Institute for Architecture of Application Systems

Master Thesis

Automated Compliance Management of Heterogeneous Application Infrastructures at Runtime

Elena Heldwein

Course of Study:	M.Sc. Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dr. rer. nat. Uwe Breitenbücher (University of Stuttgart), Ghareeb Falazi, M. Sc. (University of Stuttgart), Dipl.-Ing. ME Martin Becker (IBM)
Commenced:	7th April, 2021
Completed:	7th October, 2021

Abstract

The automation of currently manual processes is a common goal in enterprises today, with the purpose of making processes less error prone, more efficient and less costly. Compliance management in information technology (IT) is a business process that has the goal of ensuring that IT components adhere to a set of rules. Rules can stem from many sources, such as laws, regulations and enterprise-internal requirements. These rules can for example affect applications, application runtimes, or infrastructure components that enable the hosting of applications. The process of compliance management consists of multiple steps. When manually executed, these steps are error-prone and costly, as human operators tend to have a high risk of committing errors. Existing research already proposes solutions for enabling compliance management automation at different stages of an IT component's life-cycle, such as design-time, deployment-time or runtime. Research also covers different types of compliance rules, such as behavioural and structural rules. However, a generic approach for compliance management of application infrastructures at runtime was found to be missing. Rules that affect application infrastructures are structural in nature, affecting components and their relations. It cannot be assumed that infrastructures which were compliant when they were initially designed or deployed, will remain compliant during their runtime. One reason for this could be user interactions with the components, which cause changes in the components. This master thesis introduces a solution concept for a generic framework that aims to enable the automated management of currently running infrastructure. The concept takes a model-based approach, representing the currently running components through a graph-based instance model, which uses the Essential Deployment Metamodel (EDMM). Using this model, sets of compliance rules, also described as models, can be evaluated. The method of instance model retrieval, as well as the method of compliance rule description and evaluation, are designed in an extensible way. By providing a metamodel for the generic description of compliance rules, the solution concept introduced by this thesis can be used for many methods of compliance rule description. The output of an execution of the conceptualised framework is an extension of the EDMM, the Issue-Extended EDMM (I-EDMM). It allows the representation of compliance issues in a graph-based manner, by extending a given instance model, and shows which instance model elements are affected by detected issues. Furthermore, a prototype of the solution concept is described, and verified by comparing the prototype architecture and functionality to that of the proposed solution concept. Finally, an outlook consisting of future research challenges is provided.

Kurzfassung

Die Automatisierung von manuellen Prozessen ist ein häufiges Ziel in heutigen Unternehmen, mit dem Zweck, Prozesse weniger fehleranfällig, effizienter und weniger kostenintensiv zu machen. Compliance Management hat das Ziel sicherzustellen, dass IT-Komponenten vorgegebene Regeln einhalten. Regeln können von verschiedensten Quellen stammen, wie z.B. Gesetze, Richtlinien und firmeninterne Vorgaben. Diese Regeln können z.B. Applikationen, ihre Laufzeitumgebung, oder Infrastrukturkomponenten, welche die Bereitstellung von Applikationen ermöglichen, betreffen. Der Prozess des Compliance Managements besteht aus mehreren Schritten. Wenn diese Schritte manuell ausgeführt werden, sind sie fehleranfällig und kostenintensiv, da der Mensch ein hohes Risiko hat, Fehler zu begehen. Bisherige Forschung bietet bereits Lösungskonzepte um die Automatisierung von Compliance Management zu ermöglichen. Diese richtet sich auf Management zu verschiedenen Zeiten im Lebenszyklus einer IT-Komponente, wie Entwurfszeit, Deploymentzeit und Laufzeit. Desweiteren richtet sich die Forschung auf verschiedene Arten von Compliance-Regeln, wie Verhaltensregeln oder strukturelle Regeln. Allerdings fehlt ein generisches Lösungskonzept für Compliance-Management zur Laufzeit von Infrastrukturen. Regeln, welche die Infrastruktur von Applikationen betrifft sind struktureller Natur, denn sie betreffen die Komponenten und die Beziehungen untereinander. Es kann nicht davon ausgegangen werden, dass Infrastrukturen, welche zur Entwurfszeit oder Deploymentzeit Compliance-Anforderungen erfüllt haben, dieses auch während ihrer gesamten Laufzeit tun. In dieser Masterarbeit wird ein Lösungskonzept für ein generisches Framework vorgestellt, welches die Automatisierung von Compliance-Management zur Laufzeit von Infrastrukturen ermöglichen soll. Das Lösungskonzept schlägt einen modellbasierten Ansatz vor, welcher ein Instanzmodell in Form eines Graphen verwendet, um die aktuell laufende Infrastruktur darzustellen. Das Instanzmodell ist eine Instanz des Essential Deployment Metamodel (EDMM). Mithilfe dieses Modells können Mengen an Regeln evaluiert werden. Sowohl die Schnittstelle zur Gewinnung des Instanzmodells, als auch die Schnittstelle zu den Methoden, welche die Beschreibung und Evaluierung der Compliance-Regeln erlauben, sind in dem Lösungskonzept generisch gestaltet. Die Ausgabe einer Ausführung des Frameworks, für das in dieser Masterarbeit ein Konzept vorgestellt wird, ist eine Erweiterung des EDMMs, das Issue-Extended EDMM (I-EDMM). Es erlaubt die Darstellung von Regelverletzungen in einem Graphen, welcher ein gegebenes Instanzmodell erweitert. Der Graph zeigt ebenfalls, welche Elemente des Instanzmodells von den entdeckten Regelverletzungen betroffen sind. Desweiteren wird ein Prototyp vorgestellt und verifiziert, indem die Architektur und Funktionalität des Prototyps mit der des Lösungskonzeptes verglichen wird. Zuletzt wird ein Ausblick auf zukünftige Forschungsarbeiten gegeben.

Contents

1 Introduction **1**

2 Background Theory **3**
2.1 Application Infrastructure . . . 3
2.2 Cloud Computing Classification . . . 8
2.3 Application Infrastructure Layers in Cloud Services . . . 9
2.4 Compliance and Compliance Management . . . 9

3 Related Work **13**
3.1 Business Process Compliance . . . 13
3.2 Application and Infrastructure Compliance . . . 15
3.3 Miscellaneous . . . 17

4 Motivation and Research Questions **19**
4.1 Motivation . . . 19
4.2 Research Questions . . . 27

5 Solution Concept **29**
5.1 Approach . . . 29
5.2 Issue-Extended Essential Deployment Metamodel . . . 30
5.3 Architecture Concept . . . 32
5.4 Examples . . . 47

6 Proof of Concept **63**
6.1 Prototype . . . 63
6.2 Verification . . . 69

7 Evaluation **73**

8 Conclusion and Outlook **77**

Bibliography **79**

List of Figures

2.1 Essential Deployment Metamodel [WBF+20] 5

4.1 Workflow of runtime compliance management 20
4.2 (Scenario 1) An example application system infrastructure with compliance violations. 24
4.3 (Scenario 2) Example set of infrastructure components with compliance violations. 26

5.1 EDMM extended by issues (I-EDMM) 30
5.2 An example I-EDMM model for violations of CR 3 and CR 4 in motivating scenario 1 31
5.3 Overview of the compliance automation framework architecture 33
5.4 Workflow of using the framework 35
5.5 Workflow of the framework execution 37
5.6 Framework Core 38
5.7 Compliance Rule Metamodel 39
5.8 Examples Using Compliance Rule Metamodel 40
5.9 Orchestrator Control Flow 42
5.10 Required Public Interface for Compliance Rule Type Plugins 43
5.11 Instance Model Retriever Architecture 44
5.12 Required Public Interface of Retrieval Plugin 45
5.13 Architecture of Instance Model Annotator Component 46
5.14 Control Flow of Instance Model Annotator 46
5.15 Scripts Plugin 49
5.16 CR 2 applied to PrivateInternalApp stack of scenario 1 53
5.17 The I-EDMM resulting from example rule evaluations using three plugins 59
5.18 (Scenario 1) The application infrastructure without compliance violations. 60
5.19 (Scenario 2) Set of infrastructure components without compliance violations. . . 62

6.1 Architecture of prototype 63
6.2 Class diagram of prototype 65

Listings

5.1 Instance Model Description in Prolog 56
5.2 Detector Description in Prolog 56
5.3 Evaluator Description in Prolog 57

6.1 Pseudocode for interface that rules must implement in prototype 66
6.2 Pseudocode of prototype orchestrator 66
6.3 Pseudocode of detection component 68
6.4 Pseudocode of evaluation component 68

Acronyms

AWS Amazon Web Services. 6, 8, 23

BPMN Business Process Model and Notation. 19, 34

DBMS Database Management System. 9, 23, 25

EDMM Essential Deployment Metamodel. ix, 4, 5, 6, 15, 19, 22, 29, 30, 45, 48, 62, 64, 66, 67, 69, 74, 78

IaaS Infrastructure as a Service. 8, 9, 23, 25

IaC Infrastructure as Code. 6, 7, 12, 17, 48, 51

I-EDMM Issue-extended Essential Deployment Metamodel. 2, 29, 30, 32, 34, 36, 47, 60, 62, 63, 64, 67, 69, 70, 75, 77, 78

IT Information Technology. 9, 10, 11, 12, 15, 16

JRE Java Runtime Environment. 4, 23, 35, 52, 61

NIST National Institute of Standards and Technology. 8

OASIS Organization for the Advancement of Structured Information Standards. 4

OS operating system. 23, 24, 32

PaaS Platform as a Service. 8, 9, 23

SaaS Software as a Service. 8, 9

TOSCA Topology and Orchestration Specification for Cloud Applications. 4, 6, 15, 39, 48, 51, 55

UML Unified Modeling Language. 4, 6, 41, 45, 64

URL Uniform Resource Locator. 57, 58, 65

Preface

This master thesis was written in cooperation with IBM Research & Development GmbH under the technical supervision of Martin Becker. The prototype[1] was developed for IBM under the Apache 2.0 license[2].

[1]Prototype is available here: https://github.com/ElH1/masterthesis-prototype
[2]License terms and conditions can be found here: https://www.apache.org/licenses/LICENSE-2.0

1 Introduction

In modern applications, the infrastructure an application is hosted on is becoming increasingly heterogeneous [BBK+13], especially with the advance of cloud environments. Changes are deployed at a fast pace, with a paradigm shift from monolithic application architectures to micro-service based application architectures [AAE16]. This means an application consists of multiple services, which may be hosted on a variety of infrastructure components. Such an application's infrastructure must be kept compliant, meaning it must adhere to a set of rules. The on-going compliance management of said infrastructure is an increasingly challenging task [KBKL18], also due to the fact that the infrastructure is increasingly distributed and heterogeneous. Not only must its compliance be evaluated and enforced at the time of an application's design, but also whilst it is provisioned and for the complete duration while the infrastructure is up and running. Due to multiple reasons, user interaction with infrastructure components being one of them, a set of components that was compliant at the time it was provisioned cannot be guaranteed to remain compliant until it is deprovisioned [Mor20]. A further source of complexity is the variety of rules that may apply, as well as the different methods of describing and enforcing rules that may be used, even in a single application. In order to ease this process of compliance management, an approach to automating compliance management of heterogeneous application infrastructures at runtime is required.

This thesis started by reviewing related research in the area of automated compliance management, of both application infrastructure compliance, as well as the compliance of business processes, which were found to be related. This allowed the establishment of a concise research question for this work. The question this thesis focuses on is whether a generic, extensible approach to enabling the automation of compliance management at runtime, for a wide variety of use-cases, is feasible, and what is required in order to make it generic. It was concluded that a framework is needed, in order to provide an approach that facilitates the automation of heterogeneous application infrastructure compliance management at runtime. An initial collection of requirements, that the framework should fulfil, was conducted through interviews with potential framework users. Guided by these requirements and the research questions, a solution concept for a framework was designed and enhanced by making it technology-agnostic and extensible. Then a prototype was implemented in order to verify the solution concept. Finally, the solution concept was evaluated by analysing its fulfilment of the requirements.

To summarise, the contributions of the thesis are as follows:

(i) A solution concept in the form of a model-based, technology-agnostic and extensible framework, that aims to facilitate automated compliance management of heterogeneous application infrastructures at runtime. Model-based means the framework uses models of compliance rules to evaluate a model of the running infrastructure, and returns a model describing the detected compliance violations.

