *Generic Modeling Environment*. 33

HDFS *Hadoop Distributed File System*. 86, 88, 109, 111

HPC *High Performance Computing*. 12, 34, 145

IaaS *Infrastructure as a Service*. 20, 21, 24, 25, 32, 38, 72

IaC *Infrastructure as Code*. 21, 22, 30, 98, 133, 138

IDE *Integrated Development Environment*. 30, 32, 82

IP *Internet Protocol*. 89, 104, 114, 118, 136

JSON *JavaScript Object Notation*. 24

M2M *model-to-model transformation*. 15–17, 37–39

M2T *model-to-text transformation*. 16

M@R *Models at Runtime*. 17

MDA *Model-Driven Architecture*. 13, 15, 16, 38, 39, 79

MDE *Model-Driven Engineering*. 2, 12, 13, 15–17, 29, 33–35

MOCCI *Monitoring with OCCI*. 73, 74, 77

MoDEMO *Model-Driven Elasticity Management with OCCI)*. 31

MoDMaCAO *Model-Driven Configuration Management of Cloud Applications with OCCI*. 25, 26, 31, 39, 42, 46–49, 73, 74, 76, 88, 97, 103, 105, 122, 123, 140

NIST *National Institute of Standards and Technology*. 19, 20

OASIS *Organization for the Advancement of Structured Information Standards*. 1, 26

OCCI *Open Cloud Computing Interface*. 1–4, 8, 23–25, 30, 31, 35, 37, 38, 40–42, 47, 52–54, 72, 76, 82, 85, 99, 102–105, 113, 138–141, 143–145

OCL *Object Constraint Language*. 14, 25, 33, 41, 56

OGF *Open Grid Forum*. 1, 23

OMG *Object Management Group*. 13, 15, 23

OOI *OpenStack OCCI Interface*. 71, 140

OOP *Object-Oriented Programming*. 13

OS *Operating System*. 20, 22, 39, 101

PaaS *Platform as a Service*. 20, 23, 25, 31, 32

PIM *Platform Independent Model*. 16, 17, 38, 39, 74, 103

POG *Provisioning Order Graph*. 45

PSM *Platform Specific Model*. 16, 17, 38, 39, 74, 103

QVT *Query/View/Transformation*. 16

REST *Representational State Transfer*. 24, 45

S3Mining *Supporting novice data miners in Selecting Suitable mining algorithms*. 33

SaaS *Software as a Service*. 20

SmartWYRM *Smart Workflows through dYnamic Runtime Models*. 71, 72, 77, 79, 143

SNIA *Storage Networking Industry Association*. 23

SSH *Secure Shell*. 22, 48

SWF *Scientific Workflow*. 1, 9, 11, 29, 33–35

SWFMS *Scientific Workflow Management System*. 1, 11, 12, 32–35, 63

TOSCA *Topology and Orchestration Specification for Cloud Applications*. 1, 2, 4, 8, 23, 26, 30, 31, 33–35, 38, 40, 41, 82, 85, 99, 100, 102–105, 138, 143

UML *Unified Modeling Language*. 11, 12, 14, 26, 30, 32, 33, 35, 41, 55, 56, 102, 136

URL *Uniform Resource Locator*. 122

VCS *Version Control System*. 121

VM *Virtual Machine*. 20, 24–27, 39, 48, 50, 52, 54, 62, 72, 74, 76, 82, 90, 91, 93, 95–97, 101, 103, 108–113, 116–120, 124, 125, 129, 130, 134, 136, 138, 141

WASA *Workflow-based Architecture to support Scientific Applications*. 33

WOCCI *Workflows and OCCI*. 80

XML *Extensible Markup Language*. 26, 71

XSD *XML Schema*. 26, 32

YAML *YAML Ain't Markup Language*. 26, 99, 100

YARN *Yet Another Resource Negotiator*. 86, 88

1 Introduction

Nowadays, the utilization of distributed resources plays an ever-increasing role in efficiently processing huge amounts of data from various sources for various domains [1]. Experiments often contain a sequence of calculations required to achieve new and more detailed insights about specific phenomena. Combined, the individual steps form a *Scientific Workflow* (SWF) [2], for which a multitude of languages exist to ease their creation. *Scientific Workflow Management System*s (SWFMSs) automatically process the described workflows [3]. While the description of a workflow heavily depends on the SWF language, a simple and abstract example is a sequence in which the first task is responsible to gather data, while the follow-up task processes it. Each task within this example requires a different amount of computing resources with different kinds of applications. The data gathering task may require the deployment of several web crawler applications, while the processing requires the deployment of a big data framework hosted on a computation cluster.

In recent years, a multitude of orchestration techniques have been developed to adapt running cloud infrastructures. These range from abstract infrastructure descriptions in models to the utilization of scripts in form of infrastructure as code. These cover, e.g., scripting languages, idempotent configuration management, or the fast deployment of preconfigured environments in the form of containers. The trend in these developments originate from the success of cloud computing services. These services allow the consumer to dynamically rent virtualized resources on demand with the illusion of an infinite amount of available resources [4, 5]. The high demand for flexible computing resources has led to a multitude of service providers offering cloud services via different interfaces. This resulted in the *provider lock-in* problem, which essentially binds a consumer to a provider once a cloud application has been built. To tackle this issue multiple approaches have been proposed including two standards, namely the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) by the *Organization for the Advancement of Structured Information Standards* (OASIS) [6], and the *Open Cloud Computing Interface* (OCCI), by the *Open Grid Forum* (OGF) [7]. Both standards define an extensible description language for cloud deployments with OCCI additionally specifying a uniform interface to manage modeled resources. However, no standard conform approach exists that utilizes this interface to orchestrate running cloud application topologies. Furthermore, a concept to reflect operational parameters directly within an OCCI model is still missing.

Even though many of the SWFMSs utilize cloud resources, several of the existing management systems rely on a preconfigured computation cluster without granting direct access to the infrastructure layer. Therefore, scientists are often restricted to the applications and infrastructures available to them at design time. Thus, all computation clusters and applications essential for their workflow need to be set up preemptively. This design time setup presents a manual effort that may hinder scientists from replicating studies. Moreover, it may drastically increase the resource consumption and thus costs for executing a workflow, as specific applications may only be required by a specific experiment or a single task within the workflow [8]. Furthermore, this restriction complicates the integration of self-developed or emerging computation frameworks into existing solutions. In addition to an efficient resource management, one of the remaining challenges for large scale scientific workflows is the integration of a human-in-the-loop to enable the monitoring and interaction with the running workflow [9].

In this thesis, we address these issues by means of a cloud orchestration process built around the OCCI standard. In our approach, we follow the *Model-Driven Engineering* (MDE) paradigm in which formalized models are utilized to abstract the domain and ease its access. Furthermore, we combine the description of cloud application and workflow management within a runtime model that is causally connected to the abstracted system. This connection ensures that changes to the model are directly propagated to the system, i.e., the cloud, and vice versa. Especially the utilization of runtime models for workflows, according to a literature study by Bencomo et al. [10], is currently missing in the state-of-the-art. In the scope of this thesis, we implemented two environments to which the runtime model can be connected. One which connects the model to the cloud and one that connects it to a simulation environment for testing and assessment purposes. Independent of the environment chosen for the causal connection, we enable scientists to interact with the workflow model and respond to intermediate results, even at runtime. From a system point of view, the direct connection of the model to infrastructural resources allows for the dynamic provisioning of distributed resources throughout the workflow execution. To foster the reusability and portability of the proposed approach, we show the feasibility of a standard conform cloud orchestration based on two case studies. These case studies highlight the orchestration and management capabilities of an OCCI based runtime model, as well as the interoperability of the TOSCA and OCCI standards. Based on the results of this orchestration process, we assess the applicability of a runtime and model-driven workflow management. For this we use three case studies from different domains requiring changing cloud deployments and workflow capabilities. The chosen workflow case studies deal with the utilization of big data framework integration, shifting resource requirements within a dynamic simulation application, as well as the compute-intensive processes of software repository mining. Each of the chosen case studies has different requirements highlighting the individual capabilities provided by the runtime model for the scientist, as well as self-adaptive control loops using it as a knowledge base. We use the results and observations of the executed case studies to discuss the applicability of a standard based runtime model and its utilization to execute workflows for academia, education and industry. Furthermore, we use the gathered insights to discuss different viewpoints on the runtime model as well as possible improvements to the OCCI standard.

1.1 Goals and Contributions

The overall goal of this thesis is to develop and show the applicability of a concept to dynamically manage shifting infrastructure requirements for the execution of scientific workflows over a runtime model that can be easily maintained due to its abstract nature. Figure 1.1 provides a short sketch of the approach presented in this thesis which can be separated into three parts. The **Workflow** represents the design time model to be created by the scientist with each task having its individual resource requirements. This model is then used to derive the required state of the **Runtime Model** holding only the information currently required to execute this workflow. The specific characteristic of this **Runtime Model** is that we **synchronize** it with a **Cloud**, i.e., each resource in the **Runtime Model** is provisioned in the **Cloud**. The thesis comprises two core contributions providing concepts to advance the body of knowledge in the fields of model-driven engineering, cloud computing and workflow execution. In the following, these contributions are explained in more detail:

- A **Standard conform cloud orchestration** (Section 4) in which we transfer the designed cloud application to the **Runtime Model** and **synchronize** it with a **Cloud** environment using the OCCI standard. While the **Runtime Model** represents the current state of the **Cloud**, a new desired state is reached over a set of model transformations deriving the required sequence of requests. These requests manage the infrastructure, as well as deployed applications by applying either configuration or container management. To increase the reflective capabilities of the runtime model, we extend the standard with monitoring functionalities. The added capabilities enable the deployment of sensors which reflect their observed operational parameters directly in the model. Our approach demonstrates a complete cloud orchestration lifecycle fully relying on available standards. We use this process to orchestrate the infrastructures required during the workflow execution. For example, in Figure 1.1, only the infrastructure for task **D** is contained in the **Runtime Model** and thus deployed in the **Cloud**.

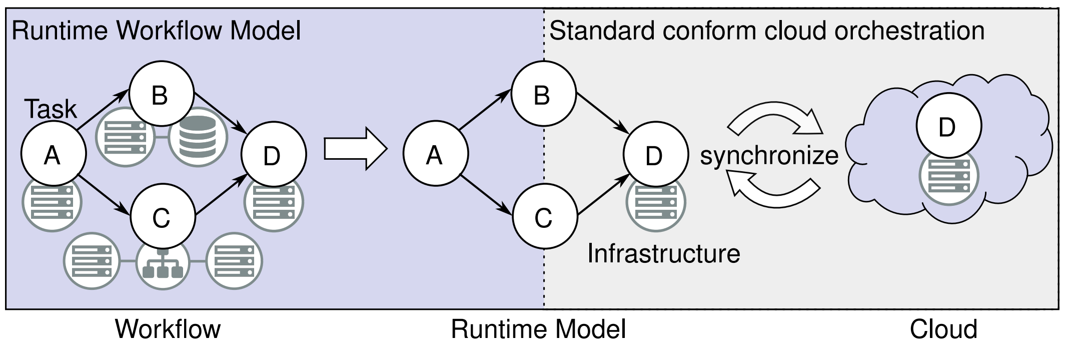


Figure 1.1: Overview: Thesis goal and contributions.

- The **Runtime Workflow Model** (Section 5) allows using arbitrary **Infrastructure** and application requirements for each individual **Task** in a **Workflow**. Therefore, we couple workflow tasks to infrastructural and application resources. The concept infuses the **Runtime Model** used to manage the **Cloud** deployment with information about the sequence of tasks to enact a **Workflow**. As part of this concept, we present a workflow execution engine that utilizes autonomous control loops with the workflow runtime model serving as a knowledge base. This engine schedules infrastructures required for individual workflow tasks by merging the design time and runtime model. The resulting model depicts the required cloud infrastructure at the specific point in time and serves as input for the **standard conform cloud orchestration** process. After orchestrating the infrastructure, the workflow engine triggers the enactment of pending tasks ready to be executed. This concept presents a novel approach to reflect and execute scientific workflows using a dynamic runtime model. The utilization of infrastructural descriptions allows for individual workflow tasks to be highly tailored to their needs, as well as the integration of a human-in-the-loop by design.

We evaluate the two contributions by performing case studies. From the results and the development of these case studies, we present three further contributions to the knowledge pool of working with cloud standards and workflows regarding the utilization of runtime models:

- **A cloud and simulation environment for the runtime workflow model** which we used to develop and execute our case studies (Section 6). This environment consists of a web application used for the cloud orchestration, as well as different effectors maintaining the runtime model. To facilitate the replication of our study, all implementations are publicly available [11].

- **Two feasibility studies of a standard conform cloud orchestration** demonstrating the runtime model capabilities of OCCI (Section 7). We highlight the reflective capabilities of OCCI (Section 7.1), as well as the compatibility of its interface to deploy cloud topologies conforming to the TOSCA standard (Section 7.2). The results demonstrate the feasibility and benefits of a cloud runtime model managed over a standardized and uniform interface and serve as a basis for further improvements of the OCCI cloud standard.

- **The assessment of the applicability of a workflow runtime model** by modeling and performing three scientific workflows showing the integration of existing frameworks (Section 8.1), the process of runtime decision-making including a human-in-the-loop (Section 8.2), as well as the deployment of self-developed frameworks requiring loops and parallelization within their workflows (Section 8.3). The results of the assessment reveal the benefits of a runtime model reflecting the workflow and its infrastructure over a sequence of states.

1.2 Impact

The results of this dissertation and further research that has been performed to enable this work have been published in two scientific journal articles and five peer-reviewed international conference proceedings. One of the author's journal publications was invited and one was accepted as a Journal First presentation at international conferences. Also, one conference publication is awarded with the "INSTICC Best Poster Award".

Journal Articles

- Stéphanie Challita, Fabian Korte, Johannes Erbel, Faiez Zalila, Jens Grabowski, and Philippe Merle. "Model-Based Cloud Resource Provisioning with TOSCA and OCCI." *Software and Systems Modeling (SoSyM)*, Springer Verlag, 2021. Invited for **presentation as a Journal First at MODELS 2021**.

  Own contributions
  I contributed to this publication by providing an initial mapping between TOSCA and OCCI elements. In this publication, I mainly focused on the orchestration of cloud deployments including the development of a required platform-specific transformation. Furthermore, I contributed by developing and executing the deployment of the presented case studies. Finally, I prepared and presented the contribution as a Journal First at the ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS).

- Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodriguez Perez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matúš Sulír, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tüzün, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James C. Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, Johannes Erbel. "Large-Scale Manual Validation of Bug Fixing Commits: A Fine-grained Analysis of Tangling." *Empirical Software Engineering (EMSE)*, Springer Verlag, 2021. Accepted for **presentation as a Journal First at ICSE 2022**.

  Own contributions
  I contributed to this publication by manually labeling data as part of a crowdsourcing approach as well as proofreading the journal article. Within this publication, the SmartSHARK framework is used which builds the basis for one of the workflow case studies presented in this thesis. The contribution got presented by Steffen Herbold at the 44th International Conference on Software Engineering (ICSE).

Conferences and Workshops

- Johannes Erbel, Alexander Trautsch, Jens Grabowski. "Simulating live cloud adaptations prior to a production deployment using a models at runtime approach ", *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 2021. Awarded with an **INSTICC Best Poster Award**.

  Own contributions
  I am the lead author of the publication. I performed most of the work for this publication, including the technical implementations. Alexander Trautsch contributed to the discussion of the simulation environment, the implementation of the replication set, and the execution of the case study.

- Johannes Erbel, Thomas Brand, Holger Giese, Jens Grabowski. "OCCI-compliant, fully causal-connected architecture runtime models supporting sensor management", *Proceedings of the 14th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019.

  Own contributions
  I am the lead author of the publication. I performed most of the work for this publication, including the technical implementations and the performance of the case studies. Thomas Brand contributed to the analysis of current problems within adaptive monitoring and the selection of an appropriate case study with respect to its external validity and the discussion of the study.

- Johannes Erbel, Stefan Wittek, Jens Grabowski, Andreas Rausch. "Dynamic Management of Multi-Level-Simulation Workflows in the Cloud", *Proceedings of the 2nd International Workshop on Simulation Science (SimScience)*, 2019.

  Own contributions
  I am the lead author of the publication. I performed most of the work for this publication, including the technical implementations, except the multi-level-simulation prototype developed by Stefan Wittek for the execution of the case study. Furthermore, Stefan Wittek contributed to the discussion of dynamic simulation requirements.

- Johannes Erbel, Fabian Korte, Jens Grabowski. "Scheduling Architectures for Scientific Workflows in the Cloud", *Proceedings of the 10th System Analysis and modeling Conference (SAM)*, 2018.

  Own contributions
  I am the lead author of this publication. All main contributions, implementations, and case studies have been done by me.

- Johannes Erbel, Fabian Korte, Jens Grabowski. “Comparison and Runtime Adaptation of Cloud Application Topologies based on OCCI”, *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)*, 2018.

  Own contributions
  I am the lead author of this publication. All main contributions, implementations, and case studies have been done by me.

Additionally, the author of this dissertation supervised two student projects and one bachelor thesis in relation to the overall topic of this thesis.

Bachelor Theses

- Lennart Thiesen. “Containerization in A Causally Connected Runtime Model for Scientific Workflows”, Bachelor Thesis, Institute of Computer Science, University of Goettingen, 2020.

Student Projects

- Nijat Rzayev. “Runtime metric detection and reflection in a workflow runtime model”, Student Project, Institute of Computer Science, University of Goettingen, 2020.

- Lennart Thiesen. “Development and integration of a Bash Connector into the MoDMaCAO framework”, Student Project, Institute of Computer Science, University of Goettingen, 2019.

1.3 Structure of the Thesis

This thesis covers several aspects related to the goals and contributions stated above. It is structured as follows.

Chapter 2 summarizes the foundations upon which the thesis is built. It includes foundations regarding scientific workflow enactment (Section 2.1), model-driven engineering (Section 2.2), cloud computing and its standards (Section 2.3).

Chapter 3 presents related work to the scientific topics to which the author contributed during his studies and puts our work into a broader research context. This chapter includes the work related to the topics of model-driven cloud orchestration (Section 3.1), model-driven workflow systems (Section 3.2) and an overview of related infrastructure aware workflow approaches (Section 3.3). In addition, a summary of the related work together with the research gap is given within this chapter (Section 3.4).

Chapter 4 introduces the standard conform cloud runtime model and its management by providing an overview while focusing on the description of the orchestration process (Section 4.1). Furthermore, we go into detail about how the runtime model is connected to the cloud to maintain a causal connection including the reflection of operational properties (Section 4.2).

Chapter 5 describes the concept of the runtime workflow model starting with an overview describing how the workflow elements are coupled to infrastructural resources, as well as their capabilities (Section 5.1). Finally, we introduce a workflow engine using the workflow runtime model as a knowledge base to schedule required infrastructures and trigger the execution of workflow tasks (Section 5.2).

Chapter 6 covers information about the ecosystem built around the OCCI cloud standard (Section 6.1). Among others, this section introduces infrastructure configurations including the connection of the runtime model to the cloud and the simulation environments. Furthermore, the chapter introduces the deployment and workflow framework to perform the case studies (Section 6.2).

Chapter 7 describes the execution of the runtime model orchestration case studies demonstrating the feasibility of a standard conform runtime model and orchestration process. This chapter provides an overview of the case study selection process, as well as the execution of a cluster scaling example (Section 7.1) and the deployment of cloud application topologies via the OCCI interface that originate from TOSCA (Section 7.2).

Chapter 8 assesses the applicability of workflow runtime models over three case studies. This chapter comprises the case study selection, followed by the description and execution of workflows from the domain of big data analytics (Section 8.1), dynamic simulations (Section 8.2), and software repository mining (Section 8.3).

Chapter 9 discusses the results of this thesis. We evaluate the applicability of a workflow runtime model for academia, education and industry (Section 9.1). Furthermore, we discuss the usefulness of the workflow runtime model from the viewpoints of personas, including scientists, cloud architects and systems utilizing the model as a knowledge pool for adaptive behavior (Section 9.2). Finally, we go over the standard conformity of the presented approach while presenting possible improvements (Section 9.3). Finally, this chapter covers the study's threats to validity (Section 9.4).

Chapter 10 concludes this thesis with a summary and an outlook on future work.

2 Fundamentals

This chapter introduces the foundations of this thesis consisting of different terminology and basic concepts. An overview of this chapter is visualized in Figure 2.1. Section 2.1 provides an introduction to the **Domain** of our research, namely **Scientific Workflows**. Section 2.2 presents the development **Technique** used throughout this study. In this section, the concept of **Model-driven Engineering** is introduced. Within this section, we especially highlight the concepts of models at runtime and runtime model based approaches. Section 2.3 gives an overview of the **Infrastructure** utilized in this thesis highlighting the advantages of **Cloud Computing** including current open standards.

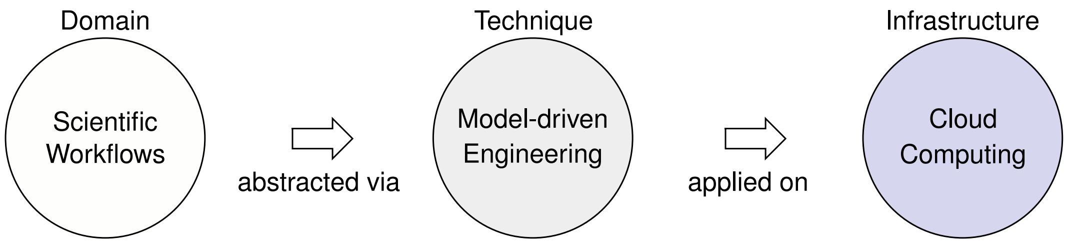


Figure 2.1: Overview: Foundations chapter.

2.1 Scientific Workflows

The term workflow is broad and can be found in several different domains ranging from manufacturing processes from the industrialization, over business processes, to information processes performing calculations [12]. Independent of the different determination of a workflow, they share the abstraction of describing a sequence of tasks that should either be performed manual or within an automated procedure. *Scientific Workflow*s (SWFs) distinguish themselves from other workflow types over the kind of activities being performed which again heavily depend on the kind of research area, experiment, and required analysis. Often, the analysis of the experiment itself can be described over a sequence of tasks or computations to be performed, also referred to as scientific workflow in the literature. In this thesis we consider a scientific workflow as a sequence of computation tasks to be performed and stick to the following definition [13]:

Definition 2.1 (Scientific Workflow)**.** *A scientific workflow is the computerized facilitation or automation of a scientific process, in whole or part, which usually streamlines a collection of scientific tasks with data channels and data flow constructs to automate data computation and analysis to enable and accelerate scientific discovery.*

While multiple definitions exist for scientific workflows, it boils down to a sequence of tasks operating on data for scientific purposes. One of the grand visions for scientific workflows is to plug in any scientific data resource and computational service [14]. Also, the inspection of data on the fly is of great interest, as well as being able to change parameters while re-executing only affected components. Workflows in general are often either control or data flow oriented. According to Ludäscher et al. [14], control flow oriented workflows are more used for business workflows, while data flow oriented are more often used for scientific reasons. The distinction between both workflow scenarios is blurry and differs mainly in the goals and priorities of the workflow [15]. This allows to intertwine techniques and frameworks from both domains as the concepts are rather similar [16–18].

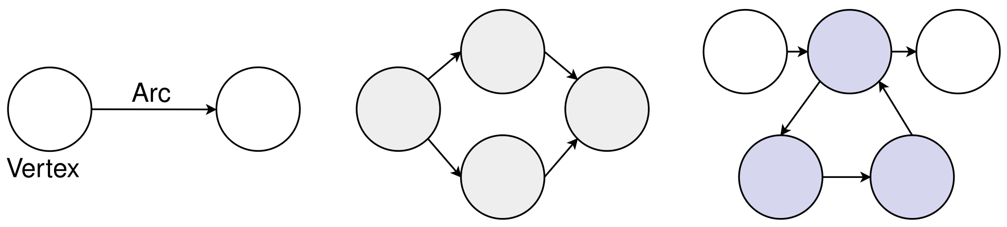
In the following, Section 2.1.1 describes the fundamentals of scientific workflows and their description languages. Section 2.1.2 introduces basics about engines and infrastructures used to process workflows.

2.1.1 Fundamentals

Workflows and thus scientific workflows are based on the same foundations. In this section, common mathematical concepts and fundamentals regarding workflows are introduced. The highest form of abstraction of a workflow is the one of a *directed graph*. Bang-Jensen and Gutin define directed graphs as follows [19]:

Definition 2.2 (Directed Graph)**.** *A directed graph (or just digraph) D consists of a non-empty finite set $V(D)$ of elements called vertices and a finite set $A(D)$ of ordered pairs of distinct vertices called arcs. We call $V(D)$ the vertex set and $A(D)$ the arc set of D. We will often write $D = (V,A)$ which means that V and A are the vertex set and arc set of D, respectively.*

Multiple notations and terms can be found in the literature defining directed graphs. Commonly used terms for vertices are, e.g., nodes and points. Arcs are also commonly known as edges or lines [20]. Throughout this thesis we use these terms interchangeably. Meanwhile, a multitude of languages have been developed around directed graphs. Often the graphs vertices $V(D)$ describe the workflow tasks while arcs $A(D)$ define the dependencies or data flow between them. In literature scientific workflow description are often separated into *Directed Acyclic Graph*s (DAGs) or a *Directed Cyclic Graph*s (DCGs), depending on whether the workflow contains loops [2]. Figure 2.2 provides examples for the different types of graphs used in this thesis. Figure 2.2 (a), shows a small directed graph, with the typical notation of a **Vertex** as a circle. In this example two vertices are connected by an **Arc**



(a) Directed graph. (b) Acyclic graph. (c) Cyclic graph.

Figure 2.2: Directed graph examples.

drawn as an arrow which directs from one vertex to another. Figure 2.2 (b) and (c) highlights the difference between an acyclic and cyclic graph. While the DAG does not contain cycles, i.e., none of the arcs in the graph create a closed loop, the DCG may consist of loops as visualized by the blue vertices.

Multiple models exist to describe the execution behavior or the logic behind a workflow, i.e., how to follow the control flow and when to execute which task. One of the most common models is the *petri net* [21] that can be used to describe a workflow execution [22, 23]. The concept of a petri net can be found, e.g., in activity diagrams from the *Unified Modeling Language* (UML) [24] or the *Business Process Model and Notation* (BPMN) [25]. These are respectively used to describe software system behavior or business processes using a sequence of actions. A petri net is a bipartite graph, i.e., a graph with two disjoint and independent sets of vertices. These sets of vertices are *places* and *transitions* which are interconnected by weighted arcs. In general, transitions represent events, e.g., the execution of a computation step or job, and places represent its pre- or post-conditions, e.g., the presence of resources or input data. Each place can be marked with a token representing that the condition of a place is met. A transition is *enabled* if each input place is marked with an amount of tokens at least equal to the weight of the arc from the input to the transition. Whether an enabled transition is fired, depends on whether the event takes place. As soon as the transition is fired the tokens are removed from the input place and transferred to each output place. To get a better insight about desired characteristics of workflows the following section goes into detail about common requirements and traits to automatically execute scientific workflows.

2.1.2 Scientific Workflow Management Systems

To interpret modeled workflows and automate their execution a *Scientific Workflow Management System* (SWFMS) is required [9]. Often these management systems directly complement the workflow language. Depending on the workflow and the infrastructure it is operating on, several kinds of system architectures exist. In complement to the definition of SWFs, Lin et al. [13] provide a definition for SWFMSs:

Definition 2.3 (Scientific Workflow Management System)**.** *A scientific workflow management system (SWFMS) is a system that completely defines, modifies, manages, monitors, and executes scientific workflows through the execution of scientific tasks whose execution order is driven by a computerized representation of the workflow logic.*

Depending on the computation infrastructure utilized workflows can be further divided into *in situ* workflows and *distributed* workflows [9]. In situ workflows exchange information over an internal storage or internal networks, e.g., of a *High Performance Computing* (HPC) system. In our approach, we focus on a distributed workflow concept in which tasks are loosely coupled and executed on distributed resources, e.g, on cluster, grid and cloud systems. As scientific calculations often require high amounts of computing resources, these languages are often tied to a specific infrastructure they are operating on. While the modeling of workflows is well investigated in the literature, we focus on how to create and infuse a basic workflow structure with infrastructural information using the technologies from the current state of the art of infrastructure management. Independent of the architecture and the infrastructure utilized, SWFMSs come with several desired characteristics and requirements. Typical requirements of scientific workflows comprise, but are not limited to seamlessly accessing remote services, reuse of workflows definitions, utilization of scalable infrastructure, reliability and fault tolerance mechanisms, data provenance, as well as a detached execution to handle long-running workflows with human interaction [14]. To cope with these challenges we combine techniques that provide practitioners with a description language for workflows and cloud deployments that operate on a high level of abstraction. These techniques refer to the area of model-driven engineering and are elaborated in the following section.

2.2 Model-Driven Engineering

To better understand and discuss software system designs and architectures, often models are used that are visualizing an abstract version of the behavior or structure of the system. Most often these models are utilized as a prescriptive or descriptive visualization to get an overview of the system and support a discussion [26]. One common example is the UML providing a standardized tool set of model elements to abstract computing systems, or the BPMN to design business processes. In the *Model-Driven Engineering* (MDE) paradigm these models are not only used as a visualisation of the system, but rather as a fundamental artifact within the software development process that describes the software system on a high level of detail [27]. Da Silva defines MDE as follows [28]:

Definition 2.4 (Model-Driven Engineering). *MDE is a software engineering approach that considers models not just as documentation artifacts but also as first-class citizens, where models might be used throughout all engineering disciplines and in any application domain.*

When applying MDE, the utilized models are formalized allowing them to be automatically processed, e.g., by transforming the model into another form or to generate code. A large community exists around this concept. Not only in science but also in industry, as reflected by the recurring Industry Day introduced in 2018 by the International Conference on Model-Driven Engineering Languages and Systems [29]. MDE brings technologies closer to non computer scientists by allowing users to work with *Domain-Specific Language*s (DSLs). These consist of a dedicated set of model elements from an application domain to ease the utilization of software solutions by domain experts. Szvetits and Zdun [30] define abstraction in the context of MDE as a way to "interact with the system by using models which are closer to the problem space". One of the most adopted MDE approaches is the *Model-Driven Architecture* (MDA) [31] specified by the *Object Management Group* (OMG), which focuses on the definition and transformation of models using several abstraction layers.

Section 2.2.1 introduces fundamental MDE terms. Section 2.2.2 describes common model-driven techniques used throughout this thesis. Section 2.2.3 explicitly highlights techniques used for runtime models.

2.2.1 Fundamentals

To understand MDE, the term *model* and its relationship to a *metamodel* is fundamental. Bézivin [32] highlights the importance of a model in MDE by comparing it to *Object-Oriented Programming* (OOP). While in OOP "everything is an object", in MDE "everything is a model". In the scope of this thesis, we refer to the term model as defined by Kühne [26]:

Definition 2.5 (Model)**.** *A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.*

Each model follows a prescriptive or descriptive purpose, describing either an already existing system or a system to be build [26]. To fulfill this purpose a model can be decomposed into three major features [33]. The *mapping*, *reduction*, and *pragmatic feature*. The mapping feature denotes that a model always refers to the system or object to be abstracted, also referred to as the *original*. Within the model, the original system is reduced to only a small selection of properties that are required to interpret the model for its specific purpose. Finally, the pragmatic feature of the model should be useful for its users, e.g., a human user or a system when using MDE. This feature allows users to interpret the model giving it a meaning in the end [27]. Each "model is an instance of a metamodel" [34] whereby the metamodel specifies elements, e.g., entities, resources, links and attributes, that can be used to create a model. In general, a metamodel can be defined as follows:

Definition 2.6 (Metamodel)**.** *A metamodel is a model of models [31] forming a language of models [35].*

The formality introduced by a metamodel allows validating its instances to check whether its "language" is correctly used. To allow for such validation, developers can infuse their

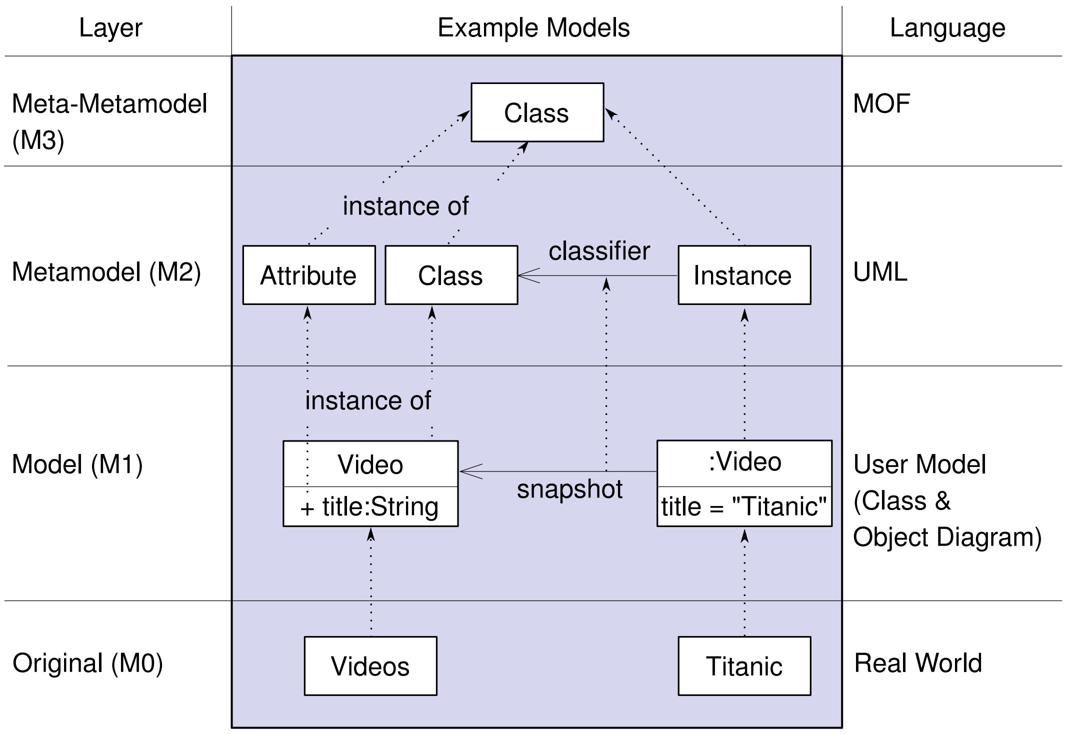


Figure 2.3: Example metamodel hierarchy (adapted from [34]).

metamodel with constraints that need to be fulfilled. For this, declarative languages, like the *Object Constraint Language* (OCL) [36], can be used, allowing for rules and expressions to be specified. In the context of an abstract DCG metamodel, a constraint may be that each arc always needs to connect to different vertices without forming a loop. This way errors in the model can be easily detected, especially when models tend to get larger or a multitude of models are used within a project. To create metamodels, a *meta-metamodel* is required which basically is a metamodel for metamodels [34]. Even though an arbitrary amount of meta-layers can be created, the most common is the utilization of four layers. Figure 2.3 provides an overview of these layers using UML class and object diagrams as an example.

In this figure, the notion of a video is abstracted over four layers. The layer **Original (M$_0$)** represents the **Real World** object to be abstracted. While **Videos** describe the type of object to be abstracted, the video **Titanic** represents a concrete instance. Within the **Model (M$_1$)** this original is abstracted as a class **Video** reducing its properties to a single attribute **title**. Based on this class, object instances can be created. These objects represent a **snapshot** of the class which in this case describes the movie **Titanic**. Both the class, and the object diagram are based on a **Metamodel (M$_2$)**, e.g., **UML**. Here, the **Video** class is an instance of **Class** and the **title** is an instance of **Attribute**. Finally, the metamodel itself is based on a description language often referred to as **Meta-Metamodel (M_3)**. One example is the *Meta-Object Facility* (**MOF**) [37], defined by the OMG. These meta-metamodels conform to themselves and provide a basic set of elements to be used for the creation of metamodels, such as a basic **Class** element. Even though the **MOF** is often referred to as meta-metamodel, it is strictly speaking a metamodel [34]. One example implementation for such a modeling stack is the *Eclipse Modeling Framework* (EMF) [38] often used within the model-driven community. In the following, techniques utilized within this thesis that operate on these different abstraction layers are introduced.

2.2.2 Techniques

Around the concept of models and metamodels several techniques, standards and frameworks have been developed to utilize the benefits of formalized information stored within models. These techniques range from but are not limited to, automatic transformations from one model to another, validation of models, and the generation of program code or graphical editors. These techniques enable the automatic creation of software artifacts which can result in reduced time efforts and lower mistakes made by manually created artifacts and improved human interaction [39]. One example framework is the *Extensible Platform for Specification of Integrated Languages for mOdel maNagement* (Epsilon) [40] which provides a set of languages compatible with EMF models that support code generation, model transformation and validation. The *Epsilon Object Language* (EOL) [41] builds the basis for the set of Epsilon languages and allows modifying EMF models by using typical programming constructs. In the following, we provide a definition of *model-to-model transformation*s (M2Ms) and an introduction to the MDA approach.

2.2.2.1 Model-to-Model Transformation

One of the most common techniques in MDE is the utilization of M2M transformations [42, 43]. These transformations define a set of transformation rules that describe how elements from one metamodel are translated into the elements of another metamodel. These rules are applied to model instances. Kleppe et al. define M2M transformations as follows [44]:

Definition 2.7 (Model-to-model transformation (M2M))**.** *A transformation is the automatic generation of a target model from a source model, according to a transformation definition.*

Definition 2.8 (Transformation definition)**.** *A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.*

Definition 2.9 (Transformation rule)**.** *A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.*

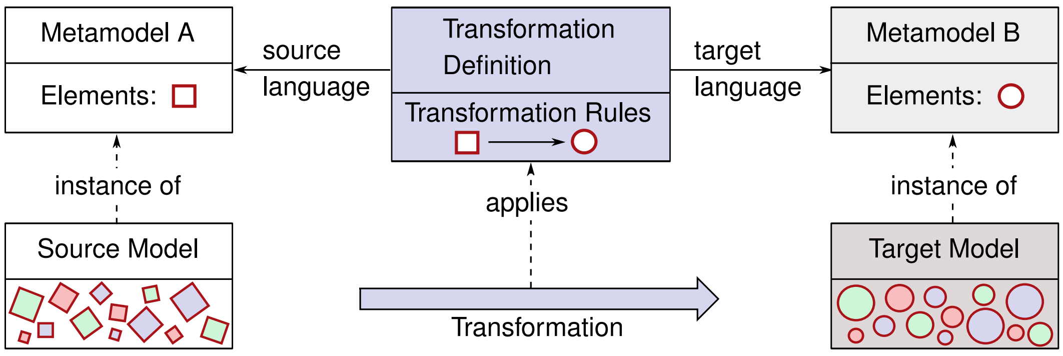


Figure 2.4: Model transformation concept (adapted from [31]).

The relationship between these different transformation terms is depicted in Figure 2.4. The figure shows an example transformation from a **Source Model** of **Metamodel A** to a **Target Model** of **Metamodel B**. Both metamodels define only one element, a square and a circle, which are identified over the shape, color and size. The **Transformation** consists of a **Transformation Definition** holding the set of **Transformation Rules**. In this example, only one rule exists which describes that squares should be transformed into circles while maintaining the remaining properties. Therefore, when this transformation is applied, the **Source Model** gets transformed into the **Target Model** so that each square is transformed into a circle. Meanwhile, a multitude of M2M transformation languages have been developed providing access to a different set of features, e.g., the *Query/View/Transformation* (QVT) [45] language, the *Epsilon Transformation Language* (ETL) [46], the *ATL Transformation Language* (ATL) [47,48] and Viatra [49–52]. One related technique is the *model-to-text transformation* (M2T) which targets a textual artifact rather than another model [28]. Common examples are the *Epsilon Generation Language* (EGL) [53] and Acceleo [54].

2.2.2.2 Model-Driven Architecture

The *Model-Driven Architecture* (MDA) is a specialized MDE approach which focuses on different viewpoints on a system. Hereby, selected details are suppressed within the model to reach a simplified version of it [31]. Within this approach, the utilization of the *Computation Independent Model* (CIM), *Platform Independent Model* (PIM) and *Platform Specific Model* (PSM) are specified which can be connected via M2M transformations. Within the MDA guide [31], Truyen [55] defines these three model abstractions as follows:

Definition 2.10 (Computation Independent Model (CIM))**.** *A CIM [...] presents exactly what the system is expected to do, but hides all information technology related specifications to remain independent of how that system will be (or currently is) implemented.*

Definition 2.11 (Platform Independent Model (PIM))**.** *A PIM exhibits a sufficient degree of independence [...] to enable its mapping to one or more platforms. This is commonly achieved by defining a set of services in a way that abstracts out technical details. Other models then specify a realization of these services in a platform specific manner.*

Definition 2.12 (Platform Specific Model (PSM))**.** *A PSM combines the specifications in the PIM with the details required to stipulate how a system uses a particular type of platform. If the PSM does not include all the details necessary to produce an implementation of that platform it is considered abstract (meaning that it relies on other explicit or implicit models which do contain the necessary details).*

Overall, the CIM hides all technological information, while the PIM is a generic representation of the model that should be applicable to any kind of related environment. The PSM, on the other hand, contains information specific to a target environment. Using this approach, the core information about a system can be retained increasing its portability. When required, this model can be infused with additional information using an M2M transformation. This approach allows preventing provider lock-ins by modeling a service at a high level of abstraction and thereafter adding provider specific information to the model in order to be automatically processed. In the following, we go into further detail about models at runtime.

2.2.3 Models at Runtime

Models at Runtime (M@R) is a subcategory of MDE that promotes the utilization of models not only as a static view on a system but rather as a live representation of it. This kind of models are named *runtime models* which M@R approaches build upon. Two definitions for a runtime model are often found in the literature which combined provide an in-depth definition of the term:

Definition 2.13 (Runtime Model)**.** *A runtime model is defined as abstraction of a running system that is being manipulated at runtime for a specific purpose [56]. [It maintains] a causally connected self-representation of the associated system that emphasizes the structure, behavior or goals of the system from a problem space perspective [57].*

The interconnection of a runtime model and the system is declared as a causal connection, which Maes defined as follows [58]:

Definition 2.14 (Causal Connection)**.** *A system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other.*

Among others, the utilization of runtime models helps to tackle inaccurate predictions and changing environments as, e.g., change impacts and system adaptation rules can be visualized before they are executed [30]. Furthermore, failing workflow resources [59], as well as violated constraints can be checked at runtime [60]. One of the major objectives of models at runtime approaches is the adaptation of the running system which needs to be adjusted due to changing requirements and other contextual changes [30]. The adaptation objective fits to the autonomic computing paradigm that fosters the vision of software systems working autonomously with each other and without the need for any human interaction or direct recognition of the system itself [61]. Common motives for adaptive computer systems are high frequencies of recurring adaptation tasks, required low reaction rates, as well as the need to process huge amounts of relevant information. Systems handling such requirements without needing human interaction are often referred to as self-adaptive systems which are defined by Lemos et al. as follows [62]:

Definition 2.15 (Self-adaptive systems)**.** *Self-adaptive systems are able to adjust their behavior or structure at runtime in response to their perception of the environment and the system itself.*

Self-adaptive systems are often implemented in form of a feedback loop. One common example of an automated control loop is the *Monitor-Analyze-Plan-Execute-Knowledge* (MAPE-K) control loop [64]. The MAPE-K loop, visualized in Figure 2.5, has four functions that share a common **Knowledge** base [63]. The **Monitor** function provides details about the system and its environment. Based on those details the **Analyze** function determines if changes are required. The **Plan** function elaborates the procedure to perform the changes and the **Execute** function triggers them. The shared **Knowledge** is the basis for adaptation and decision-making processes and consists, e.g., of policies and details about the running system that could be stored in a runtime model. The causal connection between a runtime model and the system can be used by the control loop to trigger system changes and thus often play a central role in the realization of a self-adaptive system [65].

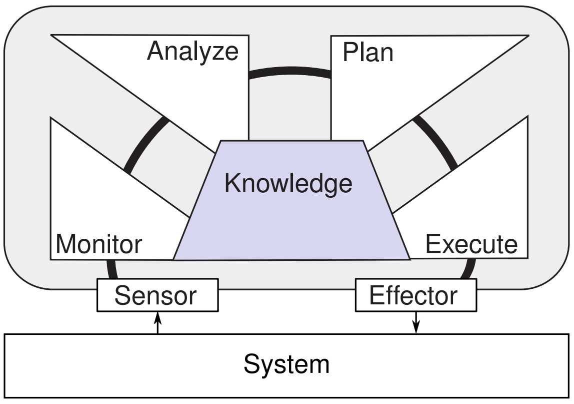


Figure 2.5: MAPE-K self-adaptive control loop (adapted from [63]).

To causally connect a **System** with a MAPE-K loop, a **Sensor** needs to be employed that grants access to it, e.g., to gather facts about the current workload of provisioned infrastructure resources. Such a **Sensor** may be used by the **Monitor** step. Furthermore, to adjust the **System** an interface to it needs to be provided. We refer to this interface as **Effector** which can be used by the **Execute** step. Among others, an **Effector** provides the logic to add, update or remove elements within the connected **System** to manage its resources.

2.3 Cloud Computing

Cloud computing has opened a new era of resource utilization and computing as a utility as an upfront commitment of required resources is no longer required [4, 5, 66]. At its core, cloud computing is a service in which virtualized chunks of a large pool of physical computing resources can be rented from a provider [67]. The sheer amount of resources that can be dynamically rented on demand removes the need to plan ahead of time. This allows to quickly react to changing computation needs and requirements. The *National Institute of Standards and Technology* (NIST) defines cloud computing as follows [67]:

Definition 2.16 (Cloud Computing)**.** *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

In the following, Section 2.3.1 goes into detail about cloud fundamentals and virtualization techniques. Section 2.3.2 describes the current state of the art to orchestrate cloud resources. Finally, Section 2.3.3 provides an overview of the cloud standards used within this thesis.

2.3.1 Fundamentals

The NIST definition for cloud computing provides an overall view on the topic and is one of the most referenced specifications in literature. The definition is separated into three categories: characteristics (i), deployment models (ii), and service models (iii).

Characteristics (i) defined by NIST form de-facto requirements for cloud services. For this paragraph, we highlight these NIST characteristics in *italic*. At its core, a cloud comprises a physical *resource pool* from which virtualized chunks can be rented using an *on-demand self-service* requiring no interaction with administrators from the provider side. For this, *broad network access* is required which supports standard mechanisms for a variety of clients. A *rapid elasticity* ensures that the offered resources can be quickly provisioned and released to enable dynamic scaling. Finally, a *measured service* needs to be provided which monitors the offered and rented resources leveraging metering capabilities for the provider and consumer.

Deployment models (ii) refer to where the cloud provider infrastructure is hosted, who is administrating it and from whom it is accessible. A public cloud is available for the general public and owned by a company, such as the *Elastic Compute Cloud* (EC2) cloud by Amazon [68]. A community cloud is a cloud environment shared by multiple organizations, while a private cloud is operated for use by a single organization. Finally, a hybrid cloud is a combination of two other deployment models which may be used, e.g, to store sensitive data in a private cloud while using computation capabilities offered by a public cloud.

Service models (iii) describe the level to which a consumer has access to the shared resources. *Software as a Service* (SaaS), *Platform as a Service* (PaaS), and *Infrastructure as a Service* (IaaS) are the most common ones and are specified in the NIST definition as well. Throughout the years, the kind of services offered by cloud computing providers drastically increased [69]. SaaS provides the end-user with a software application ready to use such as office programs and email services like Google's GSuite [70]. PaaS offers configurable runtime environments and applications ready to be used by a development team. Common examples are the Google App Engine [71] and the Elastic Beanstalk [72] from the *Amazon Web Services* (AWS). The approach introduced in this thesis attaches to IaaS as it allows to directly rent virtualized infrastructure resources and thus provides the most flexibility. Common examples of an IaaS cloud are the AWS EC2 [68] and OpenStack [73] as open-source solution. The two most common types of compute nodes offered by this service are *Virtual Machine*s (VMs) and containers. For the remainder of this thesis, we stick to the NIST definition of IaaS [67] and the definition of VMs and container instances as compute nodes by Brikman [74] :

Definition 2.17 (Infrastructure as a Service (IaaS))**.** *The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.*

Definition 2.18 (Virtual Machine)**.** *A Virtual Machine (VM) emulates an entire computer system, including the hardware. You run a hypervisor [...] to virtualize (i.e., simulate) the underlying CPU, memory, hard drive, and networking.*

Definition 2.19 (Container)**.** *A container emulates the user space of an OS [...][with] isolated processes, memory, mount points, and networking.*

VMs are fully isolated from the host machine, as well as from other VMs on the same host. Each VM runs a separate *Operating System* (OS) which results in a high start-up time as well as an overhead regarding *Central Processing Unit* (CPU) and memory utilization. To circumvent this drawback, VMs are often used to host container nodes in which cloud applications or components are deployed. These operate on a virtualized user space rather than virtualized hardware and can be created in milliseconds with next to no CPU or memory overhead [74]. This, however, comes with the drawback of sharing the hosts OS, kernel and hardware and therefore are less isolated. Next, we introduce fundamental terminology and technology regarding cloud orchestration.