(ii) A metamodel for describing issues and affected components in the form of an Issue-extended Essential Deployment Metamodel (I-EDMM) is suggested. The I-EDMM allows a generic, graph-based description of compliance violations in the form of issues, that allow a user see which issues an infrastructure has, and which infrastructure components are affected. This is the output of the framework described in the solution concept.

(iii) A metamodel for compliance rule descriptions, proposing a generic approach to describing compliance rules.

(iv) A prototypical implementation that is used to verify the solution concept's feasibility.

The remaining thesis is organised into the following chapters: Chapter 2 (Background Theory) explains the foundations that are necessary in order to understand the remainder of the thesis. Chapter 3 (Related Work) describes the state of the art in current research. Chapter 4 (Motivation and Research Questions) provides a motivation for the research conducted in this thesis, leading on to a description of the research questions this thesis aims to answer. Chapter 5 (Solution Concept) details the solution concept proposed by this thesis. The contributions consist of the Issue-Extended Essential Deployment Metamodel (EDMM), an architecture concept for a generic framework, and a metamodel for describing compliance rules in a generic way. A number of concrete examples are provided that can be used with the framework. Chapter 6 (Proof of Concept) describes the prototype that was implemented as a proof of concept, as well as the concept verification. Chapter 7 (Evaluation) evaluates to what extent the solution concept fulfils the requirements posed in chapter 4. Chapter 8 (Conclusion and Outlook) concludes the results of this thesis, and discusses future research challenges.

2 Background Theory

Managing the compliance of a heterogeneous application infrastructure consists of multiple steps and can affect different stages of an infrastructure's life-cycle. Additionally, there are many different aspects that can be managed, and can have compliance requirements. In order to better understand the challenges behind automating this process, as well as the solution concept proposed by this work, this chapter will cover some foundations. It first explains what application infrastructure is, as well as how it can be modelled and automated. Next, it covers details on cloud service types, and finally describes a compliance taxonomy, as well as what compliance is in the area of application infrastructure.

2.1 Application Infrastructure

Application Infrastructures are the set of underlying components that an application is hosted on, and which it requires to run. Application infrastructures include all of the computational and operational infrastructure components that are necessary to manage the development, deployment, and management of enterprise applications [sum21]. Infrastructure components are a part of the application architecture. Examples of infrastructure components include, but are not limited to, physical servers, virtual machines, networking, data storage, application monitoring and logging, security services, as well as various types of cloud services. Infrastructure components may be hosted, for example, on-premise or on cloud-based infrastructure, and both may be combined within a single application infrastructure. Many such infrastructures and their components can be managed using Infrastructure as Code (IaC) tools. This thesis focuses on a representative subset of components. Many components expose properties that may change over time, making it necessary to monitor their state during their lifetime, in order to ensure their adherence to requirements.

In [Mor20, Chapter 3], application systems are separated into three layers: (i) applications, (ii) application runtime platforms and (iii) infrastructure platforms.

- The ***(i) application layer*** provides capabilities to users. This can include, for example, application packages which make software available to users, such as a bundled and self-contained Java application, or otherwise executable programmes. The other layers exist to enable the application layer.
- The ***(ii) application runtime platforms layer*** includes components such as application or web servers, container clusters and database clusters. It provides capabilities to the application layer.
- The ***(iii) infrastructure platform layer*** includes components such as compute resources, network structures and storage, or a combination of these. In addition, this layer also includes the tools and services which manage the resources on said layer.

Infrastructure components can, but don't have to, span multiple layers. For example, the infrastructure for a Java application (applications layer) would require a Java Runtime Environment (JRE) (application runtime platforms layer), as well as a server that is a compute resource (infrastructure platform layer), allowing said application to be hosted. Classes and methods, i.e. the business logic of an application, are beyond the scope of this thesis. This thesis will be dealing with the application runtime platform and infrastructure platform layers of multiple services. Services are applications that provide a variety of capabilities to users. A *heterogeneous* application infrastructure means that the components included in the infrastructure are of different types, e.g. bare-metal and virtual servers, or a variety of cloud services. In a heterogeneous application infrastructure, the infrastructure is not limited to only virtual servers, or only a single type of cloud service offering.

It is important to differentiate between application architecture and application infrastructure. The former usually refers to the design of the components of an application system as well as their relationships. The latter describes the actual, existing set of running components and relationships, including all required components to make the application available to users [tec21]. The architecture is a goal, the target state, whereas the infrastructure is the actual as-is state.

2.1.1 Describing Application Infrastructure as a Model

Both application architecture and application infrastructure can be described in a model. A type of model commonly encountered when describing an application architecture is the deployment model, which describes how an application should be deployed in the form of a graph. Such a model can be represented, for example, as a TOSCA topology template[1], or as a UML deployment diagram. TOSCA stands for Topology and Orchestration Specification for Cloud Applications. It is an OASIS[2] standard that aims to enable the description of cloud applications, allowing the modelling of application topologies as well as their deployment. The models can cover the hosting and implementation of application components like middleware, web services and databases [KKW+14]. Both UML deployment diagrams and TOSCA topology tempates are designed to be technology-agnostic. However, a TOSCA topology template also integrates with deployment tools, to allow for automated deployment. Example tools for the automation of deployment are Terraform[3], Puppet[4], Chef[5] or Ansible[6]. Models of an infrastructure's runtime state would however be considered to be instance models. Both can use a similar metamodel, two differences being which life cycle stage is reflected and possibly how frequently the model needs to be updated. This allows for the methods of describing an application architecture in a model to also be used to reflect the current state of an infrastructure.

In [WBF+20], Wurster et al. have created a generic graph-based method of describing deployment models, which can also be used in order to describe infrastructure at runtime as instance models[7].

[1] Described in detail in section 3 here: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.pdf

[2] A standards organisation, OASIS stands for Organization for the Advancement of Structured Information Standards

[3] https://www.terraform.io/

[4] https://puppet.com/

[5] https://www.chef.io/

[6] https://www.ansible.com/

[7] This is for example done in the EDMM subproject on https://github.com/UST-EDMM/edmm/tree/master/edmm-instance, as well as by TOSCA, as described on https://docs.oasis-open.org/tosca/TOSCA-Instance-Model/v1.0/TOSCA-Instance-Model-v1.0.pdf

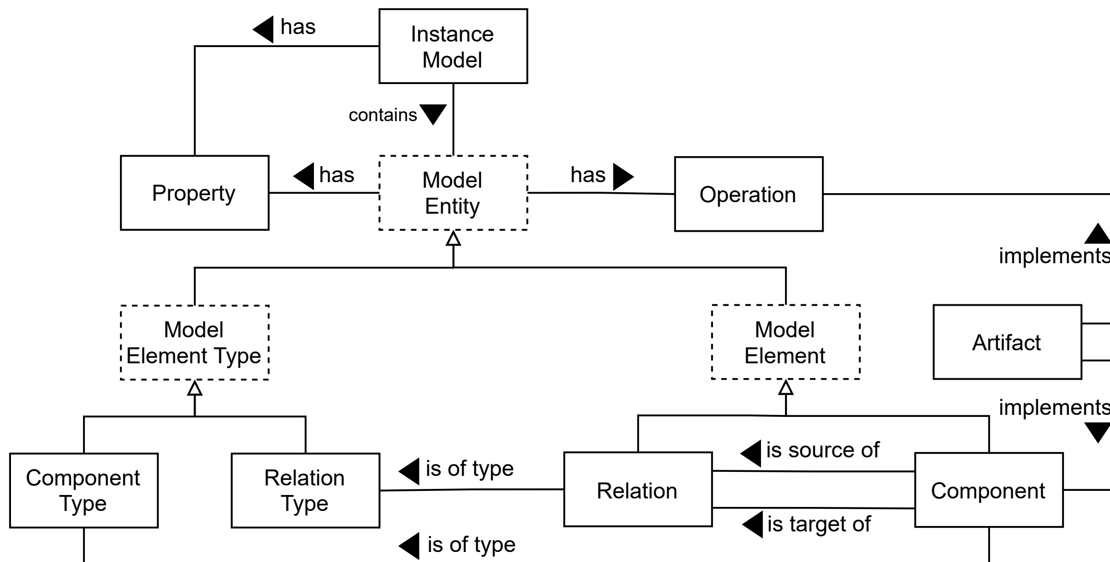

Figure 2.1: Essential Deployment Metamodel [WBF+20]

Figure 2.1 depicts the Essential Deployment Metamodel (EDMM) as described by Wurster et al. [WBF+20]. The only difference is that the depicted metamodel describes an instance model, whereas the metamodel in [WBF+20] describes a deployment model. As an instance model has the same elements as a deployment model, the definitions can be adopted. The elements of the metamodel are defined as follows:

Definition 2.1.1 (Component)
A component is a physical, functional or logical unit of an application.

Definition 2.1.2 (Component Type)
A component type is a reusable entity that specifies the semantics of a component that has this type assigned.

Definition 2.1.3 (Relation)
A relation is a directed physical, functional or logical dependency between exactly two components.

Definition 2.1.4 (Relation Type)
A relation type is a reusable entity that specifies the semantics of a relation that has this type assigned.

Definition 2.1.5 (Operation)
An operation is an executable procedure performed to manage a component or relation described in the deployment model.

Definition 2.1.6 (Property)
A property describes the current state or prescribes the desired target state or configuration of a component or relation.

Components and operations are implemented through so-called artifacts. According to the UML specification, an artifact is a physical piece of information that is created to be used for deployment and operation of a system [Obj17].

The EDMM can also be used as an instance model, similar to the approach taken by TOSCA, so these definitions can remain unchanged for an instance metamodel. A deployment model describes the goal state of an application, whereas an instance is the actual running copy of an application, and there may be multiple instances of an application.

2.1.2 Infrastructure as Code

The principles of Infrastructure as Code (IaC) are explained in the book [Mor20, Chapter 1]. IaC is a method of automating application infrastructures and allows the description of an application infrastructure in the form of code, which enables the use of software development best practices on the application infrastructure. Three core practices of IaC are (i) to define everything as code, (ii) to continuously test and deliver all work in progress and (iii) to build small, simple pieces that can be changed interdependently.

Morris also explains infrastructure stacks, as well as methods of building these as code, in [Mor20, Chapter 5], and how to configure stacks in [Mor20, Chapters 11 and 12], along with associated best practices and issues. An *infrastructure stack* is a collection of infrastructure components which are defined, deployed and updated as a unit. The IaC code defines components of the stack, such as a virtual machine (compute resource), disk volume (storage resource) and a subnet (network resource). Some best practice patterns recommend different methods of separating (or grouping) stacks. These include separating into groups of multiple related services for an application, or creating a stack per service, or even dividing the infrastructure for a single service across multiple stacks. A negative practice would be to group too many components in a single stack, making it difficult to maintain. This may defeat the purpose of automating infrastructure using IaC.

Infrastructure deployment, or *provisioning*, refers to the process of making the infrastructure components available for use, such as the initial creation of virtual servers and the installation of required runtimes. Infrastructure *management*, on the other hand, is understood to be the process of ensuring the components work correctly, allowing them to stay available for use during runtime. Deployment is an initial process, whereas management is an ongoing one. IaC tools can be divided into provisioning tools and configuration management tools [Bri17]:

- *Provisioning tools*, also called stack management tools [Mor20, Chapter 5], are capable of deploying and managing sets of infrastructure components. Hashicorp Terraform[8] and AWS' own IaC service CloudFormation[9] are examples of deployment tools.
- *Configuration management tools*, also called server configuration tools [Mor20, Chapter 11], require an existing infrastructure, of which they can then manage the configuration. Ansible, Chef and Puppet are examples of configuration management tools. They are used to configure existing infrastructure components, such as servers, although extensions exist that allow the creation and management of certain infrastructure using these tools too.

[8] https://www.terraform.io/docs/index.html
[9] https://docs.aws.amazon.com/cloudformation/index.html

Not all tools can be clearly fitted into one or the other category. Foreman is an example of a full life cycle infrastructure management tool that doesn't fully fit into either category and allows the installation of bare-metal servers, whilst it utilises IaC tools for configuration as well as ongoing management. The author of [Mor20, p. 54] recommends decoupling the stack deployment from the server configuration by passing the configuration task to a configuration management tool.

Definition of infrastructure stacks as IaC can either be declarative or imperative. *Declarative* means one specifies what result one wants, without information on how to achieve this. Terraform, for example, uses a fully declarative IaC language. *Imperative* means the code provides steps on how to set up or change the infrastructure. One example would be the option in Ansible to use procedural logic such as loops and conditionals. Another example is Pulumi[10], which is a solely imperative approach to describing IaC.