2.3.2 Cloud Orchestration

To fulfill arbitrary requirements for research infrastructures, IaaS can be used to virtualize hardware composition. Still, the provisioned resources need to be configured and managed, e.g., by deploying and monitoring a specific application. Cloud orchestration describes a managed process to automatically provision and deploy cloud applications in the correct order. Liu et al. define the orchestration of IaaS cloud resources as follows [75]:

Definition 2.20 (Cloud Orchestration)**.** *Cloud orchestration involves the creation, management and manipulation of cloud resources, i.e., compute, storage and network, in order to realize user requests in a cloud environment, or to realize operational objectives of the cloud service provider. User requests are driven by the service abstraction and service logic that the cloud environment exposes to them.*

When looking into the literature, different features required by cloud orchestration tools can be found. Baur et al. [76] define four requirements of orchestration tools to manage a cloud applications lifecycle. These include capabilities to define, deploy, monitor and manage cloud applications. Brikman [74] defines a similar feature set required for real world use cases. This feature set reflects the current capabilities of existing tools and comprises, e.g., automated healing capabilities monitoring and replacing failing nodes. But also service discovery, auto-scaling, load balancing and the functionality to roll out updates on multiple compute nodes. To achieve these orchestration goals, often *Infrastructure as Code* (IaC) techniques are used. These techniques allow specifying software artifacts that contain infrastructure compositions and application deployments in a declarative manner. A definition of IaC is given by Brikman [74]:

Definition 2.21 (Infrastructure as Code)**.** *[...] infrastructure as code [allows to] write and execute code to define, deploy, update, and destroy your infrastructure. All aspects of operations [are treated] as software – even those aspects that represent hardware (e.g., setting up physical servers).*

The treatment of infrastructure configurations in the form of software artifacts allows for "consistent, repeatable routines to provisioning and changing systems and their configuration" [77]. This results in an easier management of distributed infrastructures. Brikman [74] defines several broad categories of tools used for IaC which are used for different purposes to orchestrate cloud applications. Figure 2.6 highlights three common IaC techniques and sets them in relation to written **Script** artifacts, utilized **Engines** and access methods regarding the **Infrastructure** to be managed.

An **Infrastructure Configuration** describes the composition of infrastructural resources and serves as input for a **Provisioning Engine**. These engines, such as Terraform [78], CloudFormation [79] and OpenStack Heat [80] orchestrate the infrastructure to reach the desired state. Therefore, access to the IaaS layer is required using, e.g., the interface of the **Cloud** provider. To configure the spawned infrastructure, an **Ad hoc Script** can be utilized.

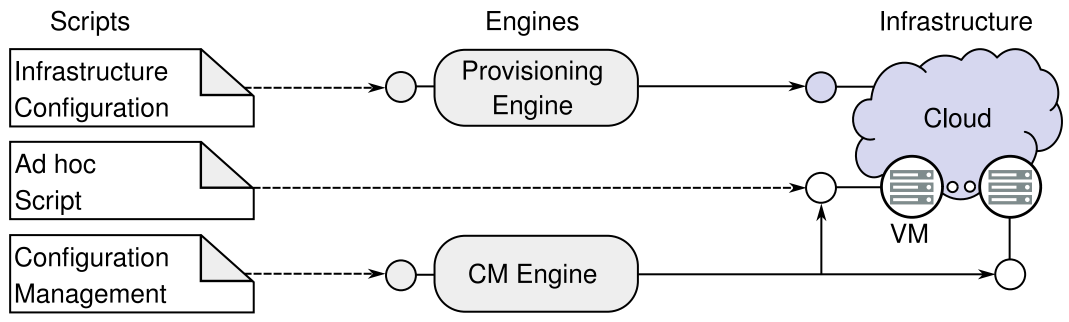


Figure 2.6: Infrastructure as Code techniques.

These refer to common scripting languages like Bash [81] and Python [82] and are used to automate tasks on the server. For this, direct access to the spawned **VM** or compute node is required. More profound IaC tools provide further and more complex functionalities, e.g., by ensuring that a consistent state of the resources is maintained. Using an **Ad hoc script** may not be sufficient to manage large computation clusters, as these kinds of scripts are typically imperative and access each **VM** individually leaving them prone to manual errors. To mitigate this problem, **Configuration Management** and a **CM Engine** can be utilized which handles the overall installation and management process of software on provisioned infrastructure [74]. In this thesis we refer to configuration management as follows [83]:

Definition 2.22 (Configuration Management)**.** *Configuration management is a way of handling changes in a system using a defined method so that the system maintains its integrity over time. A log is kept of every change made to a system along with documentation about who made the change, when the change was made, and why it was made. This allows us to know the exact state of a system at any moment in time.*

Common examples for configuration management tools are Chef [84], Puppet [85], Ansible [86], and SaltStack [87]. Compared to simple ad hoc solutions these configuration management tools are designed to configure multiple distributed resources using, e.g., *Secure Shell* (SSH) connections. Furthermore, these tools often introduce "Idempotent tasks [which] can be executed multiple times always yielding the same result" [88]. This helps to ensure that the written scripts leads to the state described in the configuration management script. Further, configuration management systems often offer the options for parallel and rolling deployments. It should be noted that applications to be deployed can already be part of, e.g., a container. This technique is referred to as templating which can be used "to create an image of a server that captures a fully self-contained "snapshot" of the OS, the software, the files and all other relevant details." [74]. This technique is often used to deploy a compute resource with common functionalities pre-installed before getting customized by additional configuration management procedures. Up next, we introduce cloud standards utilized within this thesis.

2.3.3 Cloud Standards

The success of cloud computing has led to various cloud providers each offering similar resources but through heterogeneous interfaces, semantics [89], and utilized technologies [90]. These differences result in a lack of interoperability [91], and the problem of a vendor or provider lock-in. This lock-in makes it difficult to switch from one provider to another [92]. In the literature study by Silva et al. [93] many approaches are enumerated that tackle this issue. One of these solutions is the specification and utilization of cloud standards developed by committees. Standardization groups include the OMG *Cloud Working Group* (CWG), providing vendor-neutral guidance for cloud interoperability and portability [94]. Standardized specifications for certain parts of a cloud system cover, e.g., the *Cloud Data Management Interface* (CDMI) [95] by the *Storage Networking Industry Association* (SNIA), as well as specifications on how to test these interfaces defined by the *European Telecommunications Standards Institute* (ETSI) [96]. Some standards are tailored towards specific service layers such as the *Cloud Application Management for Platforms* (CAMP) [97] for PaaS. Finally, standards specifying a data model for cloud deployments exist such as the *Open Cloud Computing Interface* (OCCI) [98] and the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [6].

In the scope of this thesis, we focus on the OCCI standard [98]. In addition, we demonstrate the interoperability of OCCI, and thus our orchestration approach, to the standardized TOSCA [6] cloud language. We describe both standards and existing extensions next.

2.3.3.1 Open Cloud Computing Interface

OCCI is a cloud standard, developed by the *Open Grid Forum* (OGF). The standard provides an extensible data model accompanied by the specification of a uniform interface to manage modeled elements.

The OCCI Core Model [98], shown in Figure 2.7, introduces a set of core elements meant to be extended in order to manage cloud resources. This extensible core is separated into two categories. The **Core base types** and the **Classification and identification** mechanisms which follow a type-instance pattern. The **Core base types** are composed of the abstract element **Entity** and its specializations **Resource** and **Link**. Instances of those types represent actual cloud resources that can be managed over the OCCI interface. OCCI models form a DAG with a **Resource** being a node that contains a set of **Link** instances. Each **Link** possesses a **target** pointing to another **Resource**. Each **Entity** is of one specific **Kind** which, combined with its **id**, uniquely identify it. **Kind** represents the main element of the **Classification and identification** mechanism of OCCI and defines a set of **Attribute**s to be filled with values by the **Entity** as well as **Action**s that can be performed on it. An **Action**, when triggered, affects the state of the resource as described by a *Finite State Machine* (FSM). In addition to a **Kind**, each **Entity** may have multiple applied **Mixin**s introducing new capabilities to the **Entity** at runtime, e.g., in form of **Attribute** and **Action** instances. **Kind**, **Mixin**, and **Action** inherit from the abstract

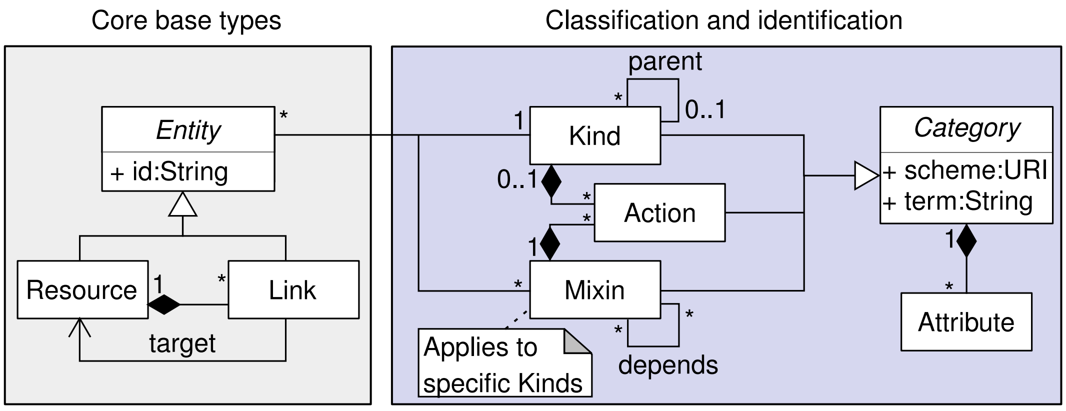


Figure 2.7: OCCI's Core Model (adapted from OCCI's Core specification [98]).

Category element and thus can be identified over a **scheme** and **term**. The **term** represents the name of the **Category** instance while the **scheme** is a namespace that often refers to the elements OCCI extension. Both **Kind** and **Mixin** possess generalization capabilities. A **Kind** can only possess a single **parent**. However, a **Mixin** can have multiple **depends** relationships allowing it to inherit and combine several capabilities at once.

The OCCI interface [99] is based on the *Representational State Transfer* (REST) [100], a stateless application-level protocol that allows managing hypermedia resources. Hereby, the interface supports the default *Create, Read, Update, Delete* (CRUD) operations, as well as the execution of defined actions on specific entities. Single entities can be addressed over their **Kind** and **id**. Also, complete collections can be managed over single requests by referring to all entities of a specific **Kind** or which have specific **Mixin** instances applied. To work with the interface, OCCI defines several renderings in its specification suite covering, e.g., a text based [101] and *JavaScript Object Notation* (JSON) based rendering [102].

2.3.3.2 Open Cloud Computing Interface Extensions

OCCI extensions utilize the classification and identification mechanisms to expand the standard with new kinds to manage arbitrary entities. The extensions used in this thesis are shown in Figure 2.8 with OCCI resource types in white and link types in gray.

The **Infrastructure** extension [103] adds specialized types to manage typical IaaS resources including **Storage**, **Network** and **Compute** elements. The **Compute** type, e.g., abstracts a VM or container resource and provides attributes to define the amount of memory, as well as actions to start and stop the machine. Each infrastructure type possesses an active and inactive state including actions to transition between them, such as start and stop. Additionally, each resource type provides several attributes, e.g., to specify the amount of memory of a compute node, the address range of a network, or the size of a storage. **Compute** nodes can be linked to a **Storage** or **Network** instance via a **StorageLink** or **NetworkInterface** respectively, serving

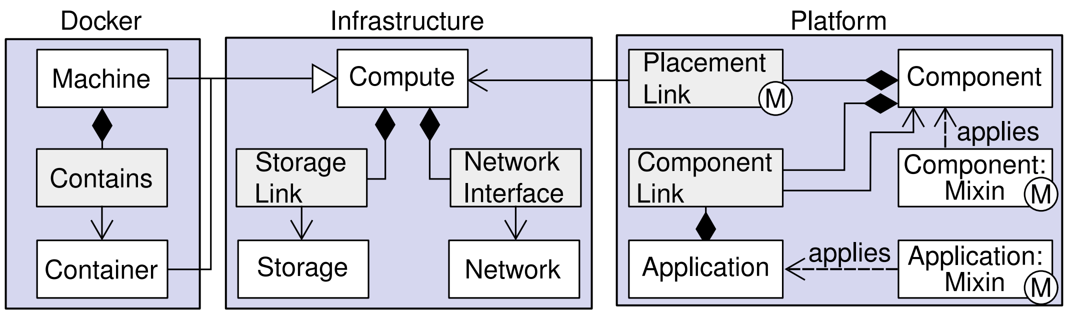


Figure 2.8: Utilized OCCI extensions with resource types in white and link types in gray.

as an interface to it. Among others, these links describe the mount point of the storage as well as the desired network address of the compute resource. The extension specifies several mixins to further extend the capabilities of infrastructure resources adding, e.g., the address range of a network or preset configurations for VMs.

The **Docker** extension by Paraiso et al. [104] introduces the management of containerized resources to OCCI by building upon the **Infrastructure** extension. It consists of two main resource types **Machine** and **Container** which are both specializations of **Compute**. The **Machine** type represents a typical VM that serves as a host for a Docker **Container** which is modeled via a **Contains** link.

The **Platform** extension [106] supports the management of PaaS. An **Application** represents a user-defined service, e.g., a computation cluster, that is composed of **Component** instances that implement the service's business logic. To abstract this behavior, the extension introduces the **ComponentLink** type to interconnect **Component** instances in order to form a larger **Application**. The *Model-Driven Configuration Management of Cloud Applications with OCCI* (MoDMaCAO) framework [105] enhances this extension and enables application management for IaaS services. For this, MoDMaCAO introduces three new elements denoted with an Ⓜ in Figure 2.8. The **PlacementLink** type describes where a **Component** is deployed by targeting a **Compute** node. The **Application:Mixin** allows specifying OCL constraints to support validation features. Finally, the **Component:Mixin** manages the lifecycle of a **Component** over a configuration management script that is executed on its compute host. For a more precise management the framework enhances the lifecycle of OCCI **Component**

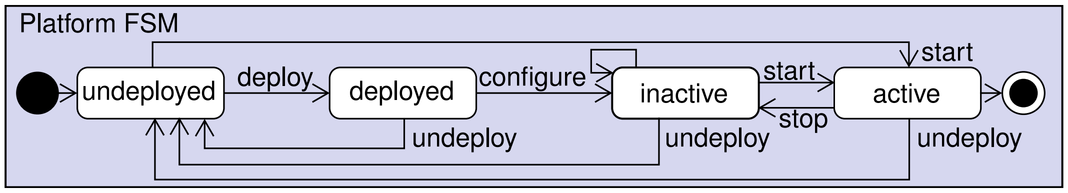


Figure 2.9: Finite state machine of applications and components (adapted from [105]).

and **Application** elements. This lifecycle is depicted in Figure 2.9 as an FSM using an UML state machine diagram. The common lifecycle follows the **undeployed**, **deployed**, **inactive** and **active** states. To traverse through these states, MoDMaCAO introduces lifecycle actions, i.e., **deploy**, **configure**, **start**, **stop** and **undeploy**. To manage this lifecycle with fewer action requests, shortcuts allow, e.g., to directly traverse the complete lifecycle from **undeploy** to **active** using a single **start** action request. When triggered, these actions execute a specific block within a linked configuration management script.

2.3.3.3 Topology and Orchestration Specification for Cloud Applications

TOSCA is a cloud standard maintained by *Organization for the Advancement of Structured Information Standards* (OASIS). Similar to OCCI, the standard provides a language to design cloud deployments [107]. In addition to the definition of deployment topologies, TOSCA allows modeling management procedures to modify services via orchestration processes called plans. In this thesis, we focus on the cloud deployment description to investigate similarities with the OCCI standard. Currently, two versions of TOSCA can be found. One is based on the *YAML Ain't Markup Language* (YAML) [108] and the other is based on the *Extensible Markup Language* (XML) [107]. While the YAML version does not provide a formal metamodel, the XML version is accompanied by an *XML Schema* (XSD). According to the TOSCA XML specification, the language's "focus is on design time aspects, i.e. the description of services to ensure their exchange. Runtime aspects are addressed by providing a container for specifying models of plans which support the management of instances of services." [107]. A simplified subset of the structure of the TOSCA metamodel is shown in Figure 2.10. Overall, the metamodel can be separated in the **Topology Template** which is accompanied by a **Type Definition**. Together, both parts form a type-instance pattern similar to the one used within the OCCI data model.

The **Topology Template**, i.e., the cloud deployment model, is composed of abstract **Entity Template** elements. These aggregate a set of **Properties** to describe the individual template elements. The **Node Template** and **Relationship Template** are the main elements of the **Topology Template** which together follow the notion of a DAG. While a **Node Template** specifies, e.g., a VM to be provisioned, a **Relationship Template** is used to model a relationship between **Node Template** instances. These can be used, e.g., to describe on which VM a specific component is hosted. TOSCA allows specifying **Capability** and **Requirement** elements that are part of a **Node Template**. While a **Capability** describes certain abilities provided by a node, a **Requirement** describes its consumption. An example of such a relationship is the capability of a node providing a runtime environment to be consumed by a node requiring one. In addition, TOSCA specifies further template types such as artifact and group templates used, e.g., to scale a group of nodes together.

The **Type Definition** provides a set of **Entity Type** instances that can be used within the topology template, as each **Entity Template** is of one **Entity Type**. Each template specialization is constrained for the utilization of a specific entity type, e.g., the type of a **Node Template** needs

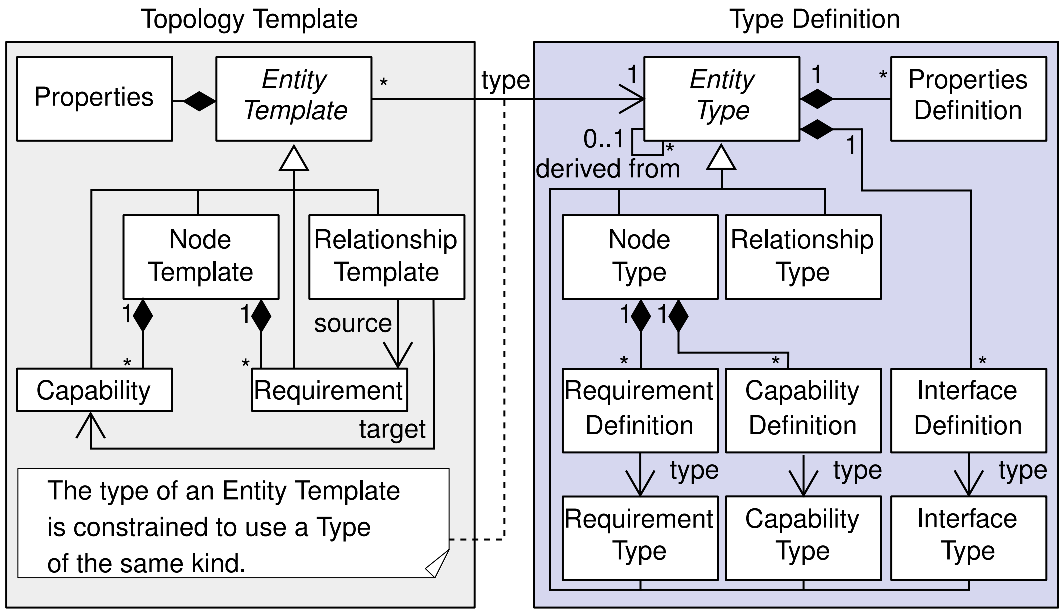


Figure 2.10: TOSCA metamodel subset.

to be **Node Type**, while the type of a **Relationship Template** needs to be a **Relationship Type**. The abstract **Entity Type** contains a **Property Definition** specifying a set of attributes to be filled by the template. To allow for generalizations of designed **Entity Type** instances, these can be **derived from** another **Entity Type**. The **Requirement Definition** describes a dependency of a **Node Type** or **Node Template** that needs to be fulfilled by a **Capability Definition** that matches it. The corresponding **Requirement Type** and **Capability Type** elements represent reusable entities to be exposed by a **Node Type**. A similar pattern can be found for **Interface Definition** and **Interface Type** which are used to define a named interface. These serve as reusable entities which define operations that can be included as a part of a **Node Type** or a **Relationship Type**. Hereby, each operation defined within the interface can be associated with code or scripts that can be executed later on by an orchestrator. These scripts implement lifecycle operations of applications and allow to transition between states, e.g., a stop action which shifts the application from an active to inactive state.

Similar to OCCI, the type-instance pattern pursued by TOSCA allows for customized extensions to be defined. While OCCI defines dedicated extensions for individual service layers, TOSCA provides a set of normative types introducing common base types to describe, e.g., VMs nodes. The standard states, that especially these normative types need to be supported by approaches implementing the standard. While the structure of the elements is rather similar, TOSCA has a more design time focus with OCCI focusing more on runtime aspects, especially due to the specification of a uniform interface corresponding to the elements within the data model.

3 Related Work

In this thesis, we orchestrate cloud deployments using a runtime model that is based on a standardized and uniform interface. Furthermore, we use this orchestration to schedule and deploy infrastructure requirements for individual tasks within a scientific workflow. Within this chapter, we introduce the related work to each of these fields and delimit our work to existing approaches. As shown in Figure 3.1, we categorize the related work into three sets based on the intersections of the thesis foundations (**MDE**, **SWF**, **Cloud**) with the **Research Gap** building the intersection of all three foundation sets.

In Section 3.1, we investigate the intersection between the utilization of model-driven techniques and cloud computing and provide an overview of cloud orchestration techniques utilizing models (**MDE ∩ Cloud**). Within this section, we especially consider the utilization of standards. Section 3.2 presents general workflow approaches that can be found in the literature with a special focus on the application of model-driven techniques in the workflow domain (**SWF ∩ MDE**). Section 3.3 introduces infrastructure aware workflow management approaches which consider the management of infrastructure directly in the workflow (**SWF ∩ Cloud**). Finally, in Section 3.4 we give a summary of the identified related work and define our **Research Gap**, covering the intersection of all three foundation sections (**SWF ∩ MDE ∩ Cloud**).

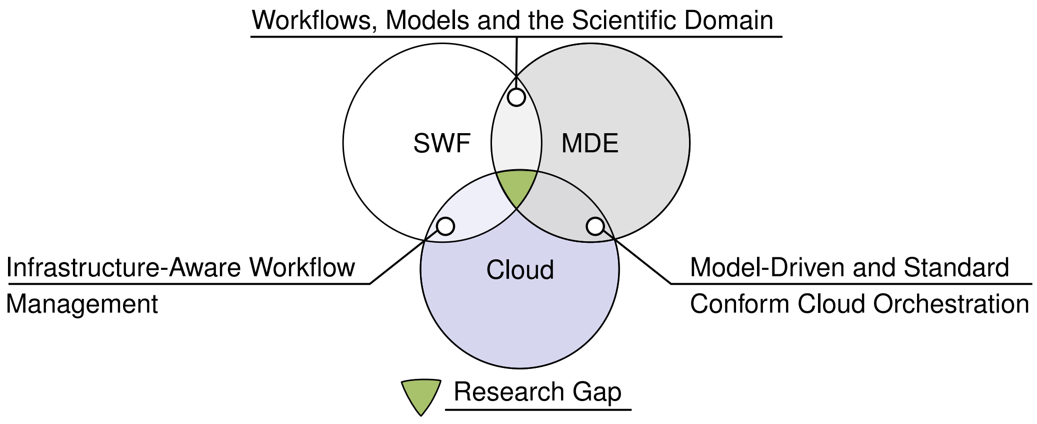

Workflows, Models and the Scientific Domain
SWF
MDE
Cloud
Infrastructure-Aware Workflow Management
Model-Driven and Standard Conform Cloud Orchestration
Research Gap

Figure 3.1: Overview: Related work.

3.1 Model-Driven and Standard Conform Cloud Orchestration

Several languages and techniques have been developed to manage distributed applications in order to focus on user domains rather than complex infrastructure configurations. Some existing solutions are directly offered by cloud providers like the OpenStack Heat Orchestration Template [80] or the CloudFormation Template format by Amazon [79]. Due to the sheer amount of cloud orchestration solutions, we focus on the state-of-the-art of working with cloud resources using model-driven techniques. Some of them are commercial, like Juju [109], which allow modeling and managing applications within hybrid cloud solutions. The research community also provides many different approaches. For example, several extensions for UML have been created to support cloud application specifics [110–112]. The *Cloud Modelling Language* (CloudML) [113] is used to design cloud deployments. The language supports the automatic provisioning of cloud infrastructure and platform resources by matching the designed application requirements to the services offered by the cloud provider. Furthermore, CloudML supports the models at runtime paradigm allowing to manage the deployed applications by changing the model. The CloudMF [114] corresponds to this language and allows modeling and maintaining multi-cloud applications. Another language to model cloud applications is the *Cloud Application Modelling and Execution Language* (CAMEL) [115] which is aligned with TOSCA. CAMEL also introduces a models at runtime approach utilizing the Cloudiator toolkit [116] to deploy and manage the lifecycle of modelled cloud deployments. The tool *An infRastructure modelinG tool for clOud provisioniNg* (ARGON) [117, 118] also provides a DSL for cloud environments and uses model-driven techniques to generate IaC artifacts in order to orchestrate modeled infrastructures. The Saloon framework [119, 120] is used to form a software product line. Hereby, it combines a feature model with a domain model to select suitable environments for multi-cloud deployments.

The related work shows plenty of model-driven cloud orchestration techniques. In the following, we further separate this section into approaches built around the OCCI or TOSCA standard. Section 3.1.1 covers OCCI approaches while Section 3.1.2 discusses TOSCA based approaches.

3.1.1 OCCI Related Approaches

OCCI represents the cloud standard we focus on in this thesis. One of the projects used within our studies is the OCCIWare project [121] which provides an *Integrated Development Environment* (IDE) to support the complete lifecycle of cloud application management using OCCI. The project is formed around the precise EMF metamodel defined by Merle et al. [122] which later got enhanced by Zalila et al. [123]. OCCIWare provides a model-driven toolchain to model and generate arbitrary OCCI extensions. A plethora of different extensions have been designed around the standard adding new capabilities to the project, highlighting the overall extensibility of OCCI. Paraiso et al. [104] extend the infrastructure layer of OCCI to support the management of containers. Within their study, Docker is used, as well as the Docker-Machine tooling to spawn Docker hosts on different cloud providers. The utilization of different cloud providers with OCCI is also shown in the study of Al-Dhuraibi et al. [124] which adds the capabilities to manage the elasticity of cloud applications. Furthermore, as part of the OCCIWare project, an extension for the simulation of cloud applications is introduced in [125]. The implementation is built around CloudSim [126] and allows modeling data centers to simulate resource utilization and pricing strategies for cloud deployments. The actual deployment of such applications can be realized using the MoDMaCAO framework [105] which enhances the OCCI platform extension with more detailed component states and actions triggering configuration management scripts. Furthermore, the *Model-Driven Elasticity Management with OCCI)* (MoDEMO) framework [127] introduces vertical and horizontal scaling of cloud resources to OCCI and can operate among multiple cloud providers. Apart from cloud infrastructures, the standard and the OCCIWare ecosystem have been used to create an extension for the management of mobile robots as a service [128].

Apart from the OCCIWare projects, further research exists that investigates the individual layers of OCCI. Prior to the initial release of the OCCI platform extension specification [106], Yangui and Tata [129] presented CloudServ as an extension to OCCI to incorporate PaaS resources. This extension is reutilized in the concept of the COAPS [130] interface for the management of cloud applications. In their approach, a platform extension is used to describe and manage PaaS resources offered by the provider, while an OCCI application extension describes application resources to be deployed in a PaaS environment [131]. Moreover, two approaches investigating OCCI based monitoring exists. Ciuffoletti [132] extends OCCI with sensor elements that are linked over collectors to the object to monitor. To form a monitoring pipeline, modeled sensors and collectors can be enhanced via different mixins. In the monitoring extension by Mohamed et al. [133] elements to monitor and reconfigure OCCI resources are discussed. In this extension, mixins are introduced to manage polling and subscription monitoring features. Additionally, the extension introduces capabilities to describe how observed monitoring data should be handled.

3.1.2 TOSCA Related Approaches

Similar to OCCI approaches, many approaches exist that built upon or adapt the TOSCA standard. While our approach focuses on cloud orchestration using OCCI, we demonstrate its compatibility with the TOSCA standard. Therefore, we provide an overview of related approaches based on TOSCA and reference a more exhaustive survey by Bellendorf and Mann [134].

Similar to OCCI, multiple ecosystems exist for TOSCA that allow modeling and deploying cloud applications using a standardized cloud modeling language. Winery [135] is a web based modeling tool using the visualization of Vino4TOSCA [136]. The OpenTOSCA ecosystem [137] can be used for a declarative deployment of the modeled TOSCA topology [138] and introduces an invocation mechanism to support management operations of different node types for different cloud providers [139]. The Eclipse Incubation Project *Cloud Application Management Framework* (CAMF) [140] describes a complete IDE built around TOSCA. To support different cloud providers, different adapters were developed and coupled with CAMF to deploy designed TOSCA topologies on different clouds. The Alien4Cloud project [141] provides a type and workflow designer that can be coupled with, e.g., Cloudify [142]. Cloudify offers an open-source orchestration framework for the management of multi-cloud deployments that can be operated over a commercial web interface. Carrasco et al. [143–145] combine TOSCA with CAMP by transforming TOSCA topologies into a CAMP-compliant YAML format allowing for an automated migration of cloud applications with the proposal of a unified interface for IaaS and PaaS. Another approach is TOSCA Light [146], a subset of TOSCA, which can be coupled with production-ready deployment technologies, e.g., Kubernetes [147], using TOSCA Lightning [148]. Several further frameworks exist that built upon the benefits of TOSCA focusing, e.g., on a model-driven approach to deploy big data cloud applications [149] or on deployment techniques to increase the portability of topologies [150].

TOSCA is often used as an inspiration for cloud modeling languages, like the *Essential Deployment Metamodel* (EDMM) [151], and also used as a mapping target for generic cloud languages like the *GEneralized Topology Language* (GENTL) [152]. Bergmayr et al. [153] demonstrate how UML can be coupled with TOSCA using an Ecore metamodel generated from the TOSCA XSD. Further approaches use TOSCA and model-driven techniques to auto-complete undeployable topologies [154] or generate resource-related DevOps tools for seamless integration with TOSCA [155]. Meanwhile, many TOSCA extensions have been developed and evaluated, e.g., to support the deployments for edge devices [156], goal modeling [157] or the management of applications based on serverless computing [158]. Moreover, there is an approach that couples TOSCA with the execution of workflow logic [159]. In the following section, an overview of existing workflow languages and modeling techniques is provided.

3.2 Workflows, Models and the Scientific Domain

In literature, a multitude of different workflow languages can be found that are designed to fulfill the requirements of different domains by tailoring the language accordingly. According to Deelman et al. [9] most SWFMS rely on scripting languages or even provide a graphical user interface to define and manage tasks as well as their dependencies. Common examples that can be found in the literature are, e.g., Swift [160], Tigres [161], Kepler [162], Trident [163], Weaver [164], Triana [165], Pegasus [166,167], Galaxy [168], Taverna [169,170], VisTrails [171], SCIRun [172], Wings [173], YAWL [174] and AVS [175]. In this section, we highlight approaches that rely on model-driven techniques as well as approaches targeting the scientific domain.

Many workflow metamodels already exist that are not exclusive to the scientific domain. A common metamodel for workflow languages is the UML [24] allowing, e.g., to describe the behavior of a workflow or to handle change requests [176]. Also, the BPMN [25] is well known and used to describe business processes for which many extensions exist to support different domains [177]. An example is the modeling of cloud orchestration processes built around TOSCA [178, 179]. Even though these workflows are not commonly used for scientific workflows, their concepts are generic and can also be applied to them. Brüning et al. [180,181] describe a workflow language based on the UML metamodel that can be adapted and evaluated during execution using the tool USE [182] which observes OCL invariants. Also, approaches exist that couple design time and runtime decision-making for the execution of business processes with resources considered as substantial requirements [183].

Specifically for the scientific domain, Cerezo et al. [16, 184] utilize multiple abstraction layers and model transformations to provide the end-user with SWF semantics capturing user goals before generating executable workflow artifacts. Similarly, in the ModFlow framework [185, 186], a domain specific, intermediate and technical layer is used to create a BPMN model which is executed over the *Business Process Execution Language* (BPEL) [187] in a Grid environment. The *Supporting novice data miners in Selecting Suitable mining algorithms* (S3Mining) framework [188] combines MDE and workflow management to support novice users to select appropriate data mining classification algorithms. In their implementation, the Taverna workflow environment is used to execute EMF artifacts via webservices. Nordstrom et al. [59] describe a model-driven approach to detect failing components or compute resources in workflow based environments. In their approach, the *Generic Modeling Environment* (GME) [189] is used to create the WorkflowML modeling language which allows modeling data stores and the synchronization of jobs. The *Workflow-based Architecture to support Scientific Applications* (WASA) project [190, 191] aims at dynamic workflow management for use in the scientific domain. As part of this project, Weske [192] describes a metamodel based on subsequent activities with specific states that can be manipulated over transitions and dynamically extended. In the following, we go into more detail about scientific workflow approaches that are coupled with dynamic infrastructure capabilities.

3.3 Infrastructure Aware Workflow Management

For nearly any SWF language an accompanying SWFMS exist, which utilizes the benefits of available distributed computing utilities developed throughout the years. For example, many approaches can be found that utilize Grid capabilities for workflow execution like DAGMan [193], ICENI [194], GrADS [195], Grid-Flow [196], UNICORE [197], Gridbus workflow [198], GridAssist [199], GridAnt [200], Askalon [201] and Chiron [202]. Meanwhile, many of the existing SWFMS, such as Pegasus [167] and Taverna [170], incorporate cloud resources due to their flexible and scalable capabilities. Often, cloud resources are utilized to scale a preconfigured workflow management system or middleware based on current workload. While scientific workflows pose several opportunities to optimize resource provisioning processes in the cloud [203, 204], we focus on concepts that allow for the management and reflection of the resources. In the following, we go into more detail about infrastructure aware workflow management utilizing the dynamic capabilities offered by cloud computing with a focus on model-driven approaches.

Vukojevic-Haupt et al. [205,206] use the cloud to spawn workflow middleware on demand requiring no pre-configured workflow cluster to be up and running. Kacsuk et al. [207] present the Flowbster system to deploy scaling architectures for the execution of workflows with large scientific data sets using the cloud orchestrator Occopus [8]. Orzechowski et al. [208] describe the deployment of HyperFlow [209], a SWFMS for distributed systems, on an automatically provisioned Kubernetes [147] cluster to manage and scale a container infrastructure for the workflow. Hoppe et al. [210] couple Hyperflow with CAMEL [115] to foster the utilization of cloud and HPC resources within a single workflow.

The ability of cloud computing to dynamically provision infrastructure on demand allows workflows to be coupled directly to the infrastructure. Qasha et al. [159, 211, 212] extend the TOSCA standard to ensure the reproducibility of workflows in the cloud utilizing the benefits of containerization. Weder et al. [213] utilize TOSCA to describe workflows and their required cloud compositions in a self-contained manner which gets deployed via OpenTOSCA [137]. Even though the TOSCA standard is extended, no model representation is considered at runtime that allows the scientist to observe and manipulate the workflow. While most of the related work does not consider the utilization of runtime reflections, some approaches exist heading in this direction. In the approach by Beni et al. [214, 215], a reflective middleware is presented that gathers information from a workflow model at runtime to optimize infrastructure deployments. Here, annotations are used to describe the workflow's resource and deployment requirements. The annotations, however, do not couple workflow tasks with modeled platform elements directly and therefore limit runtime model capabilities for deployed applications. In the following, we discuss our approach in relation to the related work and summarize the research gap addressed within this thesis.

3.4 Summary and Research Gap

The related work shows a plethora of model-driven cloud orchestration approaches and workflow languages. However, there is still a gap in the intersection of scientific workflows, cloud, and model-driven engineering. Especially for workflow approaches that combine runtime models with a standard conform cloud management. We highlight the addressed research gap by discussing the intersections of the SWF, MDE and cloud research areas.

Section 3.1 enumerated several approaches to orchestrate cloud resources using model-driven techniques. One common goal is simplifying the access to cloud infrastructures and applications with a special interest in multi-cloud deployments by leveraging upon different abstraction layers and model transformations [112–114, 119, 120]. Also, the utilization of existing standards can be seen throughout the related work utilizing the UML [110–112, 153], OCCI [104, 105, 121–124, 127–133] and TOSCA [135–146, 148–150, 153–159]. In literature, two cloud modeling languages can be found that focus on a models at runtime approach [113–115]. However, these do not explicitly conform to one of the open cloud standards. In this thesis, we address the gap of a missing orchestration process operating on the OCCI interface and the capabilities of a runtime model conforming to the OCCI data model. Utilizing the OCCIWare ecosystem [121], we enhance the capabilities of OCCI to form a fully causally connected runtime model which reflects gathered monitoring information directly in the runtime model. Furthermore, we show the capabilities of OCCI to combine container and configuration management, allowing for local simulations of cloud deployments. Additionally, we investigate the compatibility of OCCI and TOSCA to utilize the best of both standards while further tackling the provider lock-in and fostering the re-utilization of existing artifacts.

Section 3.2 listed several approaches providing concepts and implementations of SWFMS [160–175]. Independent of the workflow domain, standardized models are used to realize the creation and execution of workflows using, e.g., UML [176, 180, 181] or BPMN [178, 179, 185, 186]. In the SWF domain, MDE is often used to build multiple abstraction layers coupled via model transformations in order to create different views on the system [16, 184–186]. Also, runtime aspects are discussed in related work covering decision-making or metamodels for dynamic workflow management [183, 190–192]. In comparison to the related work, we consider the dynamic management of scientific workflows while relying on the runtime model paradigm. In this regard, we evaluate the capability of OCCI models to be orchestrated over generated UML activity diagrams. Additionally, we investigate the extent to which the standard can support a workflow layer that is coupled with infrastructural resources and managed over a standard conform orchestration process. This opens the opportunity to save resources and costs by tailoring infrastructure toward specific tasks within scientific workflows.

Section 3.3 listed even further workflow approaches with special interest in using distributed computing resources [167, 170, 193–202]. The related work shows approaches combining workflows and their execution with existing orchestration techniques [207, 209] or even cloud modeling languages [159, 210–212]. Furthermore, approaches exist that utilize a reflective middleware for workflow models using annotations to manage deployed resources at runtime [214, 215]. Compared to these approaches, we consider the utilization of a standard conform runtime model which couples workflow tasks directly to cloud resources. Our approach builds upon the OCCI standard allowing us to investigate the extent to which the data model supports creating workflows that operate on highly tailored infrastructure. Hereby, we especially investigate the benefits of reflecting the workflow within a runtime model, as well as the degree to which the uniform OCCI interface can support the execution of the workflow. The utilization of a standard that is backed up by model-driven engineering techniques opens the opportunity to foster the replication of scientific workflows and the reuse of individual deployments and artifacts for individual workflow tasks.

4 Standard Conform Cloud Runtime Model Orchestration

In this chapter, we present an approach to orchestrate cloud deployments using the standardized data model and uniform interface provided by OCCI. Due to the flexibility offered by OCCI, we follow a Models at Runtime approach and extend the standard allowing to automatically orchestrate a cloud environment. Figure 4.1 separates this approach into the **Orchestration** ① of the cloud resources and their **Causal Connection** ②. We provide a brief overview of both parts with details being explained in the remainder of this chapter.

The **Orchestration** ① covers procedures to adapt the **Runtime Model**, i.e., the abstraction of the **Cloud** deployment. Hereby, it aims at recreating the state described in the **Design Time Model** within the **Runtime Model**. The **Design Time Model** is either created by a human user or as a result of another programmatic system which is used to monitor, plan and adapt a running cloud deployment. Independent of the model's origin, several **M2M** transformations get applied on it before it serves as an **input** for the **Adaptation Engine**. The **Adaptation Engine** is used to **extract** and compare the **Runtime Model** to the **Design Time Model** and derive required **OCCI** requests to adjust it accordingly.

The **Causal Connection** ② describes how we process requested **OCCI** elements to **synchronize** the **Runtime Model** with the **Cloud**. Hereby, existing extensions for OCCI can be reutilized, e.g., extensions introducing container [104] or configuration management [105]. In this thesis, we combine both approaches to couple container and configuration management using a runtime model. Additionally, we extend the OCCI data model with monitoring instruments to reflect operational parameters of a specific node.

Section 4.1 describes the **Orchestration** ① process in more detail, while Section 4.2 covers the **Causal Connection** ② between the cloud and the runtime model.

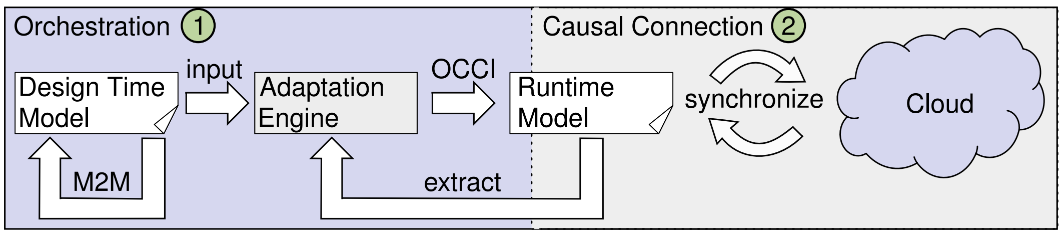


Figure 4.1: Overview: Standard conform cloud orchestration.

4.1 Model-Driven Cloud Orchestration Process

Adjusting a running cloud deployment to ever shifting environments requires a procedure that adapts the deployed cloud resources accordingly. In this section, we present an orchestration procedure that complements the OCCI data model and exploits its uniform interface to manage any kind of resource. In our preliminary work, we only covered the management of IaaS resources [216]. In this thesis, we enhance the orchestration procedure to incorporate model transformations to increase the portability of models as well as to simplify the modeling process. Furthermore, we introduce a pre-processing model transformation that adds platform specific information to manage platform related resources. To actually transfer an OCCI design time to the runtime model, an adaptation engine is used. This engine follows the concept of a MAPE-K control loop and compares the running and desired state to derive and execute required adaptive steps. Finally, to investigate the generalizability of OCCI, and thus our approach, we define a model transformation to orchestrate TOSCA topologies via the OCCI interface.

Section 4.1.1, goes into detail about the introduced pre-processing and platform specific model transformations. Section 4.1.2 defines the mapping of the TOSCA standard to the OCCI standard which represents an optional design time transformation. Finally, Section 4.1.3 describes the adaptation engine including individual steps required to adjust the runtime model to adapt it to the design time model state.

4.1.1 Design Time Abstraction Layers

The automated deployment and runtime management of a cloud topology requires different kinds of information ranging from technical to platform specific details. To address portability issues and reduce the complexity of the model at design time, we resort to the model abstraction layers defined by the MDA. Figure 4.2 depicts the **CIM**, **PIM** and **PSM** layer and how our defined **M2M** transformations apply on a small example. While individual transformations exist that transform the model from one abstraction layer to another, a validation step can be integrated at each state. This step allows checking whether defined model constraints hold, giving the user feedback on possible warnings and errors.

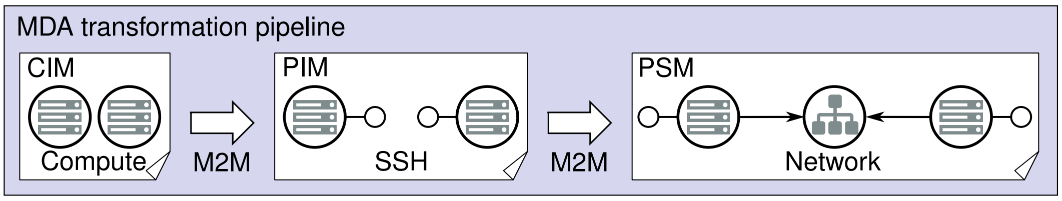


Figure 4.2: Design time model abstraction layers and transformation process.

The **CIM** presents an abstraction layer in which technical details are removed from the model. In our domain this is only possible to a certain degree, as the model is supposed to describe the composition of cloud infrastructure and deployed components. Thus, most technical information is already present in the model. We use an M2M transformation to add certain technical information in case it is missing, e.g., in form of default values.

We reach the **PIM** by adding further technical information that is cloud provider independent. This includes, e.g., the addition of preconfigured **SSH** keys or user data that is attached to compute nodes, making them accessible later on in the cloud. Additionally, the transformation checks whether certain constraints are violated, e.g., whether duplicate identifiers exist. Depending on the violated constraints either a warning is produced by the model transformation or it is automatically repaired. Overall, the transformation reduces the complexity to create the initial model, as well as the time required for it.

The **PSM** is required to automatically deploy the modeled cloud topology in the environment of a chosen cloud provider. For this, we perform an M2M transformation that adds pre-defined provider specific information, as well as elements required by utilized frameworks. Hereby, the added information largely depends on the utilized OCCI extensions and how introduced elements handle cloud provider specific behavior. For example, OCCI Resource and OS templates offered by the cloud provider can be attached to the modeled compute nodes which describe the VM size and flavor or the image to be used for a VM. While the automatic addition of provider specific information is convenient, we additionally use the M2M transformation to add elements required by the MoDMaCAO framework which is later used to configure modeled compute nodes. To fulfill these requirements, a management **Network** is added to the OCCI model. Additionally, an OCCI NetworkInterface link is added to each compute node that targets the management **Network**. This link is marked with a specific mixin indicating its use for configuration management. The transformation ensures that at least one network is present for the creation of a VM, which is required by some cloud environments, such as OpenStack. Furthermore, a mixin is added to each infrastructural resource to map the id of the OCCI resource to the id assigned by the cloud provider.

At this stage, the model is ready to be deployed and can be validated in order to check whether all constraints are fulfilled. In order to increase the portability of designed models, we additionally defined an M2M transformation that can operate on deployed runtime models to decouple them from provider specific information, so that they can be used in another environment as well. In addition to the presented MDA abstraction layer transformations, further M2M transformations can be performed at design time. For example, in the scope of this thesis, we transform a TOSCA topology into an OCCI model in order to evaluate the compatibility between both standards.

In the following, we describe a mapping between the TOSCA standard and OCCI which we use to demonstrate the generalizability of OCCI and our approach.

4.1.2 Mapping of TOSCA to OCCI

Both the TOSCA and OCCI standard specify an extensible language for cloud deployments using a type-instance pattern. In the scope of a cooperation, we designed a mapping between the standards that can be used within our orchestration process as a preliminary transformation. In this thesis, we focus on the capabilities of OCCI and use this mapping to investigate the generalizability of the OCCI interface and thus our orchestration procedure. Therefore, we briefly describe the concept of the transformation in the scope of our orchestration procedure. A concise description of the mapping and transformation can be found in the respective publication [217]. As shown in Figure 4.3, we separate the description of the transformation into a mapping of the type and instance-layer.

In the **Type Layer** we map the elements defined in the **TOSCA Types** definition to the OCCI classification and identification mechanism. This mapping is used to generate two OCCI extensions. One that represents the TOSCA normative types (**OCCI-TOSCA Normative**), and one that contains the elements of TOSCA custom types (**OCCI-TOSCA Custom**). We **register** these generated extensions within an **OCCI Interface** to allow for management requests of these specific types. In the **Instance Layer** (see Figure 4.3), we transform a **TOSCA Topology** to an **OCCI Model** that can serve as input for our **OCCI Orchestration Process**. In the following, we describe the mapping of the individual elements in both layers.

4.1.2.1 Type Layer

Table 4.1 maps **TOSCA types** to the **OCCI classification & identification** mechanism. Similar to kinds and mixins, node types and relationship types define descriptive building blocks for cloud resources. Typically, the normative types represent an abstract base from which custom types inherit. To transform normative as well as custom types we accordingly implement two subsequent transformations for this meta layer. Within these transformations, we map TOSCA types to an OCCI mixin which can be applied to a specific kind. In order to apply the generated OCCI mixins to actual OCCI resources, each mixin representing a TOSCA

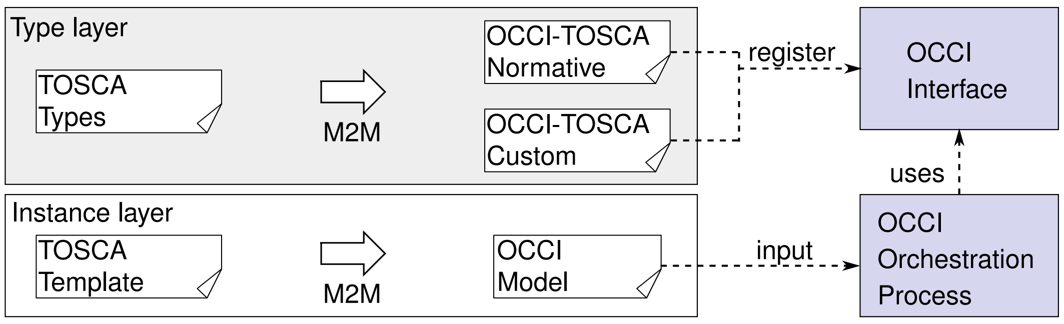


Figure 4.3: Model transformations to map TOSCA and OCCI.

TOSCA types	**OCCI classification & identification**
Node type	Mixin applied to Resource Kind
Relationship type	Mixin applied to Link Kind
Property & Attribute	Attribute
Requirement type	OCL Constraint
Capability type	Mixin
Interface type	Mixin (only actions)

Table 4.1: TOSCA to OCCI type layer mapping.

normative type can be applied to an OCCI kind from one of the standardized extensions, e.g., compute or component. For each type specified, properties and attributes are transformed into OCCI attributes. The TOSCA capability type extends another type with certain capabilities, e.g., to express the utilization of a specific service endpoint. This resembles the concept of an OCCI mixin allowing to extend the capabilities of a kind. Combined, all properties and attributes specified in the node type can be represented in OCCI. TOSCA requirement types can be implemented indirectly by transforming them to OCL within the OCCI model. The actual body of the constraint, however, has to be filled manually. An interface type defines operations that can be performed on the node, which complies to OCCI mixins specifying only actions. While most TOSCA types can be transformed into OCCI, very abstract and design time specific information that do not reflect actual cloud resources can not be mapped. This covers, e.g., scaling and artifact groups offered by TOSCA which would require additional OCCI extensions to be conceptually designed.

To visualize the concept of the transformation, Figure 4.4 provides a small example for the transformation of the normative **TOSCA Types** to an OCCI extension using a UML object diagram. This diagram visualizes an example normative node type with one of its capabilities. Hereby, we shortened the extensive TOSCA type names for brevity. The shown **database:NodeType** abstracts the utilization of database components and is typically extended for specific database frameworks. This node type possesses multiple attributes from which one is used to describe the administrative **user**. In addition, the node type specifies an

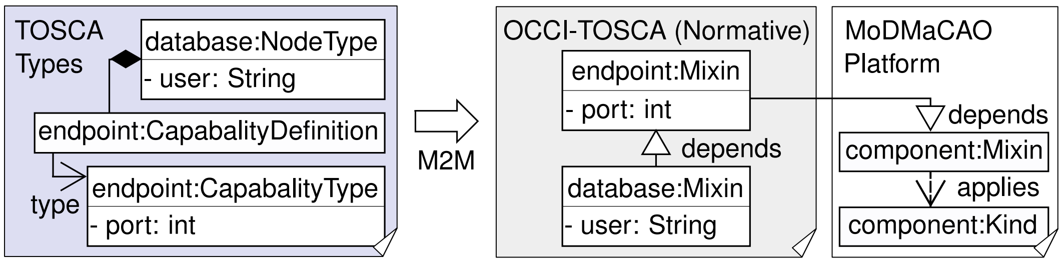


Figure 4.4: Example TOSCA to OCCI type layer transformation.

TOSCA templates	**OCCI core base types**
Node template	Resource with a Mixin instance
Relationship template	Link with a Mixin instance

Table 4.2: TOSCA to OCCI instance layer mapping.

endpoint:CapabilityDefinition which is typically used for any kind of distributed service. The properties provided by this definition are modeled in the connected **endpoint:CapabilityType** covering, e.g., a property for the **port** that can be specified for later use by a database node template. In the **OCCI-TOSCA (Normative)** extension, this node type is transformed to the **database:Mixin** which has the **user** property stored as an OCCI attribute. This mixin **depends** on the capability transformed into the **endpoint:Mixin**. This dependency relation resembles the notion of a generalization and therefore provides the **database:Mixin** with access to the **port** attribute. Moreover, due to the dependency to the MoDMaCAO **component:Mixin**, the **database:Mixin** can also apply to the **component:Kind** from the OCCI Platform extension.

4.1.2.2 Instance Layer

In Table 4.2, we map **TOSCA templates** to the **OCCI core base types** covering the instance layer. In general, templates specify the occurrence of a node or relationship in a topology template instantiating the properties given by its accompanied type. Therefore, node templates fit the notion of an OCCI resource and relationship templates the concept of an OCCI link. In addition, all transformed OCCI resources and links possess a specific mixin that matches one of the previously generated normative or custom types.

An example transformation of the instance layer is shown in Figure 4.5. Within this example, the previously generated TOSCA types from the example in Figure 4.4 are reused within the depicted templates. The **TOSCA Template** consists of a single **db:NodeTemplate** that is of the **database** type. Using this type, the template can define the administrative user account to be used by the database, which in this case is set to **admin**. Additionally, the template aggregates a **:Capability** instance of the **endpoint** type in which the **port** to be used is configured to **3306**. In the **OCCI Model** this template is transformed into the **db:Component** with a **database:Mixin**. This provides access to the modeled information during the lifecycle management of the component allowing us to configure the database accordingly.

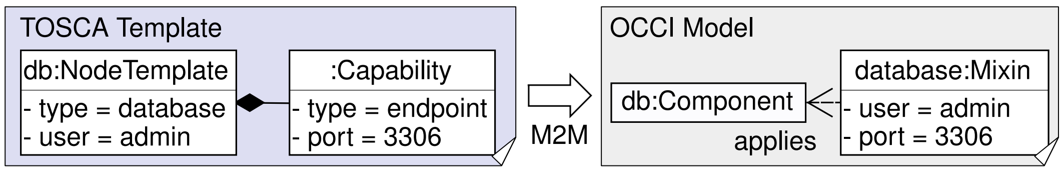


Figure 4.5: Example TOSCA to OCCI instance layer transformation.

4.1.3 Model-Driven Adaptation using OCCI

To adapt a running cloud deployment, we treat the **Design Time Model** as a high level goal to be reached, as shown in Figure 4.6. This model is an **instance of** the **OCCI Metamodel** and is either manually created by a **User** or generated by a **System**. The **Adaptation Engine** takes this model as **input** and follows the concept of a MAPE-K loop to adapt the **Runtime Model** and its causally connected system. To determine the required requests, the engine possesses a **Monitor** step that extracts a **Runtime (Snapshot)**. This snapshot contains the **Knowledge** about the system's current state which is required to perform adaptive steps. Based on this knowledge, we **Analyze** required steps by comparing the entities of both models against each other having the same id and kind. As a result, we identify whether an entity needs to be deleted, updated or removed allowing to **Plan** a sequence of requests required to reach the desired state. Finally, we **Execute** this adaptation plan resulting in adaptive requests being sent to the **OCCI Interface**. In the following, the MAPE-K loop is described in more detail.

4.1.3.1 Monitoring and Analysis Step

The monitoring step covers the extraction of the information contained within the runtime model. This procedure can be done over several means and highly depends on the capabilities provided by the OCCI interface. For example, an interface may be offered that grants direct access to the maintained runtime model or a message broker that notifies us when the runtime model has changed. Moreover, information about the running entities can be retrieved directly from the OCCI interface and be assembled into a model. To know the requests that need to be performed to gather the complete runtime model, the registered OCCI extensions can be queried. The response of this query provides information about all registered OCCI kinds and thus paths that need to be requested to gain insights about all provisioned resources. To form a knowledge base around the running cloud deployment, we store the extracted information as a runtime model snapshot.

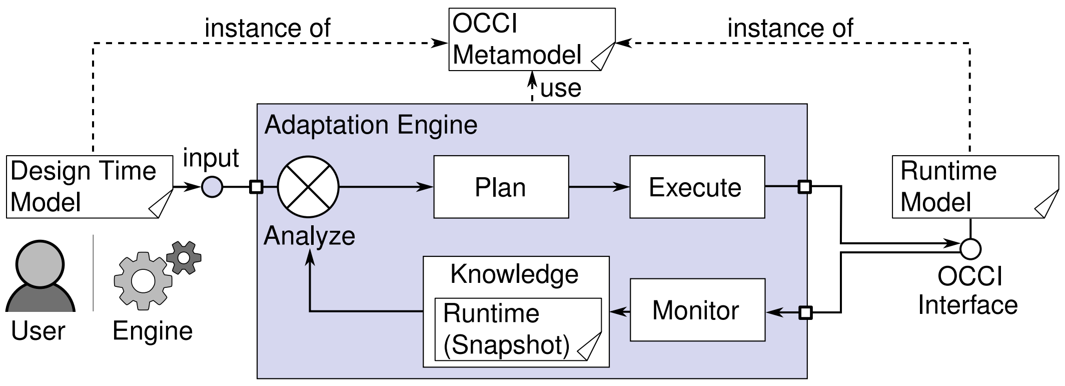


Figure 4.6: Adaptation process components.

After the extraction of the runtime model, all the information required to perform adaptive steps can be derived. As a first step, we match each resource from the design time and the runtime model. For this match we make use of the entity kind and id as they represent immutable attributes. In addition, provider specific identifiers, assigned to individual compute nodes in form of mixins, are incorporated in this matching process. Based on the comparison we determine whether a resource and its link is currently present or absent in the cloud, indicating the need for its creation or deletion. Figure 4.7 highlights the **Analyze** phase and provides a short example. In this phase, three different sets are built containing the elements from the **Design Time** and **Runtime** model. In the given example, both models contain three elements comprising one network, compute and component node. Elements only present in the **Runtime** model are mapped to a **Delete** action, as they are not required by the state indicated by the **Design Time** model. In this case, the entities to be deleted comprise **Component$_1$** and **Compute$_1$** as well as its related links. The second set is concerned with entities requiring an update of defined attributes. This set is build from the intersection of the **Runtime** and **Design Time** models, i.e., the **Network** node, which serves as input for the **Update** step. The last set comprises entities only present in the **Design Time** model which indicates that they are currently missing in the cloud. The resulting set is used as input for the **Create** step. In our example, this includes the **Compute$_2$** node, the **Component$_2$** node, and their links.

4.1.3.2 Planning and Execution

The **Plan and Execute** phase, shown in Figure 4.7, is responsible to apply the actions on the identified sets obtained during the analysis. As a first step within the plan, we **Delete** all entities not required anymore. This reduces the amount of running resources during the process to a minimum while avoiding quota conflicts. Thereafter, we **Update** entities adjusting the attributes according to the desired values. Finally, we determine the order in which requests have to be sent to **Create** the entities currently missing in the runtime model. The requests required to remove, update and add the elements to the runtime model

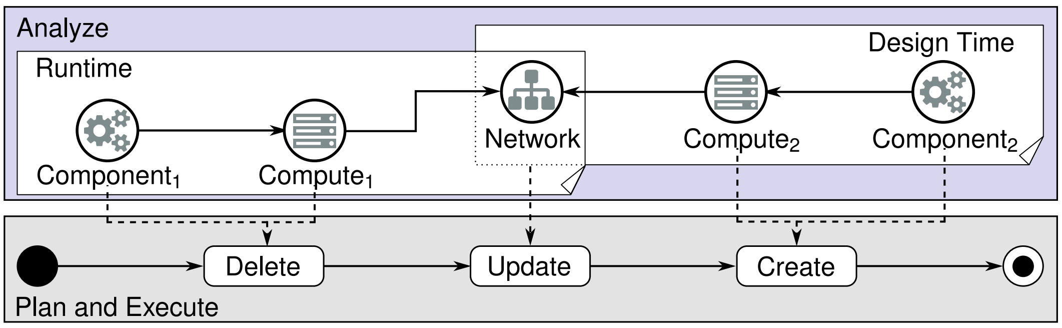


Figure 4.7: Comparison process and mapping to adaptive steps.

are automatically generated from the OCCI elements by deriving a REST request from the elements kind, mixins, and attributes that conforms to OCCI's uniform interface.

During the **Delete** step, we separate the elements to be deleted into links and resources. Based on this separation we iterate over the resources to be deleted while executing a deprovisioning procedure matching to the specific kind. In the general case, we decouple each resource before the resource itself gets deprovisioned. For this, we start by deleting all links contained within the resource from the runtime model. One exception to this rule is the deletion of components. Here, it needs to be checked whether the underlying host also gets deleted. If not and the compute node remains in the model, the component needs to be undeployed before it gets removed from the model. For this procedure, the PlacementLink connecting the component to its host component is still required. This allows removing the artifacts of the component from the host. If the host is marked as to be deleted as well, no undeploy action is triggered as the host is completely removed. Therefore, the time required to undo certain configurations and delete the artifacts of the managed component is saved.

In the **Update** phase, adaptations to single entities within the model are taken over including changes to attributes and mixins. However, not all attributes are subject to a change covering, e.g., attributes which reflect runtime information such as state messages. Therefore, we first filter for attributes of interest to be adjusted in the runtime model, such as the amount of cores of a compute node. The remaining attributes are changed in the runtime model by performing the according request. Similar to the delete step, further steps are intended to trigger update behavior based on the managed kind. While this step triggers the change of attributes in the model, the actual change in the cloud is performed by the kind's effector.

Finally, we **Create** the entities introduced in the design time model and add them to the runtime model. To resolve dependencies between the individual elements, we perform two subsequent **M2M** transformations as shown in Figure 4.8. The first **M2M** transformations creates a *Provisioning Order Graph* (POG) from the **Design Time** model, similarly to the approach presented in [138]. Subsequently, we transform the **POG** into an **Activity Diagram**. This diagram describes a sequence of OCCI requests to be executed in order to create the remaining OCCI entities in the runtime model. The **POG** is an **instance of** a **DAG Metamodel** and consists of a node for each link and resource within the **Design Time** OCCI model. Hereby, an edge in the **POG** represents a request **dependency** between two nodes. Due to the requirement of an OCCI link to have both its source and target active in the runtime model, the dependencies within the **POG** result in a pattern that determines which resources need to be created first, before being connected by links. In case of the example, the **POG** consists of five nodes that each represent one OCCI entity as well as their dependencies. While the generated **POG** contains only information from the **Design Time** model, the additional **Runtime** information needs to be incorporated. This phase of the orchestration process only considers the creation of new entities. Therefore, all entities already present in the **Runtime** model are removed from the **POG**, e.g., the **Network** node. Now that the **POG** only consists of elements that are missing within the **Runtime** model, we transform it into an **Activity Diagram** which is an **instance of** the **UML Metamodel**. This diagram represents the provisioning plan to

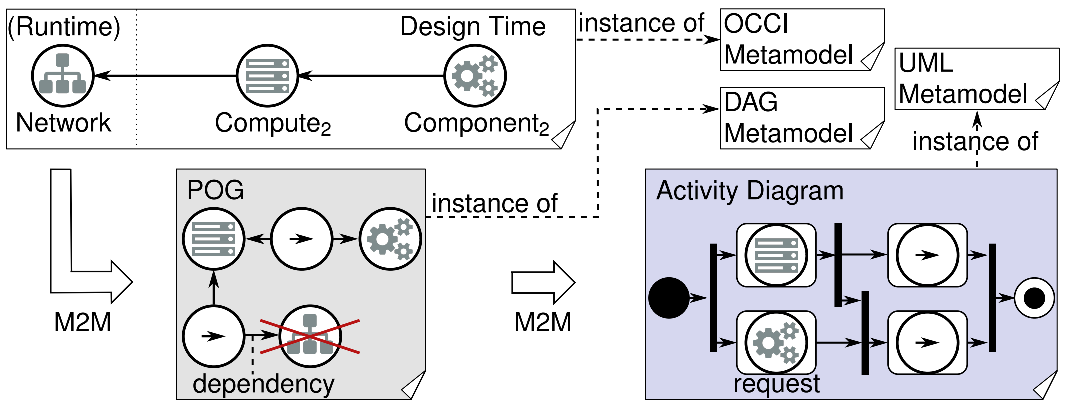


Figure 4.8: Model transformation chain generating OCCI request sequence.

be executed. To generate this plan, the transformation adds an action to the **Activity Diagram** for each **POG** node. Hereby, each action represents an OCCI **request** to provision the entity in the cloud, e.g., **Component$_2$**. Thus, each action within the activity diagram represents one OCCI element that needs to be added to the runtime model. After transforming each node to an action, the dependencies within the **POG** are incorporated to the **Activity Diagram** in form of control flows. In case of the example, this results in both the **Compute$_2$** and the **Component$_2$** node to be requested in parallel as they are independent of each other. The links on the other hand require both the target and the source resource to be created within the runtime model. Thus, for the connection to the **Network** only the **Compute$_2$** node needs to be created, while the connection of the **Component$_2$** to the **Compute$_2$** node requires both of them to be created first.

After the provisioning process has been finished, we start the deployment procedure of newly added component and application nodes. Hereby, we initially check whether the runtime model now equals the design time model regarding the entity structure. If that is the case, all infrastructure resources are up and running. For the deployment process, we utilize the actions provided by the component and application resources. In case of the extended platform extension offered by MoDMaCAO this comprises the deploy, configure and start action linked to individual configuration management scripts. Additionally, this step covers the start-up of container resources as these, similar to components, possess the requirement of an up and running host. The provisioning, deployment, and update logic itself is contained within the individual effectors of the entities, i.e., within the components maintaining the causal connection between the runtime model and the cloud.

4.2 Causal Connection to Cloud Environments

While the previous section described how an orchestration of modeled entities can be achieved, this section describes actions taken after the OCCI requests have been received. We reuse the concept of the OCCIWare runtime server [123] which manages the runtime model over a set of effectors. As shown in Figure 4.9, this server can be interpreted as a **Middleware** that registers a set of **Effector** implementations. Hereby, one **Effector** exists for each registered OCCI kind which accepts respective **OCCI Request**s to manage the **Runtime Model** and the cloud. In general, the set of effectors can be divided into **Infrastructure** and **Platform** effectors.

Infrastructure effectors translate incoming requests to the proprietary interface of the **Cloud**. Several cloud provider specific effectors for OCCI already exist. To perform our studies, e.g., we reused the OpenStack effector [105] which forwards OCCI requests to our private OpenStack cloud. Throughout our studies, we extend this effector to reflect changes made to the cloud deployment over the cloud provider interface in order to build a concise OCCI runtime model.

In this thesis, we focus on **Platform** effectors that communicate with the interface of individual **Compute** nodes using, e.g., **SSH**. To foster more complex deployments, we combine container and configuration management. Additionally, we exploit the functionality of the **OCCI** interface to deploy sensors which **monitor** events in the cloud and update the **Runtime Model** accordingly.

Section 4.2.1 highlights the combination of container and configuration management. Section 4.2.2 describes the concept of a fully causally connected OCCI runtime model which supports the management of sensors and the reflection of operational parameters.

4.2.1 Combining Container and Configuration Management

Currently, two OCCI extensions exist that enable the management of platform elements on top of infrastructure resources. The MoDMaCAO platform extension [105] and the container extension presented by Paraiso et al. [104] (see Section 2.3.3.2). In the scope of this thesis, we contribute to the MoDMaCAO framework by introducing **Container** ① management and by extending **Variable File** ② generation. The interaction between the elements of both extension is exemplified in Figure 4.10.

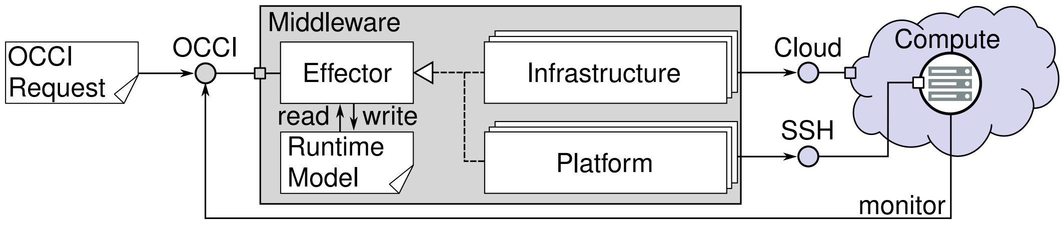


Figure 4.9: Establishing a causal connection via effectors.

In course of a supervised Bachelor's thesis [218], we combine **Container** ① and configuration management by enhancing the MoDMaCAO framework. To apply configuration management on container nodes, they need to be reachable by the platform effector. Therefore, we configure them to be accessible like a VM, e.g., by having an SSH client installed. When a VM is used to host multiple containers, each container needs to be connected to the network of the host and the SSH connection has to be established over several ports. For the configuration of the container, we inject a script that is triggered on the container's startup. When activated this script configures the container using the SSH key and port information which we extract from the modeled container node. Alternatively, existing container management systems can be utilized, as well as plugins to directly connect or configure the container. The information required is present in the runtime model, e.g., the name of the container and the address of the host network interface.

Once a lifecycle action of a component is triggered, the MoDMaCAO framework generates a **Variable File** ②. This file provides the accompanied configuration management script with access to the component's attributes. To manage more complex deployments, we extend the generation of this variable file by additionally reflecting the information of surrounding entities. For example, in case a lifecycle action of the **:Component** is triggered, the **Variable File** provides the **Management Script** access to the **port** attribute specified within the **master:Mixin**, as well as information stored within connected components. To enable the utilization of runtime model information for components that are pre-installed within container images a **mock-up:Component** can be modeled. Such components can be placed on top of container nodes and follow the purpose of interconnecting component instances while providing access to attributes. Depending on the model's level of detail, this empty component can be filled with logic to incorporate runtime model capabilities allowing to further configure or manage the component in the container. In case of the example, the **mock-up:Component** may provide management actions and access to information about the service that is pre-installed within the **image** of the **Container**.

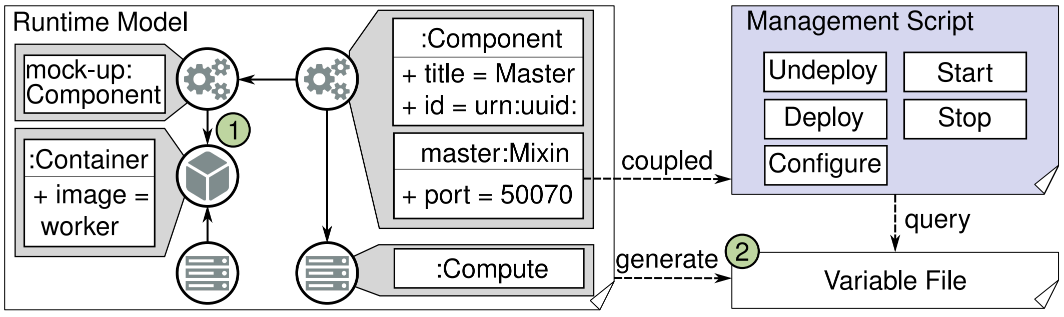


Figure 4.10: Combining container and configuration management.

4.2.2 Sensor Management and Reflection

Self-adaptive control loops require a rich knowledge base to support their decision-making, e.g., to choose which control flow to follow in a workflow. Therefore, we designed a monitoring extension for OCCI to gather and reflect monitoring results directly in the runtime model. To reduce the workload for the monitored system, our extension is designed to distribute sensors among different compute resources, while activating their monitoring capabilities on demand. To enable a dynamic management and deployment of sensor capabilities, we built the **Monitoring** extension based on the **MoDMaCAO Platform** extension [105], as shown in Figure 4.11.

Each **Sensor** is composed of several monitoring instruments. Thus, it inherits from the **Application** type which is typically used to aggregate **Component** instances. In this case, the **Component** instances are specialized to represent monitoring instruments including a **DataGatherer**, **DataProcessor** and **ResultProvider**. Each monitoring instrument inherits from the **Component** type and thus can be connected via **ComponentLink** instances to a **Sensor**. One **DataGatherer** is required in each **Sensor**, as it collects data about the object to monitor. Additionally, multiple sensors may share the same data gathering or processing device or service. The gathered data is processed by the **DataProcessor** which is responsible to derive meaningful information to be monitored. This instrument is optional, as not all data to be reflected requires post-processing. The **ResultProvider** is specific to a particular **Sensor** as it transports the monitored information to the consumer. To reflect the results directly in the runtime model, the **OCCIResultProvider:Mixin** needs to be applied. Thus, at least one **ResultProvider** should be contained in each **Sensor** to make the gathered and processed data available. This mixin provides an attribute to configure the endpoint of the OCCI interface. Subsequently, this mixin allows sending requests to the OCCI interface with updated monitoring information. A **Sensor** can contain multiple **ResultProvider** instruments. For example, one could provide the monitored information directly to a user via E-Mail notifications, while the other reflects the information within the runtime model. It should

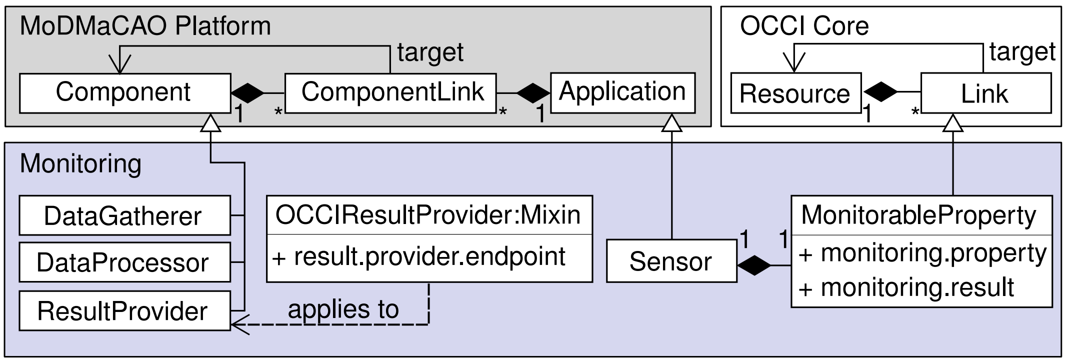


Figure 4.11: Enhanced monitoring extension for OCCI.

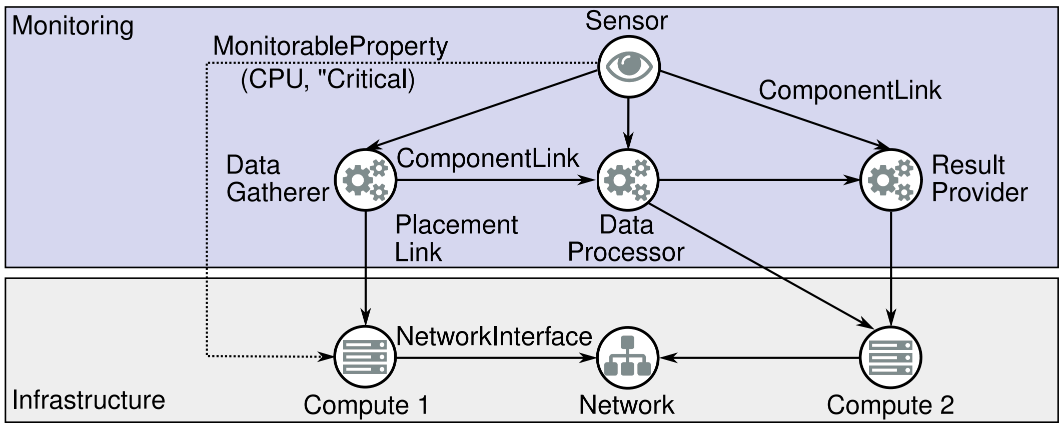


Figure 4.12: Example cloud configuration with dynamic monitoring capabilities.

be noted, that only one of them is allowed to have an **OCCIResultProvider** mixin which ensures the runtime model reflection. For this, the **MonitorableProperty** link is used. This link connects the **Sensor** to the monitored **Resource** and possesses two attributes. The **monitoring.property** can be specified by the user to label the monitored information. The **monitoring.result**, on the other hand, is filled by the **ResultProvider** to reflect the information within the runtime model.

An example instance of a cloud configuration using the aforementioned monitoring types is displayed in Figure 4.12. Here, the cloud deployment consists of two VMs described via compute type instances that are connected to a **Network** instance using **NetworkInterface** links. **Compute 1** represents the VM to be monitored hosting a **DataGatherer**. This instrument is configured to gather metrics about the CPU and memory utilization of its host. **Compute 2** hosts the **DataProcessor** and **ResultProvider**. The lifecycle of modeled monitoring devices is described over the component FSM (see Figure 2.9) and managed over dedicated configuration management scripts deploying them on the host targeted by the corresponding **PlacementLink** instance. To configure and couple the monitoring devices with each other **ComponentLink** instances can be used. This ensures their configuration and startup procedure, i.e., that the **DataGatherer** is deployed and started prior to the **DataProcessor** and **ResultProvider**. Finally, each monitoring instrument is connected over a **Sensor** which is modeled by **ComponentLink** relations. Furthermore, the link from the **Sensor** to the first compute node represents the **MonitorableProperty** holding the property **CPU**, which is currently set to **"Critical"**.

5 Runtime Workflow Model Concept

In this chapter, we present our models at runtime approach to support the execution of workflows on top of a custom infrastructure. By coupling dynamic infrastructures with workflow capabilities, we allow tailoring individual tasks of the workflow to their specific needs, e.g., by dynamically deploying computation clusters. As infrastructure and cloud knowledge is required, we assume two types of users. The scientist providing knowledge about their domain, as well as a cloud architect who can model the tasks' underlying infrastructure and applications. While the management of the infrastructure requires a separate expert, the deployment and applications can be designed and modeled in such a manner that they can be reused in multiple workflows making them more accessible to the scientist. Similar to other concepts that automate the execution of scientific workflows, the approach can be separated into the workflow modeling language and the engine interpreting it. As shown in Figure 5.1, we focus on models at runtime characteristics and divide the presentation of our approach into the **Workflow Orchestration** ① and the **Causal Workflow Connection** ②.

The **Workflow Orchestration** ① process is built around a **Workflow Engine** which takes the **Design Time Workflow Model** as input. While the **Design Time Workflow Model** represents the high level goal to be reached, i.e., the execution of the workflow, the **Runtime Workflow Model** provides valuable information about the cloud and workflow runtime state and serves as a prominent knowledge base for the execution of a workflow. By combining both runtime and design time information, the **Workflow Engine** derives a cloud runtime state at each point in time by forming a new required runtime model. The generated cloud configuration is then enacted by the **Adaptation Engine** (Chapter 4) which sends the appropriate **OCCI** requests to adjust the **Runtime Workflow Model**.

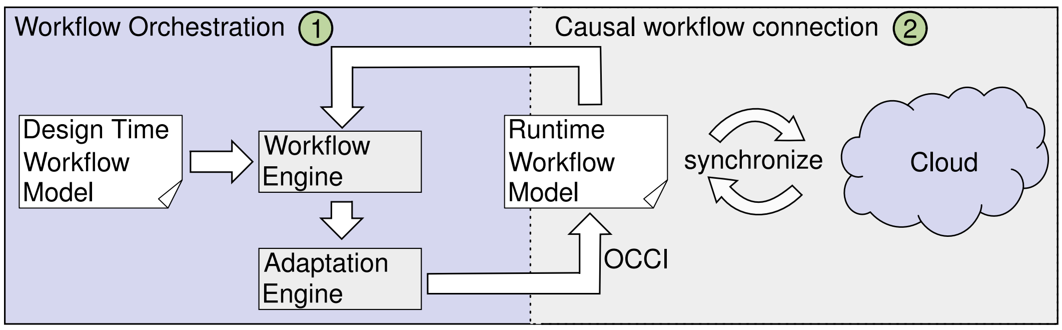


Figure 5.1: Overview: Runtime workflow model concept.

The **Causal Workflow Connection** ② forwards these **Runtime Workflow Model** adjustments to the **Cloud**. To couple workflow elements and resources from the cloud domain, we designed a concept in which workflow tasks can be attached to modeled cloud infrastructures and applications. Using this connection, we allow deploying arbitrary infrastructure that may shift throughout the execution of a workflow. Especially the abstract representation of the runtime state allows scientists to observe and interact with the workflow at runtime. We realize the concept by specifying an OCCI extension that adds workflow capabilities to the standard. This in turn provides access to already existing tooling that can dynamically manage cloud resources and provide access to runtime model capabilities.

Figure 5.2 exemplifies the goal to be reached by our extension and workflow engine. The figure shows a **Design Time Workflow** with three subsequent tasks that get decomposed into individual **Runtime States**. In this figure, we only show the **Infrastructure** requirements of each task to provide an easier overview of the concept. In this example, task **A** requires a single VM and storage which is used to gather data. Task **B** needs two VMs connected over a network that together form a computation cluster to analyze the gathered data. Finally, for task **C** a single VM is sufficient which deploys a web service that provides access to the results of the analysis. In case of this rather simple workflow, the engine derives the **Runtime States** in such a manner, that first the infrastructure of task **A** is transferred to the runtime model and thus the cloud. After task **A** has been executed the workflow requirements for task **B** are fulfilled. This leads to the deployment of its infrastructure and the execution of the task itself. Finally, after task **B** has been successfully executed, the deployment of the infrastructure for task **C** is performed. Once the deployment process is finished, the task is executed which finishes the workflow.

In Section 5.1, we describe the different capabilities provided by the OCCI workflow extension, followed by Section 5.2 which presents the engine orchestrating the workflow.

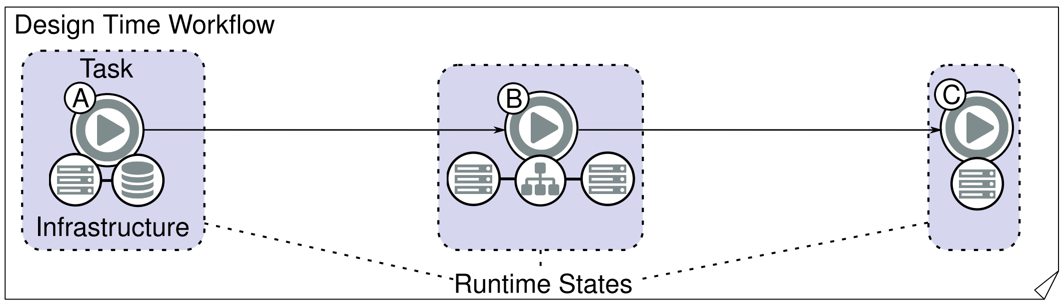


Figure 5.2: Design time workflow example with highlighted runtime states.

5.1 Runtime Workflow Metamodel

In this section we present the concept of the runtime workflow metamodel and how it can be applied as an extension to the OCCI standard. Hereby, we reflect the state of the workflow and the infrastructure within the runtime model. To conform to already existing approaches, the concept resembles the one of a directed graph and consists of resources and links in terms of the OCCI standard. A subset of utilized entity kinds, i.e., a subset of the metamodel, as well as an example model instance is shown in Figure 5.3. Each element within the metamodel is a specialization of either a resource or link in terms of OCCI. We omitted the inheritance relationships for clarity and replaced them with white boxes indicating a resource inheritance and gray boxes representing a link inheritance. We build our extension on top of existing OCCI extensions which can be separated into three layers: the **Workflow** layer, the **Platform** layer, and the **Infrastructure** layer.

The **Workflow** layer allows modeling a sequence of computations that should take place. Hereby, the computation to be performed is represented by the **Task** element, while a **TaskDependency** can be used to sequence two tasks. This link can be further specialized into a **ControlflowLink** or a **DataLink** to support control and data flows. In addition to the base task type, we introduce **Decision** and **Loop** nodes representing specialized versions of a **Task**.

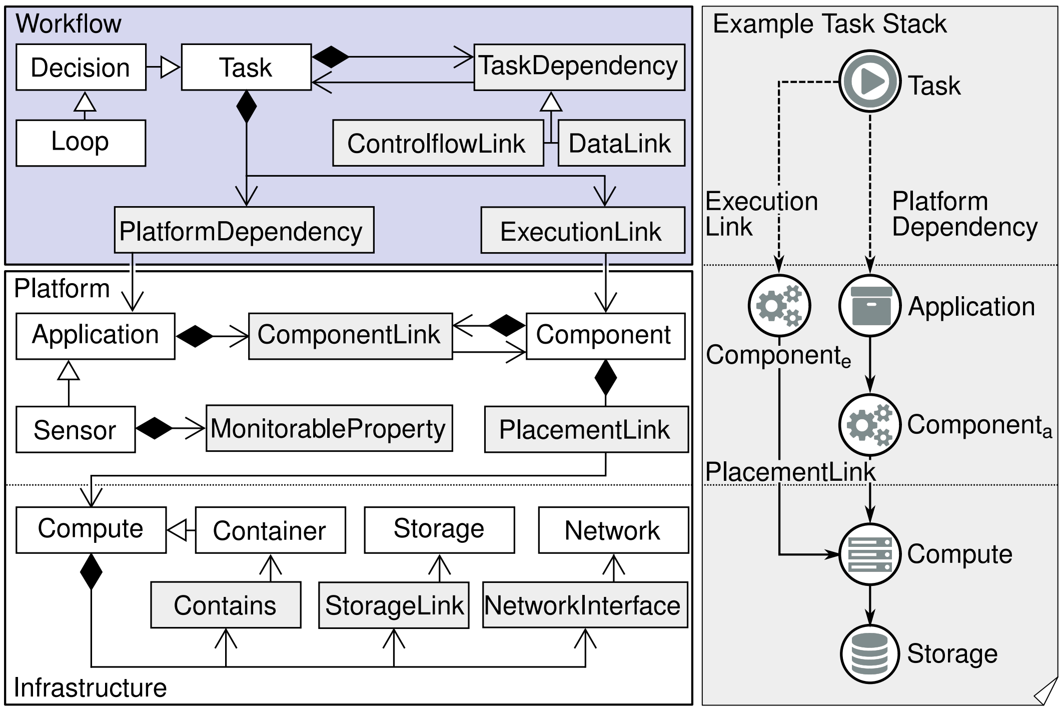


Figure 5.3: Runtime workflow concept with resource types in white and link types in gray.

These specializations allow modeling decision-making processes as well as iterations within the runtime model. To dynamically provision dedicated architectures for individual **Task** instances, its platform requirements need to be modeled as well as the computation to be performed.

The **Platform** layer is reached over the **PlatformDependency** and **ExecutionLink** types. The **PlatformDependency** connects the **Task** to an **Application** that needs to be active prior to the execution of the **Task**. The execution of the **Task** itself is modeled via an **ExecutionLink**. This link typically targets a **Component** representing the computation to be performed by the **Task**. We model the computation using **Platform** layer elements, as the executable represents an artifact that needs to be deployed on the provisioned infrastructure. As both **Task** and **Application** nodes can be connected to components, we differentiate between an *executable component* and an *application component*. This differentiation is derived over contextual information and the kind of links connecting the individual resources. An application **component** is solely connected via a **ComponentLink** to an **Application** representing, e.g., a worker node or service within a large computation cluster. An executable **Component** is targeted by an **ExecutionLink** and describes a computation to be triggered, e.g., a job analyzing data from a repository which utilizes the deployed computation cluster.

The **Infrastructure** layer is composed of **Compute** elements, e.g., VMs, serving as hosts for these **Component** elements. In OCCI these **Compute** nodes can be further connected to additional **Storage** or **Network** instances. Due to the connection of the **Workflow** layer to the **Platform** layer and the connection of the **Platform** to the **Infrastructure** layer, we can model and trace the individual application and infrastructural requirements of a **Task**.

Figure 5.3 depicts an **Example Task Stack** that highlights the structure of the first task from Figure 5.2. In this example, the **Task** represents a data gathering job that requires a specific **Application** to be deployed, e.g., data gathering functionalities. Hereby, the platform requirement of the **Task** is modeled via a **PlatformDependency** binding it to the **Application**. The **Application** itself consists of a single $\textbf{Component}_a$ which is an application component used to abstract several deployment and lifecycle procedures. To host the $\textbf{Component}_a$ a **Compute** node is modeled. In addition, a **Storage** is attached to the **Compute** node to store the data gathered by the **Task**. To trigger the execution of the **Task** an **ExecutionLink** is modeled which connects it to the executable $\textbf{Component}_e$. This executable $\textbf{Component}_e$ is also placed on the **Compute** node. In this example the $\textbf{Component}_e$ abstracts the lifecycle actions to deploy, start, and stop the computation associated with the **Task**.

While the **Platform** and **Infrastructure** layer is already discussed in Chapter 4, this chapter is build around the **Workflow** layer and the different task specializations introduced in the overview. In Section 5.1.1 we give further information about the basic runtime workflow model capabilities covering, e.g., possible task states. Section 5.1.2 discusses the decision-making procedures and the connection to monitoring capabilities. Finally, Section 5.1.3 introduces loop capabilities and their parallelization.

5.1.1 Runtime Model Capabilities

In this section, we go into further detail about the runtime model capabilities of elements in the workflow layer. To reflect and manage the execution of the workflow, each task and data flow possesses its own state. This state is described over an attribute in the model with an assigned FSM. This FSM comprises all possible states of the task or data flow and describes how actions can be enacted on them to traverse through these states. We depict this FSM in Figure 5.4 using the UML notation. The states comprise a **scheduled**, **active**, **inactive**, **skipped**, and **finished** state. The initial state of each task is **scheduled** indicating that the task can be processed as soon as its requirements are fulfilled, i.e., that the required infrastructure is available, and all preceding tasks are in a **finished** state. To trigger the execution of the task, or data transition, the **start** action can be performed. The **start** action of a task is coupled with the deployment and execution of its executable component. As defined earlier, this component describes a computation task to be performed on the compute node it is hosted on. During the execution of the task it may be stopped via the **stop** action. This action transitions the task from an **active** to an **inactive** state. From the **inactive** state, the **start** action can be triggered again transferring the task into the **active** state. Furthermore, an error state exists, which we omit for clarity. This state is reached on failed transitions, e.g., if an error occurs during the **start** action. As soon as the computation has finished, the **finished** state of the task element is reached, indicating that subsequent tasks may be executed. After the execution of a task it may be rescheduled using the **schedule** action, bringing it from the **finished** to the **scheduled** state again. Among others, this transition is used to control the execution of loop iterations allowing to re-execute modeled tasks with new input. Finally, the **skipped** state can be reached from the **scheduled** or **inactive** state. This state indicates that the workflow requirement for subsequent tasks is fulfilled even though it was not executed. Therefore, the final state can be reached from the **skipped** state. We utilize this capability for decision-making processes when only a specific control flow should be followed. The relation of the skipped state and the accompanying decision-making pipeline is described in the following section.

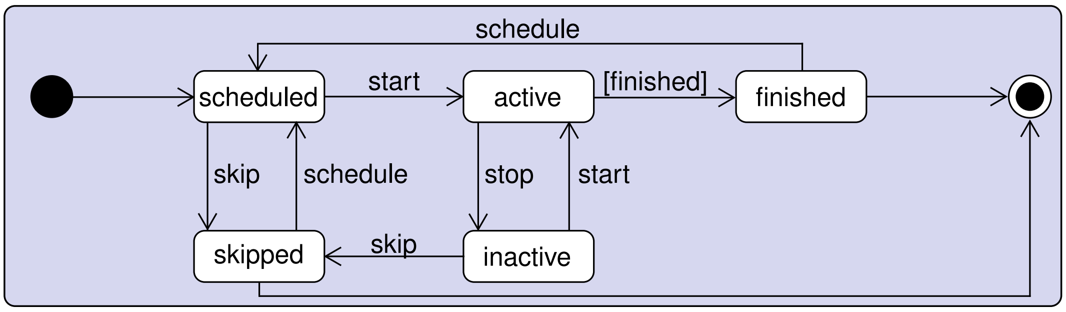


Figure 5.4: Finite state machine representing the states of a task.

5.1.2 Decision-Making Pipeline

To dynamically decide which control flow to follow, each workflow language and management system requires decision-making capabilities. Typically, the decision to be made is performed at runtime and either depends on intermediate results of individual workflow tasks or manual input manipulating which sequence to follow. In order to incorporate such decision-making capabilities in our workflow runtime model, we introduce a decision element and a control flow guard. Both these elements are inspired by the UML standard which already utilizes the concept of guards, e.g., within activity diagrams or state machines [219]. In general, a guard can be attached to a link in order to evaluate whether a certain condition is true. Based on the result it is then decided whether the task followed by the link is eligible for execution. While the utilization of these decision nodes is well known for the use in design time models, we further describe its utilization in a runtime model environment with special regards to coupled infrastructural resources. An overview of the decision-making capabilities is visualized in Figure 5.5. In the following, we provide an in detail description of the individual elements.

The **Decision** element is a specialization of a **Task**, as it represents a step to be executed within the workflow. Compared to a typical **Task** however, its purpose is to gather, process, and store runtime information which is then used to decide which control flow to follow. As the **Decision** is a **Task**, it possesses the same characteristics such as possible states and actions that can be performed on it (see Section 5.1.1). This also includes the possibility to model application and infrastructure requirements via PlatformDependency links, as well as the coupling of executable artifacts via an ExecutionLink. Compared to the usual **Task**, the **Decision** node is encouraged to utilize a **Sensor** as executable which can be, e.g., checked by an OCL constraint. The executable **Sensor** is responsible to inspect, aggregate, and reflect the information that is used to decide which control flow to follow. While a **Sensor** instance is designed to continuously produce monitoring results, we designed the **Decision** node to extract the information from the MonitorableProperty link of the **Sensor**. The extracted information is then stored within the **workflow.decision.input** attribute of the **Decision**. Thereafter, the **Sensor** is stopped, as all the information required for decision-making is available in the **Decision** node and the monitoring procedure is not required anymore. While a **Sensor** can be used to fill this attribute automatically, it can also be adjusted via manual input. This allows the user to alter the control flow dynamically at runtime. For this, a single request is sufficient to adjust the runtime model. After the **workflow.decision.input** attribute is filled, an evaluation pipeline is triggered to decide which control flow to follow by adjusting the state of subsequent **Task** instances. The evaluation itself is build around the **workflow.decision.expression** attribute which is defined in the **Decision** node. This attribute allows users to describe how the runtime information stored within the **workflow.decision.input** attribute is evaluated, e.g., using a Boolean expression. The result of the evaluation is stored within the **workflow.decision.result** and compared against attached **ControlflowGuard** instances.

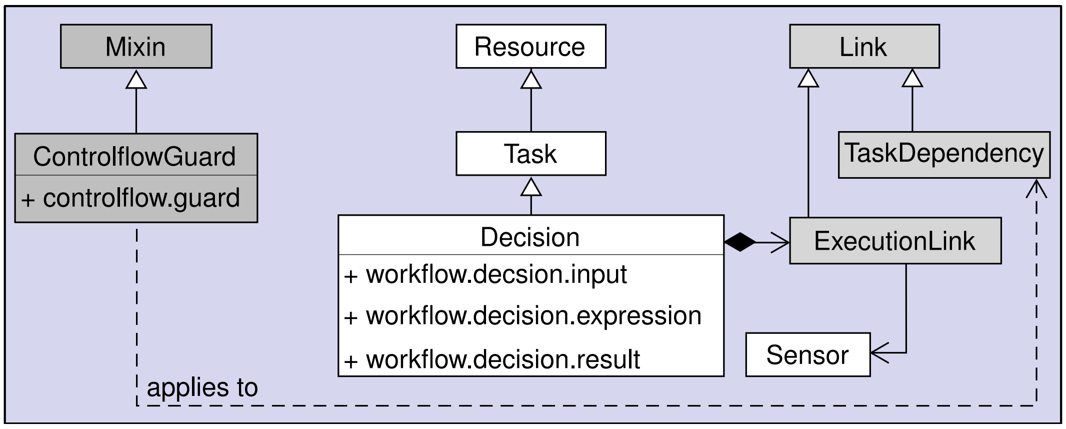


Figure 5.5: Runtime workflow decision concept.

The **ControlflowGuard** element is a specialized **Mixin** which **applies to** a specific **TaskDependency** instance. While the information used for the decision-making process is stored within the **Decision** node, the decision on whether a certain control flow should be followed is denoted within the **controlflow.guard** attribute. This attribute specifies which condition must be met to pursuit the control flow it is guarding. In case of a decision based on a Boolean expression, this guard may be true or false. If the condition of the guard matches the decision result, the **Task** targeted by the **TaskDependency** remains in a scheduled state. Therefore, it is ready to be processed as soon as its infrastructure has been provisioned. If the condition evaluates to false, the task connected by the **TaskDependency** is transferred to the skipped state, as well as all subsequent tasks that cannot be reached otherwise. The skipped state indicates that the **Task** is not performed in this iteration of the workflow. Additionally, the skipped state allows for subsequent tasks to be executed next, as the task has been successfully processed for this iteration of the workflow. This allows the engine to operate until each task is in either a finished or skipped state.

Figure 5.6 provides an example runtime workflow instance highlighting the utilization of a **Decision** node in its **active** state. The **Decision** node ensures that only one of the follow-up tasks are executed instead of both running in parallel. In this figure, we highlight the workflow layer and therefore omit infrastructural resources for the sake of simplicity. In the example workflow, **Task A** just reached the **finished** state allowing the subsequent **Decision** to be executed. This in turn leads to the deployment of the **Sensor** by the workflow engine which is then used to gather the decision-making information. After the **Sensor** has been deployed, the **Decision Pipeline** is triggered. The deployed **Sensor** fills the **input** attribute of the **Decision** with the value **"B"**. Based on the gathered input, the **expression** of the **Decision** is evaluated. In this case, it is checked whether the gathered runtime information matches **"B"** (**input.equals("B")**). Thereafter, the evaluation of the **expression** is stored in the **result** attribute, which in this case is **true**. Finally, the result attribute is compared against the control

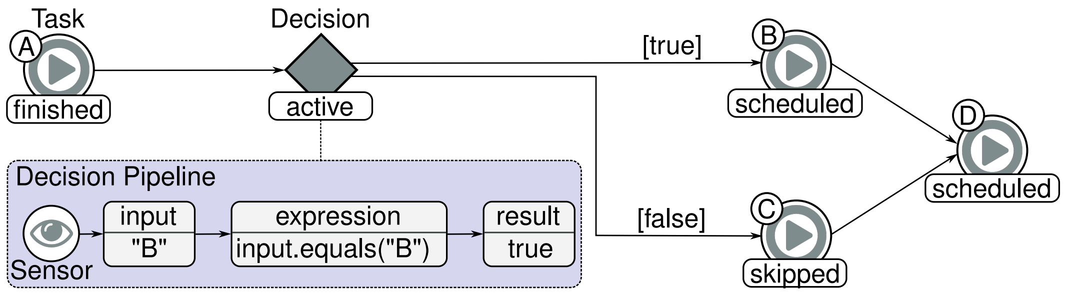


Figure 5.6: Example runtime workflow model with decision-making capabilities.

flow guards (**[true]** and **[false]**) of the connected task dependencies leading to the tasks **B** and **C**. As visualized by the figure, task **B** remains in the **scheduled** state, as only the guard to task **B** matches the decision's result (true). The guard to task **C** however does not match the decision's result and therefore is **skipped**. Even though task **C** is skipped, task **D** remains in a **scheduled** state, as it can still be reached over the control flow of task **B**. At this point the decision node has completed its decision-making process transferring it to the finished state. Therefore, all tasks prior to task **B** are now finished allowing its infrastructure to be provisioned and the task itself to be started next. After task **B** is finished, **D** can be executed as it is still **scheduled**. As soon as **D** has reached the finished state, each task within the workflow is either skipped or finished. Therefore, the termination condition of the workflow management system is reached completing the workflow. In the following, we discuss how the introduced decision-making capabilities can be extended in order to support loops within our runtime workflow model.