Configuration controls how a component, and possibly hosted applications, work. It takes place during the initial deployment of an infrastructure, however additional configuration may take place at later points during the lifetime of an infrastructure component. An example of a configuration operation is adjusting the network settings, such as assigning a server to a certain network address block and adding firewall rules. Another example can be setting up a local agent that is required for monitoring purposes and registering it with the monitoring service. The life cycle of an infrastructure component consists of the stages:

1. creating and configuring a component (deployment-time)
2. changing an existing component (runtime) and
3. destroying a component (deprovisioning).

Changing an existing component is a change at runtime and can have various effects. One such effect is *configuration drift* [Mor20, Chapter 2], which means the actual state diverges from the state that components had after initial deployment. A consequence is that initially identical components may become different to each other over the course of their lifetime and may not remain in a desired state. Configuration drift occurs particularly when making manual changes, or when executing automated code ad hoc during runtime.

Best practices for dealing with component changes [Mor20, Chapter 12] recommend either applying configuration to components on a regular basis, regardless of whether configuration has changed, or swapping out running servers against freshly built and tested servers whenever changes must be applied. Both approaches aim to reduce the risk of making changes. The first approach aims to quickly revert or make visible unexpected differences between server instances that may appear over time. The second approach aims to avoid configuration being changed on a running instance in the first place. A negative practice is to apply configuration code to a running instance only when changes occur, no matter whether manually or using an automation tool. However, components should remain compliant with certain rules regardless of how changes are applied, even if negative practices are applied.

[10] https://www.pulumi.com/

Components can be hosted using a number of methods, such as on-premise virtual components, or as an off-premise cloud service. Albeit, the responsibilities associated with the used services can vary in the cloud, depending on the used service type. In order to better understand the different types of cloud services, as well as the associated responsibilities, the next section will describe these aspects.

2.2 Cloud Computing Classification

According to the National Institute of Standards and Technology (NIST), cloud computing is “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction.”[MG+11]

There are two dimensions according to which cloud computing can be classified. One is the **deployment model**. The deployment models are private cloud, community cloud, public cloud and hybrid cloud. In a *private cloud*, the cloud infrastructure is made available for use to only a single organisation, and is commonly hosted on-premise (e.g. within an enterprise), although this is not a requirement. In a *community cloud*, the cloud infrastructure is made available only to a community of consumers from organisations with shared concerns. It may be hosted on- or off-premise. In a *public cloud*, the cloud infrastructure is provided for use by the general public, and is made available remotely by a third-party service provider. In a *hybrid cloud*, the cloud infrastructure is a combination of two or more cloud deployment models, e.g. private and public.

The second dimension is the **service model**. This allows categorisation of the service types made available by a cloud service provider, depending on the abstraction level of the service. *Infrastructure as a Service (IaaS)* provides compute resources without the need for managing the underlying hardware, however the responsibility of maintaining the compute resources remains in the hands of the service user. The user is provided with capabilities of provisioning services, such as processing, storage or networks. An example of compute resources is a service providing virtual machines to users, such as AWS EC2 service[11] and Google Cloud Compute Engine[12]. *Platform as a Service (PaaS)* provides platforms such as a database service, while abstracting away the compute layers below the platforms, such as operating systems or networks, so the service user holds no responsibility for these. Examples of such services are the database services AWS RDS[13] and Google Cloud SQL[14]. *Software as a Service (SaaS)* allows for the use of a software with no required knowledge or responsibility for the hosting and maintenance of the software. The software is accessible from multiple client devices that have network connectivity, and the user is not responsible for any aspect of the software. An example is the Microsoft Office 365[15] package. Only the IaaS and PaaS types will be of relevance to this thesis, while the cloud deployment model should be irrelevant due to the generic nature of the framework.

[11] AWS Elastic Compute Cloud: https://docs.aws.amazon.com/ec2/index.html
[12] https://cloud.google.com/compute/docs
[13] AWS Relational Database Service: https://docs.aws.amazon.com/rds/?id=docs_gateway
[14] https://cloud.google.com/sql/docs
[15] https://docs.microsoft.com/en-us/microsoft-365

2.3 Application Infrastructure Layers in Cloud Services

It is possible to map the cloud service models from section 2.2 to the application layers introduced in section 2.1.2. Services on the IaaS layer can be mapped to the infrastructure platform layer. The components on this layer, such as compute resources, network structures and storage, correlate to the services, such as compute services, virtual network services and storage services, that are made available by cloud service providers. IaaS services enable all layers above, but also leave components in overlying layers, such as operating systems and runtime environments, as a user responsibility. Services on the PaaS layer can be mapped to the application runtime platforms layer. The components a service provider makes available as PaaS services consist of runtime environments, such as Database Management System (DBMS) or web hosting environments, as well as the underlying compute resources. This allows users to host components of the application layer, such as a web application the user has developed, on PaaS services, without being aware of the underlying runtime application layer or infrastructure platform layer. Services on the SaaS layer can be mapped to the application layer. This reflects the fact that SaaS services provide applications to a user, i.e. provides capabilities to users, without the user being aware of any of the layers. In this cloud service model, the user has no responsibility for any of the described layers from section 2.1.2. The service provider makes only the application available to the user, and hides all underlying components, holding full responsibility for them.

2.4 Compliance and Compliance Management

So far, this chapter has explained what *heterogeneous application infrastructures* are understood to be, in the context of this work, and introduced related concepts. However, this thesis focuses on *automated compliance management* of such infrastructures. This means it is not only necessary to understand what components are being analysed, but also which aspects of these components require managing, and what management means in this context. The following sections will focus on explaining these details.

2.4.1 What is compliance?

Compliance in Information Technology (IT) describes a state, in which all mandatory specifications and regulations regarding the IT in an organisation are adhered to. A compliant state needs to be achieved through adequate processes. Due to a large number of regulations and the frequency of changes in regulations, as well as the potential consequences of violations [KS21], compliance is a very important topic. *Sources of compliance requirements* can be a wide variety, ranging from legal regulations, such as the laws of the Federal Office for Information Security (BSI) [KS21], or General Data Protection Regulation [GDP] , to branch-specific guidelines and enterprise-specific codes of conduct [KKR+13].

2.4.2 What is compliance management?

Compliance Management refers to the process of regularly checking and comparing a current state to a set of compliance rules. Furthermore, compliance management can also include the enforcing of compliance rules by imposing a required state on affected components. The current state in the context of IT can affect a variety of things, ranging from the state of business processes, through running infrastructure components to application components. In this thesis, only the compliance of infrastructures and infrastructure components is of relevance, with a focus on checking rather than enforcing.

Lowis [Low11] mentions the following five steps as steps that are required in order to manage compliance:

1. Know who wants you to do things *(identify the appropriate regulations that apply to your organisation)*
2. Know what to do *(interpret the regulation for the organisation's environment)*
3. Know what you do *(understand and document the organisation's processes and policies)*
4. Do what you say *(monitor for compliance and changes)*
5. Say what you know *(report as required)*

Steps 1–2 are assumed to have already been executed prior to using an automation solution. These steps will remain manual, according to Lowis [Low11]. Step 3 must also remain manual. It is, however, necessary to allow users to describe their rules in a way that the rules become machine-readable. Such a rule description should allow for an automation tool to automate the subsequent steps, leaving steps 4 and 5 as the main focus of an automation tool.

2.4.3 IT Compliance Taxonomy

To ensure compliance, the rules must be described in some way, necessary actions performed to both check and enforce, and the results must be documented. However, adaptive environments and frequent changes, to both the compliance rules and the environment under scrutiny, pose a challenge to remaining compliant. Every change could affect the overall compliance and adjustments often have to be done manually [KKR+13]. There are a few methods of differentiating types of compliance, depending on *when checks are done* and *what is being checked*.

Checks can be done at different points in an infrastructure's life cycle, as explained by Koetter et al. [KKR+13].

- *Design-time compliance* means that compliance is checked at the time of application design. The application architecture can be checked for adherence to rules.
- *Deployment-time compliance* means that compliance is checked when an application design is instantiated (i.e. implemented). In this case, a deployment model can be used to check for adherence to rules, when certain aspects of the runtime become known. Such aspects could be, for example, where compute resources will be provisioned.

- *Runtime compliance* means rule checking, and potentially enforcement, takes place whilst components are up and running. Configuration drift (see section 2.1.2) is a reason why one cannot assume that a system, which was initially compliant during design- and deployment-time, will remain so during runtime. This makes the checking of runtime compliance equally as necessary as design- or deployment-time compliance.
- *Retrospective compliance* means rule checking takes place after runtime, using, for example, log analysis.

Furthermore, checks can be done on different properties of components.

- *Behavioural compliance* means that compliance is checked by analysing events of an application, which implies that the check takes place at runtime. These analysed events are the behaviour of the application at runtime, which is compared to expected behaviour. This requires the implementation of an event emitter on components [KBF+20].
- *Structural compliance* on the other hand means that compliance is checked by analysing the structure, i.e. the topology model, for rule violations. This can take place at any point in time, as both design-time and runtime models have structural rules they are expected to fulfil. Retrospective compliance checking is only possible if the structure is monitored and stored during runtime.

Finally, compliance can also be differentiated by what areas of business and IT are being checked.

- *Business process compliance* refers to the compliance of a holistic business process. It usually includes the IT and application aspect, but business processes also consider actors and manual steps in the process that are not integrated into an application, or only interact with application components without being application components themselves. Business process compliance can be checked at any point in time. Much of the existing research focuses on this area of compliance. Business compliance is one part of business process compliance, other parts include, amongst further organisation aspects: hosting, encryption, as well as physical access control and maintenance [KKW+14]. Some of these aspects correlate to application infrastructure compliance.
- *Software or application compliance* usually refers to the compliance of application code [Low11], which is the implementation of an architecture before an application is instantiated. This can be checked by static code analysis, for example using a model of the code, such as in the approach described by Takhma et al. [TRH+15]. This type of compliance check implies that the evaluation must take place before runtime, as program code has been compiled into executable code by this point in time, and can usually no longer be checked.
- *Application infrastructure compliance*, which is the focus of this thesis, refers to the compliance of the application infrastructure, that enables the hosting of an application, as described in section 2.1. This type of compliance can be checked using either a deployment model that represents the infrastructure at deployment-time, or an instance model that represents the infrastructure at runtime.

Business process compliance is beyond the scope of this thesis, it will focus solely on the application infrastructure compliance aspect, i.e. the structural runtime compliance of application infrastructures. Compliance violations that affect the functionality of an application are also not considered. It is

assumed that violations which have to be detected cannot be detected by analysing the functionality of an application instance. The following subsection will cover examples of infrastructure compliance in more depth, and explain how this is connected to IaC.

2.4.4 Application Infrastructure Compliance

There can be many sources of architectural (i.e. structural) rules. One such source can be patterns. Patterns are templates for tried and tested solutions to common problems that may occur. One such pattern is the pipes-and-filters pattern [FLR+14], which applies to a distributed application. It is a method of decomposing distributed components by purpose, allowing independent scaling out of the components. Filters represent application components that process data, whereas pipes are the connections between filters. Another source of structural rules are reference architectures. A reference architecture describes recommended structures of IT solutions, which are accepted as best practices [CMV+10]. An example is the 3-tier architecture, also known as a client-server model [Olu14]. This separates application processing into three layers, namely the client a user directly interacts with, an application server, and a database server. These sources are in addition to the potential sources mentioned in the introduction of section 2.4.

The following rules are generic examples of rules that specifically affect the application infrastructure and are within the scope of this thesis:

- Configuration of components that depends on how they are connected, e.g. all components that host security-critical applications must be configured to fulfil certain properties.
- All components, such as operating systems, must be on a particular version, if they host certain other components, such as application servers.
- Using correct ports for communication between components of certain types, for example HTTPS port 443 must always be used for web applications and web servers, as these are not allowed to serve data using the unsecured HTTP port 80.
- Restriction of user access to all components that hold security-critical data, or are in any way connected to such components.

Infrastructure as Code compliance checking can be understood as a sub-category of infrastructure compliance checking. In this case, the infrastructure that requires checking must have been either deployed or configured, or both, using IaC tools. Design-time compliance checks can be achieved by checking the code, or a deployment model extracted from the code, for compliance to rules. Runtime compliance on the other hand poses the challenge that the infrastructure's state might diverge from the coded infrastructure, and its corresponding deployment model. It is not possible to use the code that describes a design-time or deployment-time infrastructure as a representation of the current infrastructure state.