5.1.3 Loop Reflection and Parallelization

In this section, we introduce our concepts to reflect and manage the execution of tasks that are contained within a loop. Furthermore, we discuss how these loops can be performed in parallel including a duplication of assigned infrastructural resources. Even though a task may be implemented to internally loop over a set of items, this internal procedure does not reflect the current iteration of the loop within the runtime model. Therefore, we introduce a loop element which reflects all the information required to manage a DCG. Figure 5.7 highlights the necessary additions to our runtime workflow metamodel to model and reflect loops within a scientific workflow. Section 5.1.3.1 describes how a loop can be modeled and reflected at runtime. Thereafter, in Section 5.1.3.2, we describe how loops can be modeled for a parallel execution of looped tasks.

5.1.3.1 Loop Reflection

Throughout each iteration, a loop node needs to decide whether to perform an iteration or not. For this, we define a **Loop** type that inherits from the **Decision** kind, as shown in Figure 5.7. Therefore, the same attributes and mechanics described in the decision-making process are used within the loop such as the decision input and expression attributes (see Section 5.1.2). Furthermore, a **Loop** inherits all the actions of a **Task** and can therefore be started, stopped, skipped, and scheduled (see Section 5.1.1). Additionally, due to this inheritance, a **Loop** can be connected to another **Task** using a **TaskDependency**. This **Task** may again point to a sequence of tasks which end with the **Loop** element forming the actual loop.

To trigger the execution of a **Loop**, its start action has to be requested. This action transfers the **Loop** from the scheduled to the active state. This state describes that the loop is currently processing its looped task instances. After the start action has been triggered, the decision-making process of the **Loop** is executed. This process checks the configured expression against the gathered runtime information. If an iteration is required, each task in the loop is scheduled allowing them to be processed. Otherwise, the tasks are transferred to the skipped state which indicates that they are not mandatory to complete the workflow execution. During each iteration, a loop infuses its tasks with specific items to process. For this, we utilize the **LoopIteration** mixin that can be filled with a name and the value of the variable which

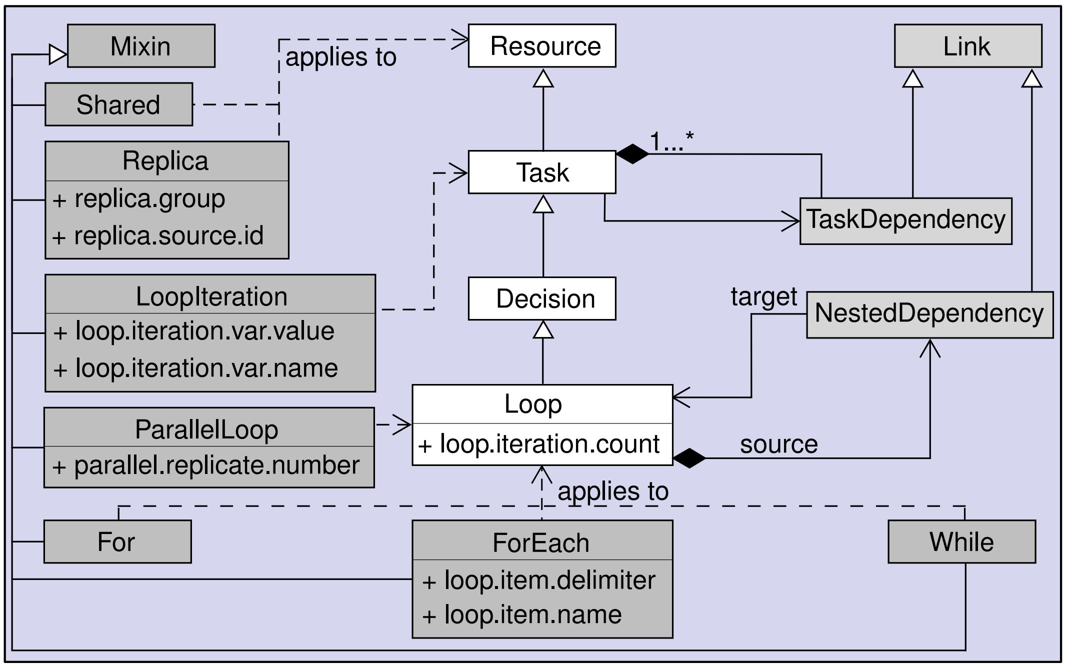


Figure 5.7: Runtime workflow loop and parallelization concept.

is introduced to the executable component of a **Task**. When the item is distributed the first task in the loop can be executed. As soon as the final looped task has reached the finished state, the **Loop** itself is notified. Thereafter, a further decision-making process is performed. Hereby, the decision result attribute is reset as it needs to be reevaluated based on the changed runtime information, e.g., the iteration count or the amount of items left to process. It should be noted, that no new runtime information is gathered by the sensor, as we assume that the items to be processed do not change after they have been gathered. After the decision result has been calculated anew, each task that possesses a control flow with a fulfilled guard is rescheduled. This means, if another iteration is required, all looped tasks are rescheduled, allowing them to be processed and started again. Otherwise, if no further iteration is required, the loop as well as its containing tasks remain in the finished state. If no iteration is required at all, all looped tasks are transferred to the skipped state. Even though no iteration is performed, the **Loop** is executed successfully reaching the finished state allowing to perform follow-up tasks.

To execute the decision-making process and support the looped tasks with dedicated information, each **Loop** requires different kinds of information and, therefore, attributes. In general, each **Loop** has access to a variable that indicates the amount of iterations already performed (**loop.iteration.count**). This attribute, e.g., helps to execute a loop a certain amount of time. Further typical loop types are the **ForEach**, **For** and **While** loops. While these can be interpreted as specializations of a **Loop**, we modeled them as mixins as they can be dynamically applied to the **Loop**. Depending on the kind of loop, different information needs to be reflected. For example, the **ForEach** loop needs to slice the gathered item set into individual items to be passed to the looped tasks. For this, the loop allows specifying **delimiter** as well as a **name** attribute to represent the item. To distribute the items among the looped tasks, multiple **LoopIteration** mixin instances are created and attached to them. Hereby, the **value** and **name** attribute are set to the value of the delimited item and the **name** attribute configured in the **ForEach** loop.

To provide a more concrete overview of the loop concept, Figure 5.8 depicts an example runtime workflow instance that contains a for-each **Loop** with three **Looped Tasks**, i.e., the tasks **B** to **D**. Again, we omit the infrastructure layer to provide a clearer overview of the workflow layer. The **Loop** itself iterates over runtime information gathered by a sensor which is stored in the **input** attribute. Based on the configured delimiter, e.g., a comma, this **input** is split into individual items to be spread among the **Looped Tasks**. In the shown iteration, the **item** $\mathbf{var_1}$ is currently processed and distributed in form of a **LoopIteration** mixin among the **Looped Tasks**. As soon as an item is shared, it is removed from the **input** attribute of the **Loop** ($\mathbf{var_2,...,var_n}$). Therefore, the **expression** attribute is set to **input.isEmpty()** allowing the **Loop** to operate until all items have been processed. Hereby, each iteration follows the same procedure provided by decision nodes. In this case, the decision **input** is not empty resulting in the **expression** to evaluate to false. This leads to task **E** being **skipped** for this iteration, as it is guarded by the condition **true**. The guard to task **B** however matches the result of the decision-making process which results in it being scheduled. As the **Loop** is

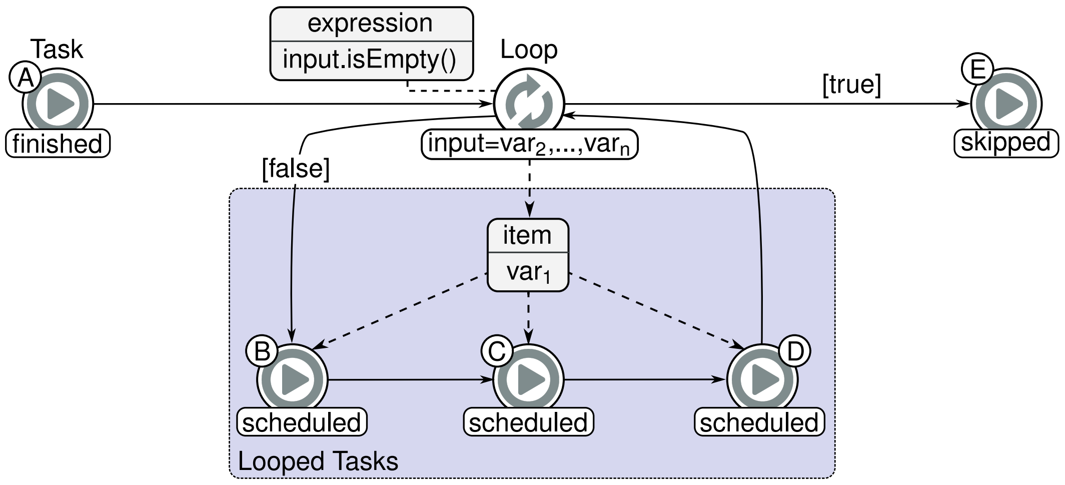


Figure 5.8: Workflow example with an active loop.

active and the task **B** is **scheduled**, its infrastructure can now be provisioned and the task can be performed. Due to the distributed information about the iteration item (**var$_1$**) task **B** and its executable have access to it. Subsequently, task **C** and **D** are executed having access to the item of the current iteration. As soon as the last task of the loop is finished, which in this case is task **D**, the decision result of the **Loop** is reset and the control flow links to follow are evaluated again. Hereby, a new decision-making process for the next iteration is performed, transferring all subsequent tasks with a fulfilled guard to the scheduled state. In this case the **input** holds another item. Therefore, the **Looped Tasks** are rescheduled. Furthermore, the **var$_2$** item is sliced from the **input** and spread among the **Looped Tasks** for another iteration. As soon as no items are left in the decision **input**, the decision-making process evaluates to **true** leading to task **E** being rescheduled. As the guard to the **Looped Tasks** area is not fulfilled, they are not rescheduled leaving them in a finished state. Additionally, this results in the **Loop** to reach the finished state. Finally, task **E** is processed finishing the workflow. In the following, we discuss how loops can be modeled in such a manner that they can be executed in parallel with shared resources.

5.1.3.2 Loop Parallelization

For the parallelization of loops we adopt the concept used by OpenMP [220], an application programming interface commonly used for the loop parallelization in C, C++ and Fortran. Among others, OpenMP allows to simply annotate the loop to be parallelized with information about which variables are shared among spawned threads. To provide similar capabilities, we introduce mixins that can be applied to a loop element to adjust its execution in parallel. These mixins are shown in Figure 5.7.

The **ParallelLoop** can be appended to a **Loop** instance to express that it should be executed in parallel. We refer to this loop as the *original*. The added mixin is recognized in the loop effector which spawns a new set of **Loop** instances on the start action. Hereby, the amount of loops spawned is taken from the **parallel.replicate.number** specified in the mixin. We refer to the newly spawned **Loop** instances as *nested*. After the nested **Loops** are spawned, the original **Loop** is promoted to administrate and supervise the duplicated elements. This covers, e.g., the distribution of items to be processed by individual loops. A connection between the original and nested **Loop** instances is established via **NestedDependency** links serving as source and target respectively. Each **Loop** consists of a duplicated set of **Task** instances formerly connected to the original. Hereby, one of the created nested loops contains the original set of modeled tasks while the other nested loops contain duplicated versions of the tasks.

The **Replica** mixin is used to mark created duplicates while providing a reference to its origin using the **replica.source.id** attribute. Each nested **Loop** forms a *replica group* which we use to determine to which **Loop** each duplicated **Task** or **Resource** belongs. We store this information in each **Replica** using the **replica.group** attribute by referencing the nested **Loop** identifier. Among others, the **replica.group** and **replica.source.id** attributes are used to derive the architecture required for each of the replicated tasks at runtime. Finally, to process the items in parallel, we distribute the items among the nested **Loop** instances and thus the workload. All spawned loops operate in parallel using the aforementioned procedures. As soon as one of the nested loops reaches the finished state, the original **Loop** is notified. When all nested loops are finished, also the behavior of the original loop is triggered rescheduling subsequent **Task** instances and reevaluating its **expression** before it is transferred to the finished state, e.g., by checking whether all items in the input have been processed.

The **Shared** mixin is used to indicate that a resource should not be duplicated but shared during parallel sections of their workflow. One example use case for is, e.g., the utilization of a database application storing the information of all tasks within the workflow. In this case, the application and its underlying infrastructure should not be replicated even though it may be part of a parallelized section of the workflow. While the mixin does not hold any behavior, it serves as a tag that supports the scheduling mechanisms to recognize elements that should not to be parallelized but rather be shared among other resources replicated for the execution of parallelized loop sections.

Figure 5.9 provides an example workflow instance in which the loop from Figure 5.8 is parallelized. In this example, the **Loop** (**O**) has a parallel mixin attached with the **p.number** set to two. In the visualized example, the original **Loop** is currently active, which results in the creation of two replica groups **Replica** and **Replica'**. The **Replica** group contains a nested loop iterating over the tasks (**B**, **C**, **D**) formerly belonging to the original loop (**O**), while the **Replica'** contains duplicated versions of them (**B'**, **C'**, **D'**). To describe the utilization of shared resources, the currently active tasks in this example are **C** and **C'** for which we highlight the **Infrastructure** currently provisioned. Task **C** originally required a single VM which is

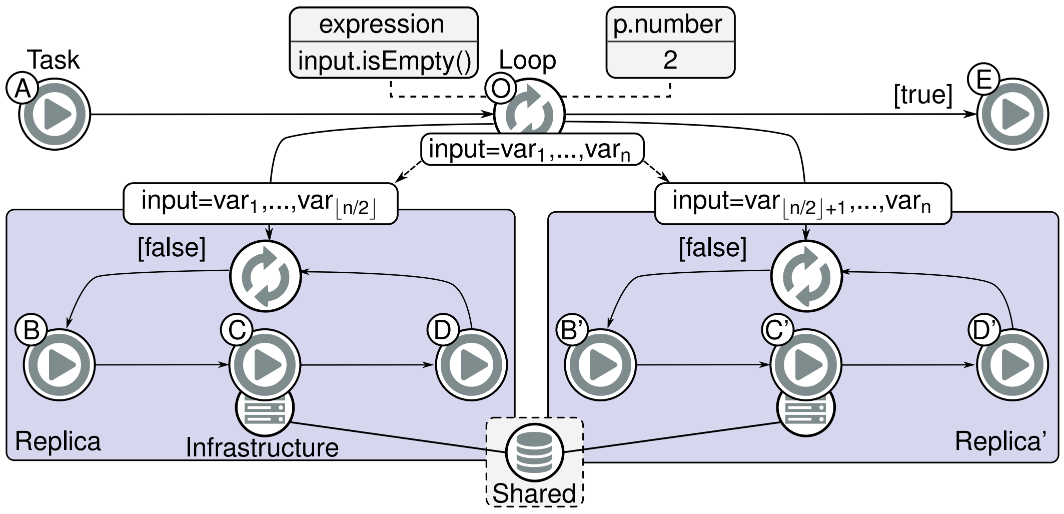


Figure 5.9: Two fold parallelization of the loop from Figure 5.8.

duplicated for the replica task **C'**. Furthermore, in this example, the storage node shown at the bottom of the figure is marked as **Shared** and therefore used for task **C** and **C'**. It should be noted, that the active tasks within the replicated loops may differ at runtime as the duration of the task may depend on its input. This could lead, e.g., to task **B** being active together with task **C'** resulting in tailored and different infrastructure compositions. The figure shows how the input of the original loop is split among the individual replication areas. Here, the **input** from the original loop ($\mathbf{var_1,...,var_n}$) is separated between the two nested loops, so that the **Replica** loop iterates over the first half of the items ($\mathbf{var_1...var_{\lfloor n/2 \rfloor}}$) and the **Replica'** iterates over the second half ($\mathbf{var_{\lfloor n/2 \rfloor+1}...var_n}$). After both replica groups have finished processing their items, the original loop **O** reaches the finished state. At this point, each item has been processed leaving the **input** empty. As a result the expression evaluates to true leaving the task **E** in the scheduled state allowing it to be processed next.

By managing the parallelization of the workflow layer and the distribution of the workload within the runtime model, we leave the provisioning and scheduling of additional required resources to the SWFMS. This allows the engine to simply duplicate the modeled infrastructure for each newly spawned task. The behavior of the workflow engine, including a description of how the required architecture of nested loops are derived, is introduced in the following section.

5.2 Runtime Workflow Execution Engine

In this section, we present the workflow execution engine that uses the capabilities of the runtime model to dynamically schedule and provision the infrastructures and applications required for the individual task within the workflow. As shown in Figure 5.10, the **Design Time Workflow Model** serves as input for the **Workflow Engine**. Overall, the engine follows the concept of a MAPE-K loop which pursuits the goal to transfer each modeled design time task into the runtime model and operate until each has either reached a finished or skipped state. To reach this goal, the **Workflow Engine** creates a **Runtime Model (Snapshot)** ① in each iteration. This snapshot serves as a knowledge pool which provides information about the current state of the workflow and the provisioned infrastructure in the cloud. To control the execution of the workflow, the engine contains two inner components. The **Architecture Scheduler** ②, which is responsible to provision the infrastructure required at each point in time, and the **Task Enactor** ③, which triggers tasks that fulfill both workflow and infrastructure requirements. In the scope of this thesis, we provide two configurations of these components running either in separate cycles or subsequent after each other.

In the following, these components are inspected in more detail. Section 5.2.1 covers the architecture scheduling process and Section 5.2.2 describes how tasks are enacted.

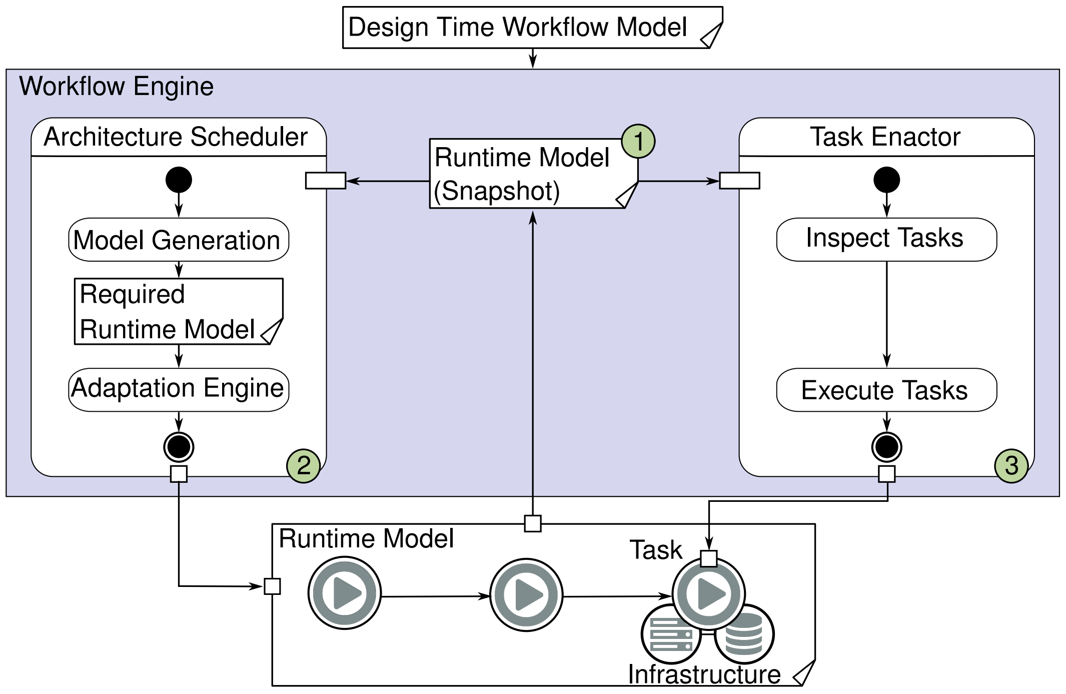


Figure 5.10: Workflow management system components.

5.2.1 Architecture Scheduler

The **Architecture Scheduler** ②, shown in Figure 5.10, orchestrates the runtime model by adapting the **Infrastructure** and application deployment during the workflow execution. For this, the process follows a control loop that analyzes and combines the information of the **Design Time Workflow Model** and **Runtime Model** in each cycle. Based on this analysis, a **Model Generation** is performed, creating a **Required Runtime Model** representing the **Runtime Model** that is required next. The generated model then serves as input for the **Adaptation Engine** which again performs the steps described in Chapter 4 to derive an adaptation plan containing a sequence of requests to adapt the **Runtime Model**. This plan is executed to adjust the **Infrastructure** in the **Runtime Model** and, thus, the provisioned cloud. The execution of the plan slightly differs between the workflow and infrastructure layer. While attributes in the infrastructure layer may be changed, the information within workflow elements may not be adjusted as these contain information to be adjusted by the model itself. Therefore, the update step for the adaptation of elements in the workflow layer is disabled. The process allows managing any kind of resource using the OCCI interface covering not only infrastructure and platform resources but also entities from the workflow layer. Among others, we use this capability to transfer the tasks modeled in the **Design Time Workflow Model** to the **Runtime Model** in the first control loop cycle. As the functionality of the **Adaptation Engine** is already discussed in Chapter 4, Section 5.2.1.1 highlights the general **Model Generation** procedure. Additionally, Section 5.2.1.2 goes into detail about the duplication of resources required for the parallel execution of loops.

5.2.1.1 Required Runtime Model Generation

To generate the required runtime model the scientist's design time workflow serves as a basis for which we simply copy it. Thereafter, we adjust the copy based on information available in the runtime model. By using both models, the generation process has access to the current state of the workflow, and the complete infrastructure required by the individual tasks. As the design time model hosts the information of all the infrastructure required throughout the whole workflow, we prune the resources not required at the current point in time. To identify the resources to remove, the actual state of each task has to be known. Therefore, as a first step, we propagate the state of the tasks from the runtime model to the required runtime model. Based on the states of the tasks, we separate them into 4 categories: *waiting*, *executing*, *finished* and *skipped*. The finished and skipped state are determined over tasks in the corresponding states. Executing tasks are tasks that have all previous tasks finished or are the first task within an active loop. Waiting tasks are tasks that are neither finished, skipped nor executing. Based on the classification, we remove the infrastructure of all tasks marked either as skipped, finished or waiting. Therefore, the required runtime model only contains the infrastructure of each task that is either active or ready for execution. As a result, only the infrastructure currently required gets provisioned.

In Figure 5.11 an example scheduling sequence is shown which highlights the generation of a **Required Runtime Model** based on a **Design Time Model** and **Runtime Model**. In this example, the **Design Time Model** consists of two **Task** instances named **A** and **B**. Task **A** requires one compute and one storage node for its **Infrastructure**, while task **B** operates on a single compute node. In the first cycle ①, the **Runtime Model** is currently empty. This results in the **Required Runtime Model** to only contain the tasks of the **Design Time Model**. Thereafter, in the second cycle ② the **Required Runtime Model** generation utilizes the information of the state information contained in the **Runtime Model** and couples it with the **Infrastructure** information of the **Design Time Model**. In this case, task **A** belongs to the executing category, as it is currently in the state **scheduled** and does not have any previous tasks. Task **B** on the other hand is in the **scheduled** state, but its previous task **A** is not finished. Therefore, the infrastructure of task **B** is removed from the **Required Runtime Model**. The generated **Required Runtime Model** is then used to adapt the **Runtime Model** in order to provision the compute and storage node. This results in the **Runtime Model** of the third cycle ③. In this cycle, task **A** is **finished**, while **B** is still **scheduled**. As task **A** is **finished** its infrastructure is removed from the **Required Runtime Model**. Furthermore, now all previous tasks of **B** are

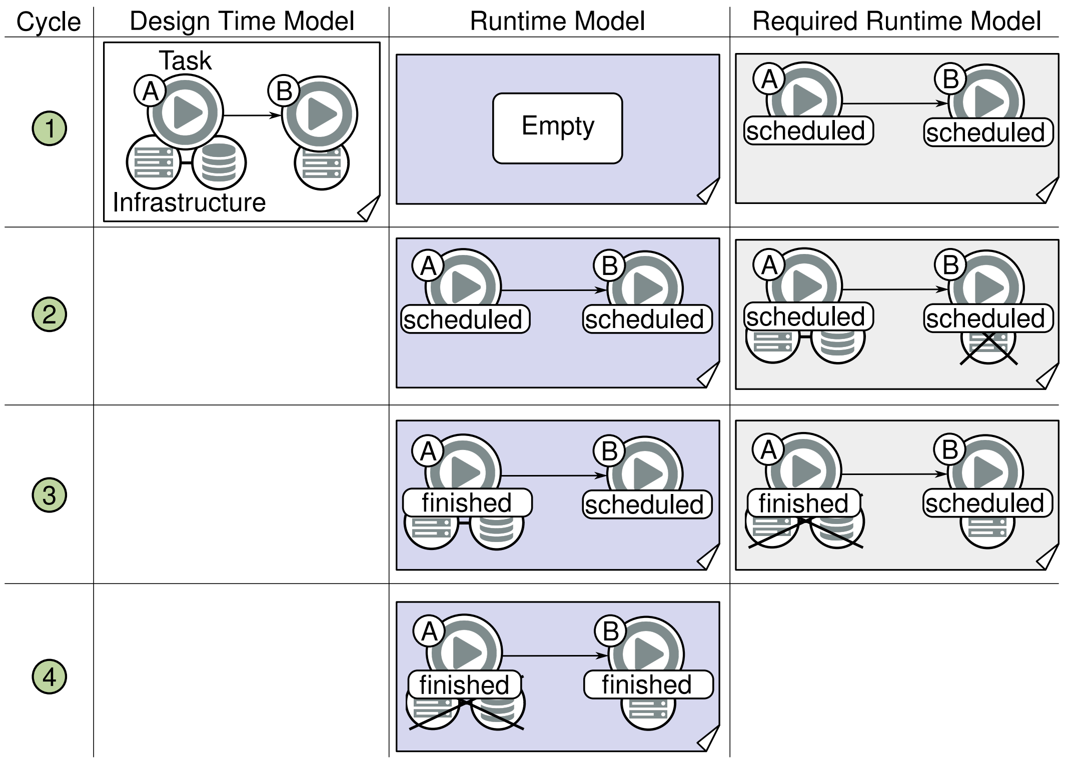


Figure 5.11: Required runtime model generation example.

finished resulting in its infrastructure to remain in the **Required Runtime Model**. Finally, the **Required Runtime Model** is used to adapt the **Runtime Model**. This leads to the fourth cycle ④ in which task **B** is processed. After task **B** has finished the scheduler recognizes that all tasks within the **Runtime Model** are now in the **finished** state completing the workflow. Therefore, no further **Required Runtime Model** is generated. It should be noted, that by default the infrastructure of the last task, in this case **B**, remains in the **Runtime Model** and therefore in the cloud environment. To deprovision all resources, a cleanup task can be modeled. This task can, e.g., store the results of the workflow in an external database and lead to another task which requires no infrastructure at all.

5.2.1.2 Management of Duplicated Resources

While the generation process follows a simple process of removing unnecessary resources, the execution of parallelized loops requires multiplying resources at runtime to fulfill the need of newly spawned tasks. For this, we add another step in the generation process to consider tasks and their links that are marked as replica. These tasks are spawned once a parallelized loop is triggered (see Section 5.1.3.2). Based on the monitored and updated runtime information, the model transformation identifies that additional infrastructure and platform resources are required. To fulfill the requirements of the additional tasks, the transformation duplicates the infrastructure and platform resources required by the original task as modeled in the design time model and attaches them to the replicated tasks. During this process, each duplicated resource is also marked with a replica mixin storing the information of the source id of the original as well as its replication group. While the id of the original entity helps to reference the source of the duplicate, the replication group is used to map each duplicated resource to a specific loop. One exception to the duplication rule are resources that are tagged as shared, including their sub-resources. Even though resources marked as shared are not replicated, the edges towards these nodes are reproduced. This provides the resources in the replica group access to the shared node.

Figure 5.12 provides an example **Required Runtime Model** generated for the execution of a parallel loop. The **Design Time Model** consists of a **Loop** that is executed with a setting of **Parallel 2**. To execute the looped **Task**, two **Component** instances are necessary. One that represents the executable of the **Task** and one which builds the **Application** to be deployed. Both, the execution and application component, are hosted on a single **Compute** node which is connected to a **Storage** node that is marked as **Shared**. The **Runtime Model** represents the state in which the loop just started. This results in the creation of two **Loop Replica** instances that are connected to the original loop via nested dependency links. Based on the **Design Time Model**, the generation of the **Required Runtime Model** creates the infrastructure required for both spawned **Loop Replica** instances. This replication process results in two replica groups, i.e., **Original** and **Replicated**. Both replica groups are identified over the id of their connected **Loop Replica** instances. Additionally, for both groups, the **Required Runtime Model** duplicates the infrastructure requirements of the original task. This includes the compute node, application, and all components. As the storage is marked as **Shared**, only links targeting it are duplicated. Therefore, the **Original**, as well as the **Replicated** section have access to the storage resource.

Overall, the generation is implemented as a model transformation with the resulting model being enacted via the adaptation engine. This allows enhancing the scheduling process by incorporating additional information in the transformation, e.g., estimated execution times of a task to reduce the overall deployment times. Furthermore, the resulting required runtime model may serve as input for further transformations, e.g., to add platform specific information like the addition of a management network. While the architecture scheduler is responsible to manage infrastructure and application resources in the runtime model, it does not trigger the execution of the tasks. For this, the task enactor component is used, which is discussed in the following.

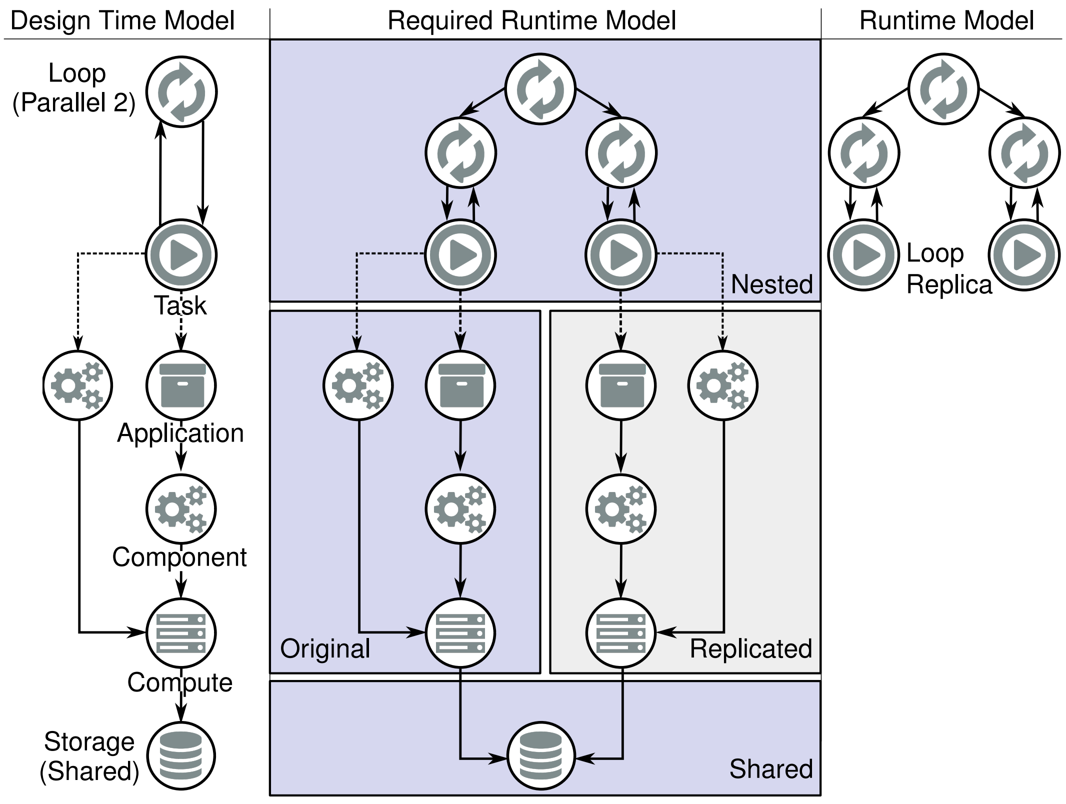


Figure 5.12: Example required runtime model generation for a parallel loop.

5.2.2 Task Enactor

In this section, we introduce the task enactor and describe its procedure to trigger the execution of tasks on provisioned cloud resources. The **Task Enactor** ③, shown in Figure 5.10, identifies tasks that are ready for their execution and enacts them accordingly. Compared to the scheduling process, the task enactor directly operates on the task elements. In most cases, this comes down to triggering the start action of a task. The behavior of the enactment process can be compared to a petri net in which the transitions represent the execution of a task that each have two places. In our case these places denote whether the task dependency if fulfilled and the required architecture is provisioned. To detect and trigger tasks, two subsequent actions are executed. First, the enactor performs an **Inspect Tasks** step, investigating a set of tasks ready for execution. Second, the tasks are triggered in the **Execute Tasks** step. In the following, both steps are described in more detail.

During the **Inspect Tasks** step, the workflow layer of the runtime model is inspected to create a set of tasks that can be executed. We filter for tasks that have all their previous tasks either in a finished or skipped state. In addition, we use the information in the design time model to identify whether all platform and infrastructure resources for the tasks to be executed are provisioned. For this purpose, we traverse the PlatformDependency instances and check whether the connected application is in an active state. We use the active state of elements from the platform layer as indicator because it describes whether the application is successfully deployed and running. Moreover, an active application implicates that the required infrastructure is up and running due to it being a prerequisite for the actual deployment of the components.

In the **Execute Tasks** step, the task enactor takes the filtered tasks ready to be executed as input. For each of these tasks a request is sent to the interface of the task, triggering its start action. Independent of the task specialization, this action transfers the task into an active state and triggers the logic within the corresponding effector. In case of a simple task, this covers, e.g., the deployment of its executable component. When the task is a decision or a loop, the decision-making process is performed additionally. If no data gathering process is modeled at design time, the action waits until the decision-making information is manually filled by a scientist at runtime.

In the following, we discuss the implementation used for the execution of the case study and demonstrate how a runtime model based approach can be used for the development of cloud deployments and workflows by coupling it to different infrastructural systems like cloud and local environments.

6 Model and Execution Environment

In this chapter, we introduce the environment and the ecosystem used to evaluate the orchestration engine and workflow runtime model built around the OCCI standard. Both concepts are implemented as part of the *Smart Workflows through dYnamic Runtime Models* (SmartWYRM) framework which is publicly available [11]. Figure 6.1 depicts the environment to perform our case studies. It consists of two deployed server instances, namely **SmartWYRM** ① and **OCCIWare Runtime** ②.

SmartWYRM ① is a web service that provides a front-end to a **Cloud Architect** or **Scientist** that allows to upload, adjust and monitor the **Cloud or Workflow Model**. To operate the chosen model, **SmartWYRM** hosts the **Adaptation & Workflow Engine** that can be triggered by the user. Both engines utilize the EOL [41] transformation in order to send requests against the **OCCI Interface** and manage resources in the cloud. OCCI implementations typically translate incoming OCCI requests directly to a specific cloud provider. Example for such an interface comprise the rOCCI implementation [221] or the *OpenStack OCCI Interface* (OOI) [222], which we initially used to develop our adaptation engine. These interface implementations are often limited in their capabilities to extend OCCI or are restricted to a single cloud provider. Therefore, we adapted our implementation to conform to the OCCIWare toolchain [123] which provides a formal **OCCI Metamodel** based on EMF with its instances being described in XML. We used the toolchain to design and generate artifacts for our monitoring and workflow **OCCI Extension** and the accompanied **OCCI Interface**.

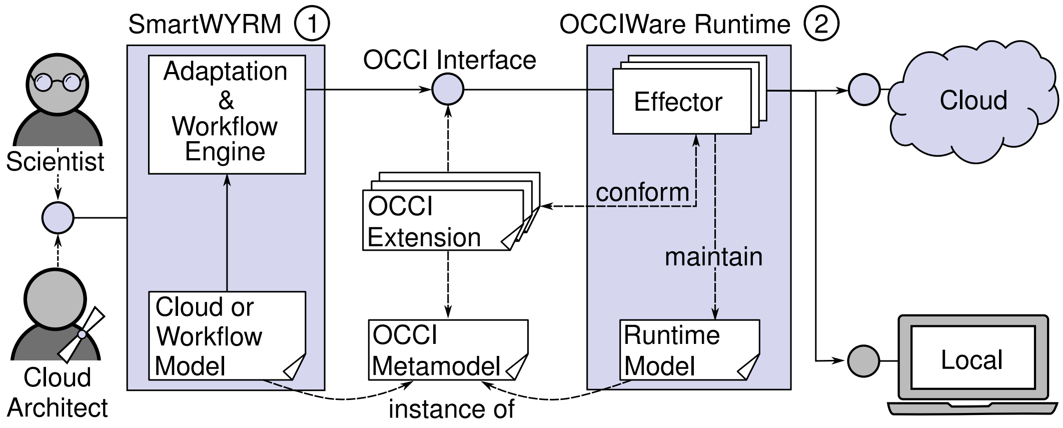


Figure 6.1: Overview: Runtime model environment.

The **OCCIWare Runtime** ② is a server application that implements the **OCCI Interface** using a plugin based approach. Each plugin consists of an **OCCI Extension** which instantiates the **OCCI metamodel**. Furthermore, the server maintains a set of **Effector** instances which **conform** to the kinds introduced by the extension. This includes, e.g., effectors managing infrastructure kinds like the compute kind to spawn VMs in the **Cloud**. In terms of OCCIWare Studio the registered **Effector** components are called connector as they do not only **maintain** the **Runtime Model**, but also inherit the logic to translate the incoming OCCI requests to a target infrastructure. The targeted infrastructure is not limited to **Cloud** environments. Within the scope of this thesis, we test the runtime model capabilities to simulate cloud adaptations and perform workflows using **Local** resources. We utilize this capability, e.g., to perform automatic tests within *Continuous Integration* (CI) environments. Moreover, we use this environment for local deployments in order to support the development of deployment models, configuration management scripts and adaptation engines.

Section 6.1 presents the infrastructure configuration of the cloud and simulation environments. Thereafter, in Section 6.2, we provide an overview of the framework used to execute the workflow and orchestration case studies.

6.1 Causal Connection Configurations

In this section, we briefly describe the OCCI extensions registered in the OCCIWare Runtime and SmartWYRM environment. We highlight the causal connection to the cloud environment, as well as the effectors used to establish a connection to our local workstation. Furthermore, we provide a detailed overview of utilized OCCI extensions and plugins including short description of their functionality. As the concepts of each element are already described in previous chapters, this section focuses more on technical aspects. Table 6.1 separates the used extensions into four categories: **Core**, **Infrastructure**, **Platform**, and **Workflow Extensions**.

The **Core Extensions** represent the extensions providing abstract elements from which other extension elements inherit. In this case, the layer only consists of the OCCI Core extension [98] allowing to model basic Resource and Link instances.

The **Infrastructure Extensions** introduce entities to model and manage resources residing on the infrastructure layer. The OCCI Infrastructure extension [103] extends the core with entities to manage a cloud's IaaS. These include Storage, Network and Compute entities, e.g., to model VMs, as well as link kinds to connect them. The Docker extension [104] adds Container and Machine entities, which are specializations of the Compute kind from the OCCI infrastructure extension. The added types are used to provision Docker containers as well as their hosts. The OpenStack Runtime extension [105] models a RuntimeId Mixin that can be attached to any kind of infrastructural resource allowing to map an OCCI id to the id assigned by the cloud provider. This information is, e.g., used in the comparison process of the adaptation engine.

Core Extensions	**Entities added**
OCCI Core [98]	Entity, Resource, Link

Infrastructure Extensions	
OCCI Infrastructure [103]	Compute, Storage, Network incl. Links
Docker extension [104]	Machine, Container, Contains link
OpenStack Runtime [105]	RuntimeId Mixin

Platform Extensions	
MoDMaCAO platform [105]	Application, Component, PlacementLink
MoDMaCAO Ansible [105]	AnsibleEndpoint Mixin
MOCCI [223]	Sensor, DataGatherer, DataProcessor, ResultProvider

Workflow Extensions	
WOCCI [224, 225]	Task, Decision, Loop, Task/PlatformDependency

Table 6.1: OCCI extensions registered by SmartWYRM and the OCCIWare Runtime.

The **Platform Extensions** introduce elements that are deployed on top of the provisioned infrastructure. The basis for these deployments is contained within the MoDMaCAO platform extension [105], improving the standardized OCCI platform extension [106]. The MoDMaCAO Ansible extension introduces a specific AnsibleEndpoint Mixin. This mixin gets attached to OCCI NetworkInterface instances marking them for the use of configuration management script executions. Using the MoDMaCAO platform extension, we created the *Monitoring with OCCI* (MOCCI) extension [223] to model and deploy Sensors and its monitoring instruments including the DataGatherer, DataProcessor and ResultProvider.

The **Workflow Extensions** extend OCCI with elements used to model and reflect workflow elements. These include Task, Decision and Loop elements as well TaskDependency instances sequencing them. Also, this extension provides PlatformDependency links allowing to attach platform and infrastructure requirements to the individual tasks.

For each of the introduced extensions one effector exists that implements the behavior of the individual resources. Throughout the development of our approach, we created several sets of effectors. One of them covers a direct connection to a cloud environment. Another one is utilized to perform automated tests and simulate the runtime model's behavior. Yet another is used to perform workflows on our local workstation. In the following, we provide a more detailed look into the different kinds of effector sets and how they handle the different extensions presented in Table 6.1. Section 6.1.1 provides implementation details about the causal connection to an OpenStack cloud, while Section 6.1.2 discusses the utilization of a simulation environment.

6.1.1 Causal Cloud Connection

In this section, we describe how the causal connection to a cloud environment is covered by referring how the elements of the individual extension layers shown in Table 6.1 are managed. For the execution of our case studies, we developed effectors providing a causal connection to a private OpenStack cloud. Hereby, the effector is configured in such a manner that it influences a single tenant in the cloud used for the orchestration of cloud adaptations and workflow executions. Especially when using this connection, the application of the PIM to PSM transformation is mandatory. Here, the transformation adds the management network to the model which is required for the deployment of component instances.

For the **Infrastructure Extensions** an OpenStack effector is implemented that translates incoming OCCI infrastructure requests, e.g., of kind Compute, Storage or Network to the OpenStack interface using the OpenStack4j library [226]. To connect to the cloud and the desired tenant, we initialize the effector using pre-configured configuration files. Overall, incoming OCCI requests are translated into requests fitting to the OpenStack interface, e.g., by referring to the requested resource attributes. Changes to the cloud environment directly are handled and integrated to the runtime model by constantly monitoring the proprietary cloud interface. If a new, deprecated or changed resource is recognized, the translation process is reverted translating OpenStack information to OCCI adding the changes to the runtime model. While the OpenStack effector has to be configured with tenant information, the Docker extension effector introduces Machine elements which allow to directly spawn VMs on the OpenStack cloud by configuring corresponding attributes within the Compute instance. These machines, however, serve the purpose of a Docker host and are spawned using the Docker-Machine tool [227]. In addition to OpenStack hosts, the Docker extension includes further Machine entities to spawn VMs on different cloud providers.

For the **Platform Extensions**, we utilize an effector for the MoDMaCAO platform extension that couples Component instances with configuration management scripts. While MoDMaCAO supports multiple configuration management systems, we focus on the utilization of Ansible configuration management scripts [86]. In the scope of this thesis, we additionally integrated basic functionalities of SaltStack [87] to MoDMaCAO to further investigate the flexibility of the framework. Furthermore, we tested the integration of Bash [81] in the scope of a supervised students project and evaluated the advantages of common configuration management approaches. The effector for the MOCCI extension functions similarly to MoDMaCAO with its elements focusing on monitoring capabilities. For example, the offered functionalities allow a Sensor to reflect its monitoring observations directly to the runtime model. The behavior of this effector is described in Section 4.2.2.

Finally, the effector for the **Workflow Extensions** manages requested workflow elements as described in Section 5.1. This management solely takes place on a model based level as no resources in the cloud are directly affected.

6.1.2 Simulation Connection

In addition to the effectors used for a cloud environment, we implemented a set of dummy effectors to simulate the behavior of the runtime model. We use this set of effectors for several purposes. Originally, the simulation effector got developed to support the implementation process of our adaptation engines including the automatic execution and validation of test scenarios. We enhanced this environment to locally deploy cloud topologies to assess the impact of adaptation changes before applying them to a cloud environment. Furthermore, the effector set allows to partially replicate our case studies as no access to an actual cloud is required.

Figure 6.2 highlights the process of fitting a cloud deployment model into a local environment. The **effector$_C$** establishes a connection to the **Cloud**, while the **effector$_L$** spawns a simulation environment on the local **Workstation**. To replicate the actual cloud deployment the **Simulation Configurator** is used. Here, we **Extract** the **Runtime Model (Cloud)** and **Transform** it by adding one *simulation tag* to each resource. These tags serve as an annotation for the **effector$_L$** to trigger a desired simulation behavior. The annotated simulation can be adjusted by the user in the **Configure Level** step. This sets up the **Runtime Model (Local)** to perform the desired simulation on top of the local **Workstation**. For example, the model can be configured in such a manner that compute resources are only partially deployed in order to develop or investigate configuration management scripts of interest. Additionally, the simulation environment can be used to develop **Scaling Rules** and runtime model engines. Depending on the effector implementation, we need to **Adjust** the scaling rules to consider the desired simulation tags. For a more convenient use, this step can be bypassed by implementing an **effector$_L$** set with pre-configured simulation levels for each resource type. This pre-configuration, however, comes with the drawback of fewer simulation options. After assessing the impact of planned changes in the local environment, the added simulation tags can be removed from the **Runtime Model (Local)**. The resulting model can then serve as input for the orchestration engine which derives the required requests to adapt the **Runtime Model (Cloud)** to the new desired state.

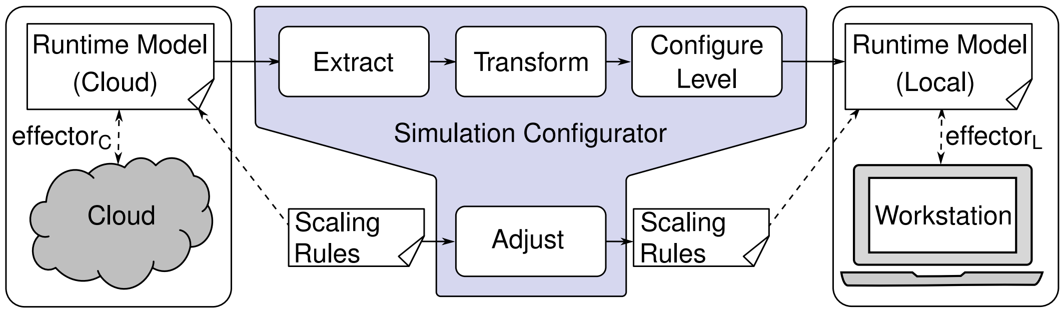


Figure 6.2: Cloud and simulation environment connection.

Figure 6.3 shows the **Simulation Extension** introducing the **SimulationTag** which we attach to each individual resource during the simulation transformation process. The **SimulationTag** is implemented as a mixin and introduces a **simulation.level** attribute. This attribute allows for a more dynamic management of simulated resources. By default, the level zero describes that each resource and operation should only be simulated on a model based level which covers, e.g., state transitions. In the scope of our OCCI **Simulation Extension**, we introduce three specializations of the **SimulationTag**. Hereby, one tag for each of the base OCCI kinds exists, e.g., **ComputeSim**, **SensorSim** and **ComponentSim**. In the following, the simulation options are described in reference to the extension categorization shown in Table 6.1.

The **Infrastructure Extensions** simulation effectors do not contain any connection to an external infrastructure interface. Instead, they mimic the existence of distributed resources. Therefore, when an infrastructure resource is requested, the corresponding entity is added to the runtime model without affecting any other system. In the current implementation, we focused on the simulation of compute nodes using the **ComputeSim** tag. In case of the default **simulation.level** of zero, only state changes are performed based on the FSM of the corresponding resource kind. When the resource is simulated with a **simulation.level** greater than zero, we locally spawn a VM, e.g., using VirtualBox, or by treating the modeled compute node as a Docker container using the Docker extension. The **virtualization.type** attribute defines the type of deployment, e.g., container or compute type while the **compute.image** refers to the desired image. Even though the OCCI extension already provides a mixin to specify an image, we utilize the information in the simulation tag as it can be easily removed once the local development process has been finished.

For the **Platform Extensions** different behaviors for the individual elements can be selected by choosing a specific **simulation.level**. For the default **simulation.level** of zero, any behavior directly affecting provisioned resources is removed. For this, we only removed a single line from the configuration manager of MoDMaCAO triggering the execution of the configuration

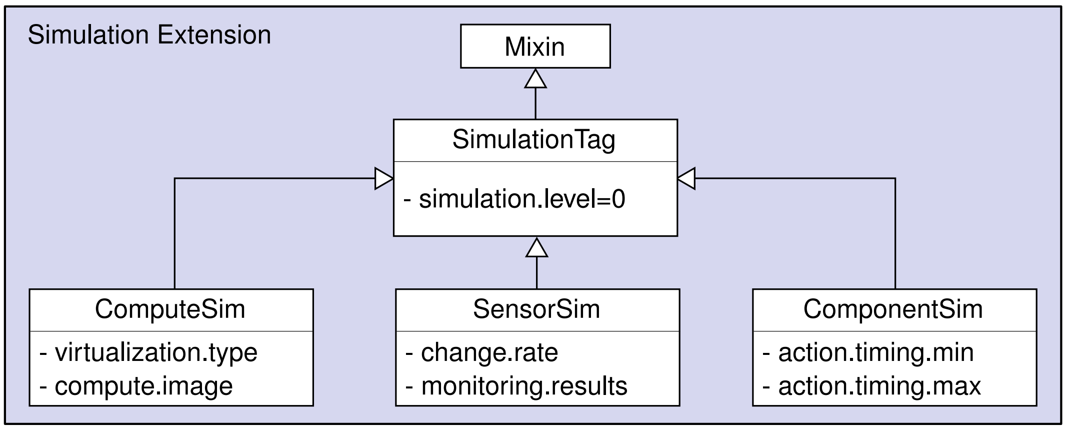


Figure 6.3: Simulation extension subset.

management script which results in a simulation of state changes. For the simulation of components we utilized different configurations of the **ComponentSim** which can be defined over the **simulation.level** and its attributes. To visually observe the behavior of deployment procedures and the changing runtime model, we slowed the adaptation process down by injecting artificial deployment durations for individual application and component lifecycle actions. These durations can be adjusted over two attributes **action.timing.min** and **action.timing.max** which represent a range from which a random value within is drawn. Finally, when setting the **simulation.level** to three, an actual deployment is performed on the attached compute host utilizing the behavior from the effector used for the actual cloud environment. For this, it must be ensured that the attached compute host has a simulation level greater than zero in order to allow for an actual deployment. This, e.g., can be checked using model validation.

For the MOCCI monitoring extension the effector can be configured to simulate the monitoring information in the runtime model using values defined by the user. For this, the **SensorSim** annotation is used which reflects artificial monitoring information in the runtime model. This capability allows testing orchestration engines and scaling rules without the time requirements for actual deployments. To generate monitoring information, the simulation annotation provides two attributes **montoring.results** and **change.rate**. These respectively describe the values to be observed by the sensor and how often they should change. Using higher simulation levels, the sensor and its components can also be deployed on local resources to monitor actual workload. Therefore, the developer can choose between actual or generated workload that can be adjusted at runtime. As a result designed cloud applications and scaling procedures can be simulated using dedicated workload scenarios.

For the **Workflow Extensions** the behavior remains the same as within the cloud effector. No simulation behavior is needed for the workflow layer, as its elements do not represent actual cloud resources and therefore can reside on a model based level.

In the following, we describe the our webserver application which implements the cloud adaptation and workflow procedure used for the execution of our case studies.

6.2 Smart Workflows Through Dynamic Runtime Models

To operate the orchestration process and workflow engine, we developed a web application called SmartWYRM. The framework and all surrounding implementations are publicly available [228]. As described in the previous section, SmartWYRM utilizes the OCCI metamodel by Merle et al. [122] which is also used in the OCCIWare toolchain. Additionally, SmartWYRM registers the generated extensions and effectors based on the metamodel concepts presented in Chapter 4 and Chapter 5. We chose to implement the proposed workflow engine in form of a webserver to neglect the need for a local workstation to run, especially as scientific workflows are often long-running processes. Furthermore, by relying on requests for communication, the engine is loosely coupled to the OCCIWare

Runtime server representing the OCCI interface. This allows connecting the application to further OCCI interface implementations. Even though other OCCI interfaces may not register the elements presented in the workflow extensions, basic functionalities of the adaptation mechanisms can be utilized. During the execution of the deployment and workflow engines, job histories are recorded and visualized by SmartWYRM. These views grant insight into the deployment and workflow execution including planned changes, provisioning and deployment times.

In Section 6.2.1, we provide an overview of the interface and notation used in the framework and the execution of our case studies. Thereafter, the implementations of the engines are briefly covered by discussing recorded job histories and their visualization. Section 6.2.2 discusses the implementation of the workflow engine. Section 6.2.3 introduces the adaptation engine implementation.

6.2.1 Interface and Notation

In this section, we introduce our concepts to support the visualization and execution of our approach. As part of our implementation, we designed a graphical interface and notation for OCCI. Hereby, we chose a notation that emphasizes the structure of cloud deployments at the cost of a limited insight to configured attributes which we circumvent by utilizing interactive capabilities offered by web frameworks. In the following, we introduce the OCCI notation used within the framework and thesis. Additionally, we briefly introduce the implemented interface used for the execution of our case studies.

Figure 6.4 provides an overview of the icons used to visualize OCCI deployment and workflow models. This figure serves as a legend that separates the different elements according to their OCCI extension, namely the **Core**, **Workflow**, **Monitoring**, **Platform**, **Infrastructure** and **Docker** extension. In general, the notation follows a graph based visualization with

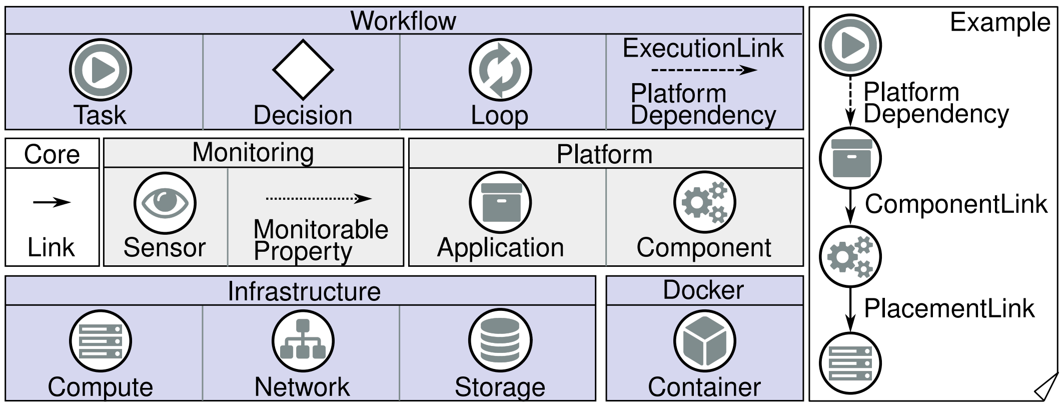
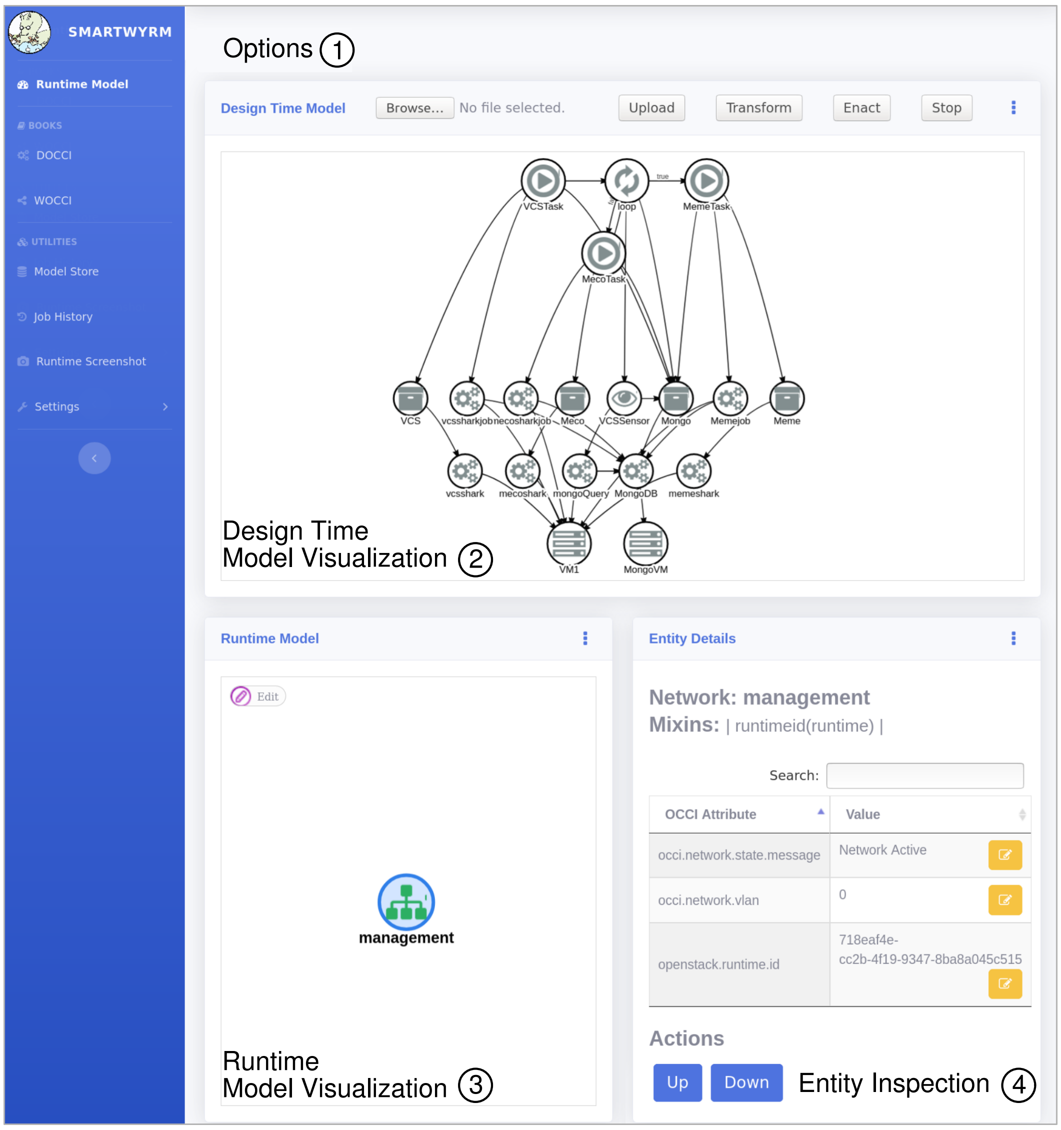


Figure 6.4: Legend: OCCI cloud notation.

each resource being a circular node inheriting an icon that corresponds to a specific kind. In addition to the graphical representation of the resources, we introduce a hierarchical separation based on a resource's kind and extension. For example, **Infrastructure** resources are depicted at the bottom of each workflow, while the **Workflow** elements remain at the top. In our graphical notation, **Link** instances and their specializations are visualized as a simple arrow. To better differentiate the connection between the workflow and platform layer, we highlight **ExecutionLink** instances and **PlatformDependency** links using a dashed line. Moreover, we visualize **MonitorableProperty** links with a dotted line as they target resources on different layers. Next, the SmartWYRM user interface is introduced which implements the introduced icons and our visualization of OCCI.

Figure 6.5 shows a screenshot of the SmartWYRM webserver interface. On the top bar the **Options** ① are displayed which provide the capability to upload and transform a design time model. After the OCCI model has been uploaded, the MDA transformation chain can be triggered on top of this model. This transformation adds, e.g., platform specific or default resources and information that can be configured in the framework. The uploaded model, as well as resources added to it can be inspected in the **Design Time Model Visualization** ②. In the figure a workflow with three tasks from which one is looped is shown. When the model is enacted, changes to the runtime model can be observed in the **Runtime Model Visualization** ③. Currently, only the management network is present in the runtime model which, e.g., can be automatically added to the design time model. Both, the runtime model and the design time model, are displayed using the same graph visualization introduced in Section 6.2.1 and used throughout the thesis. If one of the entities is clicked, as shown by the blue highlighting, the **Entity Inspection** ④ is opened. This window displays the mixins and attributes of the chosen runtime entity and allows to easily interact with the model. In addition to a visual representation of the entities attribute, this window can be used to adjust values. Also, actions that can be applied on the entity are automatically derived from the registered plugins. The runtime model visualization allows to actively observe how the system changes throughout adaptation and workflow processes. Here, new resources can be dynamically added to the runtime model using a form derived from registered OCCI extensions. For the derivation, we extracted the information from the formalized metamodel and the extensions created with it. This form navigates the user step by step through the creation process of a new element, covering the selection of the extension, resource type, desired mixins, and the definition of the resources attributes. Further features implemented in SmartWYRM that got utilized in this thesis cover, e.g., the capability to download and store models previously executed, as well the extraction of images which we use within our case study chapters.

In the following the implementation of the workflow engine and the recorded job history is explained in more detail.



Network: management
Mixins: | runtimeid(runtime) |

OCCI Attribute	Value
occi.network.state.message	Network Active
occi.network.vlan	0
openstack.runtime.id	718eaf4e-cc2b-4f19-9347-8ba8a045c515

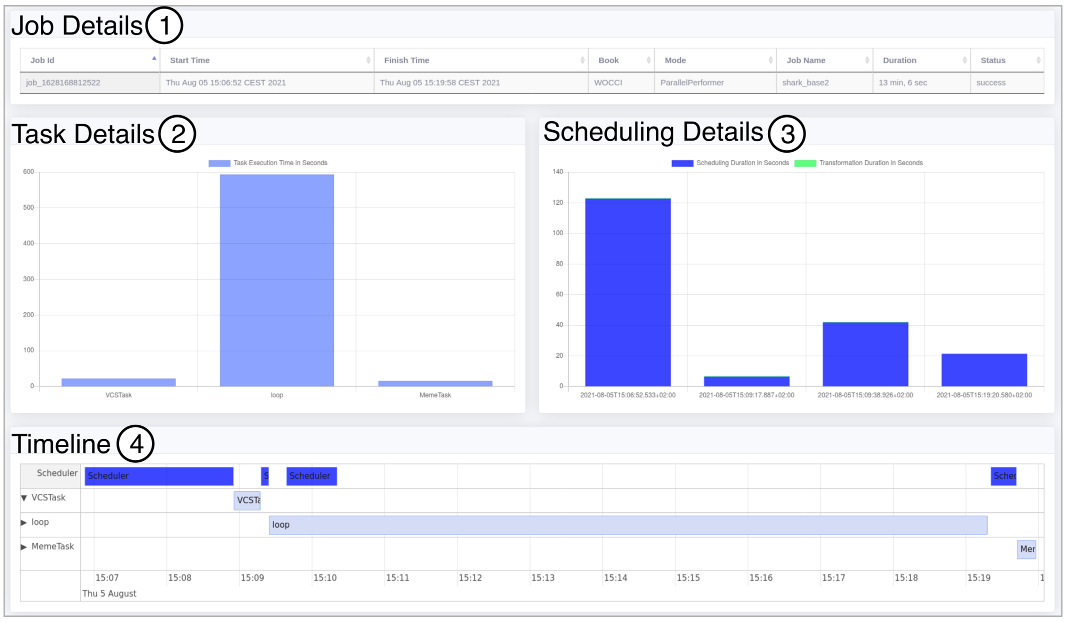
Figure 6.5: Screenshot excerpt of the SmartWYRM user interface.

6.2.2 Workflow Engine Implementation

In this section, the implementation of the workflow engine, presented in Section 5.2, is introduced. We refer to the engine and workflow extension as *Workflows and OCCI* (WOCCI) [229]. In the scope of this thesis, we implemented and tested different sequences of the architecture scheduling and task enactment process. A *sequential* arrangement of the procedures, as well as a *parallel* one.

The sequential arrangement implements alternating behavior of the architecture scheduling component and the task enacting component. This means that a task's infrastructure is first built followed by the execution of the task. To facilitate the dynamic capabilities of the runtime model, especially for the execution of parallelized workflows, we implemented the task enactor and architecture scheduler within separate control loops. This allows a task that is ready for execution to be enacted without needing to wait for the deployment process to be finished. However, for this process, the individual cycles require more information of each other to avoid race conditions. To distinguish the behavior of the different execution modes, we store the chosen procedure type within the job history visualization. Based on the design time model workflow shown in Figure 6.5, Figure 6.6 shows an example workflow job history.

The **Job Details** ① show general information about the workflow execution covering its id, name, duration status, the start and end time as well as the chosen workflow execution mode. As soon as the workflow has finished execution, i.e., all task states are finished, the job is marked as success. Further job states describe whether, e.g., failed tasks are detected in the runtime model. The **Task Details** ② show the execution durations of individual tasks, while the **Scheduling Details** ③ display the durations of performed architecture scheduling processes. In case of the shown example, three task durations are shown which are accompanied by four architecture scheduling processes. In the scope of our evaluation, we use the recorded values to quantitatively analyze and discuss our approach. Finally, a **Timeline** ④ is


Job Details ①

Job Id	Start Time	Finish Time	Book	Mode	Job Name	Duration	Status
job_1628168812522	Thu Aug 05 15:06:52 CEST 2021	Thu Aug 05 15:19:58 CEST 2021	WOCCI	ParallelPerformer	shark_base2	13 min, 6 sec	success

Figure 6.6: Screenshot of the workflow job history.

displayed, highlighting at which point in time the individual task enactment or architecture scheduling processes were triggered. This view highlights the behavior differences between the sequential and parallel execution of the workflow job. In case of the provided figure, a parallel job is shown which can be observed by the overlapping task and architecture scheduling timelines.

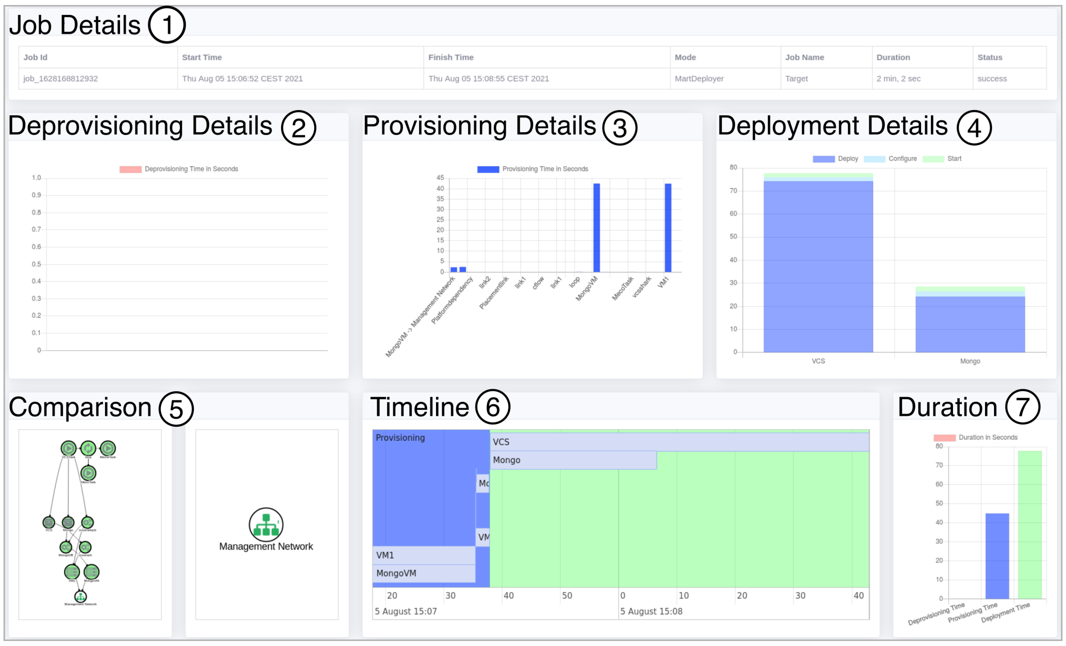
As the workflow architecture scheduling process heavily relies on the adaptation engine, we store one adaptation job history for each scheduling procedure. These scheduling histories are coupled to the individual bars shown in the **Scheduling Details** ③ which allow investigating the resource management during the workflow execution. In the following, we go into detail about the adaptation engine implementation and the accompanied history visualization.

6.2.3 Adaptation Engine Implementation

In this section, we briefly introduce the implementation of the adaptation engine presented in Section 4.1 including a description of the recorded job history visualization. We refer to the engine itself as *Deployment of OCCI* (DOCCI) [230]. This engine is not only included in the SmartWYRM webserver, but also TOSCA-Studio [217] which is an Eclipse IDE plugin that can transform TOSCA to OCCI models and deploy them via DOCCI. Independent of the wrapping framework, DOCCI takes an OCCI model representing the desired state in the cloud as input. As described in Section 4.1, it compares the input OCCI model against the currently deployed one and sends requests against the OCCI interface provided by the OCCIWare Runtime server. While we do not go into detail about the implementation specifics, we highlight the information provided by our job history as it reveals the behavior of the adaptation engine. Furthermore, the recorded values are later on used to quantitatively discuss the adaptation process performed during our case studies. An example history for an adaptation job is shown in Figure 6.7. The shown example highlights the adaptation process from the first scheduling task shown in Figure 6.6.

The **Job Details** ① depict general information such as the job id, start time, finish time, the job name, the duration and whether the job was successful or not. The state of success is hereby derived by checking whether the goal of the adaptation engine was reached by comparing the desired input model to the runtime model once the adaptation process has been finished. If all elements are the same at design and runtime, the job is considered successful. Further job states are stopped and failure depending on whether the engine was stopped during execution or whether the desired state could not be reached, e.g., due to failing component deployments. The **Deprovisioning Details** ② and the **Provisioning Details** ③ describe the time required for the individual requests. Hereby, each performed request is listed which allows us to compare their individual response times. The time is taken from the moment we send the request until an answer is received. Therefore, the observed timing heavily depend on the behavior implemented in the effector of the entity. For example, a request provisioning an actual VM in the cloud may take several seconds, while a request managing a task may only require milliseconds. The **Deployment Details** ④ display the application deployment times including the time required to perform the deploy, configure and start action of all applications in the model. In case of the figure, e.g., the duration for the deployment procedure is taking the most time. We especially investigate the deployment of applications as they represent the element managing the underlying components and is of most interest to us. The duration of update requests are not recorded nor visualized as they only change attributes in the runtime model which only takes milliseconds and thus are not of interest to us.

The boxes shown in the **Comparison** ⑤ visualize the input OCCI model, as well as the runtime model at the time the engine is triggered. In case of this example, a management network is currently present in the runtime model, with the targeted model being the required runtime model from the first adaptation cycle of the workflow execution. In addition to the visualization of the desired and current state, this view changes the background color of the nodes in the model visualizations to highlight planned adaptive actions. For example, a green background within the nodes describes the addition of new resources. The **Timeline** ⑥ displays the exact point in time of each request. Hereby, a background color highlights to which phase of the adaptation process the request belongs, e.g., blue describes the provisioning and green the deployment phase. Complementary to this, the **Duration** ⑦ box provides a more concise overview of the duration of these phases in form of a bar plot.


Job Details ①

Job Id	Start Time	Finish Time	Mode	Job Name	Duration	Status
job_1628168812932	Thu Aug 05 15:06:52 CEST 2021	Thu Aug 05 15:08:55 CEST 2021	MartDeployer	Target	2 min, 2 sec	success

Figure 6.7: Screenshot of the adaptation job history.

7 Runtime Model Orchestration Case Studies

To investigate our orchestration capabilities for OCCI, we designed several scenarios that highlight the adaptation, simulation and monitoring concepts introduced in Chapter 4. For this, we use the implementations and the notation described in Chapter 6 including the cloud and simulation connections. Due to the similar composition of our orchestration and workflows studies, we discuss the threats to validity of this study in Section 9.4. In the following, the remainder of this section is presented focusing around the scenarios shown in Figure 7.1.

Section 7.1 discusses a **Computation Cluster Scaling** ① scenario using the big data computation frameworks Apache Hadoop [231] and Apache Spark [232]. In this study, we highlight the **Orchestration** process transferring the designed model elements to the runtime model and cloud environment. Furthermore, in this case study, we highlight the utilization of the **Simulation** environment to extend the model with a new component introducing the capabilities of Apache Spark. Additionally, we extend the OCCI model with **Monitoring** functionality to directly reflect the monitoring results in the runtime model.

Section 7.2 introduces the second case study which we use to demonstrate the **Standard Interoperability** ② of OCCI with the TOSCA cloud standard. Using this study, we highlight the **Compatibility** and **Extensibility** of the OCCI data model and the **Generalizability** of its interface with special regards to our orchestration process. For this study, we utilize an existing cloud deployment model originating from the TOSCA standard describing the deployment of WordPress [233], a common content management system. To manage the cloud deployment using our OCCI orchestration process, we map both standards to perform an automated model transformation generating the required OCCI deployment model.

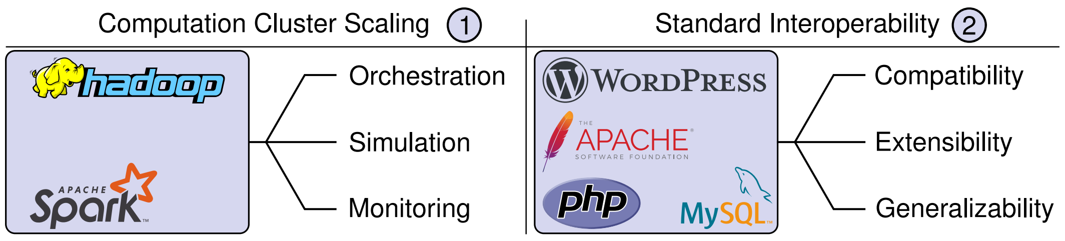


Figure 7.1: Overview: Orchestration case studies.

7.1 Case Study 1: Computation Cluster Scaling

The goal of this study is to demonstrate the feasibility of our orchestration concepts presented in Chapter 4. To reach this goal, we performed common orchestration tasks used to investigate the extent to which our orchestration, simulation and monitoring approach supports the management and development of cloud applications with OCCI. For this investigation, we modeled and orchestrated a Hadoop cluster as a representative example that represents a state-of-the-art framework for scalable, distributed computing. We split the orchestration procedures into three different scenarios that cover the utilization of our model-transformations and adaptation process (Section 4.1), the simulation environment (Section 6.1.2), and monitoring capabilities (Section 4.2). Within these scenarios, we showcase manual and automated interactions with the runtime model in order to highlight the benefits of an abstract reflection of the system. Based on the results of these scenarios, we qualitatively discuss the benefits of a model-driven orchestration process built around OCCI, the reflective capabilities of a runtime model, as well as provide quantitative insights about deployment times for the simulation and cloud environment.

Section 7.1.1 presents the artifacts required for the execution of the case study. Section 7.1.2 describes the orchestration process. Finally, in Section 7.1.3, we provide the results and observations of this case study as well as the impact of small model variations. While this section focuses on orchestrating the Hadoop computation cluster, Section 8.1 further investigates the capability to integrate the framework into a workflow execution.

7.1.1 Case Study Artifacts

To perform this case study, we developed several artifacts. Section 7.1.1.1 covers the OCCI deployment model. This model represents the primary artifact of the study and the basis for our scenarios. Section 7.1.1.2 introduces a subset of the Ansible configuration management scripts used to deploy and configure the Hadoop cluster with special regards to capabilities provided by the runtime model information.

7.1.1.1 Deployment Model

The Hadoop framework [231] consists of *NameNodes* and *DataNodes* which we abstract in our deployment model. A NameNode is responsible for management activities such as storing metadata of the *Hadoop Distributed File System* (HDFS) and administer the *Yet Another Resource Negotiator* (YARN) service. A DataNode stores the data to be processed in the distributed file system and thus represents a resource that should be monitored and scaled. Moreover, the framework can be extended with, e.g., Spark [232], that makes use of the HDFS and YARN to introduce streaming capabilities. To evaluate our orchestration process, simulation environment, and monitoring extension, we created the deployment model shown in Figure 7.2 which is composed of three increments. The individual increments serve as a basis to evaluate our approach within the different adaptation scenarios of this case study.

The **Hadoop Core** is the deployment model which we use to initially deploy the framework (Section 7.1.2.1). This model consists of three compute nodes, namely **Hadoop-master**, **Hadoop-worker-1** and **Hadoop-worker-2**. The **hMaster** and the two **hWorker** component instances are hosted on these compute nodes which we modeled via PlacementLinks. The **hMaster** component describes the lifecycle of a Hadoop NameNode and therefore requires information from the **hWorker** instances representing DataNodes. For this, we modeled ComponentLinks with ExecutionDependency mixins (**ed**) which indicate that the initial setup of the **hWorker** nodes need to be started prior to the **hMaster** node. All components are aggregated under one application node called **HadoopCluster** which unites the management of its components.

The **Spark** compartment enhances the **Hadoop Core** and is used to evaluate our simulation environment (Section 7.1.2.2). This increment of the model introduces an additional component called **spark**. We use this component to evaluate the feasibility of our local simulation environment by filling its attached configuration management script with directives to handle the lifecycle of Spark prior to an adaptation of the actual cloud deployment. For this, we placed the **spark** component on the **Hadoop-master** compute node. Additionally, we connected the **spark** component to the **hMaster** component over a ComponentLink with an attached InstallationDependency (**id**). We used this specialized dependency to indicate that the DataNode needs to be running before Spark should be installed and started by the orchestration process.

The **Monitoring** compartment introduces sensor elements and is used to evaluate our monitoring extension in the scope of a scaling scenario (Section 7.1.2.3). In this model extension, we added one **Sensor** for each worker node. Each **Sensor** application is composed

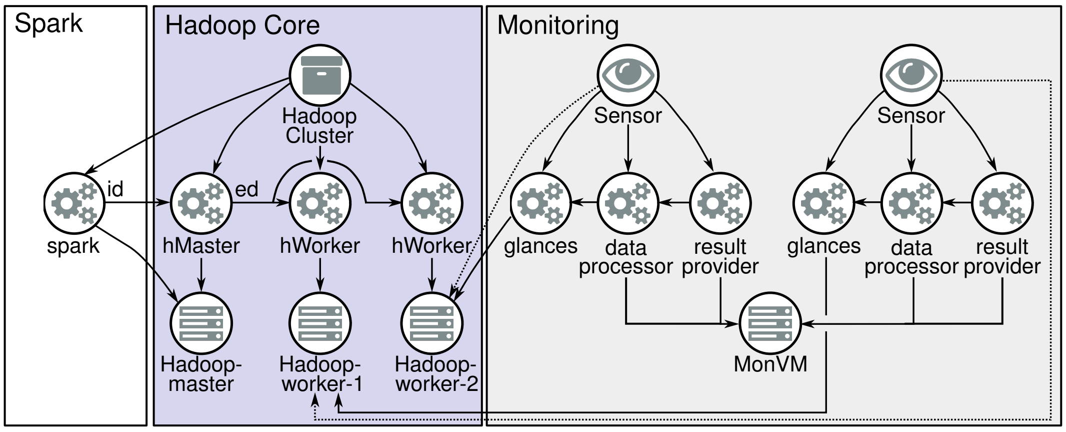


Figure 7.2: Hadoop cluster deployment model.

of three component nodes representing its monitoring instruments. The DataGatherer component is represented by **glances**. This component deploys the Glances [234] monitoring software on the worker nodes and provides an interface to request different performance metrics such as the current CPU or memory utilization. The gathered information is retrieved by the **dataprocessor** requesting and aggregating the CPU utilization of the worker nodes. Finally, the **resultprovider** publishes the gathered information to the runtime model showing whether the workload is, e.g., low or critical. To distribute the monitoring workload, the **dataprocessor** component as well as the **resultprovider** are hosted on the **MonVM**. This compute node is dedicated to host the monitoring pipeline and lift the monitoring workload from the worker nodes. To allow for a configuration of the individual monitoring instruments, we connected them via ComponentLinks. These connections ensure that the configuration management scripts have access to the attributes of the connected components. In the following, we go into further detail about the structure of the configuration management scripts describing the deployment procedures of the individual components.

7.1.1.2 Configuration Management Scripts

For each component in the OCCI model, a corresponding configuration management script exists that manages its lifecycle. This comprises the scripts to orchestrate the Hadoop cluster, Spark, and the monitoring instruments. Due to the similarity of these scripts, we provide an excerpt from two chosen configuration management scripts which represent their general structure. To explain how applications are deployed, we briefly introduce the hMaster component (Listing 7.1). Moreover, to exemplify the utilization of runtime model attributes via the generated variable file, we describe the dataprocessor instrument (Listing 7.2).

Listing 7.1 displays an artifact snippet from the Hadoop master configuration management script used to handle the component's lifecycle actions. The MoDMaCAO framework indicates that the individual lifecycle actions are performed within blocks. The **DEPLOY** task manages the deployment of the component (1-7). As the deployment process itself is unrelated to other components in the process, mainly package management statements are used, e.g., to upload and **unarchive** the Hadoop binaries. In the **CONFIGURE** block, the snippet shows the adjustment of the **core-site** configuration file (9-13). Additional configuration lifecycle tasks include the configuration of further files, as well as the setup of known hosts in the cluster. In the **START** block the start lifecycle action is described (15-19). Within this action, the HDFS and YARN services, required for the Hadoop master to run, are started. In this excerpt, the startup of the **namenode** is shown which manages data stored in the HDFS. For brevity, we omitted further blocks used to manage the stop and undeploy lifecycle actions.

Listing 7.2 highlights the utilization of attributes from the runtime model in configuration management scripts. Once a lifecycle action of a component is triggered the MoDMaCAO framework generates a variable file which contains the attributes of the component as well as the attributes of surrounding entities. The shown snippet depicts the **START** block of our DataProcessor monitoring instrument which starts a previously deployed **processor.sh**

Listing 7.1: Hadoop master configuration management code snippet (Ansible).

```
block:
- name: Upload hadoop
  unarchive:
    src: hadoop-2.9.2.tar.gz
    dest: /opt
  [...]
when: task == "DEPLOY"

block:
- name: edit core-site.xml configuration
  shell: echo [...] > /etc/hadoop/core-site.xml
  [...]
when: task == "CONFIGURE"

block:
- name: Start HDFS
  shell: yes "yes" | /sbin/hadoop-daemon.sh start namenode
  [...]
when: task == "START"
```

script using a **start-stop-daemon** (3). This script implements directives to query the CPU utilization currently monitored by the DataGatherer component. The queried monitoring information is aggregated and stored to be accessed by the ResultProvider. To run multiple scripts on the same host, it has to be uniquely identified. For this, we query the variable file generated from the runtime model while filtering the **id** of the DataProcessor instance (4). As a second argument, the script requires the information about the *Internet Protocol* (IP) address of the compute node hosting the glances DataGatherer. This information is similarly retrieved. First, the **links** from the DataProcessor are queried (5). These links are filtered for **componentlink** instances with their **target** resource being of interest (6). For each target that is a **datagatherer** we extract the **ipaddresses** from the queried results and pass it to the data processing script (7).

Listing 7.2: DataProcessor code snippet showing the start action (Ansible).

```
1 block:
2 - name: Execute processor script
3   command: start-stop-daemon processor.sh
4   {{ id }}
5   {{ links |
6   json_query('[?kind == `componentlink`].target') |
7   json_query('[?kind == `datagatherer`].ipaddresses[] | [0]') }}
8   [...]
9 when: task == "START"
```

7.1.2 Orchestration Process

In this section, the execution of the initial Hadoop cluster deployment is described as well as additional simulation and scaling scenarios built around it. Section 7.1.2.1 covers the initial deployment of the Hadoop cluster on our private OpenStack cloud. Section 7.1.2.2 describes the utilization of the simulation environment to extend the Hadoop cluster with additional capabilities offered by Spark solely relying on local resources. Finally, Section 7.1.2.3 presents the addition of monitoring capabilities and highlights the interaction of the runtime model with an additional scaling engine.

7.1.2.1 Initial Cloud Deployment

We use the initial deployment scenario to demonstrate the feasibility of our model-driven and OCCI based orchestration process (Section 4.1). Before the deployment, we adjust the Hadoop cluster model using our automated model transformations as shown in Figure 7.3. The Hadoop core model itself represents a **CIM**, as compute related information is still missing. The transformation to the **PIM** enhanced each compute node with mixins so that they utilize our default SSH **key** and initialization **script**. In the **PSM**, further provider specific compute mixins are added to use Ubuntu 18.04 with python pre-installed in a medium sized VM having two virtual CPUs and four gigabyte memory. Also, the **ManagementNetwork** is added to the model including NetworkInterfaces (**nwi**) to each compute node.

To describe the orchestration process, we investigate intermediate runtime model states reached throughout the execution of the adaptation engine. For the initial deployment, we empty the runtime model so that no resources are currently provisioned in the cloud.

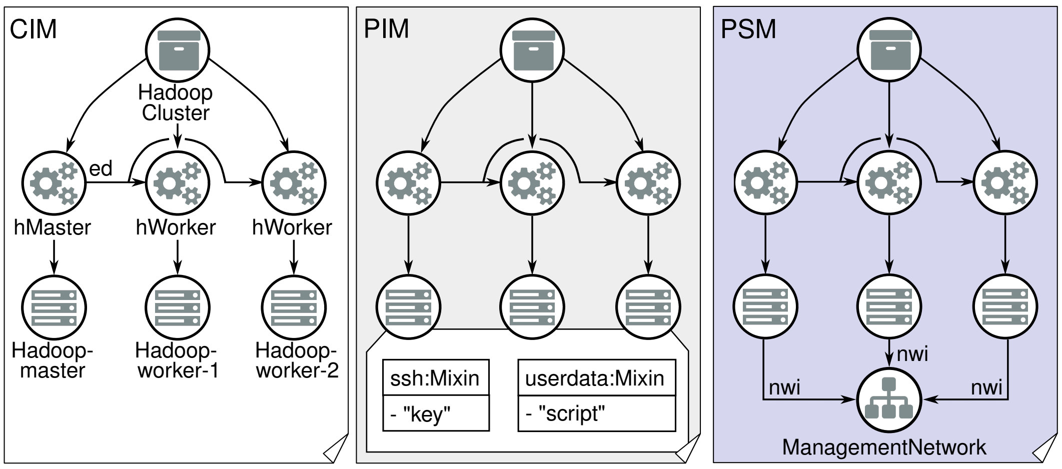
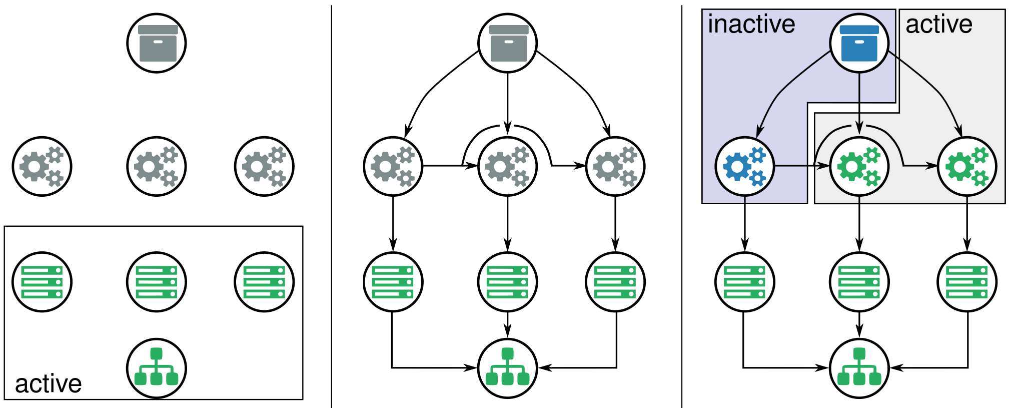


Figure 7.3: Design time Hadoop cluster model with applied model transformations.

Figure 7.4 (a) shows the first runtime model state in which all resource entities are requested. This covers the requests to create the compute, component, network and application nodes. In case of the infrastructure resources the requests are directly forwarded to our OpenStack cloud. Once the startup of the VM and network has finished, they are transferred to state **active** allowing the orchestration process to continue.

Figure 7.4 (b) depicts the time step in which each link is provisioned in the runtime model. As the platform elements are first provisioned at the end of the orchestration process, they are already interlinked with the ComponentLinks during the startup of the VMs. Once the compute nodes are active, the PlacementLinks are requested connecting each component to its host. Simultaneously, the connections from the compute nodes to the management network are established. At this point in time, the runtime model contains all entities of the design time model which leads to the next phase.

Figure 7.4 (c) shows the runtime state during the execution of the start action on the application node. During each execution of a lifecycle action, the variable file for the individual component is generated providing up-to-date information about the element's environment and configuration. The figure highlights the orchestration process at the time in which the components transition from an **inactive** to an **active** state. Hereby, the Hadoop master component is currently still **inactive** with the start action being currently processed. Once the lifecycle action has finished processing, the master component reaches the **active** state. This leads to the application node also transitioning to the state **active**. Finally, the deployment process is finished, and the runtime model is compared to the design time model. As both models match each other the Hadoop cluster is successfully deployed.



(a) Resource provisioning. (b) Link provisioning. (c) Application deployment.

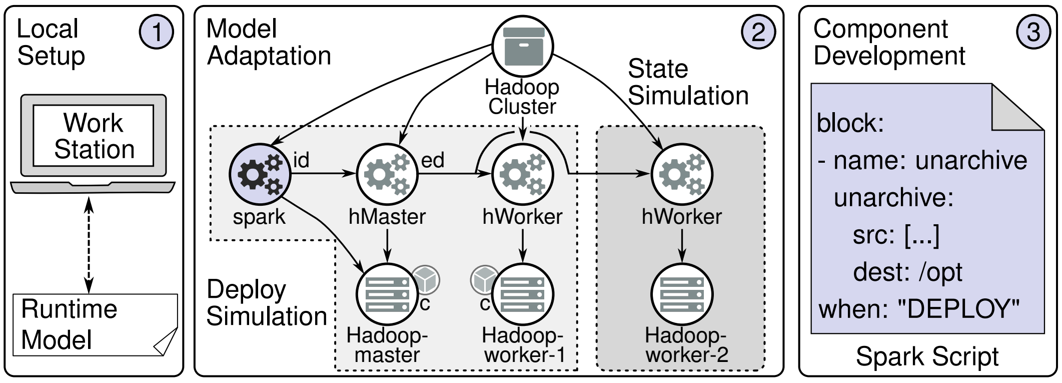
Figure 7.4: Runtime states of the initial Hadoop cluster deployment.

7.1.2.2 Cluster Simulation Environment

In this scenario of the study, we highlight the causal connection of the runtime model (Section 4.2). We use this connection to evaluate the extent to which our simulation environment (Section 6.1.2) can be used to develop and plan adaptive changes prior to a deployment in the cloud. For this, we add additional capabilities to the Hadoop model to include Apache Spark [232]. As shown in Figure 7.5, we separate the scenario in three steps. The **Local Setup** ① of the simulation environment, the **Model Adaptation** ②, and the **Component Development** ③. In this scenario, individual steps of the process are showcased in isolation to highlight particular aspects.

For the **Local Setup** ①, we spawned an instance of SmartWYRM (Section 6.2) and the OCCIWare Runtime system [123] on our **Workstation** running a i7-7600U CPU2.80GHz with 16 GB of memory. Using this environment, we simulate the productive cloud environment by replicating its runtime model in order to test and assess the impact of planned changes locally. To utilize the benefits of the simulation environment, we first extracted the runtime model currently connected to the cloud. Thereafter, we triggered an automated transformation on the model which adds a simulation mixin of the appropriate type to each entity. This covers a compute simulation mixin added to each compute node and a component simulation mixin added to each component.

For the **Model Adaptation** ② we utilize the default configuration of the simulation level which indicates that the effector only performs state changes. Thereafter, we deployed the annotated model in our local environment using the orchestration engine. In this simulation, no virtualized computing resources got provisioned. This allows us to operate on a model-based level with next to no resource consumption in which each entity only reflects its state. To integrate the **spark** component into the Hadoop cluster model, we connected it to the application node and placed it on the **Hadoop-master** compute node. In preparation for the development of the **spark** configuration management script, we assume that a partial



```
block:
- name: unarchive
  unarchive:
    src: [...]
    dest: /opt
when: "DEPLOY"
```

Figure 7.5: Local deployment simulation setup and runtime model.

deployment of the **HadoopCluster** application is sufficient. For this, we reduce the amount of required local resources by adjusting the simulation mixins to only perform a **State Simulation** for **Hadoop-worker-2**. Additionally, we set up a **Deploy Simulation** for the **Hadoop-master** and **Hadoop-worker-1** compute nodes which get simulated as containers (**c**). To ensure a correct provisioning of the resources, we cleared the runtime model residing in our workstation and performed a fresh deployment with the adjusted simulation levels. This deployment results in containers being spawned in our workstation for the **Hadoop-master** and **Hadoop-worker-1** which simulate the behavior of VMs in the cloud. To increase the validity of the simulation, we utilized a container image which replicates the operating system used in the VMs of the cloud. While we initially performed these changes in the runtime model, we exported the adjusted model for an automated orchestration of the cloud deployment.

The **Component Development** ③ describes the implementation process of the configuration management script attached to the **spark** component. After the provisioning of the infrastructure, the platform elements are deployed with the **Spark Script** still being empty. This allowed us to develop the corresponding configuration management script with a simulated state of the Hadoop cluster running in the cloud. During the development of the script, we manually altered attributes within the runtime model and triggered individual lifecycle actions of the component, e.g., **DEPLOY**, to exploit the direct connection between the runtime model and configuration management script. After finishing the development of the additional component, we replicate the adaptation scenario to be performed on the cloud environment. For this, we used the orchestration process to first deploy the initial Hadoop cluster model followed by triggering the orchestration process for the adjusted model containing the additional **spark** component. After performing the local adaptation, we removed the simulation tags from the model and applied it on the cloud runtime model. Due to the utilization of the same ids in both environments, only the **spark** component is added to the cloud runtime model introducing its functionality.

7.1.2.3 Cluster Monitoring and Scaling Scenario

In this section, we further extend the deployed Hadoop cluster model with monitoring capabilities. We use this scenario to highlight the self-reflective capabilities of the runtime model. Hereby, we investigate the extent to which the monitoring information can be utilized by additional adaptation engines. Moreover, we evaluate the capability of the runtime model to reflect artificial workload chosen by the developer to manipulate and test decision-making and scaling processes. As shown in Figure 7.6, we attached a **Scaling Engine** that utilizes the self-reflective capabilities of our **Runtime Model** for decision-making purposes.

The **Scaling Engine** is a MAPE-K loop which can be started next to the runtime model. This engine monitors and scales the the Hadoop worker nodes in our deployment and was specifically developed for this scenario. In the **Monitor** phase a **sensor query** requests the information of all CPU related information contained in MonitorableProperty links. In the case of Figure 7.6 the current workload of **Hadoop-worker-2** is **"Critical"**. In the **Analyze** step, the gathered information is used to check whether the deployed cluster needs to be scaled. The engine differentiates between a *scale down* and a *scale up* procedure. Scale down is chosen if no compute node has a critical workload with one having no workload at all. The scale up procedure is called when the majority of compute nodes have a critical workload. In Figure 7.6, a **"Critical"** workload is detected leading to the decision that a **scale up** procedure has to be performed. This decision in then passed to the **Plan** phase which, depending on the chosen scenario, builds a new desired state for the runtime model. In the scale up scenario a new worker node is added comprising a compute resource and a Hadoop worker component. Additionally, a sensor is added which monitors the CPU utilization of the new worker node. The added elements resemble the structure of the **Runtime Model** shown in Figure 7.6. In the scale down scenario, the machine currently having a "None" CPU utilization is removed from the model including its Hadoop worker component and sensor. Independent of the scaling scenario chosen, the resulting model is transformed to add platform specific elements such as the management network. Finally, the resulting model is passed to the **Execute** phase which triggers the **Adaptation Engine** with the planned model serving as input. The engine then extracts and compares the runtime model to the new desired state and performs the adaptive actions over requests sent to the **OCCI** interface.

To develop the scaling engine, we used the simulation environment to assess the impact of our scaling scenarios. For this, we adjusted the **Sensor** simulation mixin in such a manner that it alternates between different artificial monitoring information that we predefined in the model. Especially changes to the values "Critical" and "None" were of interest as these trigger the scaling procedures. Furthermore, we utilized the added simulation capabilities to adjust the observed monitoring results manually. This forced specific scaling procedures to be triggered. For example, to investigate the upscale behavior we configured the sensor to only monitor critical workload. Additionally, to test the scaling in the actual cloud we modeled example Hadoop jobs to stress the provisioned compute nodes.

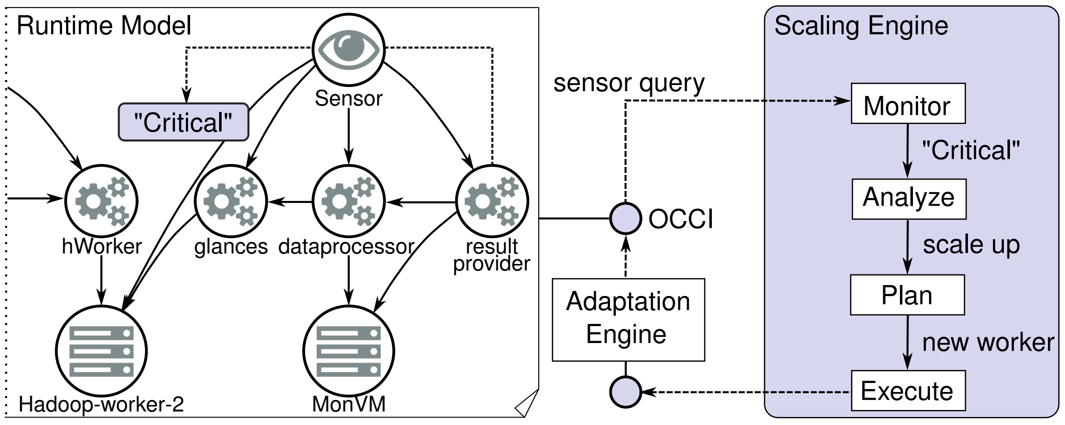


Figure 7.6: Hadoop model excerpt with attached scaling scenario.

7.1.3 Results and Observations

Within this case study, we demonstrate the initial deployment, as well as the adaptation of a scalable computation framework in a cloud and a simulation environment. We show that the OCCI standard is suitable to provide adaptive capabilities by demonstrating how a OCCI based runtime model can serve as a knowledge base in a self-adaptive control loop. Furthermore, the study demonstrates that this runtime model can be used to not only reflect structural but also operational parameters by modeling sensors with runtime model access. In the following, we describe our observations made about the modeling, execution and simulation process at design and runtime. Moreover, we briefly highlight the impact of small model and scenario variations and conclude this study with a summary.

7.1.3.1 Initial Deployment

The initial deployment scenario shows the feasibility of our orchestration process and highlights our transformation and deployment process.

The model-transformations infuse our Hadoop model with provider specific information required for an actual provisioning in the cloud. This allows focusing on the structure and behavior of the model. By introducing the transformation process, we save time throughout the development of the model as less information has to be added to the model in the first place. However, most values to be added need to be preconfigured and attuned to the provider covering, e.g., specific images and VM sizes or the authentication keys to be used.