This chapter has covered the required foundations in order to understand the topic of this thesis, and the chapters that follow. After describing what application infrastructures are understood to be in the context of this work, the chapter explained various aspects of compliance and compliance management, including a taxonomy which should allow for evaluation of existing research. The next chapter will describe the current state of the art in the field, in the form of a literature review.

3 Related Work

In this chapter, currently existing research is reviewed. The research covers compliance management in various areas, and determines the current state of the art in this field of research. The literature is separated into two categories. First is business process compliance, which is related to infrastructure compliance as explained in section 2.4.3. Second is existing work on application and infrastructure compliance.

3.1 Business Process Compliance

IT components may execute steps in a business process, making them an element (such as an activity) in a business process. Thus, IT compliance can be understood as a part of an overall business process, also constituting an element of business process compliance. Because of this, research on automation of business process compliance was included in this literature review. Relevant terms that differentiate the time of compliance evaluation, and the type of compliance which is being evaluated, will be italicised. The description of these terms can be found in section 2.4.3.

García-Galán et al. [GPGN16] executed a survey on the approaches that exist for dealing with runtime variability of compliance rules, in this case for business processes. Variability means the changing of compliance requirements at runtime. Their work provides a good overview of what research exists on compliance management of business processes, and where research in that area is still lacking. Furthermore, they provide a comparison of existing approaches to automated management of business process compliance. In particular, the automation of compliance management of operating environments is stated to be lacking, and infrastructure is a part of the operating environment. This helps point out and motivate the need for further research in the area of compliance management of running infrastructure.

Anstett at al. [AKL+09] propose a solution to proving and enforcing compliance in a distributed setting, where systems have a heterogeneous architecture. Their approach aids compliance management in a situation, where the compliance status of certain activities in a business process cannot be ascertained locally or internally. They have to be retrieved from, and enforced through, an outsourcing partner instead. An example of an outsourcing partner is a cloud provider. Their general compliance architecture allows the monitoring and enforcement of services deployed in any cloud environment. This includes the aggregation of collected compliance data from various providers. The approach focuses on runtime compliance, however it uses events to analyse the behaviour of a business process, not the structural compliance of an application infrastructure. Additionally, Schleicher et al. have done extensive research in the area of business process compliance. This research includes proposing a number of complex compliance rules to describe control-flow and data-related compliance rules [SGL+11], as well as an approach to assist business process designers in modelling compliant business processes when creating processes in the field of

cloud computing [SFG+11]. A third article by Schleicher et al. [SLSW10] introduces the concept of compliance scopes, which allows compliance rules to be attached to partial business process templates, thus enforcing compliance at design-time. None of this research, however, focuses on automating the evaluation of structural runtime compliance.

In [Awa10], Awad proposes a formal approach that aims to support automated business process compliance checking at *design-time*. Their approach checks the compliance rules against a model of a business process. Many of the business process modelling languages are graph-based, expressing activities (steps in a process) and execution orders between activities (control flow), as well as data flow between activities. However, these models cannot represent a topology of components, as well as the connections between them that do not constitute control or data flow. Awad uses patterns to model the compliance requirements and later map them into logical formulae. In their work, the compliance rule types are separated into rules which affect the control flow, data flow or conditional flow. Their rules are checked against design-time models of the business process under scrutiny. Furthermore, Awad addresses the issues that stem from having multiple sources of compliance requirements, which may be inconsistent. This approach however does not address runtime compliance, which cannot be checked using the approach used at design-time.

In a later piece of work, Awad et al. [ABE+15] do focus on the automated *runtime* compliance management of business processes. They present a generic compliance monitoring framework for business processes at runtime, which they have named BP-MaaS. In this approach, they use anti-patterns and an event-based framework to detect runtime compliance violations. By using anti-patterns for monitoring, they assure that their approach is independent from underlying technology. As a requirement towards their framework is that violations are detected as soon as they occur, an event-based approach with monitoring is required. Their proof of concept adopts a complex event processing technology as a possible framework realisation. However, this approach is not suitable when provided with, for example, components which cannot be modified to work with a framework. Such components would not be allowed to add event processing components that emit events. Events are required by the approach in this work, as the approach assumes there is an event stream. The evaluation of events means the approach by Awad et al. checks behavioural compliance. This stands in contrast to the checking of structural compliance, where the compliance rules affect, and allow evaluation of, the topology of an infrastructure.

Barnawi et al. [BAE+15] cover automated runtime monitoring of business processes which execute in a cloud environment, in order to ensure the compliance of said business processes. Their approach aims to allow the management of compliance without needing to rely on external monitoring components, that must be linked to process execution environments. Most of these approaches require an event stream that can be monitored, and Barnawi et al.'s approach aims to avoid this overhead. They highlight that, especially in a cloud environment, it is not always possible to monitor all components of an execution environment using this approach. Their approach instead allows the reporting of a compliance status by the process instance itself. To achieve this, they use compliance patterns which have to be defined by a compliance expert, and apply these to a business process model, which a business expert has to define. A business process is augmented in a way that allows monitoring with the above models. They focus on compliance patterns that enforce the order and owner of task execution, or the response time of a task, which does not include compliance rules that affect the topology of an infrastructure. Furthermore, this approach requires a model of the

overall business process, and still requires augmentation of monitored instances, both of which should not be a requirement in the approach this thesis proposes. The business process surrounding the compliance of a given infrastructure is assumed to be transparent to us.

3.2 Application and Infrastructure Compliance

In this section, a number of papers are presented that have looked into, and proposed, solutions to aid in the management of application and infrastructure compliance. This puts a greater focus on the IT aspect, and treats any related business processes transparently.

In [GKN08], Ganesan et al. have looked into automating architecture compliance checking at *runtime*. Architecture, by definition of this paper, refers to the software architecture of applications. Thus, compliance rules affect relationships among code elements. Their approach is to derive coloured Petri-nets (CP-nets) from a given running software. The CP-nets are responsible for architecture extraction (from an event bus that produces run-time traces) and compliance checking at run-time, with a focus on structural aspects of the architecture. This is achieved in a non-intrusive way. Their approach may be interesting as an element of a generic compliance checking solution, however they restrict the user to having to model compliance rules in a way that can be interpreted as CP-nets. This is a non-trivial task for many users that may be responsible for managing compliance of components. The non-triviality of the task of creating a mapping between architectural elements and runtime traces, as well as the fact that this is an iterative process, is stated by Ganesan et al. themselves. Our approach, on the other hand, should allow for a variety of methods of modelling and checking compliance rules, allowing users to choose a preferred method that they are most familiar with. This means the approach described in this paper could be used as one method of rule evaluation, however not as the sole method. Furthermore, it is not clear from this paper whether the approach also includes components on the *infrastructure platform and runtime layer*, or exclusively analyses the *application layer*.

Krieger et al. [KBKL18], Zimmermann et al. [ZBKL18] and Fischer et al. [FBKL17] all propose approaches to modelling and checking *structural compliance* rules at *deployment-time*, or *design-time* in the case of the work of Fischer et al. All three propose generic, graph-based methods of modelling compliance rules, which allow their evaluation by comparing them to declarative deployment models of applications. Krieger et al. and Fischer et al. provide a method of modelling structural rules that separates the detection of rule applicability and the evaluation of rule fulfilment into two separate aspects. Zimmermann et al., on the other hand, combines detection and evaluation into one rule description, allowing not only whitelisting rules (describing permitted structures), but also blacklisting rules (describing forbidden structures). The deployment models in Krieger et al.'s work are described using the same metamodel that is introduced by the EDMM described in section 2.1.1. Zimmerman et al. and Fischer et al. use TOSCA deployment models, which follow a very similar metamodel to the EDMM. Since all of these approaches are aimed at compliance evaluation prior to runtime, they do not provide a method of retrieving and evaluating an infrastructure's current status. It was however found that these approaches could be extended to also model and evaluate rules that apply to infrastructures at runtime, given an instance model, and an example based on Krieger et al.'s work is covered in section 5.4.2.1.

In a more recent paper, Krieger et al. [KBF+20] propose a method of checking *behavioural compliance* at *runtime*, however their method does not check for structural compliance and is not topology-based. It instead allows for event-driven management of software compliance during runtime. Krieger et al. present an approach to monitoring the compliance of software architectures using architectural software patterns and complex event processing. They aim to answer the question how they can automatically monitor a system's architectural compliance, based on behavioural aspects described in architectural patterns. However, they do not provide a solution for how to manage structural compliance of an infrastructure.

In the paper [SBKL19], Saatkamp et al. take a different approach to describing and evaluating *structural compliance* rules at *deployment-time* than Krieger et al. [KBKL18], Zimmermann et al. [ZBKL18] and Fischer et al. [FBKL17]. Saatkamp et al. describe and evaluate rules using logic clauses, using the logic programming language Prolog in their prototype. This is again an interesting approach to integrate into this thesis, however it is too restrictive to require all users of a solution concept to model their rules as logic clauses, given that the solution concept should be generic. The solution proposed by this thesis is usable with an extensible variety of methods of describing compliance rules. This allows experts, who are responsible for describing rules, to use a method they are most familiar with. However, the approach described in [SBKL19] is used as an example method of describing and evaluating rules in section 5.4.2.2.

3.2.1 Extracting Instance Models At Runtime

The papers in this section provide approaches that can be used to extract instance models from applications and infrastructures at runtime. They are relevant to the approach proposed in this thesis, as it requires a representation of the infrastructure's current status, in order to allow for its evaluation. Integrating a generic solution concept with approaches, such as the ones described in the following papers, would allow for a wide applicability of such a solution concept.

Binz et al. [BBKL13] propose a generic, extensible framework for extracting so-called Enterprise Topology Graphs (ETGs), which represent a snapshot of the complete enterprise IT. The framework is plugin-based, successfully discovering and maintaining the status of an ETG, so that it continues to reflect the actual current status. This approach is very well suited to extracting a graph representation of existing infrastructure at runtime, however it requires the development of plugins, and the prototype is no longer available.

A more recent research project by Kleehaus et al. [KUSM18] proposes a method of extracting the current status of a microservice system's architecture. This takes into consideration the status extraction at runtime, without stopping a running system. However, this method relies on an existing service discovery tool being in use. A service discovery tool cannot be expected to always exist in the types of infrastructures this thesis looks into, making it unsuitable as a generic extraction method. This project may be suitable as a method for managing the compliance of microservice infrastructures during runtime, however not as the sole method for a generic solution concept.

Harzenetter et al. [HBB+21] propose a method of deriving, and if necessary enriching, instance models of applications at runtime. This is once again not a sufficiently generic approach for all use-cases the solution concept is required to be applicable to, however it does provide an approach for one method that can be used to retrieve an instance model of the current infrastructure status.

3.3 Miscellaneous

The following papers provide general useful information that is related to this thesis, however does not fit any of the previous categories. [KKR+13] and [KKW+14] provide a good overview of a compliance taxonomy, especially with regards to when compliance is checked, as well as what type of compliance is being checked. In [Spe19], Speth proposes a concept for unifying issue management across distributed teams and various issue tracking tools. Since compliance violations are also a type of issue which may require tracking, especially if linked to development, it could be of interest to integrate with a method such as the one proposed in the aforementioned thesis. Endres [EBF+17] describes two different methods of modelling deployment descriptions of applications, namely declarative and imperative procedural deployments. For each of these two methods, they identify basic pattern primitives and document the approaches as patterns which point out frequently occurring problems. This paper provides a good reference for methods of IaC deployments.

4 Motivation and Research Questions

This chapter covers the motivation behind this thesis and the research questions which it aims to answer. The motivation section explains the general process of managing infrastructure compliance at runtime, provides an overview of requirements the solution concept must fulfil, as well as some motivating scenarios. The research questions section describes the questions at hand, as well as a discussion of further questions which arise from posed requirements.

4.1 Motivation

This section first explains the general compliance management workflow required in an enterprise in order to keep their running infrastructure compliant. Next, an overview of the requirements that the solution concept is expected to fulfil will be given. Finally, it provides two motivating scenarios, described as EDMM-based instance models, together with some example rules that apply to these scenarios. All of this motivates the research questions which will be covered in the next section.