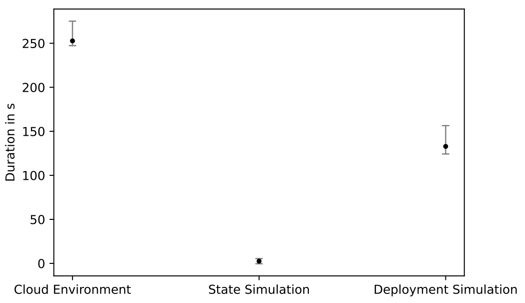
The orchestration procedure derives the dependencies between the individual OCCI entities to ensure an automatic deployment of the Hadoop cluster model. Hereby, resources are requested first, followed by their links (see Figure 7.4). This comes due to the fact, that both resources need to be present in the runtime model, before they can be connected. Furthermore, depending on the kind of resource, a different state needs to be reached before they can be connected. For example, in case of the connection of a compute node to a network resource, the compute node needs to be active. For this, we directly start the compute node within the initial resource request. During the deployment of the individual Hadoop cluster components, the generated variable file, to be used by the configuration management scripts, provides access to runtime information. Among others, this allows us, e.g., to utilize IP addresses that are dynamically assigned by the management network to establish a communication between the master and worker nodes. From a timing perspective, our observations show that requesting OCCI entities without cloud behavior only takes a few milliseconds. For example, in five successive executions of the deployment scenario, the PlacementLink requests took 14 milliseconds in the mean and ComponentLink requests took 33 milliseconds in the mean. On the other hand, the provisioning of infrastructure resources, like a VM, took 59 seconds in the mean with the deployment of the application taking up to three minutes. Therefore, the model-driven derivation of cloud management requests produces a negligible overhead when compared to the time required to spawn the infrastructure and deploy the application.

To diversify the initial deployment study, we additional test different kinds of VMs images and topologies including the utilization of a dedicated network for the communication of nodes in the cluster. For this, we add a network node as well as network interfaces to the individual compute nodes to the Hadoop core model. As a result, each VM spawns with a predefined IP address. We observe that this variation allows for a static development of configuration management scripts due to the utilization of hard-coded address ranges. In comparison to the utilization of dynamic addresses that we derive from the runtime model, these hard-coded scripts are only suitable for testing purposes due to the constraint of requiring predefined addresses.

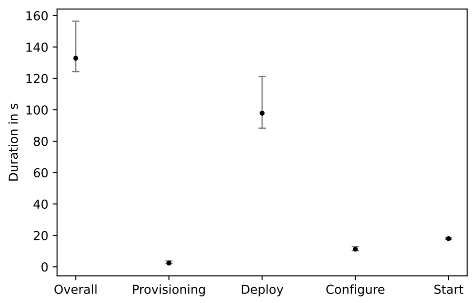
7.1.3.2 Simulation Environment

In this scenario we evaluate the extent to which our simulation environment can be used to assess and test the impact of planed adaptive changes. For this, we use a state simulation to extend the model itself and a deployment simulation for the development of the Spark configuration management script. To substantiate our observation with quantitative data, we executed the study five times per used environment with the 95% confidence intervals shown in Figure 7.7 (a).

The **State Simulation** setup extracts and deploys the state of the cloud runtime model on our local workstation only taking 2 seconds in the mean. As the runtime model in this environment is not causally connected to an actual system, we are able to develop adaptive changes directly in the runtime model using small iteration cycles. Moreover, due to the low time requirements, the state simulation environment proves to be useful to define and execute automated tests for our orchestration, workflow and scaling engine within a CI pipeline. Still, in this setup, no configuration management scripts are executed which limits the assessment of planned changes to the behavior of the runtime model.



(a) Deployment duration per environment.



(b) Detailed Deployment Simulation.

Figure 7.7: 95% confidence intervals of deployment duration per environment.

The **Deployment Simulation** environment can be used to mitigate the limitations of the state simulation as configuration management scripts are actually executed. During the development of our spark configuration management script, the runtime model allows us to manually trigger single actions on the deployment. Hereby, the impact of the action is directly reflected in the runtime model visualization notifying us, e.g., on failing deployment actions with the error message updating appropriately. Compared to the 253 seconds deployment time required in the **Cloud Environment**, the utilization of the **Deployment Simulation** takes only 133 seconds in the mean. As shown in Figure 7.7 (b), the simulation reduces the **Provisioning** time from roughly a minute in the cloud to a few seconds in our local environment. In both settings the individual **Deploy**, **Configure**, and **Start** action require roughly the same amount of time. However, the **Deployment Simulation** allows reducing the time required for the application deployment by a third, as we configure the simulation to partially deploy the model covering only a single master and worker. Overall, the deployment times, as well as the utilization of a local workstation as computation resource, allow for smaller iteration cycles without the need to be connected to a cloud. Still, the utilization of different virtualization technologies as substitute in the simulation pose some constraints that have to be considered, e.g., in case of Docker the access to specific configuration files is restricted. The **Provisioning** step in the **Deployment Simulation** takes only a few seconds rather than minute to spawn the infrastructure due to the utilization of containers to simulate VM behavior.

7.1.3.3 Monitoring and Scaling Capabilities

To highlight the reflective capabilities of our OCCI extension, we add sensors to the Hadoop cluster model and attach a scaling engine that utilizes the monitoring information. We separate our results into observations made while extending the model with monitoring capabilities and observations made about the reflective capabilities of the runtime model during the development of the self-adaptive control loop.

The process of extending the Hadoop cluster model with monitoring capabilities resembles the addition of the Spark component due to the inheritance of the MoDMaCAO framework. Due to the manual configuration of the monitoring pipeline, the distribution and expressiveness of the reflected values depends on the aggregation process modeled by the user. Thus, monitoring workload can be distributed across multiple machines with the ability to push arbitrary monitoring properties to the runtime model. In its current state, the runtime model is only capable of reflecting the "as is" state. As a result, historical information needs to be collected and stored using an additional engine.

By reflecting monitoring information in the runtime model, the scaling engine is able to monitor the structural and operational information of the deployment using simple OCCI queries. While formerly only the information about the amount of resources and their connections were available, our monitoring extension provides access to monitored deployment specifics such as the cluster's workload. This ensures that all relevant information is available in the monitored runtime model to decide when and how to scale the cluster. As we reuse the orchestration engine for our scaling engine, no development effort is required to build the execution phase. Therefore, the standard allows focusing on domain specific behavior rather than re-implementing already existing approaches. In the case of our scenario, we are able to focus on the scaling engine's analyze and plan step. Especially in these steps the simulation environment supports the development of adaptive behavior, as specific scaling actions can be forced by injecting specific monitoring information. At runtime, we observe that several subsequent scale up actions quickly clutter the runtime model visualization. Per scale up six new resources are added from which four are used for monitoring capabilities. Once more than two worker nodes are deployed, the amount of monitoring entities make up most of the resources in the runtime model even though they are automatically generated. This hints at a trade-off between the readability of the runtime model for a user and the fine-grained representation of the system with detailed monitoring information.

In a second variation of the scaling engine, we utilize a simple Bash script that scales the individual nodes in the cluster vertically. In this scenario, the engine directly queries the monitoring results of a single node over the OCCI interface to adjust its amount of virtual CPU. During this scenario we learn that managing attributes via simple scripts are sufficient, but may be limited for multiple resources as request dependencies need to be manually resolved. This highlights the usefulness of our developed approach as it allows focusing on desired states rather than adaptive actions.

7.1.3.4 Summary

In the following, we summarize our results of the performed use case.

Summary: Our model-driven approach allows to automatically orchestrate cloud deployments using the OCCI standard. At design time, model-transformations reduce the amount of information to be modeled by automatically adding provider specific information which, however, must be previously defined by a user. At runtime, the deployment is reflected in the runtime model providing an abstract view on the system. By connecting the runtime model to different environments, cloud deployments can be simulated on different levels of detail. These support the change assessment and development of IaC artifacts and automated test cases by reducing deployment times and running outside the cloud. Reflecting operational parameters in the runtime model fosters decision-making processes which are required, e.g., by scaling engines. By introducing monitoring elements to OCCI, sensors can be distributed and individually managed to mitigate potential monitoring overheads. Still, only the "as-is" state can be reflected with no historic information.

7.2 Case Study 2: Standard Interoperability

Similar to the first study, the goal of this study is to demonstrate the feasibility of our orchestration concepts presented in Chapter 4. In this study, we especially emphasize the generalizability of OCCI, and thus our orchestration approach, by managing TOSCA models via the OCCI interface. Moreover, we use this study to discuss the mapping of TOSCA to OCCI (Section 4.1.2) with special regards to the orchestration process presented in this thesis. Hereby, we highlight the conceptual differences of both standards to distinguish their individual advantages and drawbacks. As TOSCA provides two kinds of specification, we choose to focus on the TOSCA YAML version [108] as it is more widely adopted by the community and more actively maintained.

In the scope of our studies, we transformed and deployed several TOSCA topologies of various sizes [217]. In this thesis, we present the transformation and deployment of the TOSCA WordPress topology. We choose this topology, as WordPress [233] is one of the most common content management systems. Therefore, it represents a common use case for cloud deployments. Moreover, a complete WordPress stack is available within the TOSCA YAML specification [108]. This allows us to extract a precise TOSCA topology to determine the interoperability of OCCI and TOSCA. Finally, the topology is suitable to demonstrate the feasibility of the transformation and orchestration process while maintaining a concise description of the study.

The remainder of this section is structured as follows. Section 7.2.1 introduces the artifacts implemented for the execution of the case study, covering the example TOSCA topology. Section 7.2.2 describes the orchestration and transformation process. Finally, Section 7.2.3 discusses observations made at design and runtime.

7.2.1 Case Study Artifacts

To mitigate the threat of creating a model that perfectly fits our designed transformation, we reutilize an existing model of the WordPress example as well as accompanied deployment scripts from the Alien4Cloud Project [141].

Similar to OCCI, TOSCA is built around a type-instance pattern (see Section 2.3.3.3). In general, TOSCA separates this pattern using the term type, e.g., for node types, and the term template, e.g., node template, for the instance layer. Together, node templates and relationship templates fill out the TOSCA topology, with each template relating to a specific type describing the properties to instantiate.

The artifacts specifying the types and templates are described via several YAML files. Section 7.2.1.1 introduces the TOSCA WordPress types, and Section 7.2.1.2, the TOSCA template.

7.2.1.1 TOSCA Types

Listing 7.3 shows the **node_types** definition of **tosca.nodes.Wordpress** which is a custom type that is **derived_from** the **tosca.nodes.WebApplication** normative type (1-3). In addition to a **description**, the type specifies two **properties** and one **attributes** type (4-9). For example, the **context_root** describes the folder path to be used by the wordpress application. Additionally, **requirements** are specified that need to be fulfilled by node templates instantiating it (10-21). The **host** requirement is a **HostedOn** relationship describing where the node template is deployed. Also, a **database** connection needs to be specified, as well as a **php** component to which the wordpress application **ConnectsTo**. Finally, the type introduces several **interfaces** (22). These describe actions that can be applied on wordpress node templates for which individual scripts are assigned.

7.2.1.2 TOSCA Template

Listing 7.4 shows the YAML TOSCA template modeling the WordPress stack. In this excerpt the **node_templates** (1) section of the topology is highlighted covering six nodes. The **wordpress** template is of the previously discussed **type: tosca.nodes.Wordpress** (2-11). Here, the individual **requirements** specified by the type are instantiated. As **host** the **apache** node template is chosen, the **database** requirement is fulfilled by the **mysql** node and the **php** requirement by the **php** node. To deploy all these requirements, the topology contains two

Listing 7.3: TOSCA WordPress node type definition excerpt.

```
1  node_types:
2    tosca.nodes.Wordpress:
3      derived_from: tosca.nodes.WebApplication
4      description: The TOSCA Wordpress Node Type represents[...]
5      properties:
6        zip_url: #omitted for brevity
7        context_root: #omitted for brevity
8      attributes:
9        max_user_number: #omitted for brevity
10     requirements:
11       - host:
12           capability: tosca.capabilities.Container
13           relationship: tosca.relationships.HostedOn
14       - database:
15           capability: tosca.capabilities.MysqlDatabaseEndpoint
16           relationship: tosca.capabilities.Endpoint.Database
17           occurrences: [1, 1]
18       - php:
19           capability: tosca.capabilities.Root
20           relationship: tosca.relationships.ConnectsTo
21           occurrences: [1, 1]
22     interfaces: #omitted for brevity
```

node templates of type **tosca.nodes.Compute** named **computeWww** and **computeDb** (13-18). Both node templates represent VMs to be provisioned in the cloud and define **capabilities** covering OS and sizes to be used. We omitted this information from the listing for brevity. The **apache** node template provides information about the **port** and **document_root** to be used for the component (20-26). Both the **apache** and the **php** node templates are hosted on the **computeWww** node template (27-30). Finally, to fulfill the database requirement, the **mysql** template is modeled and deployed on the **computeDb** (31-34).

Listing 7.4: TOSCA topology excerpt of defined node templates.

```
1  node_templates:
2      wordpress:
3        type: tosca.nodes.Wordpress
4        requirements:
5          - host: apache
6          - database:
7              node: mysql
8              capability: tosca.capabilities.Endpoint.Database
9          - php:
10             node: php
11             capability: tosca.capabilities.Root

13     computeWww:
14       type: tosca.nodes.Compute
15       capabilities: #omitted for brevity
16     computeDb:
17       type: tosca.nodes.Compute
18       capabilities: #omitted for brevity

20     apache:
21       type: tosca.nodes.Apache
22       properties: #omitted for brevity
23         port: 80
24         document_root: "/var/www"
25       requirements:
26         - host: computeWww
27     php:
28       type: tosca.nodes.PHP
29       requirements:
30         - host: computeWww
31     mysql:
32       type: tosca.nodes.Mysql
33       requirements:
34         - host: computeDb
```

7.2.2 Orchestration Process

In this section, the execution of the transformation from the WordPress TOSCA topology to the OCCI artifacts is described. Additionally, we describe the orchestration process of the transformed model to discuss the interoperability and generalizability of our orchestration process. Section 7.2.2.1 describes the OCCI extension and model generated from the TOSCA artifacts. Section 7.2.2.2 presents the deployment of the generated OCCI model.

7.2.2.1 Generated OCCI Model

In this section, the automatically transformed OCCI model is described that serves as input for the orchestration process. The model itself is shown in Figure 7.8 highlighting the **OCCI Deployment Model**, as well as exemplifying the transformed **OCCI Extensions for TOSCA** in form of a UML class diagram.

We generated two **OCCI Extensions for TOSCA** to instantiate an OCCI model that complies to the information of the original TOSCA topology. These comprise the **TOSCA custom** extension, covering user defined types and the **TOSCA normative** extension introducing common types specified by the standard. We visualize the **wordpress:Mixin** as an example for a **TOSCA custom** type. This type consists of the attributes **zip.url** and **context.root** which comply to the properties defined in the corresponding TOSCA node type (see Listing 7.3). Similar to most custom mixins, this mixin depends on the **webserver:Mixin**. This mixin is part of the **TOSCA normative** extension and **depends** on the **component:Mixin** of the **MoDMaCAO** extension. This dependency is created as the **webserver:Mixin** and its specialization represent components to be deployed and therefore need a configuration management script attached. The same relation to the OCCI component kind can be found for the **php**, **apache**, and **mysql** resources. The mixins generated for the infrastructure layer, i.e., for both compute nodes, can be applied to the compute kind of the OCCI infrastructure extension.

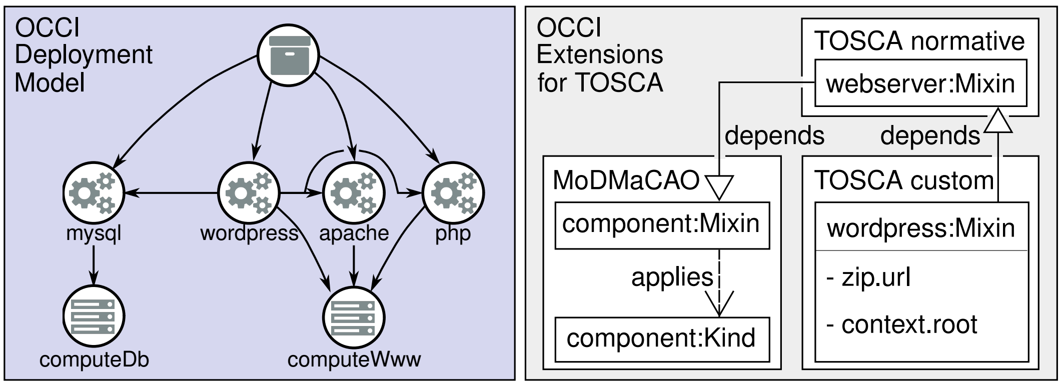


Figure 7.8: OCCI model transformed from the TOSCA WordPress topology.

The **OCCI Deployment Model** is generated from the TOSCA topology template. The infrastructure is represented by the compute nodes **computeDb**, **computeWww**. The application layer is represented by the components **mysql**, **wordpress**, **apache**, **php** which are aggregated under a generic application node. Each of these resources has a mixin instance attached which corresponds to the **TOSCA custom** node types. In case of the **wordpress** component, e.g., an instance of the **wordpress:Mixin** is attached with the accompanying **zip.url** and **context.root** attributes. The specific attribute values of the individual resources in Figure 7.8 are omitted for brevity. To ensure the replicability, we reuse the configuration management scripts provided by the Alien4Cloud project to manage each individual component. In addition to resources, several links are generated during the transformation. The **wordpress** component, is hosted on an **apache** web server and is connected to a **mysql** database and a **php** software component. These links utilize the kinds **tosca.relationships.hostedon** and **connectsto** mixin being part of the generated TOSCA normative extension. Furthermore, each of the TOSCA host requirements from the original topology were transformed to a PlacementLink connecting the component to the compute node it is deployed on. Before the orchestration is performed, a PIM to PSM transformation is executed on the OCCI model to fulfill the requirements given by the OCCI MoDMaCAO framework.

7.2.2.2 Provisioning and Deployment Procedure

In the first step, the orchestration process inspects the currently running cloud deployment, which we emptied for this use case. Therefore, a provisioning plan is generated that includes requests to create all elements being part of the model. The requests are sequenced by creating the resource elements first, i.e., the **computeDB**, **computeWww**, **mysql**, **wordpress**, **apache**, **php** and the general application resource. While the effector of the compute nodes directly performs a request to the cloud environment spawning up the modeled VM, the component nodes reside in a undeployed state. Once the VMs have reached an active state, the orchestration process continues by sending requests to create the links interconnecting the resources in the runtime model. In case of the given model, this includes the connection of the VMs to the management network, the creation of PlacementLinks describing where each component is hosted, as well as ComponentLinks interconnecting the individual components. After validating the infrastructure, the application deployment process is taking place starting the deploy, configure and start action on the generated application node. During this process the information modeled within the added TOSCA mixins is taken into consideration and passed to the configuration management script in order to be used. After the request has been sent, the effector identifies the correct order of lifecycle actions to be triggered among the components by utilizing the information modeled within the ComponentLinks and mixins. In case of this deployment model, first the **php** and **apache** component are deployed, followed by the **mysql** database and finishing with the **wordpress** component. The orchestration of the configuration and start actions follow the same sequence. After each component has reached the active state, the application is also set to active which closes the deployment procedure.

7.2.3 Results and Observations

This case study demonstrates the cooperation between the two cloud standards TOSCA and OCCI while utilizing the orchestration engine and effectors presented in this thesis. We show that the OCCI standard is feasible to serve as an interface that may be used to deploy TOSCA topologies using a uniform and standardized interface. In the following, we describe observations about the compatibility of TOSCA and OCCI at design and runtime.

7.2.3.1 Design Time

In general, a standard-driven approach to model cloud deployments is quite advantageous. This is, because the metamodel or language is designed by experts which agree about key aspects in the cloud domain. Thus, the standard presents a sophisticated abstraction of the cloud domain which is at least partially accepted by the community. In case of this study, we demonstrate that TOSCA can be transformed to OCCI in the scope of our model-driven environment. Even though both standards possess similarities, they differ in their focus and therefore cannot be mapped one to one. TOSCA provides concepts to model on a higher abstraction level providing elements which, e.g., allow grouping different elements and scale them. OCCI on the other hand focuses with its uniform interface on a runtime perspective, introducing elements such as mixins that provide the capability to dynamically add behavior. Therefore, elements such as TOSCA scaling groups, as well as OCCI applications, do not possess equivalents in the respective standards.

Overall, the transformation of TOSCA to OCCI transfers all information that refers to actual cloud resources by generating corresponding mixins. However, not all information can be directly utilized for an actual deployment over the OCCI interface. The elements in the platform layer, i.e., components, can utilize the transformed information within the mixins in the scope of the variable file generated for the configuration management scripts. However, some information at the infrastructure layer require a more concise mapping. Contrary to OCCI, TOSCA allows compute templates to define a default IP address without requiring a network to be defined. To correctly transform this representation, an abstract network is required in OCCI, which, e.g., represents a provider network. This, however, represents a resource that a user is not allowed to manage which contradicts the purpose of OCCI. While this hints at required adjustments to optimize the mapping in some use cases, the translated information of the TOSCA to OCCI transformation is sufficient to deploy a cloud application using our orchestration process.

7.2.3.2 Runtime

While the similarities and differences of the standardized languages can be identified by comparing their specifications, the orchestration process is used to compare the implications of managing the modeled resources at runtime. While multiple approaches exist that interpret TOSCA topologies and orchestrate their deployment, our approach focuses on the orchestration of such topologies over the uniform and standardized interface provided by OCCI. Overall, our observations about the management of the resources can be divided into the logic of the orchestration engine and the logic implemented in the effector managing the resources of a specific kind. The logic within the orchestration engine does not have to be adjusted, as each transformed TOSCA resource corresponds to a kind specified within the OCCI standard, i.e., compute, component, or application. The utilized model transformation solely generates TOSCA related mixins that can be attached to these kinds. As a result, no new effectors need to be generated. Still, the specific behavior provided as capabilities by the attached mixins must be implemented by the corresponding OCCI effector. It should be noted, that in case of our case study no specialized implementations of our effectors are necessary. This is because the default variable generation provides the configuration management scripts with all attributes of attached mixins.

The configuration management scripts attached to the individual components play a major role during the deployment of the topology. In case of this study, we utilize the scripts provided by the Alien4Cloud project [141]. In the TOSCA topology these scripts are directly attached to the type definition among the interfaces of a specific type. In case of OCCI, we map these scripts to the individual MoDMaCAO actions, i.e., deploy, configure and start, and store them within our runtime environment. In general, the design-time attachment of the scripts to the TOSCA type definition makes them more portable. However, the utilization of the scripts within the OCCI environment allows making use of runtime information. In case of this deployment, the runtime information comprises, e.g., the IP addresses and ports of the MySQL database. During the deployment of the wordpress component, this up-to-date information is retrieved from the runtime model and passed over the variable file generation to the configuration management script.

7.2.3.3 Summary

In the following, we summarize our results of the performed use case.

> **Summary**: The TOSCA and OCCI standards both offer different advantages and can be combined using a set of model transformations. Thus, our orchestration process can be used to deploy and manage OCCI models that originate from the TOSCA standard. Still, for design time aspects, TOSCA has a more sophisticated metamodel. Therefore, elements that do not represent direct cloud resources, such as scaling capabilities, can not be directly mapped to OCCI, while elements such as OCCI applications do not have a direct TOSCA correspondent. Compared to TOSCA, OCCI benefits from a simple core model with the advantage of being accompanied by a uniform interface to abstract and manage modeled cloud resources.

8 Runtime Workflow Model Case Studies

In this chapter, we demonstrate the applicability of the runtime workflow model concept presented in Chapter 5. For this we designed, modeled and reflected scientific workflows from multiple domains using the notation and configuration introduced in Chapter 6. Due to the similar composition of our orchestration and workflows studies, we discuss the threats to validity in Section 9.4. As shown in Figure 8.1, this chapter comprises three case studies that highlight individual capabilities, advantages and drawbacks offered by our workflow runtime model. While focusing on different aspects of our concept, each of these case studies possess the same structure. First we identify the need for shifting resource requirements within the scientific domain for which we provide a brief introduction. Based on the gathered insights, we create a corresponding workflow model and investigate how it operates at runtime by visualizing reached runtime states. At the end of each case study we conclude our design and runtime observations and discuss potential timing overheads.

Section 8.1 covers the **Big Data Framework** ① case study which focuses on a **Task** and **DataLink** communication to perform a **Map Reduce** job hosted on distributed machines.

Section 8.2 covers a **Dynamic Simulation** ② scenario that highlights the **Decision** and **Sensor** functionality within a **Multi-Level Simulation** application which requires shifting amounts of computing resources. Moreover, this case demonstrates the utilization of specialized infrastructure resources within the workflow.

Section 8.3 presents our **Repository Mining** ③ case study which demonstrates the utilization of **Loop** instances and their **Parallelization** within a software repository mining workflow built around the **SmartSHARK** framework.

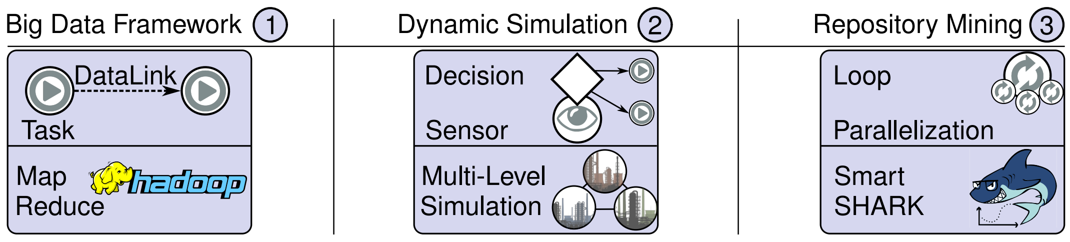


Figure 8.1: Overview: Workflow case studies.

8.1 Case Study 1: On-Demand Big Data Framework

The ability to gather large amounts of data has led to many methodologies to efficiently analyze them using distributed resources. As already described in Section 7.1, Apache Hadoop [231] is a common big data analytic framework. This framework is built around the MapReduce programming model [235] which needs to be further explained for this case study. In general, this programming model can be separated into the map and reduce step. The map step filters and sorts the data, while a reduce method performs a subsequent summarizing operation. Combined huge amounts of data can be analyzed by distributing the individual steps of this computation to the compute nodes storing the data.

In this case study, we demonstrate the applicability of our workflow approach to dynamically spawn a big data framework. Hereby, we highlight the general structure of our workflow approach (Section 5.1.1). To discuss the feasibility of our approach, we review the complexity of creating an infrastructure aware workflow model at design time. Moreover, we investigate the behavior of the workflow model and the workflow engine at runtime while using measured timings to discuss potential overheads.

As basis for our study, we model a workflow in which a MapReduce program is used to calculate a *bag of words* that counts the occurrences of words from previously gathered text data. In this scenario, a simple deployment of a single compute node is sufficient for the data gathering step. However, to analyze it a more complex deployment of a Hadoop cluster is required.

Section 8.1.1 describes the workflow model of this study which abstracts the different tasks and their requirements. Section 8.1.2 introduces the runtime model states reached during the execution of the workflow. Finally, Section 8.1.3 presents the results of the described scenario highlighting design time, runtime, and timing observations.

8.1.1 Workflow Model

In Figure 8.2 the Hadoop workflow is shown using our notation for OCCI introduced in Figure 6.4. The workflow consists of three tasks each requiring a specific deployment: **Data Fetching** ①, **Data Distribution** ② and **MapReduce** ③.

The **Data Fetching** ① task requires a single VM named **FetchVM**. This compute resource is used to download a set of text files to be analyzed. For this, the **Wget** executable component is used. This component describes the deployment and execution of the wget software package which we configured to download a set of public text books. After the download process is finished, the data flow is triggered which transfers the fetched data from the **FetchVM** to the **Hadoop-master** VM. The information to transfer the data between the tasks is derived from the **DataLink** by configuring the desired input and output location of the data. Here, different communication channels can be chosen to transfer the data by referring to corresponding elements in the infrastructure layer. In this case, the management network serves as default

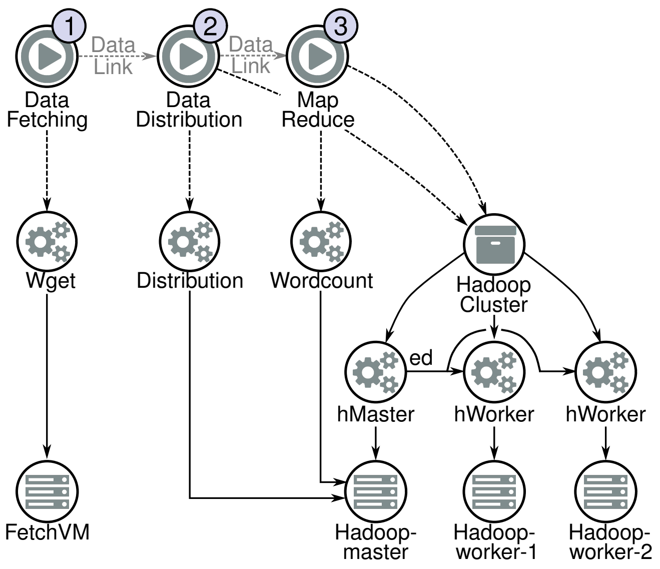


Figure 8.2: On-demand big data workflow.

communication channel as it is automatically added by the orchestration process. We omitted this default network from the visualization for brevity.

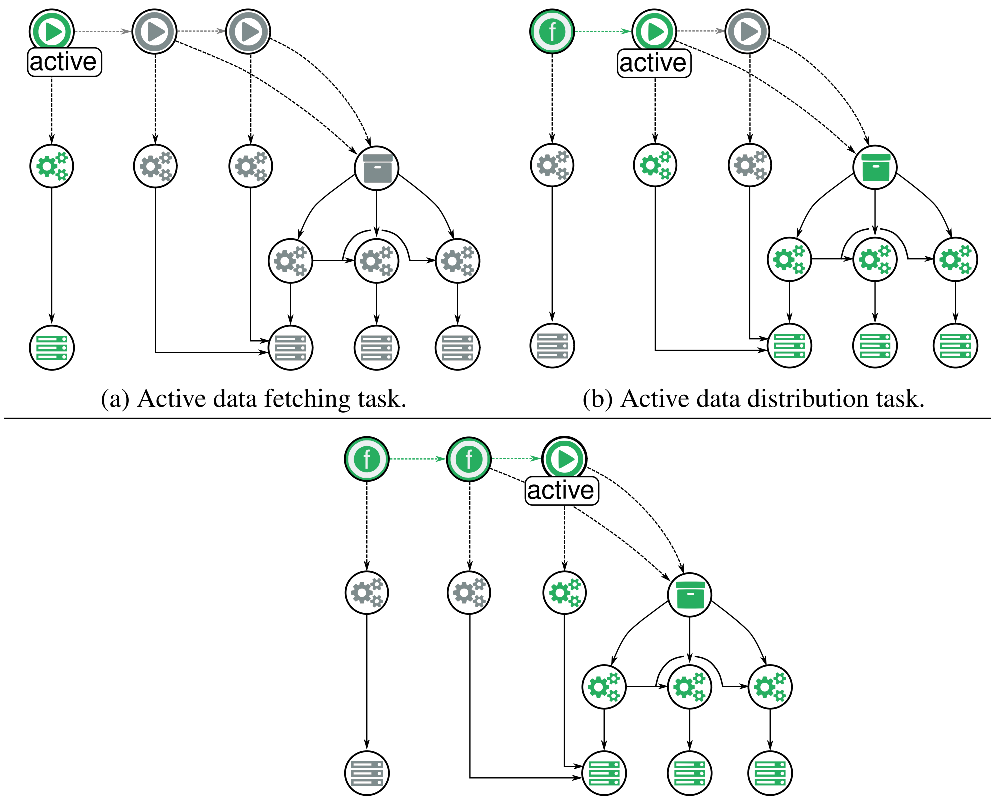
The **Data Distribution** ② task is responsible to store the downloaded files in the HDFS [236], a distributed file system on which the Hadoop computation cluster resides. The commands to store the files in the HDFS are described within the configuration management script attached to the **Distribution** component. To trigger this component the deployment of the Hadoop cluster is required. This requirement is modeled via a PlatformDependency link targeting the **Hadoop Cluster** application. To fulfill this dependency, we reused the core model of the Hadoop application created in the previous case study. A detailed description can be found in Section 7.1.1.1. In general, the application consists of three components that are deployed on three separate VMs posing a typical master worker architecture. The **Hadoop-master** compute node hosts the **hMaster** component, while the **Hadoop-worker-1** and **Hadoop-worker-2** compute nodes host **hWorker** components. To describe the interconnection between the individual components, ComponentLinks are used with an attached ExecutionDependency mixin (**ed**).

The **MapReduce** ③ task finalizes the workflow. It is linked to the **Wordcount** executable which itself is hosted on the **Hadoop-master** compute node. This component possesses a configuration management script that deploys the artifacts to calculate a bag of words from the gathered text books using the Hadoop framework. Similar to the previous task, the **MapReduce** task requires the **Hadoop Cluster** to be deployed as it utilizes the capabilities provided by the Hadoop framework, as well as the data stored within the HDFS.

8.1.2 Workflow Execution

To discuss the dynamic capabilities of our approach, we describe the execution of the workflow by visualizing individual runtime states. These states are shown in Figure 8.3 with deployed resources visualized in green and undeployed resources in gray. In this figure, the individual time steps represent the runtime state and therefore the results of the generated required runtime model. To initialize the workflow, the first required runtime model only contains the tasks to be executed which are correspondingly transferred to the runtime model.

In Figure 8.3 (a) the first cycle of the workflow execution is depicted with the Data Fetching task currently being in the **active** state. This task is the first one to be triggered, as it has no previous tasks. Therefore, the generation of the required runtime model results in a model that only contains the infrastructure of the Data Fetching task, i.e., a single VM. After the architecture is successfully deployed, the Data Fetching task is triggered as part of the same cycle. This results in the deployment, configuration, and start of its executable component



(a) Active data fetching task. (b) Active data distribution task.

(c) Active map reduce job.

Figure 8.3: On-demand big data workflow execution.

fetching a set of text books. Once the download is completed, the Data Fetching task reaches the state finished. The finalization of this task indicates the availability of the fetched data so that the DataLink can be triggered. For this, both VMs hosting the executable components get started by generating a corresponding required runtime model. Once triggered, the data flow ensures that the fetched data is available for the Data Distribution task.

Figure 8.3 (b) depicts the second cycle of the workflow after the DataFetching task is finished (**f**) and the files have been transferred. To trigger the Data Distribution task (**active**), the deployment of the Hadoop Cluster application is required. Therefore, the VMs of the Hadoop cluster are provisioned, and the corresponding components are deployed. Additionally, the VM used to fetch the data is removed from the required runtime model as it is not needed for the execution of the Data Distribution task. Once the cluster is deployed, the Data Distribution is triggered which transfers the fetched data to the HDFS.

Figure 8.3 (c) visualizes the state in which the MapReduce task is executed and **active**. To reach this state, only the executable component is deployed, as the infrastructure requirements are already fulfilled by the previous task. Consequently, the task enactor triggers the execution of the Hadoop job to compute the bag of words from the data stored within the HDFS. After the Hadoop job has finished processing, each task within the workflow has reached the finished state fulfilling the condition to end the execution of the workflow engine. This leads to the Hadoop cluster being provisioned at the end of the workflow which can be released by modeling an extra clean-up task.

8.1.3 Results and Observations

With this case study, we demonstrate the compatibility of our approach with a big-data framework. Hereby, we highlight the capability of modeling infrastructures that can be dynamically deployed and reflected during the workflow execution. To foster the discussion of the applicability of our approach, we describe the complexity of creating a runtime workflow model at design time, and provide observations made at runtime.

8.1.3.1 Design Time

At design time, we observe that the modeling process can be separated into two layers. The workflow layer and the infrastructure layer. As the workflow layer only consists of three subsequent tasks with clear requirements it represents a rather simple layer to model for this scenario. Also, the modeling process of components describing the executable part of the task requires next to no effort as they represent simple scripts triggering, e.g., command line tools such as wget. Still, the scripts have to be configured to utilize the information within the runtime model, e.g., to extract the information from the DataLink about desired input and output files. The highest complexity of this workflow model resides within the deployment of the Hadoop application. However, the model structure and the deployment artifacts could be reused and therefore represent a one time effort.

8.1.3.2 Runtime

Performing the workflow reveals that the execution can be described via subsequent runtime states. Hereby, each runtime state covers the complete workflow layer, indicating the goal to be reached, as well as the infrastructure of the currently running workflow task. Independent of whether a task is currently performed and active or whether the infrastructure is provisioned and deployed, the visualization of the runtime model provides insights about the current state of an application and its components. The visualization and the distinct runtime states allow observing the individual attributes of each resource and link and even trigger their actions. Combined, the runtime model provides an abstract and visualized interface to the system that a user can interact with. During the execution of the workflow we identify that the execution of the workflow and, therefore, the time requirements can be separated into three steps. The first step comprises the generation of the required runtime model, i.e., the analysis and planning of the next state required in the cloud. The second step considers the time required to provision the cloud infrastructure and deploy the cloud application. Finally, the third step comprises the time required to execute the modeled workflow task. Based on this insight, we gathered the corresponding data regarding the time requirements of the presented workflow which are discussed in the following section.

8.1.3.3 Timing Measurements

To provide a better overview of the workflow timings, we executed the case study five times and visualized the 95% confidence interval and mean of the overall workflow, task execution, scheduling, and model generation duration in Figure 8.4. The timings were recorded with each compute node being a VM with two virtual CPUs and four gigabyte memory.

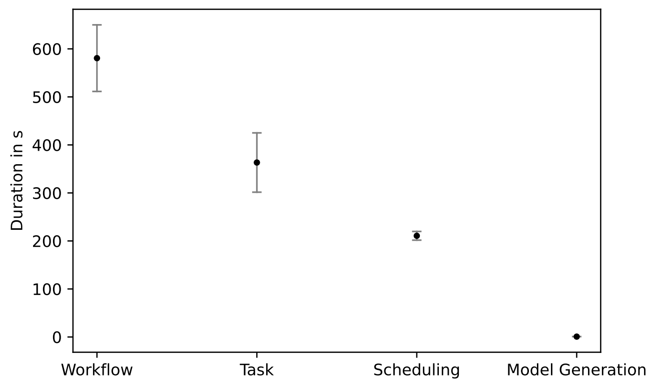


Figure 8.4: 95% confidence interval plot of the Hadoop workflow duration.

The **Workflow** execution for the analysis of 22 text books takes 580 seconds in the mean. When executing the workflow multiple times, a small variance in the time required for its execution can be detected. This variance correlates with the time required to execute the individual **Task** instances which takes 360 seconds in the mean. We explain the observed variance with the Data Fetching task which utilizes an external mirror with shifting download speeds. The **Scheduling** of the architecture on the other hand, has a rather small variance which mainly relates to the time required to spawn VMs in our private cloud. Within our experiment, the provisioning and deployment of the modeled resources take 210 seconds in the mean. In this case study, the deployment time of the Hadoop application is rather constant, as we upload the binaries directly from the OCCI server to the compute nodes rather than relying on external download servers. Finally, the execution of all required **Model Generation** procedures together take 0.7 seconds in the mean with each cycle taking less than 0.3 seconds per iteration. Thus, the generation of the required runtime model is negligible for this workflow model.

While experimenting with the workflow, we observe that small adjustments can greatly impact the execution times. By increasing the amount of text books to analyze, more time is required for the execution of the tasks. To compensate the longer task execution, further worker and master nodes can be added to the Hadoop cluster. This, however, results in more time required to spawn the infrastructure. Ultimately, the size of the model has to be well-chosen to trade off the time to perform the tasks and the time to deploy the cloud topology. Additionally, tailoring the workflow itself can improve the time required for the overall execution. For example, the Data Fetching task can be modeled in such a manner that its computation is performed on the Hadoop-master VM. In this case, the VM is reused for multiple purposes which saves provisioning times and time required to transfer the data.

8.1.3.4 Summary

In the following, we summarize our results of the performed use case.

Summary: The execution of the workflow can be described via distinct runtime model states in which the cloud infrastructure for each individual task is subsequently deployed. At design time, existing artifacts can be reutilized allowing the user to focus on the workflow layer to incorporate, e.g., data flows. The workflow execution time is determined by the provisioning and deployment of the infrastructure as well as the execution of the task. Compared to these timings, the time required for a model based generation of required runtime represents a negligible overhead. By performing small adjustments to the model, we observe that the time requirements can be highly tuned to fit the scope of the experiment.

8.2 Case Study 2: Dynamic Simulation

To gain knowledge about specific behavior within a domain, it is common practice to perform simulations which artificially recreate scenarios of interest. Depending on the level of detail targeted by the simulation, different amounts of computing resources are required. Therefore, the utilization of distributed environments can be of great benefit as cloud resources can be dynamically scaled to reduce cost and energy consumption. In this section, we present a workflow for a dynamic multi-level-simulation that simulates individual parts of a system using different levels of detail [237].

Within this case study we investigate the applicability of our approach to enable dynamic and distributed simulation while highlighting the decision-making process provided by the runtime model (Section 5.1.2). Moreover, we evaluate the capability of a human-in-the-loop that uses the runtime model as an abstract interface to influence the workflow and choose desired simulation details.

As basis for our study, we utilize a multi-level-simulation that abstracts the supply chain of a factory. While at default the factory is simulated at a coarse level, detailed simulations describing quality assurance and order picking processes can be added.

Section 8.2.1 describes the workflow model of the simulation study. Section 8.2.2 introduces the runtime states during the execution of the workflow. Section 8.2.3 discusses observations about the workflow creation, execution and time requirements.

8.2.1 Workflow Model

The workflow model created for the execution of the multi-level-simulation application is shown in Figure 8.5. Overall, the model comprises four tasks and one decision, namely, **Coarse** ①, **Decision** ②, **Picking** ③, **QA** ④ and **SimAll** ⑤.

Coarse ① represents the first task of the workflow with no previous tasks. This task is responsible to perform the standard execution of the factory simulation, i.e., without specific detailed simulations added. The simulation itself is packaged in the component of the **CApp** application which needs to be deployed in order to perform the task. This relation is modeled via a PlatformDependency going from the **Coarse** to the **CApp** node. Additionally, the task possesses an execution link targeting the **CJob** component representing the task's executable. This job triggers the deployed simulation application with predefined parameters such as the amount of simulation steps to be performed. To deploy both the executable and the application, the dedicated **CoarseVM** is modeled. The composition of task, application and executable can be seen throughout the whole workflow as each task requires the deployment of a simulation application on top of virtual hardware. As the simulation requires individual and distributed resources for each simulation detail, each simulation application is hosted on a separate compute node. Each compute node is connected to the **mlsNetwork** with a pre-defined IP. This network is dedicated to manage the message exchange of the simulation.

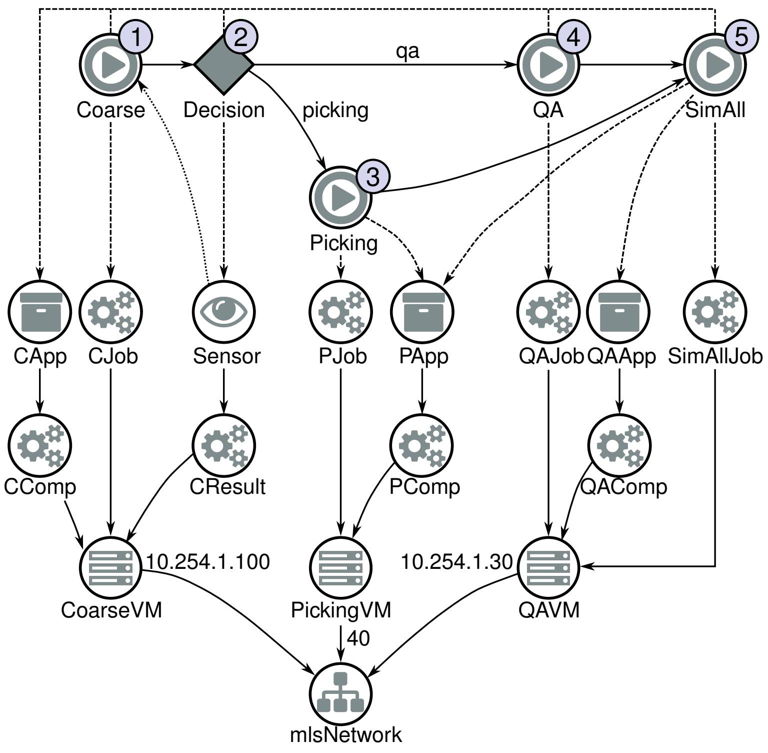


Figure 8.5: Multi-level-simulation workflow.

The second task is represented by the **Decision** ② node. This task is responsible to investigate the results of the previously performed simulation and to decide which detailed simulation to add. To investigate the results of the first simulation, the attached **Sensor** is used. This sensor represents the PlatformDependency of the **Decision** task and therefore is deployed in order to perform the decision-making process. The deployment of the **Sensor** is mainly concerned with the installation of the **CResult** component. This component is connected over a PlacementLink to the **CoarseVM** which serves as a host. Once deployed, the component reads the output of the coarse simulation and aggregates it to decide which detailed simulation to add. In this case, the output volume of processed packages is checked in order to respond to unexpected results with a more fine-grained simulation of the picking process. To ease the decision-making process, we configured the monitoring instrument in such a manner that the observed information is aggregated to match the modeled guards, i.e., **qa** or **picking**. These express the need for a detailed quality assurance simulation, or the addition of a detailed order picking simulation. Finally, using the behavior of a ResultProvider, the aggregated information is reflected within the MonitorableProperty link that points from the **Sensor** to the **Coarse** task.

The **Picking** ③ task and the **QA** ④ task represent detailed simulations of the factory's order picking and quality assurance processes. Both simulation tasks extend the coarse simulation and therefore possess a PlatformDependency targeting the **CApp** application. Additionally, both simulation tasks have a PlatformDependency targeting their own simulation application, i.e., **PApp** and **QAApp**, as well as an ExecutionLink targeting their executable component, i.e., **PJob** and **QAJob**. The components of both tasks are hosted on separate VMs being either the **PickingVM** or the **QAVM** which are both connected to the **mlsNetwork**.

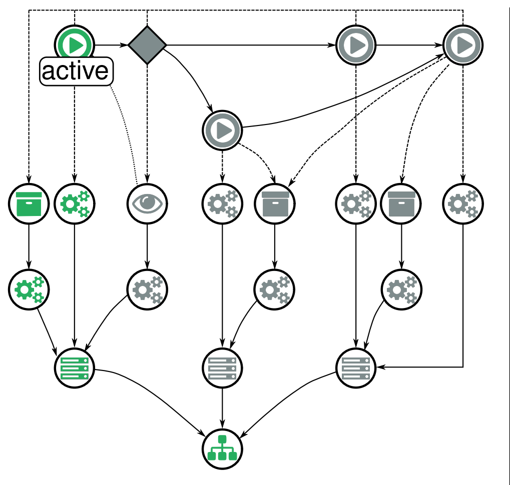
The **SimAll** ⑤ task finishes the workflow and performs the final steps within the simulation by adding all levels of detail to the coarse simulation. For this, the task has a Platform-Dependency to each simulation application, i.e., **CApp**, **PApp**, and **QAApp**. Furthermore, the task is connected to the **SimAllJob** executable component which triggers all simulations at once to inspect the behavior of the factory on the highest level of detail.

8.2.2 Workflow Execution

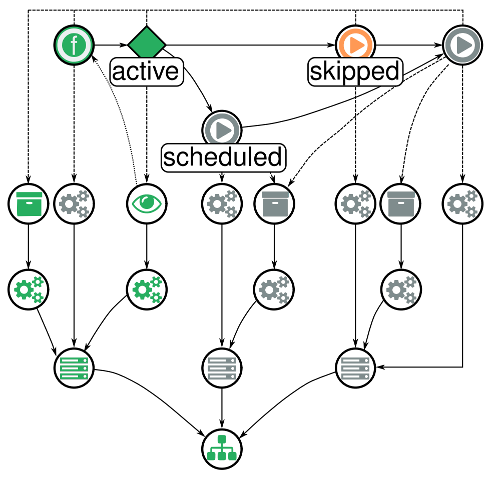
The execution of the workflow shown in Figure 8.5 starts with the transmission of the task sequence into the runtime model. In Figure 8.6, we visualize the individual runtime states reached during the workflow execution. Hereby, we reuse the original model visualization and highlight deployed and active resources in green and undeployed resources in gray.

In Figure 8.6 (a) the first cycle of the workflow is shown. At this point in time, the Coarse task is determined as ready for execution as it has no preceding tasks. Thus, a required runtime model is generated containing only the infrastructure and platform elements required by the Coarse task, followed by the deployment of the generated model. The reached state comprises the VM on which the coarse application is deployed, as well as the executable which triggers the application to perform the simulation. Once the orchestration engine has finished, the VM and the application are successfully deployed allowing the task enactor to request the execution of the Coarse task. This in turn transfers the Coarse task to an **active** state and triggers the accompanied job executable starting the coarse simulation.