4.1.1 Compliance Management Workflow

The automation of currently manual processes is a common goal in enterprises today, with the goal of making processes less error prone, more efficient and less costly [OGP03]. Compliance management is a business process which takes place in all enterprises, with compliance rules stemming from a variety of sources, such as laws and internal regulations [KBKL18]. As defined in [HN05], a *process* consists of a set of connected activities that must be executed in a specific order, in order to achieve a defined goal. The manual execution of such activities means they are error-prone and costly, as human operators tend to have the highest risk of failure [OGP03]. A *workflow* is understood to be a fully or partially automated process. In order to better understand the problem at hand, the general process of managing runtime compliance in an enterprise is depicted in figure 4.1 as a workflow using BPMN[1]. BPMN is a standard designed for modelling business processes and workflows.

[1]The current BPMN specifications can be found here: `https://www.omg.org/spec/BPMN/2.0/PDF`

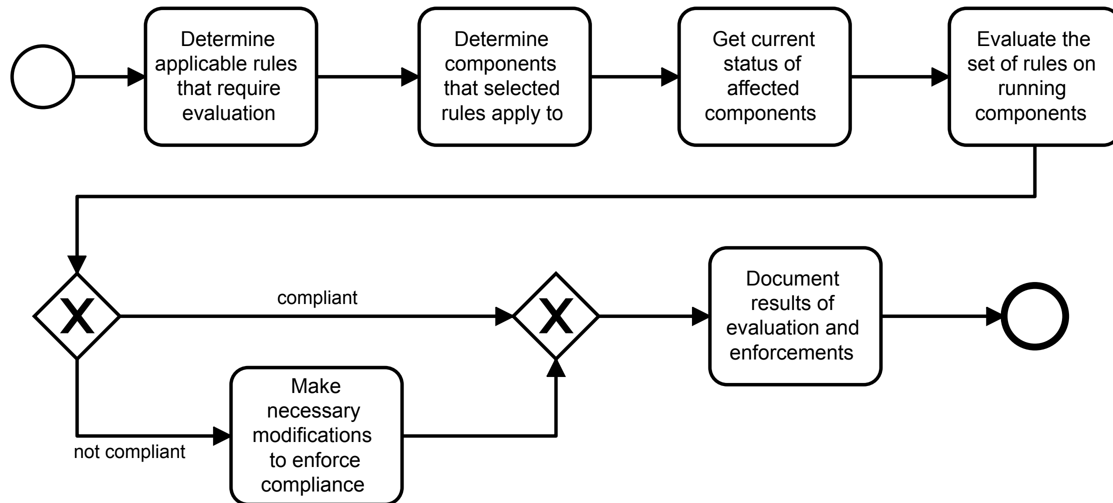

Figure 4.1: Workflow of runtime compliance management

The workflow consists of a number of activities and is triggered either when an infrastructure's compliance requires re-validation, or when it requires an initial validation after setup. First, a person must determine which rules have to be evaluated. This activity assumes that rules have previously been defined in sufficient detail in order to execute evaluation and to determine which components the rules apply to. Defining the rules is a separate process. Next, a person must determine which set of components the rules apply to, so as to know which components have to be evaluated using the previously chosen rules. In the following activity, the current status of the components, which were determined in the previous activity, must be ascertained. After applicable rules, affected components and their current status have been determined, the chosen set of rules can be evaluated on the set of running components. This can be achieved by a person or a programme which has access to all relevant components. In this activity, it is decided whether the components are compliant or not, thus influencing which activity is executed next. If any of the components are found to be non-compliant, the next activity is to make any necessary modifications in order to achieve compliance. If all of the components are compliant, this activity can be skipped. In both cases, the final activity is to document the results of rule evaluation and which modifications were made, so as to provide transparency and traceability. Finishing the documentation activity then also terminates one execution of the workflow.

All of these steps must be repeated on a regular basis. They should be repeated within a pre-defined time frame every time a rule or component changes, or on a regular basis regardless of changes. Components which have not been checked after such a time frame must be assumed to be non-compliant. In enterprises, many of these steps are executed manually; however, as mentioned previously, the process of manually managing compliance is highly error-prone, complicated, and intense in both time and cost [OGP03]. In-depth technical and security know-how, as well as awareness, is required to achieve system compliance, which the responsible person that is not a security expert (e.g. a developer) may not have [MMT+14]. The lack of maintenance or know-how of responsible people leads to vulnerabilities and thus a risk of security breaches as well as legal consequences [LSK09]. For this reason, it is important to automate as much of this

process as possible. Assuming that the rules and a representation of the components are available in a machine-readable language, it should be possible to automate many of the aforementioned activities.

Because of the risk and effort that is associated with repeatedly manually executing many of these steps, this work aims to assist with the automation of runtime compliance management, by proposing a design for a generic framework. In order to allow the creation of a framework architecture that is capable of automating a number of these activities in a way that is helpful to potential users, a set of requirements had to be collected to guide the solution concept.

4.1.2 Framework Requirements

The requirements covered in this section aim to provide an overview of what a solution concept should fulfil, in order to automate parts of the previously explained workflow, which have not yet been automated. These requirements will guide both the concept, as well as enable evaluation.

They were collected primarily by discussing the issue with potential framework users in an enterprise. Users of the framework can, for example, be developers or system administrators, and are not assumed to be security experts. Quantifying non-functional requirements was found to be beyond the scope of this thesis, so only functional requirements were collected. Each requirement is provided with a short, memorable title for future reference, followed by a very short definition. This definition will use terms as described in [Bra97] in order to allow concise definitions. These terms can be recognised by the words MUST, MUST NOT, SHOULD and SHOULD NOT, which are always in all capitals. For more details on their meaning, please refer to [Bra97]. Finally, following title and definition, each requirement will be described.

Req 1 (Add compliance rule)
The framework MUST allow users to add compliance rules.

Users must be able to describe new rules, which can be evaluated on different infrastructures and infrastructure components. This means users want to be able to re-use the compliance rules in multiple environments.

Req 2 (Extend compliance rule description methods)
The framework MUST allow users to extend the ways in which they can describe their rules.

Users should not be restricted to describing their compliance rules in a single, pre-defined way. They should instead be able to choose either an existing way of describing rules, or to add their own way of describing rules to the framework if required.

Req 3 (Extend infrastructure retrieval methods)
The framework MUST allow users to extend the methods of retrieving the current infrastructure status.

In order to be able to evaluate a running infrastructure, the status of all relevant components has to be retrieved. Users must not be restricted to a single method of retrieving the status of their infrastructure. This means they must be able to add own, new methods of retrieving an infrastructure's status, thus extending the framework in a way which suits their specific needs.

Req 4 (Re-use the framework)
Users MUST be able to re-use compliance rules and framework plugins, which means they MUST be able to set configuration values. In order to allow for re-use, the framework MUST be generic.

Users are not required to create custom solutions for every rule and every infrastructure they have to assess. Instead, they can re-use existing solutions for new use-cases with little effort. Use-case specific settings can be execution times or periods, having to assess new infrastructure components with private access credentials, and new values for compliance rules. All of these aspects may change for a single user at any point in time; however it should not require that the user starts the complete compliance management process over from the start.

Req 5 (Repeat rule evaluation regularly)
Users MUST be able to evaluate rules repeatedly. They MUST NOT be required to manually trigger an evaluation every time rules have to be evaluated.

Users have to execute the activities required for compliance management on a regular basis. These repeated activities should be decoupled from a user's interaction with the framework. This means it should be possible for a user to set up the compliance management activities once and then let the framework trigger future executions.

Req 6 (Evaluate rule fulfilment)
Users MUST be able to find out whether a provided rule or set of rules is fulfilled or not. Users SHOULD be able to find out which infrastructure components are affected by compliance violations.

Users have to be able to find out whether the rules they provided the framework with, for evaluation on the infrastructure the user provided, are fulfilled or violated. The user should also find out if an error occurred. An error should not be masked as a compliant or non-compliant state, the user should get a response that clearly indicates either compliance, non-compliance or an error.

The requirements show what users expect from a solution concept, and help show the functionality which a solution concept must provide. The next section provides two motivating scenarios, which aim to provide a better understanding of the types of infrastructures and compliance rules users may work with, and which this thesis will be looking at.

4.1.3 Motivating Scenarios

In order to better understand what users work with, that are involved in the runtime compliance management process, as well as to better understand the solution concept, a motivating scenario will be used throughout the rest of the thesis. Two infrastructure instances will be introduced as motivating scenarios. For each motivating scenario, a non-compliant instance model will be shown, together with a set of compliance rules in text form. Primarily the first of the scenarios will be used as a running example throughout the remainder of the thesis.

The motivating scenarios will be described using the EDMM metamodel explained in section 2.1.1. Nodes in the graph represent Components, whereas edges represent Relations. The Component Types are written in brackets and the name of the component, which is also its unique identifier, is written before the brackets. Any number of Properties can be associated with either a Component

or a Relation, Properties are depicted as key-value pairs in the boxes below the Component Type and associated name. Relations can also have any number of key-value Properties, however they are not depicted in the motivating scenarios. Relations' source and target are represented using arrows. Relation Types are shown as a legend in the models, only the *hostedOn* and *connectsTo* relation types are used in the following scenarios. Operations and Artifacts are not included in the motivating scenarios. Approaches to evaluating the presented rules, how the solution concept of this thesis captures the compliance violations, and finally a compliant instance model, will be discussed in section 5.4.

4.1.3.1 Scenario 1

Figure 4.2 shows an application system instance which contains multiple violations of structural compliance rules. From right to left, the instance consists of a web application that connects to a database containing user data, which is also used by a Java application. All of these components are hosted in the cloud. The web and Java applications are hosted on AWS IaaS services which provide compute resources to a client[2]. The database is hosted on an AWS PaaS service that provides clients with relational data storage[3], without requiring that clients manage the DBMS and compute layers. The web server uses Apache[4] to serve web applications. In the leftmost stack of the figure, a second Java application is shown, which is assumed to be private and hosted on-premise, as access to this application must be restricted. The hosting operating system (OS) has the property "Frozen", which indicates a component with approval to run an outdated Ubuntu OS. This instance model shows a *heterogeneous* infrastructure, as multiple types of infrastructure components (physical, IaaS, PaaS) are included, as well as an infrastructure *at runtime* as it is assumed to be a currently running instance. Between the deployment of these instances and the current status, user interaction has modified any number of attributes, leading to a non-compliant instance.

A number of rules are assumed to apply to this application system and all of them are violated in figure 4.2. In the following, a subset of rules is described which can be applied to the figure. Compliance Rule is shortened to CR for easy future reference.

CR 1: *All web applications must be hosted in the same region as connected databases.* A reason for such a rule could be to prevent transferring sensitive data across regions, or to reduce the probability of data loss during transfers. Furthermore, restricting cross-region communication could help reduce data transfer periods. In scenario 1, this rule is violated by the instance of WebApp and the UserData database, as they are hosted in two different regions.

CR 2: *All instances of the Java application PrivateInternalApp must be hosted on a JRE version 15.* This rule implies that no earlier or later versions are allowed, it is assumed to be a specific requirement of the PrivateInternalApp. The instance in motivating scenario 1 is however running on version 16.0.2 of the JRE, making this version non-compliant with CR 2.