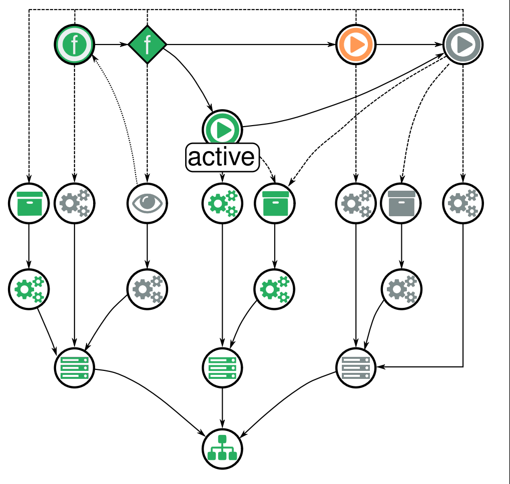
Figure 8.6 (b) depicts the second cycle of the workflow execution which is reached once the Coarse task is finished (**f**). Consequently, the Decision task is performed next as visualized by the figure. Here, the task enactor triggers the execution of the Decision node which transfers it to the **active** state and starts the monitoring process. The monitor comprises the activation of the Sensor application which is configured to reflect the coarse simulation result in the MonitorableProperty. After the sensor has finished processing the monitored information, the reflected information is used as decision input and checked against the modeled control flow guards. In case of the visualized execution, the decision input is set to "picking". Therefore, the QA task is transferred to the **skipped** state with the Picking task remaining in the **scheduled** state. It should be noted, that in a second experiment we adjust the control flow as a human-in-the-loop to test the QA task. Independent of the observed results, the Decision node is finished (**f**) with one of its following tasks being scheduled allowing it to be executed.



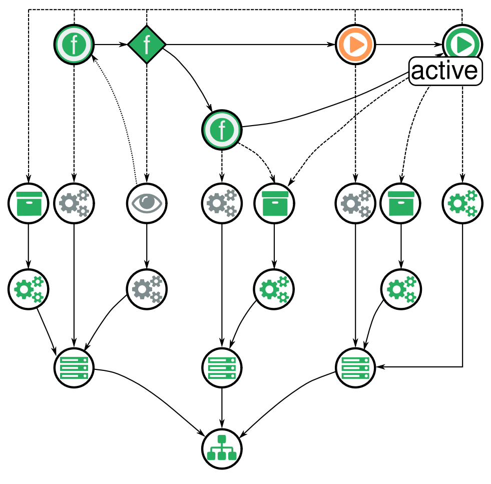
(a) Triggered coarse simulation.



(b) Decision skipping the QA simulation.



(c) Activation of the picking simulation.



(d) Activation of all detailed simulations.

Figure 8.6: Dynamic simulation workflow execution.

Figure 8.6 (c) depicts the third cycle in which the Picking task is executed. For this, a new required infrastructure is derived from the design and runtime model. The resulting required runtime model state comprises a new VM as well as the simulation application and job component of the picking simulation. Additionally, as the Decision task is finished, the Sensor is not needed anymore and therefore deprovisioned. After the architecture scheduling process, the task enactor transfers the Picking task to the **active** state which starts the simulation. After the detailed simulation is performed, the Picking task reaches the finished state.

Figure 8.6 (d) visualizes the fourth cycle of the workflow execution. At this point in time, the state of the runtime model fulfills all workflow requirements to execute the final SimAll task as every previous task is either finished or skipped. The generated required runtime model for this step comprises the VMs for each simulation detail with the corresponding application being provisioned on top. However, only a single executable is deployed corresponding to the SimAll task. This task is executed as soon as the required infrastructure is provisioned, and the applications are deployed. Triggering the task transfers it to the **active** state with the overall simulation job being executed. After the simulation is processed, the workflow is completed with each task being either finished or skipped. Again the infrastructure for the last task remains active after this workflow which may be adjusted by modeling a cleanup step to store the results of the simulation and release excess resources.

8.2.3 Results and Observations

With this study, we demonstrate the applicability of a workflow runtime model for the execution of a dynamic simulation. Hereby, we highlight the monitoring and decision-making capabilities provided by the runtime model and the monitoring extension presented in Section 4.2.2. Furthermore, we use the model of this case study to perform direct interaction with the model by a human-in-the-loop adjusting, e.g., the decision input at runtime. In order to foster the discussion of the applicability of our approach, we provide our lessons learned about this workflows design time, runtime and overall timing aspects.

8.2.3.1 Design Time

The presented workflow follows the general scheme of our runtime workflow model concept. For each task this scheme comprises a VM, the executable job component, as well as the application node and the component describing its deployment. Compared to the former study, this model additionally considers a sensor node and a dedicated network for the communication of individual parts of the simulation. Hereby, we observe that the pre-configured IP addresses can be either manually defined or dynamically retrieved by the the configuration management scripts of the individual simulation components. While the decision-making process is performed at runtime, the modeling at design time only requires the addition of a sensor application. This capability allows reflecting intermediate results in the runtime model which are then used to influence the control flow. We notice that the target of the sensor's MonitorableProperty is flexible and may point to different resources, e.g., the task it observes or the job executable producing the results. This allows for more flexibility as the information of the MonitorableProperty target can be used within configuration management scripts connected to the monitoring instruments modeled. Finally, as each task within the workflow requires the application of the coarse simulation to be deployed, a lot of PlatformDependency links are present within the model. These connections clutter the visualization of the model at design time.

8.2.3.2 Runtime

At runtime, the problem of a highly cluttered visualization due to many PlatformDependency links connecting the tasks to the coarse application is mitigated. Here, only the PlatformDependency links required at the current point in time are provisioned including the infrastructure targeted. Therefore, a clear visualization of the workflow state is provided, as well as the state of the infrastructure currently provisioned.

Apart from the visual benefits of a runtime workflow model, we observe that the integration of sensors from our monitoring extension fosters decision-making capabilities. Combined with the capability to spawn highly tailored infrastructure, this functionality allows tailoring workflows to dynamically decide on required tasks and thus infrastructure. In this scenario, we use a decision node which deploys a sensor that automatically inspects intermediate results. Based on these results follow-up tasks are chosen including the required infrastructure. During this decision-making process, the causal connection of the runtime model provides access to the monitored intermediate results which helps to reason about the workflow and the experiment. In addition to the representation of the monitoring information, the actual decision of this process is reflected in the runtime model as well. Hereby, the runtime states of the guarded tasks are directly manipulated to ensure the correct orchestration of the workflow. In addition, the utilization of these states provide the user with information about the workflow's current state.

While we mainly discuss the automated decision-making process using modeled sensor applications, we additionally investigate the extent to which a scientist can serve as a human-in-the-loop. For this, we simply removed the sensor application from the model. This results in the Decision node blocking the execution of the workflow, as long as no decision input is set. Due to the causal connection of the model and the workflow, we are able to choose the desired control flow to follow, i.e., the desired detailed simulation, by simply adjusting the decision input attribute.

8.2.3.3 Timing Measurements

In this section we compare the workflow time, scheduling and task execution time, as well as the accumulated time required to generate the required runtime model states. The workflow model got executed five times with the 95% confidence interval of the timing measurements shown in Figure 8.7. It should be noted, that in each execution of the workflow the decision-making process decided for the picking assurance simulation. Furthermore, we configured each VM to be of medium size comprising two virtual CPUs and four gigabyte memory.

Overall, the **Workflow** execution takes 344 seconds in the mean with a small variance. This variance matches the one observed by the **Scheduling** process which takes 284 seconds in the mean. We explain the variance with the individual start-up times of requested VMs. Compared to the other timings, the variance for the **Task** execution is rather constant with a mean duration of 50 seconds. Also, the accumulated times of all required runtime

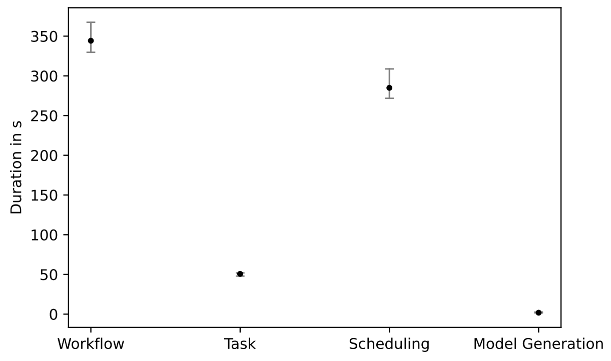


Figure 8.7: 95% confidence interval plot of the simulation workflow duration.

Model Generation executions take 1.8 seconds in the mean. The observed values show that the model generation does not significantly impact the time required to execute the workflow and therefore is negligible. All in all, the scheduling takes up the most of the workflow execution time, mainly because of the time required to spawn the VMs. In this case, the dynamic provisioning of distributed resources is only justifiable if the individual simulation tasks represent long-running processes. This, e.g., can be reached by increasing the amounts of simulation steps to be performed. Alternatively, the scheduling time can be reduced by modeling container nodes instead of VMs.

8.2.3.4 Summary

In the following, we summarize our results of the performed use case.

> **Summary**: By attaching workflow tasks to individual infrastructures deployed architectures can be dynamically shifted. Still, spawning new infrastructure is costly and may induce an overhead for short executing tasks. Therefore, the workflow model needs to be well designed by the user to ensure an efficient resource management. The integration of sensors and the reflection of arbitrary information fosters the decision-making process, e.g., by monitoring intermediate task results to decide which control flow to follow. Hereby, the causal connection of the runtime model ensures that users can simply influence the execution of the workflow at runtime by performing small attribute adjustments.

8.3 Case Study 3: Software Repository Mining

Software repositories and *Version Control System*s (VCSs) support the software implementation process by providing a traceable way to store artifacts and how they change over time. Combined with the information that is stored within mailing lists and issue tracking systems, huge amounts of data are available to draw insights about how software is developed, implemented, and maintained. This process is commonly referred to as software repository mining. When mining software repositories, different amounts of computing resources are required, e.g., for the data gathering and analysis tasks.

In this case study, we demonstrate the applicability of our runtime workflow model concept for the software repository mining domain. Hereby, we highlight our loop concept including their parallelization (Section 5.1.3). Moreover, we use the insights gathered from the previous study to investigate the extent to which a domain specific OCCI extension can support the workflow runtime model.

As basis for our study, we recreate the mining and analytic process formed around the SmartSHARK platform [238]. This platform allows scientists to extract software quality metrics from software projects in order to perform empirical studies [239–242]. To allow for a precise configuration of the repository mining workflow, we create a domain specific extension to support the three core plugins of SmartSHARK. The VCSShark plugin extracts metadata from the software repository and stores it in a database. The MecoShark plugin utilizes this metadata to analyze the metric values of each commit in corresponding software projects. The MemeShark plugin is used to reduce the amount of required storage and cleans redundancies in the database. While gathering metadata and the removal of redundancies in the database require only a few computing resources, the analysis of the individual commits of the software project may utilize as many computing resources as available. Especially the analysis of the individual commits can be performed in parallel.

Section 8.3.1 describes the workflow model and extension. In Section 8.3.2 the workflow model runtime states are described including the state when the loop is marked as parallel. Section 8.3.3 discusses results and observations made during the study.

8.3.1 Workflow Model

The SmartSHARK framework follows a plugin based approach and therefore consists of multiple components introducing functionalities to support the software repository mining process. To incorporate the individual plugins on a higher level of detail, we designed a dedicated OCCI extension. In Section 8.3.1.1 this domain specific extension is introduced. Thereafter, in Section 8.3.1.2, we describe the software repository mining workflow built around the SmartSHARK framework.

8.3.1.1 SmartSHARK Repository Mining Extension

For the execution of this case study, we created an extension for the three major plugins of the SmartSHARK framework. The VCSShark, MecoShark, and MemeShark. Additionally, we use this extension to introduce dedicated elements for individual job components of these plugins. Among others, these allow scientists to configure the repository to analyze or the desired log level. A subset of the OCCI SmartSHARK extension, including its relation to the MoDMaCAO platform extension, is shown in Figure 8.8. All elements within the OCCI **SmartSHARK Extension** are mixin instances. Most of them depend on the MoDMaCAO **component:Mixin**. This ensures that a configuration management script can be attached in order to deploy and manage the domain specific framework. In case of this extension, we allow the deployment of the SmartSHARK plugins using the **MemeShark**, **MecoShark** and **VCSShark** mixins. These mixins do not contain any attributes as the actual configuration of the framework is determined on the job level, e.g., over command line arguments. This job level is reflected by the **SharkJob** mixin. This mixin mainly serves as an abstract parent class introducing common attributes shared among the different plugins. The **shark.project.repository** describes, e.g., the *Uniform Resource Locator* (URL) of the repository to analyze. For each plugin we modeled one specialization which represents the corresponding executable, i.e., **VCSJob**, **MemeJob** and **MecoJob**. Each of these mixins introduce additional attributes that allow for a more concise representation of the workflow runtime model. For example, the **shark.project.revision** attribute of the **SharkJobRevision** mixin is used to automatically configure the revision which is analyzed during each iteration of a loop. To determine the revision hashes of a software repository, the **MongoQuery** mixin

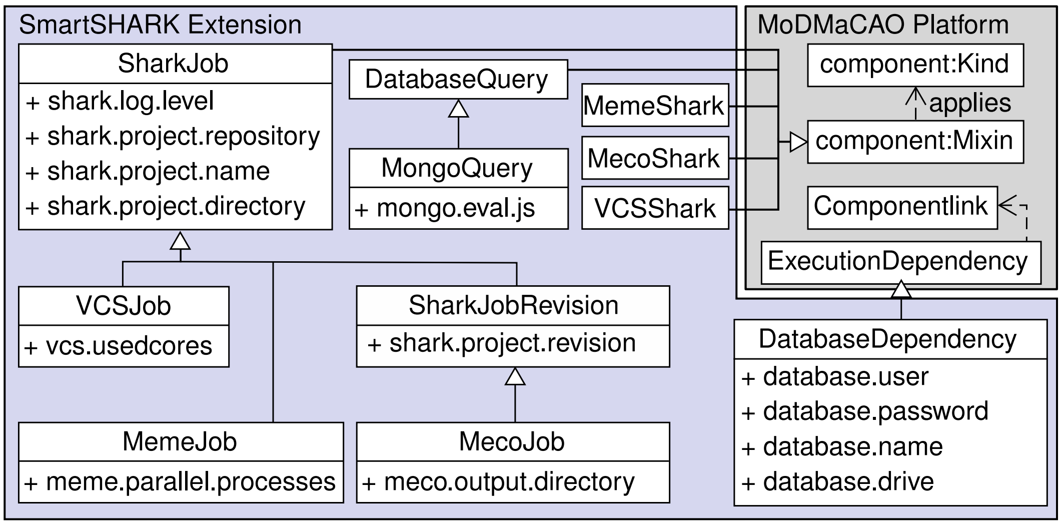


Figure 8.8: Subset of the OCCI SmartSHARK extension.

is used. Using this mixin, the scientist can configure the exact query to perform against the database storing the mined information. Due to the heavy utilization of databases within the SmartSHARK environment, the extension introduces a **DatabaseDependency** mixin which specializes the MoDMaCAO **ExecutionDependency**. The **DatabaseDependency** provides multiple attributes to configure the connection to the database, e.g., the **database.user** and **database.password**. Among others, this information is used within the configuration management scripts of the job components.

8.3.1.2 SmartSHARK Repository Mining Workflow Model

The software repository mining workflow, shown in Figure 8.9, exemplifies a common software repository mining sequence covering a loop to iterate over the commits to analyze. This sequence includes the **VCSTask** ①, **Loop** ②, **MecoTask** ③ and **MemeTask** ④.

Except of the loop, each task possesses a PlatformDependency to a modeled SmartSHARK application. This structure is accompanied by an ExecutionLink that targets the corresponding executable job component that is used to start the mining task. For these components the domain specific SmartSHARK extension is applied that we introduced in Section 8.3.1.1. To clarify the utilization of the modeled elements, we equally name the component instances to the domain specific mixin utilized. To ensure a constant deployment of the database, each task is connected over a PlatformDependency to the **MongoDB**. Moreover, each executable component is connected to the database. The corresponding ComponentLinks are modeled with DatabaseDependency mixins, which we visualize as dotted gray lines.

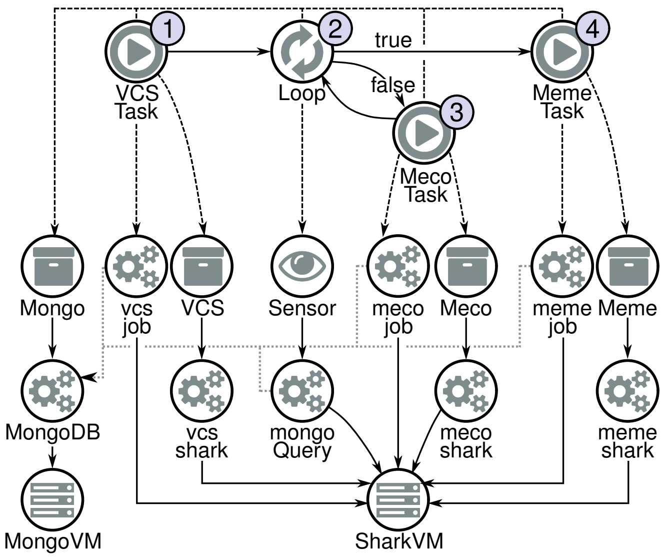


Figure 8.9: Software repository mining workflow.

The **VCSTask** ① gathers the metadata of the software repository to mine. For this, it requires the **VCS** application and the **MongoDB** to be deployed. The executable of the task is represented by the **vcsjob** component. When triggered, this component starts the **VCS** application and therefore the gathering process. Hereby, the attributes of the component can be adjusted by the scientist to define the repository to mine.

The **Loop** ② task is modeled with an attached ParallelLoop mixin that we set to zero and three in our experiment. The loop iterates over the commit revision hashes gathered beforehand. To retrieve this metadata the **MongoDB** is queried using the **VCSSensor** and its **mongoQuery** ResultProvider. This monitoring instrument has an attached MongoQuery mixin which we configure to gather the metrics of each commit available for the given project stored within the database. Once executed, the queried information is stored within the decision input of the **VCSSensor** containing delimited revision hashes. The decision expression is configured in such a manner, that the loop iterates as long as the decision input is not empty. As each iteration consumes one item, the decision input is emptied once all revision hashes have been processed. Therefore, the guards are modelled with **true** and **false** which respectively describe that a further revision has to be analyzed or that the loop is finished.

The **MecoTask** ③ represents the looped task to which the revision hashes are passed in order to extract metrics of the software project at a specific point in time. In order to analyze these metrics, the **Meco** application has to be deployed. To trigger the execution of the analysis job, the **mecojob** executable component is used which locally replicates the software repository, followed by a checkout and analysis of the revision hash.

The **MemeTask** ④ represents the final task within the software repository mining workflow. It requires the **Meme** application to be deployed. This application, when triggered, shrinks the amount of storage required in the database by removing redundancies. To trigger the application the **memejob** executable component is activated which is filled with the information about the software project to process.

At the infrastructural layer, the workflow model consists of two VMs. The **MongoVM** hosts the **MongoDB** in which the mined metrics are stored. The **SharkVM** is used as host for the SmartSHARK plugins. This VM is reutilized as each plugin requires similar packages reducing the overall deployment time. It should be noted, that this VM is duplicated when executing the workflow with a parallelized loop which spawns multiple **MecoTask** with individual applications and VMs and commits to analyze.

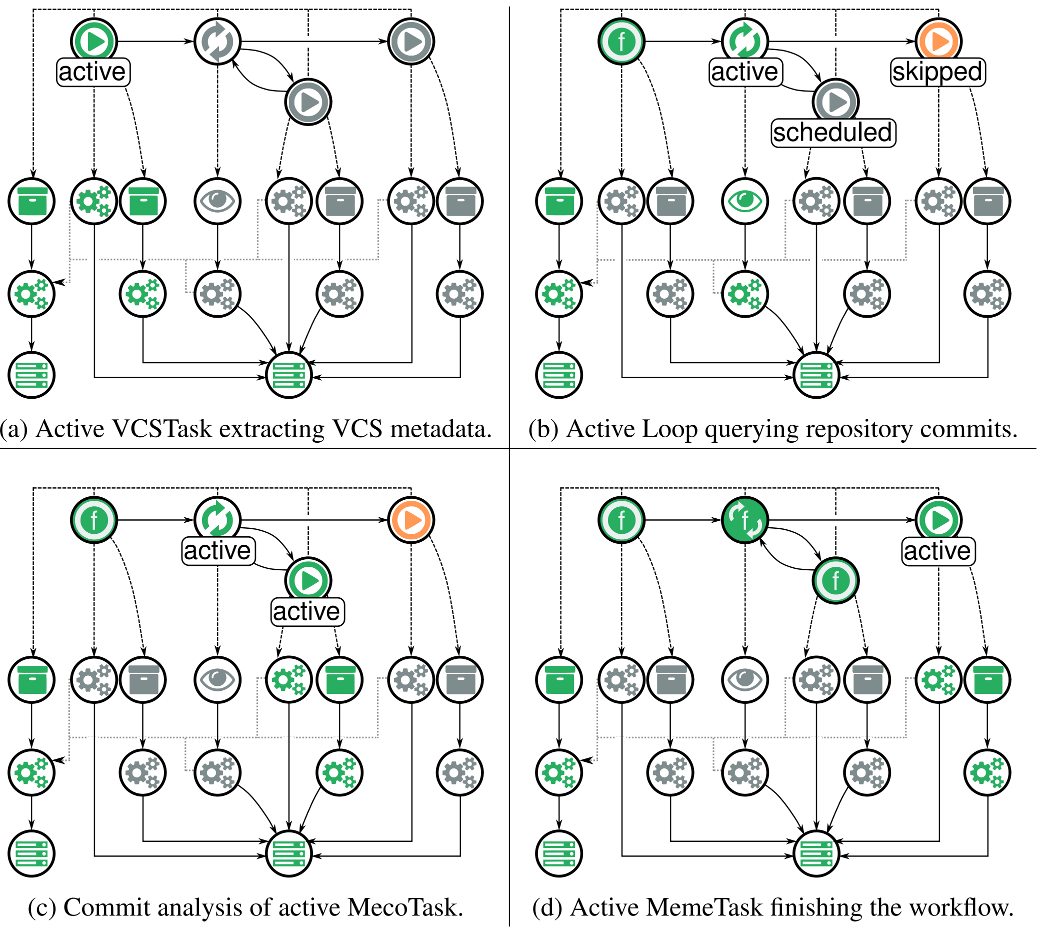
8.3.2 Workflow Execution

In the scope of this study, we execute multiple configurations of the mining workflow to analyze the different implementations of our approach. Independent of the chosen project, the sequence of reached runtime model states are the exact same with only the execution duration varying. Section 8.3.2.1 describes these runtime states for a serial execution of the loop, while Section 8.3.2.2 highlights the parallel execution.

8.3.2.1 Serial Workflow Execution

Similarly to the previous workflow studies, we start with an empty runtime model. The runtime states reached during the execution of the workflow are visualized in Figure 8.10. In this figure, we marked the undeployed resources in gray, deployed resources in green and skipped tasks in orange.

Figure 8.10 (a) depicts the first major runtime state highlighting the workflow with the VCSTask being currently **active**. The modeled PlatformDependency links from the VCSTask result in a required runtime model containing both the SharkVM and MongoVM, as well as the components and application nodes of the MongoDB and VCS plugin. The resulting required runtime model then serves as input for the orchestration engine which leads to the provisioning of the aforementioned VMs with the corresponding applications being deployed



(a) Active VCSTask extracting VCS metadata. (b) Active Loop querying repository commits.
(c) Commit analysis of active MecoTask. (d) Active MemeTask finishing the workflow.

Figure 8.10: Software repository mining workflow execution.

on top. After the orchestration process has finished the deployment of both applications, the task enactor triggers the execution of the VCSTask resulting in the depicted **active** state. Once triggered, the vcsjob executable component is deployed, configured and started leading to the VCSShark plugin being executed against the defined software repository. After the VCSShark has finished mining the metadata of the repository, the VCSTask transfers to the finished state. This leads to a deprovisioning of the VCS application and job component, as well as the Loop task to be triggered next. It should be noted, that for the remainder of the workflow execution the Mongo application, MongoDB component, and MongoVM persist in an active state. These elements remain in the state active, due to each task possessing a PlatformDependency targeting the Mongo application.

Figure 8.10 (b) depicts the second major time step during the execution of the software repository mining workflow, after the VCS task is finished (**f**). In this time step, the Loop task is currently in an **active** state with the VCSSensor being triggered. By triggering the sensor, also the mongoQuery monitoring instrument is started. As a result, the revisions of the project are queried from the database and stored within the decision input attribute of the loop. Thereafter, the VCSSensor is stopped. In our case study, the projects to analyze contain multiple commits leading to the decision input being not empty. Therefore, the MemeTask is transferred to the **skipped** state with the MecoTask remaining in the **scheduled** state. Finally, the revision hash of the commit to analyze is passed to the MecoTask by attaching a LoopIteration mixin.

Figure 8.10 (c) highlights the runtime model state during the execution of the loop. Here, the Meco application is deployed with both the MecoTask and the Loop task being in the state **active**. It should be noted, that in this state the executable VCSSensor is deprovisioned as the Loop already contains a decision input and therefore is not required anymore. Finally, the mecojob is triggered. This executable utilizes the passed revision hash to gather and store its metric values within the MongoDB database. This loop continues until each revision hash has been passed from the Loop task to the MecoTask. Thereafter, the decision input attribute of the Loop is emptied. Once emptied, the evaluation of the loop expression results in the MemeTask being scheduled. This allows it to be executed next.

Figure 8.10 (d) shows the runtime model state during the execution of the now **active** MemeTask. During the transition to this state, the Meco application is deprovisioned with the Meme application being deployed. Once deployed, the platform dependencies for the MemeTask are fulfilled allowing for its memejob executable component to be triggered. This results in the deletion of created duplicates from previous mining steps and therefore in smaller storage required within the deployed MongoDB. As the MemeTask represents the final task, its platform dependencies remain deployed. Again a cleanup step can be added which, e.g., can be used to deploy a Jupyter Notebook [243] to analyze the mined data.

In the following, we describe a variant of the repository mining workflow in which the loop is tagged as parallel leading to multiple MecoTask instances being spawned.

8.3.2.2 Parallel Workflow Execution

The **MecoTask** shown in Figure 8.9 can be parallelized as the metric gathering process of the individual commits is independent of each other. To enable a parallel computation, we added a parallel mixin to the **Loop** task and set the parallelization level to three. Additionally, we infused the **MongoDB** with a shared mixin as all the mined data should be stored within the same database. Compared to the sequential execution of the workflow, this change results in nearly the same sequence of runtime model states. One exception is the execution of the loop (see Figure 8.10 (c)) which we further discuss in this section. We visualized the differing runtime model state for the three-fold parallelization of the loop in Figure 8.11. The execution of the parallelization process is separated into two parts. The MecoTask replication on the workflow layer ① and the additional provisioning of infrastructure ②.

The task replication ① starts once the main **Loop** is triggered. For each parallelization level one nested **ParLoop** is created. Each **ParLoop** is connected to the main **Loop** via a NestedDependency link. Additionally, each looped task is replicated for each nested loop. In this case, the **MecoTask** is replicated two times in form of a **Replica MecoTask**. To distribute the workload, the decision input of the main **Loop**, that is filled with the revision hashes, is evenly divided among the **ParLoop** instances.

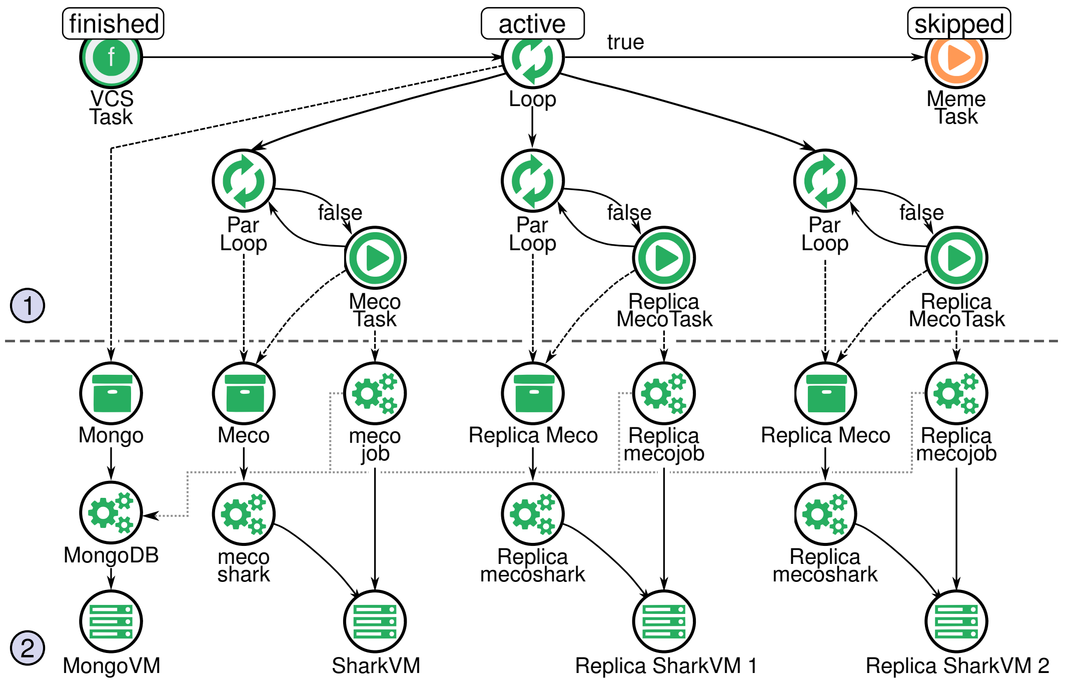


Figure 8.11: Three-fold parallelization of the repository mining loop.

To provision additional infrastructure ②, the resources required by the **MecoTask** are replicated. Hereby, one **Meco** application and one **mecoshark** application component is created for each **Replica MecoTask**, as well as an executable **mecojob** component. Each of these components is hosted on a replicated version of the **SharkVM**, i.e., **Replica SharkVM 1-2**. It should be noted, that the **MongoDB** is tagged as shared and is therefore not duplicated. Moreover, by tagging the database as shared, the **mecojob** executable and its replicas are connected to the same database. In the figure, the corresponding ComponentLinks are visualized in gray.

Throughout this step of the workflow execution, the main **Loop** is notified once one of its nested loops reaches the finished state. Each time the main **Loop** is notified it checks whether all nested loops are finished. Once the last **ParLoop** is finished, the main **Loop** also transfers to the finished state. At this time a further evaluation is performed. As the decision input is emptied, the control flow guard targeting the **MemeTask** is evaluated to true. Therefore, the **MemeTask** is scheduled again which allows for its execution. As the finalizing task does not need the replicated resources anymore, they are released from the runtime model and deprovisioned in the cloud. Thus, only the **MongoVM** and the **SharkVM** remain. As a last step, the **MemeTask** is executed which finalizes the workflow.

8.3.3 Results and Observations

Within this case study we highlight the feasibility of loops and parallelization within our runtime workflow model approach. We show that the decision-making process can be reused to iterate over a set of tasks. Additionally, we demonstrate how a loop and its tasks can be parallelized by provisioning and reflecting additional infrastructure. Furthermore, as part of this study, we investigate the benefits and drawbacks of a domain specific extension. In the following, we discuss the impact of these individual aspects on the workflow at design time, runtime, and general time requirements.

8.3.3.1 Design Time

The repository mining workflow is designed in the same pattern as the workflows presented in the previous studies. Again, each task requires a specific application to run which is accompanied by an executable component that is triggered once the task is started. Compared to the other studies, each executable component requires direct access to a database and therefore is connected over a link. This link ensures that the generation of the variable file contains the information to connect to the database. While these connections foster a concise representation of the runtime model, it represents an extra modeling effort for the user that could be automated, e.g., by a model transformation. Modeling the loop is similar to the concept of the decision node including the need for a sensor. The modeled sensor partially reflects the intermediate results of the VCS task providing us with information about the amount of revision hashes to analyze.

To create a sophisticated software repository mining workflow model, we designed and generated a domain specific OCCI extension. The utilization of domain specific extensions provides a pre-defined set of attributes that can be configured by the scientist. In case of this domain, these attributes allow specifying the exact repository to analyze. Thus, the repository to analyze can be quickly adjusted by changing the corresponding attribute. Once designed, the implementation represents a small one time effort as corresponding artifacts can be partially generated from the model.

8.3.3.2 Runtime

During the execution of the workflow, the runtime model allows observing the current state of the infrastructure and the overall workflow. Especially the reflection of the loop provides insight about the amount of iterations already performed. Moreover, it displays which commit is currently processed by which task. Also, we observe that the runtime visualization supports the transparency of the parallelization strategy by directly showing added and removed resources. In case of the three-fold parallelization, the addition of the two additional VMs is clearly visible including the deployment procedure of the required applications. Furthermore, the observation of the loop parallelization shows that the individual task iterations require different amounts of time. Therefore, different infrastructure configurations are scheduled which depend on the state of the nested loops. Once a nested loop finishes, its associated infrastructure is deprovisioned. This, however, does not affect the infrastructure of still active loops which remain deployed.

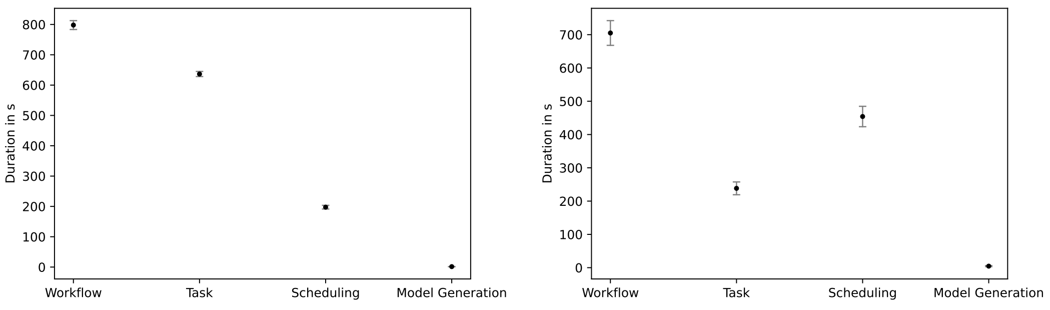
Over the course of the parallel execution study, we observe that PlatformDependency links play an important role to tune individual runtime states. For example, if a Platform-Dependency to the Meco application is modeled, the infrastructure for each replicated resource exists until all loops are finished. This, however, leads to an excess of infrastructural resources. On the other hand, if no PlatformDependency is modeled the infrastructure of a finished loop is deprovisioned even though it may be required by the a subsequent workflow task. While not affecting the concept of our workflow runtime model, these observations reveal possible improvements in the required runtime model generation. For example, subsequent model transformations may incorporate state-of-the-art scheduling techniques.

8.3.3.3 Timing Measurements

In this section we describe the timing measurements for the repository mining workflow. For this, we configured the workflow to mine the software repository of our *Comparing OCCI* (COCCI) [244] project. This project implements the model transformations used in our approach and consists of roughly 60 commits. Similar to the previous workflow studies, we set up each VM to be of medium size comprising two virtual CPUs and four gigabyte memory. Figure 8.12 visualizes the 95% confidence interval and mean of the duration for both the sequential and parallel execution.

Figure 8.12 (a) shows the duration values for the sequential execution of the workflow. The mean of the overall **Workflow** execution amounts to 798 seconds. This timing is composed of a **Task** duration with a mean of 636 seconds and a **Scheduling** duration of 197 seconds in the mean. Hereby, the scheduling of the architecture for the first task takes up the most time with roughly 130 seconds. This timing occurs due to the provisioning of the VMs required throughout the workflow, as well as the deployment of the database. For the **Task** duration, the looped task that iterates over the individual commits takes up the most time with roughly 600 seconds. Finally, the accumulated time for the **Model Generation** processes requires 1.4 seconds. Each of these values show a low variance with an overall potential to speed up the workflow using parallelization.

Figure 8.12 (b) depicts the confidence interval and means for the execution of the three-fold parallelization of the workflow. This visualization shows that the overall workflow requires less time with 705 seconds in the mean. In this experiment the duration of the **Scheduling** process takes up 454 seconds. While this duration is higher compared to the sequential execution, the **Task** duration is reduced to 238 seconds in the mean. This shift occurs as the parallelization spawns multiple additional VMs at runtime that are subsequently used to speed up the calculations within the workload. It should be noted, that the time required to provision the infrastructure and deploy the applications does not scale with the size of the project to analyze. Therefore, once a larger software project is mined, the extended time to spawn up additional infrastructure is quickly made up with the faster processing of the loop iterations. In the parallelized version of the workflow, an overall higher variance can be observed. This is likely related to the parallel management of the tasks, as well as the additional workload put on our small private cloud. Finally, the accumulated duration of the **Model Generation** processes rises to 4.6 seconds in the mean. Still, the duration is negligible compared to the task and scheduling execution times.



(a) Sequential workflow execution. (b) Three-fold parallel workflow execution.

Figure 8.12: 95% confidence interval plot of the repository mining workflow duration.

8.3.3.4 Summary

In the following, we summarize our results of the performed use case.

> **Summary**: Modeling a domain specific extension allows for a more concise management and reflection of the runtime workflow model. This opens the opportunity to influence the workflow by adjusting pre-defined attributes. Overall, the creation of such an extension represents a one time effort that raises the reusability of the workflow model. By reflecting loops in the runtime model, the current iteration of the workflow is revealed. Therefore, users can directly inspect and interact with it. When executing loops in parallel, a speed-up of the workflow can be achieved. However, in order to avoid overheads, the additional time required to dynamically spawn new infrastructure needs to be considered. In summary, the runtime workflow model offers the capability to tailor scientific workflows to their individual needs while simultaneously maintaining a visualization and interface to interact with the system at runtime.

9 Discussion

In this chapter, we discuss the extent to which the OCCI standard can be used to combine a runtime model management of workflows and cloud deployments. The chapter itself is structured into four parts. Section 9.1 discusses the applicability of a standard based cloud orchestration and workflow runtime model in academia, industry and education. Section 9.2 considers different viewpoints on the presented runtime model from the perspective of practitioners and software components. Section 9.3 discusses lessons learned about OCCI and highlights recommendations to improve the standard. Finally, Section 9.4 discusses the threats to validity we identified.

9.1 Applicability

By performing the orchestration case studies (Section 7), we show that an OCCI conform orchestration process can support the management and development of cloud deployments. Furthermore, the workflow execution case studies (Section 8) show the feasibility of extending OCCI with workflow management capabilities to utilize dynamic and arbitrary infrastructures, as well as a human-in-the-loop. To extend the discussion to a point beyond the feasibility of the approach, this section investigates the extent to which a standardized and runtime model-driven management of cloud applications and workflows can be beneficial in academia, industry and education.

9.1.1 Academia

The amount of existing workflow systems, such as Pegasus [167] and Taverna [170], reflect their need and usefulness in academia. We identify four advantages of a standardized cloud workflow runtime model for academic purposes: 1) the utilization of task tailored infrastructure, 2) a human-in-the-loop that can interact with the model, 3) the distribution of domain specific extensions that conform to a cloud standard and 4) the opportunity to replicate cloud research using local resources.

In this thesis, we emphasize the flexibility offered by cloud computing and current IaC tools to show how infrastructures can be dynamically orchestrated throughout the execution of a workflow. While the runtime workflow model is potentially applicable on industry use cases, this thesis is focused on workflow case studies from the scientific domain. Our approach allows scientists to choose their desired infrastructure for individual tasks in the workflow and tailor the infrastructure to their needs while using the high level of abstraction provided by the model. The big data workflow (Section 8.1) shows, e.g., that large clusters can be dynamically spawned for analysis tasks while using only a small set of resources to fetch data. While a dynamically spawned infrastructure stretches the overall execution time of the workflow, preconfigured computation clusters are no longer needed, allowing to save resources and costs.

In context of scientific workflows, a human-in-the-loop enables scientists to directly interact with the workflow throughout its execution. Also, the runtime model provides a visual reflection of the infrastructure and workflow that can be observed by the scientist. In our dynamic simulation case study (Section 8.2), we demonstrate how a human-in-the-loop can change the workflow's control flow at important decision-making points using intermediate results directly reflected in the runtime model. Furthermore, we demonstrate how the interface to the model can be used to provide scaling mechanisms to be influenced by the scientist. Here, we adjust, e.g., the amount of parallel running worker nodes or the size of provisioned VMs. This allows us to trade resource consumption and cost for execution time and faster results.

Building upon an open cloud standard, our approach introduces platform, monitoring and workflow capabilities extending the OCCI ecosystem. Compared to non-standard conform solutions, standard based approaches can be reused which supports the exchange of ideas among scientists to even further extend the standard. In the software repository mining workflow (Section 8.3), we develop a domain specific extension for OCCI that integrates the SmartSHARK framework into our workflow approach. This extension may be shared among researchers to utilize the tooling in various workflows with only minimum knowledge required due to the abstract model representation.

Replicating cloud deployment studies from literature often requires access to a cloud environment which limits the reproducibility of the research. Runtime models can help researchers to mitigate this issue by providing different effectors linking, e.g., to a cloud or local resources. We show this exemplary in our simulation study (Section 7.1.2.2). Here, we create a local environment to develop cloud deployments and assess, e.g., the impact of adaptive changes. Among others, this environment is used to create a replication kit for our studies. This replication kit mirrors a cloud runtime model behavior using container instances that run on a local workstation.

9.1.2 Industry

The benefits of using a model-driven language for cloud deployments is already discussed in literature and utilized in many related approaches, such as CloudML [114], CAMEL [115] and OpenTOSCA [137]. In this thesis we work with a standard based runtime model which allows focusing on the domain while supporting monitoring and testing of cloud deployments. In the previous section, we have discussed the applicability of the orchestration process for the provisioning of infrastructures for scientific workflows and thus in an academic context. In this section, we discuss the applicability of our approach from a more industry based perspective, even though the offered capabilities can be applied on research projects. In the following, we discuss our observations based on the three common industry scenarios executed in Section 7.1.2.

The standard conform orchestration process allows practitioners to focus on the cloud application domain rather than provider or framework specific details. Especially the utilization of model transformations enable the addition of default or provider specific information which eases the transition between providers. Within our initial deployment scenario (Section 7.1.2.1), we demonstrate how the orchestration process automatically adjusts a platform independent model to be deployed in an OpenStack. Especially the utilization of a standard allows us to reuse existing approaches, e.g., to integrate container [104] or configuration management [105] which are based on state-of-the-art implementations.

During our studies, we observe that the runtime model is able to aid in the development process of configuration management scripts, as parts of the scripts can be individually triggered and observed through the model. Furthermore, different environments can be used to mirror the behavior of the deployed model in different scenarios. Therefore, the impact of planned adaptive actions can be locally assessed before an actual enactment on the production environment. The simulation scenario (Section 7.1.2.2) shows how the capabilities of the runtime model can be exploited to partially deploy a computation cluster using only local resources. This local simulation allows to incrementally test the different lifecycle actions by triggering them through the runtime model interface. The quick redeployment of the model results in a quick recovery of erroneous states as it requires less time than the deployment in an actual cloud environment. Additionally, the runtime model supports the development of test cases as individual states could be stored and reused, e.g., in CI-pipelines. Overall, using a simulation environment the runtime model possesses a resemblance to a digital twin allowing the operators to work on a high level of abstraction with a predefined language.

Another major benefit of a runtime model approach is its reflective nature which allows monitoring and observing the system. With our monitoring approach, we extend the OCCI runtime model to reflect operational parameters in addition to structural resource compositions. Combined with a visual representation of individual runtime states, changes to the system can be monitored which helps to understand how certain adaptation or scaling procedures behave. Within the scaling engine scenario (Section 7.1.2.3) we demonstrate that the added reflection of monitoring information in the runtime model supports the development of adaptation engines. Due to the automatic orchestration of OCCI adaptation, practitioners can focus on the analyze step in the self-adaptive control loop as the monitoring, planning and execution phase is handled by the runtime model. Moreover, the utilization of model transformations results in a loose coupling of engines that can be chained together, e.g., to further optimize the scheduling mechanism and subsequently save resources.

9.1.3 Education

Abstraction layers and metamodels are often used to teach about domains as they provide an abstract description language that is typically supported by an accompanied visual notation. While the UML standard is the de-facto staple for teaching about software architectures and design, cloud languages also have been proven useful to simplify the utilization of cloud deployments [117]. In this section, we discuss the potential impact of our approach for educational purposes.

In general, the OCCI metamodel layer can be taught using, e.g., UML class diagrams. In turn, also the instance layer can be visualized and explained using the UML object diagram notation. We argue that this notation is not suited for runtime representation as it visualizes OCCI links in form of rectangles which increases the amount of displayed elements and thus clutters the overall visualization. In the scope of this thesis, we propose a graphical notation (Section 6.2.1) which emphasizes the graph based nature of the standard that supports interactions with the runtime model. In this notation, resources are represented by nodes with a color code based on their state. Links are depicted in form of arrows connecting them with their most important attribute represented as label. In addition, our approach can be used by lecturers to spawn infrastructures and deploy applications quickly and without much manual effort. For example, computation clusters such as Apache Hadoop [231] can be automatically provisioned. Moreover, the spawned deployment can be observed and monitored due to the models high level of abstraction.

The runtime model may foster an interactive learning experience for students as changes to the cloud deployment result in immediate feedback. Furthermore, using the abstract representation of the model, implementation specific details can be partially neglected by the student such as setting up a specific IP address of a VM in network. This may foster a more incremental learning procedure as students can focus on core cloud elements before being confronted with all available configurations. Moreover, due to the reflection of operational parameters students may benefit from a reflection of workload they are producing to get a better understanding of how certain computations are distributed. Additionally, due to the runtime model being able to connect to different systems, students do not need access to large computation clusters to start working with specific frameworks as they can also be spawned locally at a much smaller scale. We observe that only a few kinds need to be known to start creating larger deployment models with details of the application represented by mixins. Additionally, we discover a reoccurring pattern in all of our case study models which mainly differed in utilized mixins and their configuration. Combined with the extension based structure of the standard, students may incrementally and interactively learn about the individual capabilities of OCCI and the cloud domain.

9.2 Model Viewpoints

The pragmatic feature of a model [33] describes that the usefulness is partially dictated by the user of the model. In this section, we discuss the potential usefulness of the workflow runtime model from two different perspectives. First, from the perspective of the scientist and cloud architect which observe and interact with the runtime state of the workflow, and second, from the self-adaptive control loop utilizing the runtime model as a knowledge base and interface.

9.2.1 Workflow and Cloud Architect

From a workflow and cloud architect point of view, the runtime model supports a human-in-the-loop that can interact with the states and attributes of the modeled elements. During the development of our case study models this capability allowed us to test and interact with individual parts of cloud and workflow deployments. Especially the visualization of resource states helped us during the development of the case study models as failing nodes were directly highlighted. To actively cooperate with the workflow at runtime, parameters in the individual entities can be changed to incorporate, e.g., information gained based on intermediate results. For this, domain specific knowledge is required to adjust deployed applications or provisioned infrastructure as changes are directly propagated to the system. To potentially ease the integration of domain specific elements, the model-driven approach allows generating extensions which include pre-defined and highly abstracted actions that can be applied on individual elements. This may include, e.g., an action to automatically up or downscale a cluster application with the click of a button. To foster the interaction with the model, specialized model perspectives could be beneficial that focus on an individual user. For example, complete OCCI extensions can be faded out. This would allow hiding the workflow layer for the cloud architect and the platform layer for the scientist. Furthermore, elements such as applications and loops can be collapsed to reduce the amount of visualized entities within the model which we, e.g., use in the cluster scaling scenario (Section 7.1.2.3) and the software repository mining case study (Section 8.3.2.2). Next, we discuss the viewpoint of engines using the runtime model as an interface to the system to be managed.

9.2.2 Self-Adaptive Control Loops

From a system point of view, self-adaptive control loops benefit from a highly detailed runtime model. A highly detailed model results in more information available for the self-adaptive control loops and eventually more options to control the system. For example, in our cluster scaling scenario (Section 7.1), the addition of new workers and accompanied sensors can be programmatically orchestrated. However, the added sensors and monitoring instruments quickly clutter the visualization. This highlights the need for different runtime model viewpoints, especially when human users and systems utilize the same interface.

In our workflow engine we combine the information contained within the design time and runtime model. Therefore, it is worthwhile to discuss which of both models contains the more important information for attached systems. The design time model describes the high level goal to be achieved within the workflow. However, the runtime model may contain additional capabilities not intended by the design time model. In the proposed approach we value the information in the runtime model higher, as it allows other autonomic control loops to be attached to the process, e.g., to add scaling capabilities to databases. For example, within the repository mining process another control loop may observe the size of the MongoDB and scale it by adding additional VMs that are not considered in the design time model. Therefore, design time capabilities that introduce scalable parts are beneficial such as the ones provided by TOSCA. While approaches exist that allow scaling plans or groups to be modeled, behavioral models are not envisioned by the OCCI standard. Especially as individual activities do not refer to an actual cloud resource to be managed.

In the following, we discuss interoperability of OCCI to other standards and systems, as well as provide insights about lessons learned working with it.

9.3 Standard Conformity

In our studies, we work with the OCCI standard including its data model, uniform interface, and several extensions. Even though the OCCIWare ecosystem [123] introduces extended functionalities not directly part of the standard, like abstract data types, we only utilize OCCI's core features. In Section 9.3.1, we provide a short discussion about the versatility of the OCCI data model and its extension capabilities in regard to other existing cloud modeling approaches. Additionally, in Section 9.3.2, we provide lessons learned working with OCCI and give our recommendations on how to further improve the standard.

9.3.1 OCCI Interoperability

In our studies we create and work with multiple extensions for OCCI to further support the infrastructure, platform and workflow layer. Overall, the structure of OCCI resembles the one of a DAG, i.e., nodes connected by arcs. In case of this study, we use this structure to instantiate monitoring and workflow elements that are based on dedicated OCCI extensions. Within these extensions, we heavily utilize the capabilities provided by the OCCI mixin elements. The versatility of this element allows for the creation of annotation or tags that dynamically add information or capabilities to an entity at runtime.