[2] AWS EC2: https://docs.aws.amazon.com/ec2/index.html
[3] AWS RDS: https://docs.aws.amazon.com/rds/?id=docs_gateway
[4] Apache documentation can be found here: https://httpd.apache.org/docs/2.4/ and the product page here: https://httpd.apache.org/

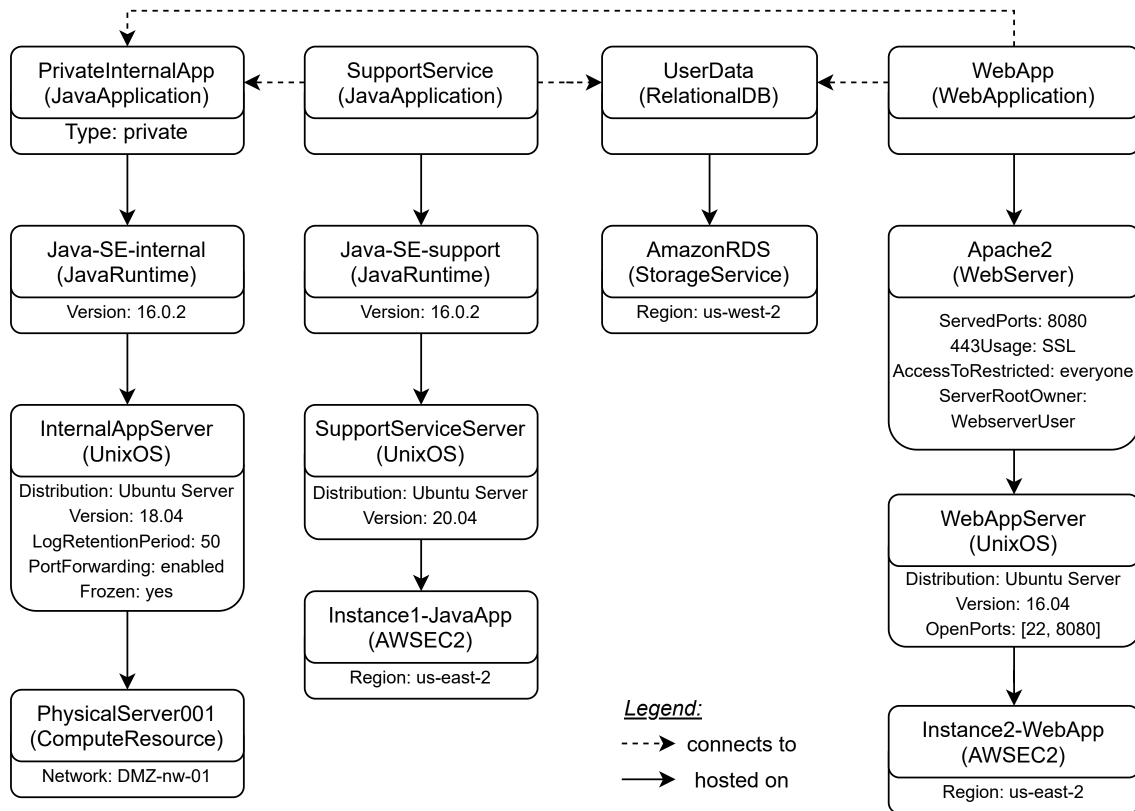

Figure 4.2: (Scenario 1) An example application system infrastructure with compliance violations.

CR 3: *All web servers must serve their web applications on ports 80 or 443, and operating systems they are hosted on must not have any additional ports open.* This rule checks that all web servers in an infrastructure serve the correct ports, as well as the hosting operating systems. Any web servers and hosting operating systems which serve additional ports must show up as non-compliant, because these allow room for potential security breaches. In scenario 1, the Apache2 web server, and the OS it is hosted on, are assumed to have been unintentionally configured using development configuration, instead of production configuration. For this reason, the web server is still serving its application on port 8080, and the SSH port of the OS is open, thus violating CR 3.

CR 4: *Only instances of SupportService are allowed to connect to instances of PrivateInternalApp.* As instances of PrivateInternalApp are assumed to perform sensitive operations, in this application system, any other connections are not permitted for security reasons. Access regulation to this component must be more restrictive than on other applications and components, so permitting connections would risk leaking sensitive information.

CR 5: *All data that is transferred across a network must be encrypted.* Encryption for data in transit, i.e. when being transferred, is required to prevent data leaks. If such data is not encrypted, an attacker can easily obtain access to the data. The encrypted property is not shown in the figure of the motivating scenario, however connections are assumed to be unencrypted, thus violating this rule.

The aforementioned example rules are of a *structural* nature because they are all violations of the instance model structure, which the application system should fulfil. The rules and their violation affect the infrastructure components and relations between components, and the violations can be detected by analysing the topology. This is in accordance with the definition of structural compliance, which was presented together with other kinds of compliance rules (e.g. behavioural) in section 2.4.3. Furthermore, although many of them can be checked during *deployment-time* and some even during *design-time* (see section 2.4.3 for explanations), the affected aspects of infrastructure components in this instance cannot be assumed to remain the same during *runtime*. As a result, these rules aim at checking *structural* compliance *at runtime*. See section 2.4 for a taxonomy of the italicised terms.

4.1.3.2 Scenario 2

This second scenario represents a set of multiple independent services provided by one service provider and hosted on one service provider infrastructure, that however do not connect to each other. This scenario can be extended to any number of physical or virtual components that make up a service provider's infrastructure, grouped in whatever way the provider sees fit. An example grouping could be by department responsibility or area of use.

Figure 4.3 shows two stacks, the left of which provides relational data storage, and the right stack provides a web server infrastructure for multiple web applications. The IBM DB2 DBMS[5], which can host relational databases, has multiple properties. DB-territory shows the territory used to create the Database1 database and is used by the database manager when processing data that is territory sensitive. DISCOVER_DB controls whether information about a database is returned to a client when a discovery request is received at the server (enable) or not (disable). CF_DIAGLEVEL specifies the type of diagnostic errors that will be recorded in logs, where 0 means no diagnostic data is captured, 1 means only severe errors are captured, and 2 means all errors are captured. Further levels allow the additional capturing of warnings and informational messages. This stack is hosted on-premise on a physical server. A possible explanation for this sort of stack is the storage of sensitive data which has to be stored on-premise for confidentiality and privacy reasons. The web server component, which hosts multiple web applications, is itself hosted on a virtual machine, which is provided as an IaaS service, in the cloud. It uses NGINX to host the web applications.

This scenario is also *heterogeneous* due to the different types of infrastructure components, these being an IaaS service and a self-hosted physical server. It is again assumed to be currently running, and any of the components may have been modified by user interaction.

All rules from scenario 1 also apply to this scenario, however none of them are violated. For demonstration purposes, two additional rules are defined for this scenario.

CR 6: *All databases must be hosted on an internal, protected network and must be located within Germany.* This is a requirement for sensitive data, which must be treated in accordance to German law and enterprise-specific regulations. It must thus be avoided that the data is stored on servers anywhere where other, less strict, rules may apply. The data must also not be stored anywhere

[5] IBM DB2 documentation can be found here: https://www.ibm.com/support/pages/db2-database-product-documentation

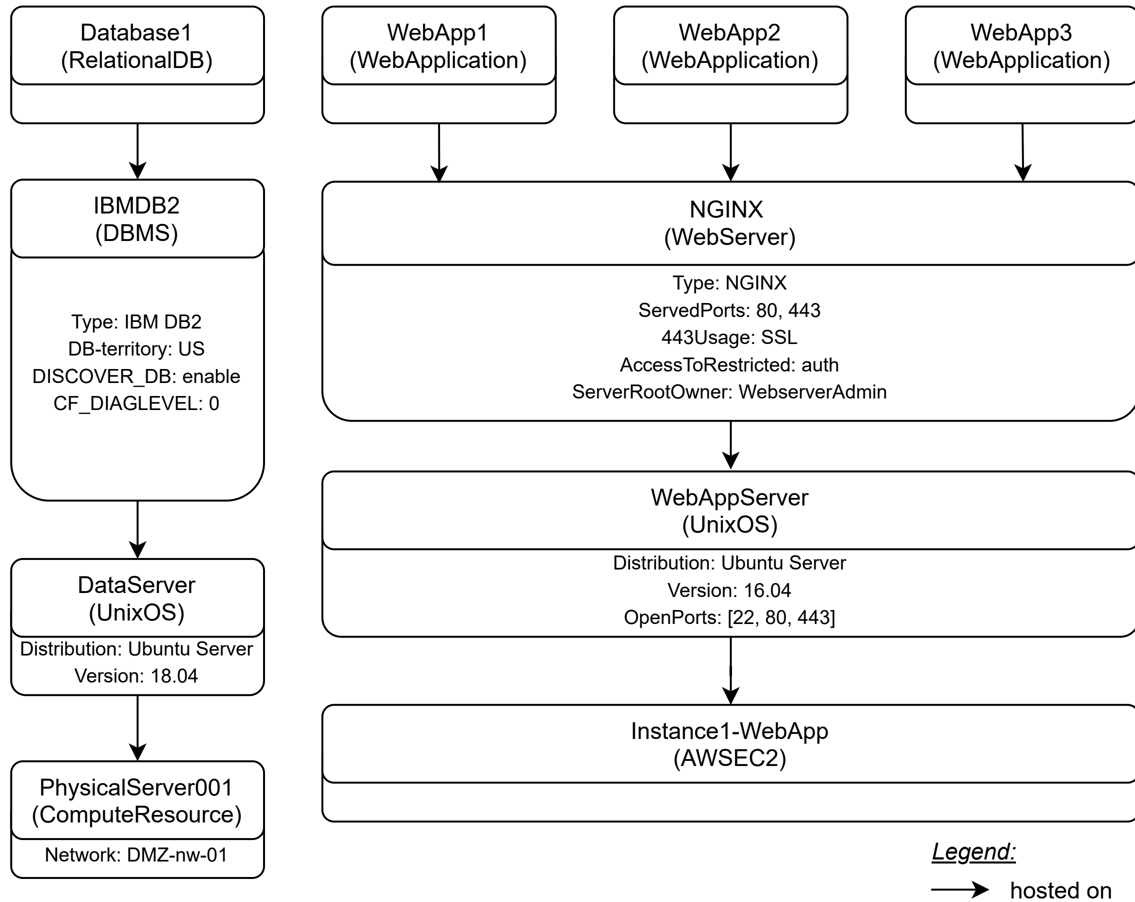

Figure 4.3: (Scenario 2) Example set of infrastructure components with compliance violations.

that is directly accessible from external networks, thus the requirement for it to be located on an internal network. The database in this second scenario is located in the US, and on a network that is accessible from the internet, which leads to a violation of this rule.

CR 7: *All IBM DB2 servers must have DISCOVER_DB set to disable and CF_DIAGLEVEL set to at least 2.* This rule ensures that databases storing sensitive data are configured in a way that conforms with further requirements for sensitive data. Information about the database must not be leaked to database users, which is why DISCOVER_DB must be disabled. The diagnostic level must be high enough for all errors to be captured, so as to be able to diagnose problems.

These rules are of an equally structural nature as in scenario 1. They affect the infrastructure instance and can be detected by analysing the instance model. This section has described the general workflow of runtime compliance management, as well as an overview of the requirements a solution concept should fulfil, and finally presented two motivating scenarios, the first of which will primarily be used as a running example. The next section will describe the research questions that were found to be unanswered thus far in the context of the problem.

4.2 Research Questions

As mentioned in chapter 3, extensive research has already been done on automating business processes in general, as well as automating compliance management in particular. Existing automation approaches aim to automate compliance management at various phases of an application infrastructure's life cycle, as explained in section 2.4.3. However, there does not yet appear to be any research on the management of structural compliance (meaning compliance of the application infrastructure's topology), at the time when infrastructure is already up and running. In order to fill this gap in research, this thesis will attempt to answer the following research questions (RQ):

RQ 1: *How can runtime structural compliance rules be described?* The description must be machine-readable, and at the same time human-readable so as to allow rules to be defined by experts. Taking this into account, methods of describing rules which affect infrastructure components at runtime must be looked into.

RQ 2: *How can the compliance rule description be made generic?* As there is no generic means of describing compliance rules yet, this thesis will look into the possibility of generalising different methods of describing compliance rules into one, more generic, method. For this purpose, existing methods of describing rules must be looked into, and how they can be made generic enough for a compliance management framework to work with a variety of methods.

RQ 3: *How can a framework be developed that automates compliance management at runtime for various rule types and various infrastructures, allowing extensibility in the future?* Finally, this thesis aims to propose a feasible solution concept for automating a subset of the activities in the described compliance management process, or to rule out the feasibility of chosen approaches.

4.2.1 Discussion

During initial research, requirements collection and the definition of the research questions, a number of questions arose that must be taken into consideration during the framework design.

In order to be able to analyse the current infrastructure status, the framework design must take into consideration how the infrastructure status can be represented. This representation must be machine-readable in order to allow automation. Furthermore, this status must be newly retrieved for each set of rules that the framework evaluates, i.e. for each framework execution. This should ensure that the framework evaluates a state which is as current as possible, not a hypothetical state. The framework design must provide methods of answering how to represent and retrieve the current infrastructure status.

A further question is how to retrieve and describe the compliance rules which the framework must evaluate. To allow automation, these must be machine-readable too. In order to make the framework as generic as possible, framework design should take into consideration possibilities of generalising the description of compliance rules. This should prevent limiting the variety of supported compliance rule descriptions to a small, non-expandable subset.