In our computation cluster scaling study (Section 7.1) we demonstrate the feasibility of combining the management of different state of the art IaC technologies within a runtime model that can be managed over a single interface. As a result, our OCCI based approach allows cloud architects to choose the tool most suitable for a given project while keeping the same abstraction and orchestration procedure.

Similarly, in our standard interoperability study (Section 7.2), we investigate the extent to which the uniform OCCI interface can be used to manage cloud resources modeled with TOSCA. We show, that both the type and instance layer can be transformed from TOSCA to OCCI in order to be managed by the interface. Only few elements cannot be automatically mapped in this process such as scaling rules for which OCCI currently does not possess an extension. Overall, we demonstrate the feasibility of combining both standards to manage cloud systems allowing users to utilize the extensive design time model of TOSCA while relying on OCCI's uniform interface to manage actual resources in the cloud.

9.3.2 OCCI Recommendations

Even though the OCCI data model is versatile and can be extended for many use cases, the specification was not updated since 2016 and is in need of a revision to foster the reusability and interoperability of designed extensions. In this section, we discuss our recommendations on how the standard could be improved to allow for a more pleasant developer and end-user experience. Overall, we recommend an update of state descriptions and attributes, as well as standardized guidelines for upcoming extensions.

The state information within the individual components is crucial to correctly reflect the current state of a deployed application. While the enhanced platform extension [105] covers additional states for OCCI components, the error state is not detailed enough. For example, when not actively observing the model during the deployment it remains hidden how the component reached the error state. In this case, the only possibility is to trigger the undeploy action even though the component may be correctly deployed already. Therefore, a more detailed FSM would be beneficial to add to the standard that, e.g., supports hierarchical states and histories. Combined, these would allow performing rollbacks on failed actions and navigate to the last functional state.

To further sophisticate the notion of an OCCI based runtime model, it would be beneficial to add *runtime attributes* to the standard. This kind of attributes could be automatically detected and hidden from graphical user interfaces reducing the complexity the user is confronted with at design time. Furthermore, a mechanism that restricts adjustments to the model from certain entities would be beneficial, e.g., to scope the access of attached adaptation engines. Another major limitation observed while designing OCCI extensions is the rather simple attribute structure of OCCI that does not allow any kind of abstract data types. Thus, extensions are limited to attributes holding single values or values divided by a delimiter. This limits us in certain settings, e.g., to monitor only one attribute by each sensor in the monitoring extension, as well as utilize a delimited string in the loop element. The OCCIWare metamodel [123] already addresses this issue by proposing an extended OCCI metamodel introducing, e.g., abstract data types. However, these are not yet integrated into the standard's specification and would require accompanied reworks of the interface.

The extension capability of OCCI represents one of its biggest advantages, especially, with the availability of a model-driven toolchain, like OCCIWare [123], to design and automatically generate arbitrary extensions. A document covering best practices to extend the standard would support the interoperability of designed extensions and tooling. We especially recommend specifying a reading direction for links connecting two extensions. Contradicting reading directions of links connecting two layers can be observed, e.g., in MoDMaCAO [105] and the Docker extension [104]. In MoDMaCAO, a platform component is *placed* on a compute node having the link going from the platform to the infrastructure extension. In the Docker extension, the compute node *contains* a container directing from the OCCI infrastructure to the Docker extension. Unifying reading directions is of importance for surrounding tooling as links are typically traversed to access the information contained within connected elements. A uniform reading direction would improve the reusability and interoperability of designed OCCI extensions while reducing the need for implementation adjustments and an improved navigation within the model.

9.4 Threats to Validity

In this section, we discuss the threats to validity following the criteria by Wohlin et al. [245]. Section 9.4.1 covers the construct validity, Section 9.4.2 the internal validity, and Section 9.4.3 the external validity of our study.

9.4.1 Construct Validity

Construct validity refers to the extent to which a study conforms with its investigation intents [245]. In our case, this aspect of validity is concerned with the extent to which we conform to the OCCI standard, including its data model, interface, and utilization as a runtime model.

To perform our study, we designed the extension solely relying on the classification and identification mechanisms provided by the OCCI core specification [98] that is implemented as an EMF metamodel [122]. Even though this metamodel was later enhanced as part of the OCCIWare toolchain [123], we restricted our workflow and monitoring extension design to only use elements available in the OCCI specification, such as simple attributes. Our adaptation engine is designed to perform model transformations based on this meta information and derive required OCCI requests. Throughout the course of our studies, we test two OCCI interfaces that conform to the specification. In the first version of this engine, the generated OCCI requests were sent against the OOI [222], before we later on switched to the OCCIWare runtime server that provides a better compatibility with generated extensions. Even though the workflow execution is only tested in the OCCIWare environment, we evaluate the generation of requests for multiple implementations of the standardized interface. Therefore, we mitigate the threat of our implementation not conforming to the specification of the standard's interface.

9.4.2 Internal Validity

Internal validity is concerned with causal relations examined by the study [245]. As our study focuses on the feasibility and applicability of our approach, the internal validity is concerned with threats about our recorded timing measurements and diversity of case studies.

To mitigate the threat of hidden influencing variables, we created multiple environments to perform the case studies in isolation. Furthermore, to cope with changing workloads in our cloud environment we performed each study multiple times to derive our presented timing measurements. It should be noted, that the observed timings originate from an implemented research prototype used to investigate the feasibility of the runtime model approach. Thus, the implementation and timings can benefit from optimized application and infrastructure scheduling mechanisms. To investigate the extent of a model-driven management overhead, we measured the time required for communications of our engines with the cloud and compared it against the time requirements of the performed required runtime model transformations. By performing a variety of different workflow domains and workflow sizes, we mitigate the influence of highly similar model structures.

9.4.3 External Validity

External validity is concerned with the general applicability of an approach, so that observed results not only apply on settings of a specific environment [245]. In our approach the external validity is concerned with the selected case studies and the extent to which they underline the generalizability of our approach.

In this thesis, we describe two orchestration and three workflow scenarios in detail. We chose the presented scenarios from different domains to examine the general applicability of the OCCI orchestration process and the workflow runtime model. Furthermore, we chose the case studies to cover workflow models of different sizes, while describing its main features including its decision-making process, loops, and data flows between distributed hosts. While only one of the orchestration case studies utilized a scaling engine for the deployment, the orchestration capabilities are demonstrated within the workflow scenarios.

Another threat to the external validity of our study is the amount of virtualization techniques and cloud providers used. To perform our case studies, we utilize a private OpenStack cloud. Even though the proposed approach is only tested on a single cloud provider, the utilization of the OCCI cloud standard allows incorporating other provider specific interfaces. Furthermore, our provisioned computation resources mostly referred to VMs even though the use of containers could speed up the provisioning process. However, for the demonstration of the usefulness of runtime models for workflow execution the utilized virtualization technique is negligible.

10 Conclusion

In this section, we conclude the thesis. Overall, we discover that a model-driven cloud orchestration process based on OCCI is feasible to fulfill arbitrary and shifting resource requirements during workflow executions. While the OCCI standard serves as baseline concept of our study, we implement our workflow and orchestration concepts within the publicly available SmartWYRM framework [228]. Hereby, we design the framework to integrate seamlessly into the existing OCCIWare ecosystem [123]. In the following, we provide a summary of both the orchestration and workflow contributions. Moreover, we present an outlook into future work.

10.1 Summary

In this thesis, we investigate the orchestration capabilities of OCCI and the utilization of a workflow runtime model on top of it. Our model-driven cloud orchestration compares the state of a running cloud application to a desired one and performs adaptive actions by sending OCCI conform requests. During this process, the current state of the cloud is reflected as a runtime model which we utilize in a chain of model transformations. In addition to a standard conform orchestration process, we extend existing platform management solutions to combine configuration and container management within the OCCI runtime model. Furthermore, we increase the self-reflectiveness of OCCI by designing an extension which introduces monitoring capabilities that allow reflecting operational properties directly in the runtime model.

To demonstrate the generalizability of our adaption procedure and the capabilities of an OCCI runtime model, we performed two case studies. We selected the case studies to highlight the usefulness of a runtime model as a knowledge base, the interoperability of our orchestration approach to the OCCI and the TOSCA standard, as well as the possibility for the utilization of a simulation environment. Our studies reveal that the utilization of a standard conform runtime model possesses several advantages. Practitioners can directly interact with the system over a highly abstracted representation of it. We observe that the causal connection of the system and the model, as well as its visual representation supports the development process of, e.g., configuration management scripts or scaling engines. By attaching the runtime model to a local simulation environment cloud developers can be supported as planned adaptations can be designed and tested before getting applied on a production environment.

Moreover, we utilize the advantages of the model-driven cloud orchestration to design and dynamically shift infrastructures throughout the execution of scientific workflows. We demonstrate the feasibility of a runtime model to couple individual workflow tasks with tailored infrastructure and application requirements. To realize this concept, we create an OCCI workflow extension that connects to the standardized OCCI platform and infrastructure extension. To study the extent to which a runtime workflow model can be beneficial for the execution of workflows, we develop concepts for typical workflow functionalities and couple them with the infrastructure aware capabilities of the runtime model. Among others, this includes the integration of data flows, dynamic decision-making points, as well as the modeling of loops and parallelization techniques.

To discuss the extent to which a workflow execution may benefit from an infrastructure aware runtime model, we performed several case studies from different domains. In this thesis, we selected three of them to highlight the individual functionalities offered by the runtime model. Throughout the execution of the study, we discover that the runtime model provides a prominent knowledge base that can be manipulated over the adaptation process described in this thesis. Therefore, we are able to schedule required resources solely relying on model transformations forming the next required runtime state. Thus, the overall workflow execution can be described over a set of changing runtime model states that are easy to observe. Furthermore, these states support the development process of workflows as they can be easily extracted, e.g., for test purposes. Overall, the utilization of a standard conform runtime model for workflows possesses several benefits for scientists. The causal connection of the runtime model supports direct interaction with the system over an abstract representation fostering the integration of a human-in-the-loop. This allows scientist to inspect intermediate results and, e.g., change the active control flow or adjust parallelization values to trade of time for resources. Furthermore, as the workflow dynamically spawns the modeled infrastructure, no preconfigured computation clusters are required, allowing for an easier replication of the study. Especially the utilization of an open standard encourages the active exchange of workflow and infrastructure extension from different domains and communities.

10.2 Outlook

The concepts and implementations presented in this thesis demonstrate the feasibility of a workflow runtime model and open up further research opportunities. In this section, we discuss future work regarding a model-driven cloud orchestration and workflow management.

To foster the use of standards for cloud orchestration, an automated extraction mechanisms may be beneficial that builds a standardized runtime model from a currently running cloud deployment. While a derivation of infrastructure resources can be reached by requesting provider specific interfaces, the replication of deployed configuration remains open and requires further investigations, e.g., by checking for certain services which could be reflected in the runtime model. Ultimately, the derived model can then be deployed at the site of a new cloud provider using our orchestration engine. Additionally, this allows utilizing the benefits of the standards' ecosystem. Moreover, an update of the standard's specification would be worthwhile to include, e.g., abstract data types, improved state machine definitions. Also, further specifications may be introduced that cover best practices to design extensions, or collections of mappings between OCCI and, e.g., Amazon EC2, Microsoft Azure and the Google Cloud Platform.

To further support the development of cloud applications and the replication of cloud research, the local simulation environment can be extended to support more realistic simulation scenarios. In future work, the utilized simulation can be extended by adding tags to test the robustness of cloud applications, e.g., by injecting faults into existing networks. Moreover, to improve the usability of the simulation, size proportions could be derived to automatically fit the model to the local environment.

In addition to an improved simulation extension, the workflow runtime model may benefit from several extensions introducing, e.g., task estimation. Also, a direct mapping of data to the infrastructure storing it may be beneficial to support the provenance of processed data. Even further infrastructure types may be considered and evaluated to, e.g., integrate HPC systems within the workflow runtime model. Additionally, we plan to investigate the extent to which OCCI supports actions that heavily adjust the runtime model, e.g., to automatically up or downscale a cluster application. Furthermore, we plan to incorporate real world sensors in the runtime model to dynamically spawn virtualized computing resources based on the sensor results observed within a workflow.

Apart from several functionalities that may be added to the runtime model, one major area of interest is the evaluation of suitable views on the model. As a cloud runtime model provides the capability to directly interact and reflect the current system, it represents a point of access for several stakeholders. In our case, these comprise the cloud architect interested in the infrastructure, the scientist designing the workflow, as well as the self-adaptive control loops managing the system. All views, however, have totally different requirements on the model regarding its level of abstraction. Therefore, as future work, an investigation of individual and interoperable views on the runtime model system is required, as well as mechanisms to seal off the access of different parts of the model from specific users.

Bibliography

[1] Y. Zhao, Y. Li, I. Raicu, S. Lu, W. Tian, and H. Liu, "Enabling scalable scientific workflow management in the cloud," *Future Generation Computer Systems*, vol. 46, pp. 3–16, 2015.

[2] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528 – 540, 2009.

[3] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, vol. 13, no. 4, pp. 457–493, 2015.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," *Electrical Engineering and Computer Sciences, University of California at Berkeley*, 2009.

[6] Organization for the Advancement of Structured Information Standards, "Topology and Orchestration Specification for Cloud Applications," 2013, Available online: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca, last retrieved: 31.05.2022.

[7] Open Grid Forum, "Open Cloud Computing Interface," 2010, Available online: http://occi-wg.org/, last retrieved: 31.05.2022.

[8] J. Kovács and P. Kacsuk, "Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures," *Journal of Grid Computing*, vol. 16, no. 1, pp. 19–37, 2018.

[9] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[10] N. Bencomo, S. Götz, and H. Song, “Models@run.time: a guided tour of the state of the art and research challenges,” *Software & Systems Modeling*, vol. 18, no. 5, pp. 3049–3082, 2019.

[11] J. Erbel, “Runtime Workflow Modelling,” 2021, Available online: https://doi.org/10.25625/OE4AN3 or https://gitlab.gwdg.de/rwm, last retrieved: 31.05.2022.

[12] D. Georgakopoulos, M. Hornick, and A. Sheth, “An overview of workflow management: From process modeling to workflow automation infrastructure,” *Distributed and parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.

[13] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, and J. Hua, “A reference architecture for scientific workflow management systems and the view soa solution,” *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 79–92, 2009.

[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the kepler system,” *Concurrency and computation: Practice and experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[15] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers, “Scientific workflows: Business as usual?” in *Proceedings of the 7th International Conference on Business Process Management (BPM)*. Springer Berlin Heidelberg, 2009.

[16] N. Cerezo, J. Montagnat, and M. Blay-Fornarino, “Computer-assisted scientific workflow design,” *Journal of grid computing*, vol. 11, no. 3, pp. 585–612, 2013.

[17] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, and M. Reiter, *Conventional Workflow Technology for Scientific Simulation*. Springer London, 2011, pp. 323–352.

[18] M. Sonntag, D. Karastoyanova, and F. Leymann, “The missing features of workflow systems for scientific computations,” in *Proceedings of the 3rd Grid Workflow Workshop (GWW)*, 2010.

[19] J. Bang-Jensen and G. Z. Gutin, *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2000.

[20] R. Diestel, *Graph Theory: 5th edition*, ser. Springer Graduate Texts in Mathematics. Springer-Verlag, 2017.

[21] C. A. Petri, “Kommunikation mit Automaten,” Dissertation, Technische Hochschule Darmstadt, 1962.

[22] N. R. Adam, V. Atluri, and W.-K. Huang, “Modeling and analysis of workflows using petri nets,” *Journal of Intelligent Information Systems*, vol. 10, no. 2, pp. 131–158, 1998.

[23] K. Salimifard and M. Wright, “Petri net-based modelling of workflow systems: An overview,” *European journal of operational research*, vol. 134, no. 3, pp. 664–676, 2001.

[24] Object Management Group, “Unified Modeling Language,” 2015, Available online: http://www.omg.org/spec/UML/2.5/PDF, last retrieved: 31.05.2022.

[25] ——, “OMG: Business Process Model and Notation,” 2011, Available online: http://www.omg.org/spec/BPMN/2.0/PDF, last retrieved: 31.05.2022.

[26] T. Kühne, “Matters of (meta-) modeling,” *Software & Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[27] E. Seidewitz, “What Models Mean,” *IEEE software*, vol. 20, no. 5, pp. 26–32, 2003.

[28] A. R. Da Silva, “Model-driven engineering: A survey supported by the unified conceptual model,” *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015.

[29] International Conference on Model Driven Engineering Languages and Systems, “Industry Day,” 2018, Available online: https://modelsconf2018.github.io/program/industryday/, last retrieved: 31.05.2022.

[30] M. Szvetits and U. Zdun, “Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime,” *Software & Systems Modeling*, 2016.

[31] Object Management Group, “MDA Guide Version 1.0.1,” 2003, Available online: http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf, last retrieved: 31.05.2022.

[32] J. Bézivin, “In Search of a Basic Principle for Model Driven Engineering,” *Novatica Journal, Special Issue*, vol. 5, no. 2, pp. 21–24, 2004.

[33] H. Stachowiak, *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[34] Object Management Group, “Unified Modeling Language Infrastructure Specification,” 2011, Available online: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF, last retrieved: 31.05.2022.

[35] J.-M. Favre, “Towards a Basic Theory to Model Model Driven Engineering,” in *Proceedings of the 3rd UML Workshop in Software Model Engineering (WiSME)*, 2004.

[36] Object Management Group, “Object Constraint Language,” 2016, Available online: http://www.omg.org/spec/OCL/2.4/PDF/, last retrieved: 31.05.2022.

[37] ——, "OMG Meta Object Facility (MOF) Core Specification," 2016, Available online: http://www.omg.org/spec/MOF/2.5.1/PDF/, last retrieved: 31.05.2022.

[38] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[39] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007.

[40] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. Polack, "The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering," in *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2009.

[41] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The epsilon object language (eol)," in *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg, 2006.

[42] E. Syriani, J. Gray, and H. Vangheluwe, "Modeling a model transformation language," in *Domain Engineering*. Springer, 2013, pp. 211–237.

[43] C. Gomes, B. Barroca, and V. Amaral, "Classification of model transformation tools: Pattern matching techniques," in *Proceedings of the 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer International Publishing, 2014.

[44] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[45] Object Management Group, "OMG Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification," 2016, Available online: https://www.omg.org/spec/QVT/1.3/PDF, last retrieved: 31.05.2022.

[46] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer, 2008.

[47] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: A qvt-like transformation language," in *Proceedings of the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Association for Computing Machinery, 2006.

[48] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1, pp. 31–39, 2008.

[49] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the viatra framework," *Software & Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016.

[50] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro, "Viatra - visual automated transformations for formal verification and validation of uml models," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE)*, 2002.

[51] D. Varró and A. Balogh, "The model transformation language of the viatra2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214–234, 2007.

[52] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró, "Viatra 3: A reactive model transformation platform," in *Proceedings of the 8th International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer International Publishing, 2015.

[53] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "The epsilon generation language," in *Proceedings of 4th the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*. Springer Berlin Heidelberg, 2008.

[54] Eclipse Foundation, "Acceleo," 2020, Available online: https://www.eclipse.org/acceleo/, last retrieved: 31.05.2022.

[55] F. Truyen, "The fast guide to model driven architecture the basics of model driven architecture: The basics of model driven architecture," *Whitepaper; Cephas Consulting Corp, Architecture Oriented Services*, 2006.

[56] N. Bencomo, G. Blair, S. Götz, B. Morin, and B. Rumpe, "Report on the 7th international workshop on models@run.time," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 1, p. 27–30, 2013.

[57] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[58] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. Association for Computing Machinery, 1987.

[59] S. Nordstrom, A. Dubey, T. Keskinpala, R. Datta, S. Neema, and T. Bapty, "Model predictive analysis for autonomic workflow management in large-scale scientific computing environments," in *Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASE)*, 2007.

[60] G. Waignier, A.-F. Le Meur, and L. Duchien, “A Model-Based Framework to Design and Debug Safe Component-Based Autonomic Systems,” in *Proceedings of the 5th International Conference on the Quality of Software Architectures (QoSA).* Springer, 2009.

[61] J. O. Kephart and D. M. Chess, “The Vision of Autonomic Computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[62] R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, *Software Engineering for Self-Adaptive Systems III. Assurances.* Springer International Publishing, 2017.

[63] I. B. M. Corporation, “An architectural blueprint for autonomic computing,” *IBM White paper*, 2005.

[64] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, “Engineering self-adaptive systems through feedback loops: Software engineering for self-adaptive systems,” in *Software Engineering for Self-Adaptive Systems*, 2009.

[65] D. Weyns, “Software engineering of self-adaptive systems: An organised tour and future challenges,” in *Handbook of Software Engineering*, 2018.

[66] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[67] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” *National Institute of Standards and Technology*, 2011.

[68] Amazon, “Amazon Web Services Elastic Compute Cloud,” Available online: https://aws.amazon.com/ec2/, last retrieved: 31.05.2022.

[69] S. Kächele, C. Spann, F. J. Hauck, and J. Domaschka, “Beyond iaas and paas: An extended cloud taxonomy for computation, storage and networking,” in *Proceedings of the 6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2013.

[70] Google, “GSuite,” Available online: https://gsuite.google.com/index.html, last retrieved: 31.05.2022.

[71] ——, “Google App Engine,” Available online: https://cloud.google.com/appengine/, last retrieved: 31.05.2022.

[72] Amazon, “Amazon Web Services Elastic Beanstalk,” Available online: https://aws.amazon.com/elasticbeanstalk/, last retrieved: 31.05.2022.

[73] OpenStack, "Newton," 2016, Available online: https://releases.openstack.org/newton/, last retrieved: 31.05.2022.

[74] Y. Brikman, *Terraform: Up & Running: Writing Infrastructure as Code*. O'Reilly Media, 2019.

[75] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez, "Cloud resource orchestration: A data-centric approach," in *Proceedings of the 5th biennial Conference on Innovative Data Systems Research (CIDR)*. Citeseer, 2011.

[76] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and J. Domaschka, "Cloud orchestration features: Are tools fit for purpose?" in *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2015.

[77] K. Morris, *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[78] HashiCorp, "Terraform," 2021, Available online: https://www.terraform.io/, last retrieved: 31.05.2022.

[79] Amazon Web Services, "Cloudformation," 2020, Available online: https://aws.amazon.com/de/cloudformation/, last retrieved: 31.05.2022.

[80] OpenStack, "Heat - 15.0.0," 2020, Available online: https://wiki.openstack.org/wiki/Heat/, last retrieved: 31.05.2022.

[81] GNU Project, "Bourne-again shell," 2021, Available online: https://www.gnu.org/software/bash/, last retrieved: 31.05.2022.

[82] Python Software Foundation, "Python," 2021, Available online: https://www.python.org/, last retrieved: 31.05.2022.

[83] M. Heap, *Ansible: from beginner to pro*. Springer, 2016.

[84] Progress, "Chef," 2021, Available online: https://www.chef.io/, last retrieved: 31.05.2022.

[85] Puppet, "Puppet," 2021, Available online: https://puppet.com/, last retrieved: 31.05.2022.

[86] Red Hat, "Ansible," 2021, Available online: https://www.ansible.com/, last retrieved: 31.05.2022.

[87] VMWare, "SaltStack," 2021, Available online: https://www.vmware.com/support/acquisitions/saltstack.html, last retrieved: 31.05.2022.

[88] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *Proceedings of the 14th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*. Springer Berlin Heidelberg, 2013.

[89] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, and K. Tarabanis, "Towards a reference architecture for semantically interoperable clouds," in *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010.

[90] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 1–11, 2009.

[91] P. Hofmann and D. Woods, "Cloud computing: The limits of public clouds for business applications," *IEEE Internet Computing*, vol. 14, no. 6, pp. 90–93, 2010.

[92] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2010.

[93] G. C. Silva, L. M. Rose, and R. Calinescu, "A systematic review of cloud lock-in solutions," in *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2013.

[94] Cloud Standards Customer Council, "Interoperability and portability for cloud computing: A guide version 2.0," 2017, Available online: https://www.omg.org/cloud/deliverables/CSCC-Interoperability-and-Portability-for-Cloud-Computing-A-Guide.pdf, last retrieved: 31.05.2022.

[95] Storage Networking Industry Association, "Cloud data management interface (cdmi) version 2.0.0," 2020, Available online: https://www.snia.org/sites/default/files/technical_work/CDMI/CDMI_v2.0.0.pdf, last retrieved: 31.05.2022.

[96] European Telecommunications Standards Institute, "Etsi ts 103 142 v1.1.1 cloud; test descriptions for cloud interoperability," 2013, Available online: https://www.etsi.org/deliver/etsi_ts/103100_103199/103142/01.01.01_60/ts_103142v010101p.pdf, last retrieved: 31.05.2022.

[97] Organization for the Advancement of Structured Information Standards, "Cloud Application Management for Platforms Version 1.2," 2018, Available online: http://docs.oasis-open.org/camp/camp-spec/v1.2/cs01/camp-spec-v1.2-cs01.pdf, last retrieved: 31.05.2022.

[98] Open Grid Forum, “Open Cloud Computing Interface - Core,” 2016, Available online: https://www.ogf.org/documents/GFD.221.pdf, last retrieved: 31.05.2022.

[99] ——, “Open Cloud Computing Interface - HTTP Protocol,” 2016, Available online: https://www.ogf.org/documents/GFD.223.pdf, last retrieved: 31.05.2022.

[100] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[101] Open Grid Forum, “Open Cloud Computing Interface - Text Rendering,” 2016, Available online: https://www.ogf.org/documents/GFD.229.pdf, last retrieved: 31.05.2022.

[102] ——, “Open Cloud Computing Interface - JSON Rendering,” 2016, Available online: https://www.ogf.org/documents/GFD.226.pdf, last retrieved: 31.05.2022.

[103] ——, “Open Cloud Computing Interface - Infrastructure,” 2016, Available online: https://www.ogf.org/documents/GFD.224.pdf, last retrieved: 31.05.2022.

[104] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, “Model-Driven Management of Docker Containers,” in *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD)*, 2016.

[105] F. Korte, S. Challita, F. Zalila, P. Merle, and J. Grabowski, “Model-driven configuration management of cloud applications with occi,” in *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)*, 2018.

[106] Open Grid Forum, “Open Cloud Computing Interface - Platform,” 2016, Available online: https://www.ogf.org/documents/GFD.227.pdf, last retrieved: 31.05.2022.

[107] Organization for the Advancement of Structured Information Standards, “Topology and Orchestration Specification for Cloud Applications Version 1.0.” November 2013, Available online: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, last retrieved: 31.05.2022.

[108] ——, “TOSCA Simple Profile in YAML Version 1.3,” February 2020, Available online: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html, last retrieved: 31.05.2022.

[109] Canonical, “Juju,” 2021, Available online: https://juju.is/, last retrieved: 31.05.2022.

[110] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel, “UML-based Cloud Application Modeling with Libraries, Profiles, and Templates,” in *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, 2014.

[111] A. Kamali, S. Mohammadi, and A. A. Barforoush, "UCC: UML profile to cloud computing modeling: Using stereotypes and tag values," in *Proceedings of the 7th International Symposium on Telecommunications (IST)*. IEEE, 2014.

[112] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, "A UML Profile for Modeling Multicloud Applications," in *Proceedings of the 2nd European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer, 2013.

[113] N. Ferry, G. Brataas, A. Rossini, F. Chauvel, and A. Solberg, "Towards bridging the gap between scalability and elasticity," in *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER)*, 2014.

[114] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "Cloudmf: Model-driven management of multi-cloud applications," *ACM Transactions on Internet Technology*, vol. 18, no. 2, pp. 1–24, 2018.

[115] A. P. Achilleos, K. Kritikos, A. Rossini, G. M. Kapitsaki, J. Domaschka, M. Orzechowski, D. Seybold, F. Griesinger, N. Nikolov, D. Romero *et al.*, "The cloud application modelling and execution language," *Journal of Cloud Computing*, vol. 8, no. 1, p. 20, 2019.

[116] D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka, "A provider-agnostic approach to multi-cloud orchestration using a constraint language," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Press, 2018.

[117] J. Sandobalin, E. Insfran, and S. Abrahao, "On the effectiveness of tools to support infrastructure as code: Model-driven versus code-centric," *IEEE Access*, vol. 8, pp. 17 734–17 761, 2020.

[118] J. Sandobalin, E. Insfran, S. Abrah *et al.*, "A model-driven migration approach among cloud providers," in *Proceedings of the XIV Jornadas de Ciencia e Ingeniería de Servicios (JCIS)*. SISTEDES, 2018.

[119] C. Quinton, N. Haderer, R. Rouvoy, and L. Duchien, "Towards multi-cloud configurations using feature models and ontologies," in *Proceedings of the 1st International Workshop on Multi-Cloud Applications and Federated Clouds (MultiCloud)*. Association for Computing Machinery, 2013.

[120] C. Quinton, D. Romero, and L. Duchien, "Saloon: a platform for selecting and configuring cloud environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016.

[121] OCCIWare - Project, "A formal framework for the management of any digital resource in the cloud," 2014 - 2017, Available online: http://occiware.github.io/, last retrieved: 31.05.2022.

[122] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata, "A Precise Metamodel for Open Cloud Computing Interface," in *Proceedings of 8th IEEE International Conference on Cloud Computing (CLOUD)*, 2015.

[123] F. Zalila, S. Challita, and P. Merle, "A model-driven tool chain for OCCI," in *Proceedings of the 25th International Conference on Cooperative Information Systems (CoopIS)*, 2017.

[124] Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle, "Model-driven elasticity management with occi," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.

[125] M. Ahmed-Nacer, S. Kallel, F. Zalila, P. Merle, and W. Gaaloul, "Model-Driven Simulation of Elastic OCCI Cloud Resources," *The Computer Journal*, 2020.

[126] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, p. 23–50, 2011.

[127] Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle, "Model-driven elasticity management with occi," *IEEE Transactions on Cloud Computing*, vol. 9, no. 4, pp. 1549–1562, 2021.

[128] P. Merle, C. Gourdin, and N. Mitton, "Mobile cloud robotics as a service with occiware," in *Proceedings of the 2nd IEEE International Congress on Internet of Things (ICIOT)*, 2017.

[129] S. Yangui and S. Tata, "Cloudserv: Paas resources provisioning for service-based applications," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2013.

[130] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, "Paas-independent provisioning and management of applications in the cloud," in *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*, 2013.

[131] S. Yangui and S. Tata, "An OCCI Compliant Model for PaaS Resources Description and Provisioning," *The Computer Journal*, vol. 59, no. 3, pp. 308–324, 2014.

[132] A. Ciuffoletti, "A simple and generic interface for a cloud monitoring service," in *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER)*, 2014.

[133] M. Mohamed, D. Belaid, and S. Tata, “Monitoring and reconfiguration for occi resources,” in *Proceedings of the 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2013.

[134] J. Bellendorf and Z. A. Mann, “Specification of cloud topologies and orchestration using tosca: a survey,” *Computing*, pp. 1–23, 2019.

[135] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, “Winery–a modeling tool for tosca-based cloud applications,” in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2013.

[136] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and D. Schumm, “Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA,” in *Proceedings of the Confederated International Conferences "On the Move to Meaningful Internet Systems" (OTM)*. Springer, 2012.

[137] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, and J. W. M. Zimmermann, “The OpenTOSCA Ecosystem –Concepts & Tools,” *European Space project on Smart Systems, Big Data, Future Internet -Towards Serving the Grand Societal Challenges*, pp. 112–130, 2016.

[138] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, “Combining declarative and imperative cloud application provisioning based on tosca,” in *Proceedings of the 2nd IEEE International Conference on Cloud Engineering (IC2E)*, 2014.

[139] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, and M. Zimmermann, “Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA,” in *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER)*, 2014.

[140] N. Loulloudes, C. Sofokleous, D. Trihinas, M. D. Dikaiakos, and G. Pallis, “Enabling Interoperable Cloud Application Management through an Open Source Ecosystem,” *IEEE Internet Computing*, vol. 19, no. 3, pp. 54–59, 2015.

[141] Alien4Cloud - Project, “Application LIfecycle ENablement for Cloud,” 2021, Available online: https://alien4cloud.github.io, last retrieved: 31.05.2022.

[142] Cloudify, “Version 5.1,” 2021, Available online: https://cloudify.co/, last retrieved: 31.05.2022.

[143] J. Carrasco, J. Cubo, and E. Pimentel, “Towards a Flexible Deployment of Multi-Cloud Applications Based on TOSCA and CAMP,” in *Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer, 2014.

[144] J. Carrasco, J. Cubo, E. Pimentel, and F. Durán, “Deployment over Heterogeneous Clouds with TOSCA and CAMP,” in *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER)*, 2016.

[145] J. Carrasco, F. Durán, and E. Pimentel, “Trans-cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud,” *Computer Standards & Interfaces*, vol. 58, pp. 167–179, 2018.

[146] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, “Tosca light: Bridging the gap between the tosca specification and production-ready deployment technologies,” in *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER)*, 2020.

[147] Cloud Native Computing Foundation, “Kubernetes,” 2021, Available online: https://kubernetes.io/, last retrieved: 31.05.2022.

[148] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, and J. Soldani, “Tosca lightning: An integrated toolchain for transforming tosca light into production-ready deployment technologies,” in *Proceedings of 32nd International Conference on Advanced Information Systems Engineering (CAiSE).* Springer International Publishing, 2020.

[149] M. Artač, T. Borovšak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, “Model-driven continuous deployment for quality devops,” in *Proceedings of the 2nd International Workshop on Quality-Aware DevOps (QUDOS).* Association for Computing Machinery, 2016.

[150] J. DesLauriers, T. Kiss, R. C. Ariyattu, H.-V. Dang, A. Ullah, J. Bowden, D. Krefting, G. Pierantoni, and G. Terstyanszky, “Cloud apps to-go: Cloud portability with tosca and micado,” *Concurrency and Computation: Practice and Experience*, p. e6093, 2020.

[151] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani, “The Essential Deployment Metamodel: a Systematic Review of Deployment Automation Technologies,” *SICS Software-Intensive Cyber-Physical Systems*, pp. 1–13, 2019.

[152] V. Andrikopoulos, A. Reuter, S. G. Sáez, and F. Leymann, “A GENTL Approach for Cloud Application Topologies,” in *Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC).* Springer, 2014.

[153] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann, “From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA,” in *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER)*, 2016.

[154] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann *et al.*, "Automatic Topology Completion of TOSCA-based Cloud Applications," in *Proceedings of the 44th Jahrestagung der Gesellschaft für Informatik (INFORMATIK)*, 2014.

[155] H. Brabra, A. Mtibaa, W. Gaaloul, B. Benatallah, and F. Gargouri, "Model-driven orchestration for cloud resources," in *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*, 2019.

[156] A. Tsagkaropoulos, Y. Verginadis, M. Compastié, D. Apostolou, and G. Mentzas, "Extending tosca for edge and fog deployment support," *Electronics*, vol. 10, no. 6, p. 737, 2021.

[157] D. A. Tamburri, W.-J. Van den Heuvel, C. Lauwers, P. Lipton, D. Palma, and M. Rutkowski, "Tosca-based intent modelling: goal-modelling for infrastructure-as-code," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2, pp. 163–172, 2019.

[158] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo *et al.*, "Radon: rational decomposition and orchestration for serverless computing," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1, pp. 77–87, 2020.

[159] R. Qasha, J. Cala, and P. Watson, "Dynamic deployment of scientific workflows in the cloud using container virtualization," in *Proceedings of the 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016.

[160] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[161] L. Ramakrishnan, S. Poon, V. Hendrix, D. Gunter, G. Z. Pastorello, and D. Agarwal, "Experiences with user-centered design for the tigres workflow api," in *Proceedings of the 10th IEEE International Conference on e-Science (e-Science)*, 2014.

[162] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*. IEEE, 2004.

[163] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan, "The trident scientific workflow workbench," in *In Proceedings of the 4th IEEE International Conference on e-Science (e-Science)*, 2008.

[164] P. Bui, L. Yu, and D. Thain, "Weaver: Integrating distributed computing abstractions into scientific workflows using python," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[165] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming scientific and distributed workflow with triana services," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1021–1037, 2006.

[166] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.

[167] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[168] J. Goecks, A. Nekrutenko, J. Taylor, G. Team *et al.*, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[169] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin *et al.*, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

[170] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic Acids Research*, vol. 41, no. W1, pp. W557–W561, 2013.

[171] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Enabling interactive multiple-view visualizations," in *Proceedings of the 16th IEEE Conference on Visualization (VIS)*, 2005.

[172] S. G. Parker and C. R. Johnson, "Scirun: a scientific programming environment for computational steering," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC)*, 1995.

[173] R. Bergmann and Y. Gil, "Similarity assessment and efficient retrieval of semantic workflows," *Information Systems*, vol. 40, pp. 115–127, 2014.

[174] A. Ter Hofstede and W. Van der Aalst, “Yawl: yet another workflow language,” *Information Systems*, vol. 30, pp. 245–275, 2005.

[175] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. Van Dam, “The application visualization system: A computational environment for scientific visualization,” *IEEE Computer Graphics and Applications*, vol. 9, no. 4, pp. 30–42, 1989.

[176] A. Schleicher and B. Westfechtel, “Beyond stereotyping: Metamodeling approaches for the uml,” in *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2001.

[177] R. Braun and W. Esswein, “Classification of domain-specific bpmn extensions,” in *Proceedings of the 7th IFIP Working Conference on The Practice of Enterprise Modeling (PoEM)*. Springer Berlin Heidelberg, 2014.

[178] D. Calcaterra, V. Cartelli, G. Di Modica, and O. Tomarchio, “Combining tosca and bpmn to enable automated cloud service provisioning,” in *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*, 2017.

[179] ——, “Exploiting bpmn features to design a fault-aware tosca orchestrator,” in *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER)*, 2018.

[180] J. Brüning and M. Gogolla, “Uml metamodel-based workflow modeling and execution,” in *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference (EDOCW)*, 2011.

[181] J. Brüning, M. Gogolla, and P. Forbrig, “Modeling and formally checking workflow properties using uml and ocl,” in *Proceedings on the 9th International Conference on Perspectives in Business Informatics Research (BIR)*. Springer Berlin Heidelberg, 2010.

[182] M. Gogolla, F. Büttner, and M. Richters, “Use: A uml-based specification environment for validating uml and ocl,” *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.

[183] R. Ramdoyal, C. Ponsard, M.-A. Derbali, G. Schwanen, I. Linden, and J.-M. Jacquet, “A generic workflow metamodel to support resource-aware decision making,” in *Proceedings of the 15th International Conference on Enterprise Information Systems (ICEIS)*, 2013.

[184] N. Cerezo and J. Montagnat, “Scientific workflow reuse through conceptual workflows on the virtual imaging platform,” in *Proceedings of the 6th workshop on Workflows in support of large-scale science (WORKS)*, 2011.

[185] G. Scherp, “A Framework for Model-Driven Scientific Workflow Engineering,” Ph.D. dissertation, Christian-Albrechts-University, Kiel, 2013.

[186] G. Scherp and W. Hasselbring, “Towards a model-driven transformation framework for scientific workflows,” *Procedia Computer Science*, vol. 1, no. 1, pp. 1519–1526, 2010.

[187] Organization for the Advancement of Structured Information Standards, “Web services business process execution language version 2.0,” 2007, Available online: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, last retrieved: 31.05.2022.

[188] R. Espinosa, D. García-Saiz, M. Zorrilla, J. J. Zubcoff, and J.-N. Mazón, “S3mining: A model-driven engineering approach for supporting novice data miners in selecting suitable classifiers,” *Computer Standards & Interfaces*, vol. 65, pp. 143–158, 2019.

[189] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, “The generic modeling environment,” in *Proceedings of the IEEE International Workshop on Intelligent Signal Processing (WISP)*, 2001.

[190] G. Vossen and M. Weske, *The WASA Approach to Workflow Management for Scientific Applications*. Springer Berlin Heidelberg, 1998, pp. 145–164.

[191] M. Weske, G. Vossen, and C. B. Medeiros, *Scientific workflow management: WASA architecture and applications*. Citeseer, 1996.

[192] M. Weske, “Flexible modeling and execution of workflow activities,” in *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS)*. IEEE, 1998.

[193] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, “Condor: a distributed job scheduler,” in *Beowulf Cluster Computing with Windows*. MIT Press, 2001.

[194] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington, “Workflow enactment in iceni,” in *Proceedings of the UK e-Science All Hands Meeting*, 2004.

[195] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crumme *et al.*, “The grads project: Software support for high-level grid application development,” *The International Journal of High Performance Computing Applications*, vol. 15, no. 4, pp. 327–344, 2001.

[196] Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu, “Grid-flow: a grid-enabled scientific workflow system with a petri-net-based interface,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1115–1140, 2006.

[197] J. Almond and D. Snelling, "Unicore: Secure and uniform access to distributed resources via the world wide web," *White Paper, October*, 1998.

[198] J. Yu and R. Buyya, "A novel architecture for realizing grid workflow using tuple spaces," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID)*, 2004.

[199] M. Ter Linden, H. De Wolf, and R. Grim, "Gridassist, a user friendly grid-based workflow management tool," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2005.

[200] K. Amin, G. Von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi, "Gridant: A client-controllable grid workflow system," in *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2004.

[201] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto Jr, and H.-L. Truong, "Askalon: a tool set for cluster and grid computing," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 143–169, 2005.

[202] E. Ogasawara, J. Dias, V. Silva, F. Chirigati, D. de Oliveira, F. Porto, P. Valduriez, and M. Mattoso, "Chiron: a parallel engine for algebraic scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 16, pp. 2327–2341, 2013.

[203] R. Oda, D. Cordeiro, and K. R. Braghetto, "Dynamic resource provisioning for scientific workflow executions in clouds," in *Proceedings of the 15th IEEE International Conference on Services Computing (SCC)*, 2018.

[204] A. C. Zhou, B. He, X. Cheng, and C. T. Lau, "A declarative optimization engine for resource provisioning of scientific workflows in iaas clouds," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. Association for Computing Machinery, 2015.

[205] K. Vukojevic-Haupt, F. Haupt, and F. Leymann, "On-demand provisioning of workflow middleware and services into the cloud: An overview," *Computing*, vol. 99, no. 2, p. 147–162, 2017.

[206] K. Vukojevic-Haupt, D. Karastoyanova, and F. Leymann, "On-demand provisioning of infrastructure, middleware and services for simulation workflows," in *Proceedings of the 6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2013.

[207] P. Kacsuk, J. Kovács, and Z. Farkas, "The flowbster cloud-oriented workflow system to process large scientific data sets," *Journal of Grid Computing*, vol. 16, no. 1, pp. 55–83, 2018.

[208] M. Orzechowski, B. Balis, K. Pawlik, M. Pawlik, and M. Malawski, “Transparent deployment of scientific workflows across clouds - kubernetes approach,” in *Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2018.

[209] B. Balis, “Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows,” *Future Generation Computer Systems*, vol. 55, pp. 147–162, 2016.

[210] D. Hoppe, Y. Sandoval, A. Sulistio, M. Malawski, B. Balis, M. Pawlik, K. Figiela, D. Krol, M. Orzechowski, J. Kitowski *et al.*, “Bridging the gap between hpc and cloud using hyperflow and paasage,” in *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, 2017.

[211] R. Qasha, J. Cala, and P. Watson, “Towards automated workflow deployment in the cloud using TOSCA,” in *Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD)*, 2015.

[212] R. Qasha, J. Cała, and P. Watson, “A framework for scientific workflow reproducibility in the cloud,” in *Proceedings of the 12th IEEE International Conference on e-Science (e-Science)*, 2016.

[213] B. Weder, U. Breitenbücher, K. Képes, F. Leymann, and M. Zimmermann, “Deployable self-contained workflow models,” in *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, 2020.

[214] E. H. Beni, B. Lagaisse, and W. Joosen, “Infracomposer: Policy-driven adaptive and reflective middleware for the cloudification of simulation & optimization workflows,” *Journal of Systems Architecture*, vol. 95, pp. 36–46, 2019.

[215] ——, “Adaptive and reflective middleware for the cloudification of simulation & optimization workflows,” in *Proceedings of the 16th Workshop on Adaptive and Reflective Middleware (ARM)*, 2017.

[216] J. Erbel, “Comparison And Adaptation Of Cloud Application Topologies Using Models At Runtime,” Master’s thesis, Institute of Computer Science, University of Goettingen, 2017.

[217] S. Challita, F. Korte, J. Erbel, F. Zalila, J. Grabowski, and P. Merle, “Model-Based Cloud Resource Management with TOSCA and OCCI,” *Software & Systems Modeling*, 2021.

[218] L. Thiesen, “Containerization in a causally connected runtime model for scientific workflows,” Bachelor’s Thesis, Institute of Computer Science, University of Goettingen, 2020.

[219] Eclipse, "UML2, an EMF-based implementation of the Unified Modeling Language (UML) 2.x," Available online: https://wiki.eclipse.org/MDT/UML2, last retrieved: 31.05.2022.

[220] OpenMP Architecture Review Board, "OpenMP Application Programming Interface Version 5.1," 2020, Available online: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, last retrieved: 31.05.2022.

[221] B. Parák, Z. Šustr, F. Feldhaus, P. Kasprzakc, and M. Srbac, "The rocci project–providing cloud interoperability with occi 1.1," in *Proceedings of the International Symposium on Grids and Clouds (ISGC)*, 2014.

[222] OpenStack OCCI Interface, "OpenStack OCCI Interface," 2015, Available online: http://ooi.readthedocs.io/en/stable/, last retrieved: 31.05.2022.

[223] J. Erbel, T. Brand, H. Giese, and J. Grabowski, "OCCI-compliant, fully causal-connected architecture runtime models supporting sensor management," in *Proceedings of the 14th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019.

[224] J. Erbel, S. Wittek, J. Grabowski, and A. Rausch, "Dynamic Management of Multi-Level-Simulation Workflows in the Cloud," in *Proceedings of the 2nd International Workshop on Simulation Science (SimScience)*, 2019.

[225] J. Erbel, F. Korte, and J. Grabowski, "Scheduling architectures for scientific workflows in the cloud," in *Proceedings of the 10th International Conference on System Analysis and Modeling (SAM)*, 2018.

[226] OpenStack4j, "OpenStack4j," 2021, Available online: http://www.openstack4j.com/, last retrieved: 31.05.2022.

[227] Docker, Inc., "Docker Machine," 2021, Available online: https://docs.docker.com/machine/, last retrieved: 31.05.2022.

[228] J. Erbel, "Smart Workflows through dYnamic Runtime Models," 2021, Available online: https://doi.org/10.25625/OE4AN3 or https://gitlab.gwdg.de/rwm/smartwyrm, last retrieved: 31.05.2022.

[229] ——, "Workflows and OCCI (WOCCI)," 2021, Available online: https://doi.org/10.25625/OE4AN3 or https://gitlab.gwdg.de/rwm/de.ugoe.cs.rwm.wocci, last retrieved: 31.05.2022.

[230] ——, "Deployment of OCCI (DOCCI)," 2021, Available online: https://doi.org/10.25625/OE4AN3 or https://gitlab.gwdg.de/rwm/de.ugoe.cs.rwm.docci, last retrieved: 31.05.2022.

[231] Apache Software Foundation, “Hadoop,” 2020, Available online: https://hadoop.apache.org/, last retrieved: 31.05.2022.

[232] ——, “Spark,” 2020, Available online: https://spark.apache.org/, last retrieved: 31.05.2022.

[233] WordPress Foundation, “WordPress,” 2021, Available online: https://wordpress.com/, last retrieved: 31.05.2022.

[234] Nicolas Hennion, “Glances an Eye on your system,” 2021, Available online: https://github.com/nicolargo/glances, last retrieved: 31.05.2022.

[235] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, p. 107–113, 2008.

[236] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[237] S. Wittek and A. Rausch, “Learning state mappings in multi-level-simulation,” in *Proceedings of the 1st International Workshop on Simulation Science (SimScience)*, 2017.

[238] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, “Addressing problems with replicability and validity of repository mining studies through a smart data platform,” *Empirical Software Engineering*, vol. 23, no. 2, pp. 1036–1083, 2018.

[239] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. Ghaleb, K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. Nili Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodriguez Perez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaeej, I. Amit, B. Turhan, S. Eismann, A.-K. Wickert, I. Malavolta, M. Sulír, F. Fard, A. Henley, S. Kourtzanidis, E. Tüzün, C. Treude, S. Maleki Shamasbi, I. Pashchenko, M. Wyrich, J. Davis, A. Serebrenik, E. Albrecht, E. Aktas, D. Strüber, and J. Erbel, “A fine-grained data set and analysis of tangling in bug fixing commits,” *Empirical Software Engineering*, 2021.

[240] A. Trautsch, S. Herbold, and J. Grabowski, “A Longitudinal Study of Static Analysis Warning Evolution and the Effects of PMD on Software Quality in Apache Open Source Projects,” *Empirical Software Engineering*, vol. 25, no. 6, p. 5137–5192, 2020.

[241] F. Trautsch, S. Herbold, and J. Grabowski, “Are Unit and Integration Test Definitions Still Valid for Modern Java Projects? An Empirical Study on Open-Source Projects,” *Journal of Systems and Software*, vol. 159, p. 110421, 2019.

[242] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, “Addressing problems with replicability and validity of repository mining studies through a smart data platform,” *Empirical Software Engineering*, vol. 23, no. 2, p. 1036–1083, 2017.

[243] Project Jupyter, “Jupyter Notebook,” 2021, Available online: https://jupyter.org/, last retrieved: 31.05.2022.

[244] J. Erbel, “Comparing OCCI (COCCI),” 2021, Available online: https://doi.org/10.25625/OE4AN3 or https://gitlab.gwdg.de/rwm/de.ugoe.cs.rwm.cocci, last retrieved: 31.05.2022.

[245] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.