When looking at what the user gets back from the framework, it becomes clear from Req 6 that a simple boolean compliant/non-compliant response is not sufficient. This raises the question how best to model the result which the framework returns. As far as possible, the result should not restrict what is possible in steps following the rule evaluation in Req 6. As is visible in the

workflow of runtime compliance management in figure 4.1, evaluation is not the final step of managing compliance, so the framework design should also not assume this. All of this must be taken into consideration when designing how the framework describes and reports back the compliance violations it discovers.

Finally, Req 2 and Req 3 make it clear that the framework cannot know which types of infrastructure components and compliance rule it may be confronted with in the future. This means the framework design must answer the question how the framework will be able to work with currently unknown infrastructure components and compliance rule descriptions. It must do this, as far as possible, without limiting the framework functionality and generality.

This chapter has motivated the reason behind the research conducted in this thesis, and presented two motivating scenarios as well as the research questions. In the next chapter, the approach taken in this thesis is described, in order to answer both the questions mentioned in this subsection, as well as the research questions.

5 Solution Concept

This chapter describes the approach taken in this thesis to provide a solution concept for the problems posed in the previous chapter. The solution should provide a way of managing infrastructure compliance at runtime in a heterogeneous environment. The first section explains the overall approach and contributions made by this thesis. The next section covers how detected violations can be described in a model. Afterwards, the architecture concept as well as the single components are described in detail. In the final section, multiple examples are used to describe how the framework can be extended. These will include descriptions of example rule evaluations using each of the example methods.

5.1 Approach

The solution approach chosen in this thesis is model-based, meaning the inputs to the framework are models that describe the compliance rules, and a model of the instance of the infrastructure which is being assessed. Evaluation of the compliance rules takes place by assessing the provided models. The structure of the infrastructure instance is pre-defined, whereas there are multiple types of rule descriptions which can be used. Several examples of both rule description and evaluation will be covered in section 5.4, however the framework is designed in an extensible way, so that these example methods aren't the only working methods. The framework also requires a fixed output structure for compliance violations.

This thesis makes four contributions to its field of research. The first contribution is an Issue-extended Essential Deployment Metamodel (I-EDMM), which is an extension of the EDMM metamodel, and allows the description of an instance model with compliance violations. I-EDMM is the required output structure of the framework, the architecture of which is the second contribution. The framework architecture contributed by this thesis is a reusable, extensible solution approach to allowing the automated management of infrastructure compliance at runtime. The third contribution is a metamodel for rule descriptions, that allows a generic description of compliance rules. The final contribution is a prototype which takes an instance model, provided as a graph description, and analyses it using compliance rules which are described as programmes. After analysis, the prototype returns an I-EDMM which describes the detected compliance violations. The prototype can be provided with multiple rules, allowing a set of rules to be checked for a given instance model. The following sections 5.2 and 5.3 will describe the I-EDMM and proposed framework architecture respectively. The prototype will be described in chapter 6.

5.2 Issue-Extended Essential Deployment Metamodel

The I-EDMM is a metamodel, which allows the description of an instance model with issues by adding these to the model as additional nodes, with relations to any components or relations that a given issue affects. Compliance violations are a type of issue, thus they can be represented using issues in this metamodel. However, a variety of issues can be described using the I-EDMM. As the I-EDMM is based on the EDMM, please refer to section 2.1.1 for details on EDMM.

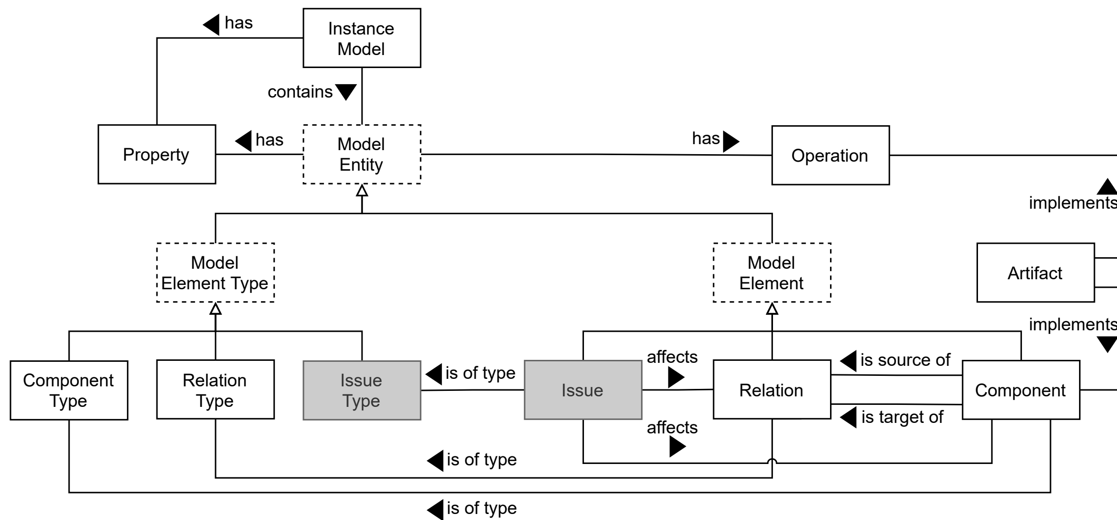

Figure 5.1: EDMM extended by issues (I-EDMM)

The extension, shown in figure 5.1, consists of an additional Model Element, the Issue, and an additional Model Element Type, the Issue Type. A single issue can attach to any number of relations or components. If there is an edge from an issue to another model element, this means that the issue affects the element it attaches to. Thus, an issue attaches to other model elements using a relation of the *affects* type.

Due to the fact that issues inherit from model element and model element inherits from model entity, there exists a transitive inheritance from model entity to issues. This means an issue can also have an operation and an artifact which implements the operation. This can be used in the future for optional issue resolving, meaning an operation could be a fixing operation associated with a particular issue and issue type, which is implemented by an artifact that can be used for all issues of this type. Furthermore, issues can have properties, just like any other model element. These properties can for example be used to describe what the issue is.

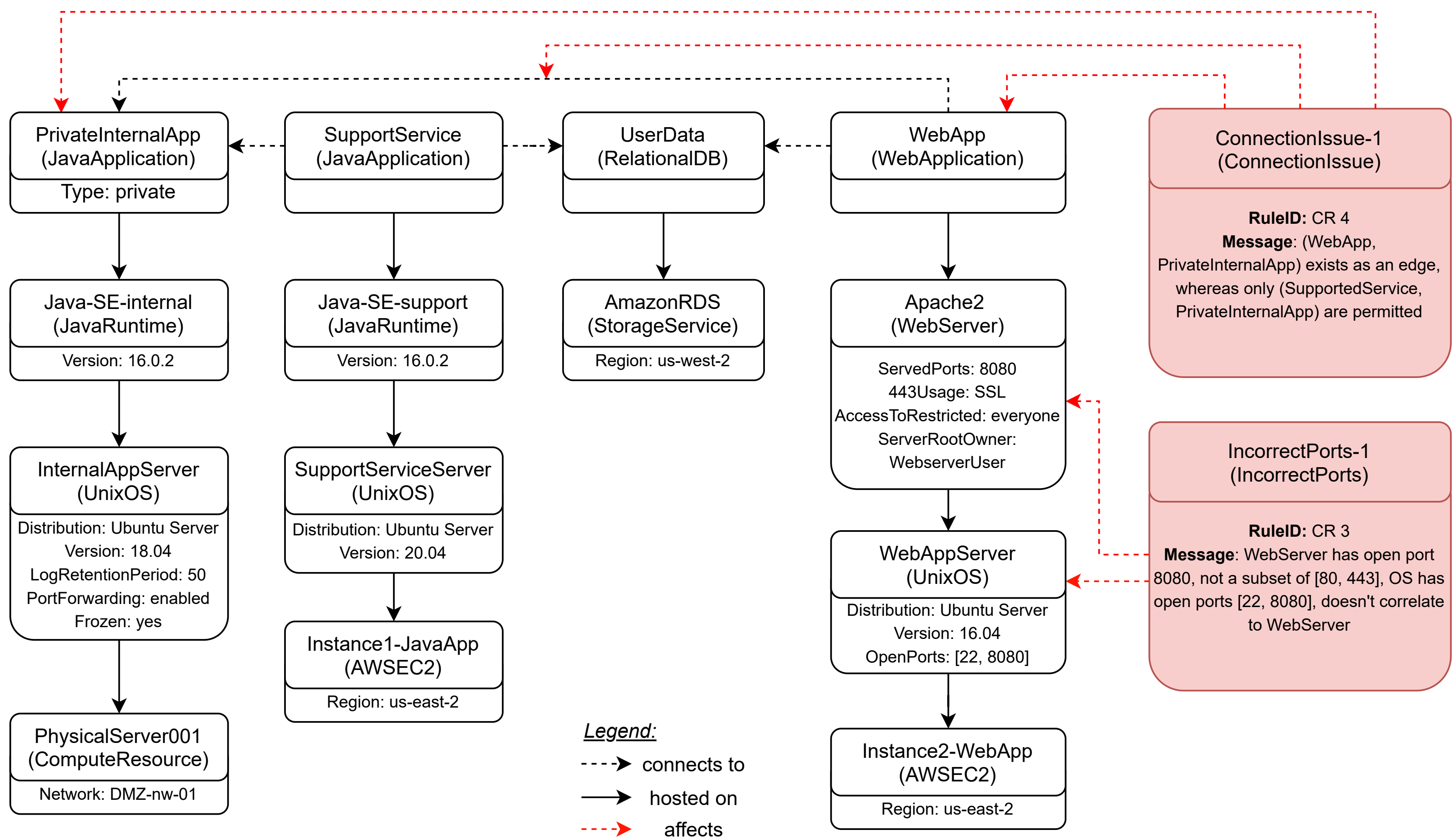

Figure 5.2: An example I-EDMM model for violations of CR 3 and CR 4 in motivating scenario 1

In figure 5.2, an example I-EDMM is depicted, assuming that rules **CR 3** and **CR 4** in scenario 1 have been evaluated. The instance model is identical to scenario 1 with violations, as shown in figure 4.2, except that it is now annotated with an issue element and multiple connecting edges per violated rule. These elements can be seen in highlighted in red. The violation of **CR 3** is represented in the "IncorrectPorts-1" issue of type "IncorrectPorts" that attaches to the WebServer and OS components using an "affects" relation. **CR 3** states that all components of type WebServer must only serve ports 80, 443 or both, and that the hosting OS must correlate to the WebServer ports. The issue nodes have a type (shown in brackets), a name (shown before brackets) as well as properties RuleID and Message, shown as labels. Each property provides an insight into details of the detected issue. The violation of **CR 4**, stating that there must only be connections to instances of PrivateInternalApp from instances of SupportService, is shown in the issue of type ConnectionIssue. Same as the first issue, it has a name, issue type, as well as properties RuleID and Message, providing an insight into what is violated. This issue connects to multiple model entities with an "affects" relation, as it affects not only the relation between the instances of WebApp and PrivateInternalApp, but also the components themselves.

As shown with these two issues, any one issue can affect multiple model entities, however an issue cannot affect another issue. In addition, one rule cannot produce multiple issues, which is a convention introduced by the approach in this thesis. However, multiple rules being checked in succession on a single instance model can lead to multiple issues attaching to one model element, so the I-EDMM metamodel allows for this. This could for example happen if there were an additional rule which is violated by the Apache2 component's configuration, leading to this rule also attaching a different issue to the WebServer component. If no violations exist, the instance model remains unmodified, with no Issue elements. An I-EDMM is what results from a framework execution. The framework architecture will be presented in the following section.

5.3 Architecture Concept

In order to describe the framework architecture which this thesis proposes as a solution concept, this section will first give an overview of the architecture, along with a description of two workflows. The first of these describes the workflow of compliance management and the required user activities from a user perspective when using the framework. The second workflow describes the interaction between the elements of the framework, representing the process of framework execution and its activities from a framework perspective. After this, each architecture component will be described in greater detail, and finally the concept will be made tangible using examples for the framework extensions and their respective evaluation methods.

5.3.1 Architecture Overview

This section starts with an overview of the final framework architecture, which is shown in figure 5.3. The grey elements in the overview are components of the framework. Each component fulfils a separate task. The framework provides extensibility using plugins, which are depicted in the figure using the white elements that latch onto the Instance Model Retriever (left) and the Framework Core (right). Plugins are themselves programmes that add new functions to a host program without altering the host programme itself.

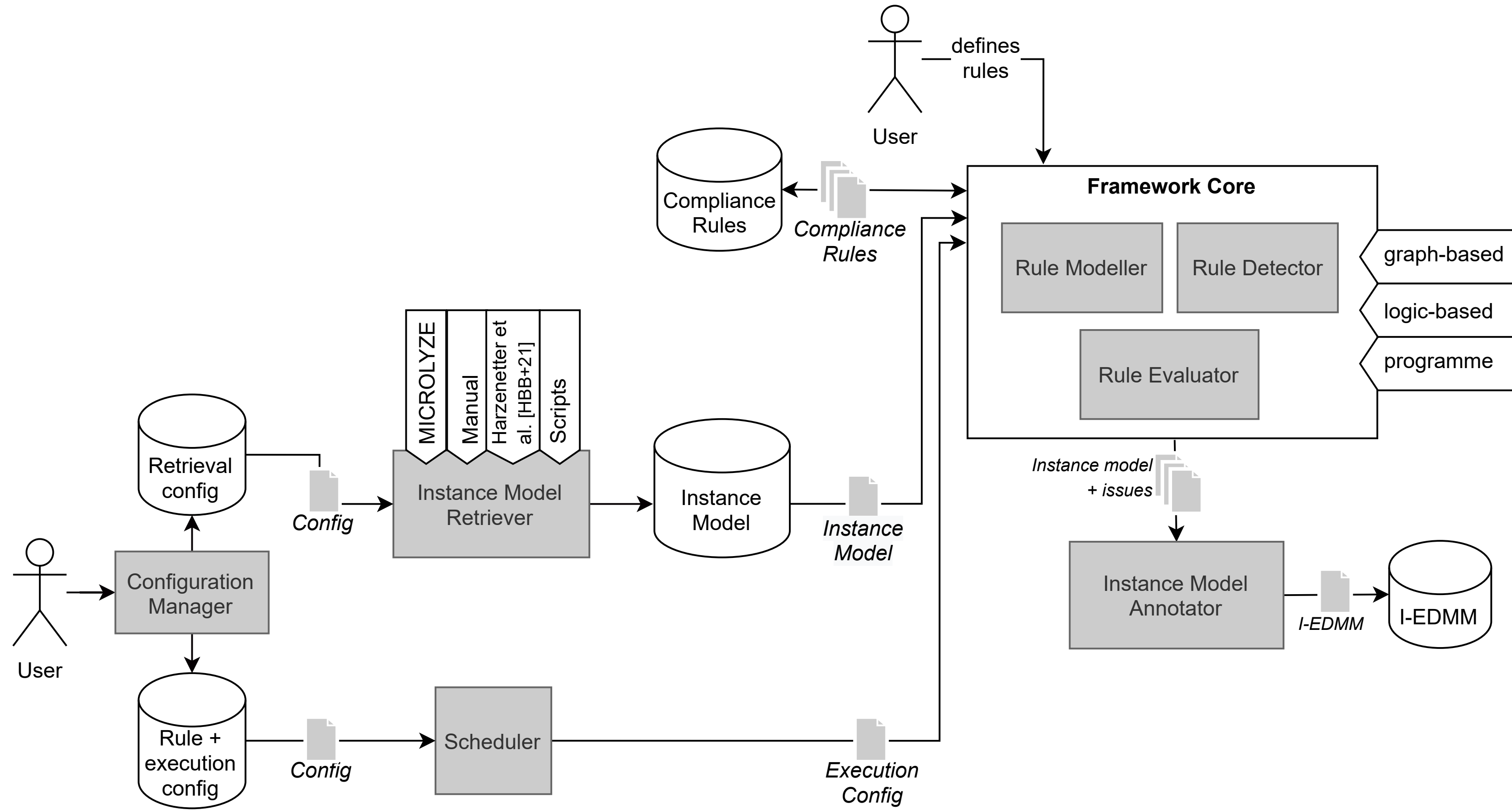

Figure 5.3: Overview of the compliance automation framework architecture

Plugins allow programmers to extend a host programme, while at the same time allowing the user to remain within the host programme's environment. They are a method of adding flexibility to a programme, in this case the framework, when one cannot anticipate all possible functions future users may want in the design phase [Ste19].

Looking at the components depicted in figure 5.3, the plugin-based framework core consists of a Rule Modeller, Detector and Evaluator. Furthermore, the core has an *Orchestrator*, which is not depicted in the figure. The Orchestrator manages the execution of sub-components, plugins and the storage of intermediate values. A framework user can add rule descriptions through the *Rule Modeller* component. This component uses a plugin's validator to validate rules the user wishes to store, and stores the compliance rules in a repository for future access and reuse, if the description was found to be valid. Furthermore, once rules have been added, a user configures required rules, an execution schedule, and one or more retrieval method plugins using the *Configuration Manager*. This completes all initially required user interaction with the framework. The *Instance Model Retriever* component uses the chosen plugins and corresponding configuration in order to get the current infrastructure status. The resulting instance model is temporarily stored by the Orchestrator for the duration of the framework execution. The *Scheduler* component uses an execution schedule to trigger a framework execution at a time chosen by the user. The framework core's *Rule Detector* component and *Rule Evaluator* component each execute a plugin's detection and evaluation method, respectively, when called by the Orchestrator. Using the retrieved instance model and configured compliance rules, the detector can use a plugin's detection method to decide whether a rule is applicable to the instance model or not. The *Rule Evaluator* is triggered by the Orchestrator for each rule that applies to the current instance model. Using a suitable plugin's evaluation method, it can evaluate the rule for fulfilment and receives one of two things back from the plugin. The evaluator either receives back nothing, if the rule was fulfilled, or it receives back an issue together with a list of the unique identifiers of relations and components which are affected in the instance model. The Orchestrator collects this information until all rules have been processed. Finally, after rule processing is finished, the Orchestrator calls the *Instance Model Annotator*, which annotates the instance model with the detected issues, resulting in an I-EDMM. The Instance Model Annotator then stores this I-EDMM, which the user can retrieve. In the solution concept, a plugin can either be a combination of a rule description validation, rule detection and rule evaluation method, which is referred to as a *compliance rule type plugin*, or an instance retrieval method, which is referred to as an *instance retrieval plugin*. In order to allow the adding of plugins, one more component is required: a *plugin manager*. This component is not depicted in figure 5.3. The plugin manager must allow the user to choose and add new plugins, which means plugins are installed and registered by the manager. After this has been done, the plugins are available to the framework.

5.3.2 Using the Framework

In order to use the framework, provided that the required plugins already exist and have been installed using the plugin manager, the activities described in figure 5.4 must be executed. This figure uses BPMN notation once again, as it represents the now partially automated business process of compliance management at runtime, which the proposed framework makes possible. The framework is treated as a black box in this workflow.

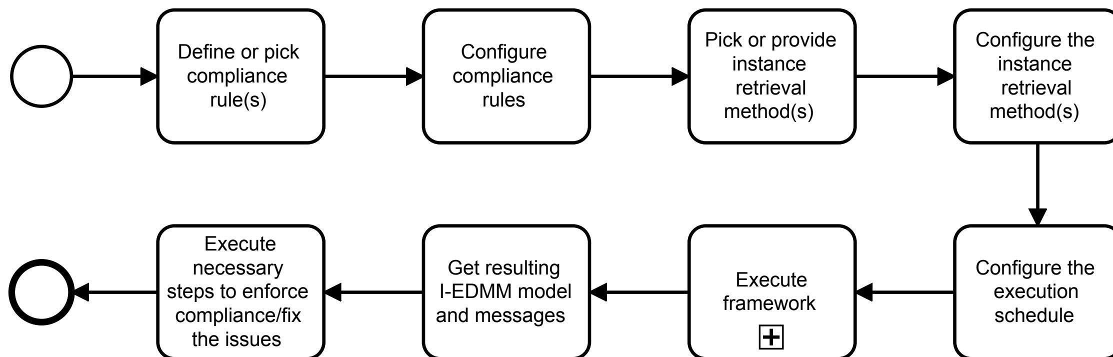

Figure 5.4: Workflow of using the framework

The workflow is triggered when a user, such as a developer or administrator, wishes to get the compliance status of a running set of infrastructure components. The user must first **select one or more existing compliance rules** which they wish to have evaluated. If a compliance rule, which the user requires to be evaluated, is not available, the user must either define this rule themselves or find an expert that can do so. After this activity is completed, the second activity is triggered. It consists of **configuring the selected compliance rules**. The configuration in this step includes the assignment of any values which are required in order to bind the selected rules to the infrastructure under assessment. Assignments can for example be values for a set of labels (i.e. properties) that may be associated with a component in the rule. An example binding in motivating scenario 1 (see figure 4.2) would be the name and value of the version property of the JRE hosting the PrivateInternalApp. Initially, in a rule definition, this should be two variables, which are assigned the values “Version” and “16.0.2” upon binding of the rule to the infrastructure. After rule configuration and thus binding is completed, the user must **pick or provide one or more instance retrieval methods**, which the framework can use to get the current infrastructure status. If the required instance retrieval methods are not yet available as a plugin, the user must add a new plugin to the framework. Next, **the selected retrieval methods must be configured**, similar to the rules previously. This activity binds the chosen methods to a running infrastructure or application instance. What exactly the configuration requires depends on the specific method, however examples could be providing access credentials to infrastructure components. If there is no suitable automated method of retrieving an instance model of the current state, it is possible to manually describe the status as a model. This would be the manual method and does not require configuration, as the framework does not care how it retrieves the instance model, as long as it receives a model with the required structure. Finally, combining the previous two configuration activities, **execution schedule configuration** is the last step before the framework can be executed. A schedule always includes the time of execution (e.g. immediately, at night, on a specific day), and the number of executions (e.g. once, daily, weekly). Executing once immediately would be a manually triggered framework execution. Furthermore, the execution configuration requires that bound compliance rules (i.e. including configuration) are provided, as well as the instance retrieval methods together with their configuration. For all executions which are scheduled to recur or to occur in the future, the configuration should be stored, else the execution would require user interaction at the time of execution. Storing configuration allows for all activities executed thus far to be skipped in recurring executions. Now that everything has been selected and configured, the next activity is **framework execution**, which is triggered by the scheduler in accordance with a schedule. Execution includes

multiple subprocesses, but is treated as a black box in this workflow, as the inner workings of the framework are not relevant to the user. See figure 5.5 for the subprocess. Once the framework has finished this activity, the user **gets the resulting I-EDMM model**. This allows the user to deduce any compliance violations which the framework has found. The final step of this workflow is to execute any steps necessary to **resolve found issues**. In the current concept, this step is manual, however it should be possible to automate this activity in the future. The concept has taken future automation of this activity into consideration. With this activity, the workflow terminates.

5.3.3 Framework Execution

The workflow depicted in figure 5.5 shows the activities in a framework execution, which is the sixth activity of the workflow in figure 5.4, from the previous section. This workflow is triggered by the scheduler, depending on an execution configuration. The execution of the activities in this workflow, as well as the passing of data between them, is managed by the framework's *Orchestrator*. In the first activity, the **Instance Model Retriever component** of the framework is executed using the plugins associated with the execution configuration, in order to retrieve the current infrastructure status, as an instance model. For this purpose, the retriever must use the retrieval configurations and plugins associated with the schedule. It then passes the instance model to the next activity, which **executes the Detector component** of the framework. Using the instance model it receives from the previous activity, and the rules bound to this instance, which are retrieved from the rule configuration repository, the detector decides whether an associated rule applies to the instance or not. For this purpose, it uses the detection part of the plugin associated with the current rule type. This activity iterates for each rule that is bound to the instance model. The next activity is chosen, on a per-rule basis, depending on the result of the detector.

If the detector does not find the current rule is applicable, it skips the evaluator activity and immediately checks whether the current rule was the last rule or not. If the detector decides that a rule is applicable, the next activity is to **execute the Evaluator component** of the framework. The evaluator receives the instance model and applicable rule. Using these and the evaluation plugin associated with the rule type, it decides whether the rule is fulfilled or not, i.e. whether the instance is compliant with the provided rule. The evaluator receives back either the detected issues together with the affected model elements, or the information that no rule was violated.

If the current rule was not the final rule in the set, the next activity is a loop back to the detection activity. However, when the last rule in the set has been reached, the next and final activity is to merge all found issues into a single I-EDMM. The *Instance Model Annotator* component of the framework is responsible for this activity and finally stores the resulting I-EDMM in a repository which the user can access. After having stored the I-EDMM model, the workflow can terminate.

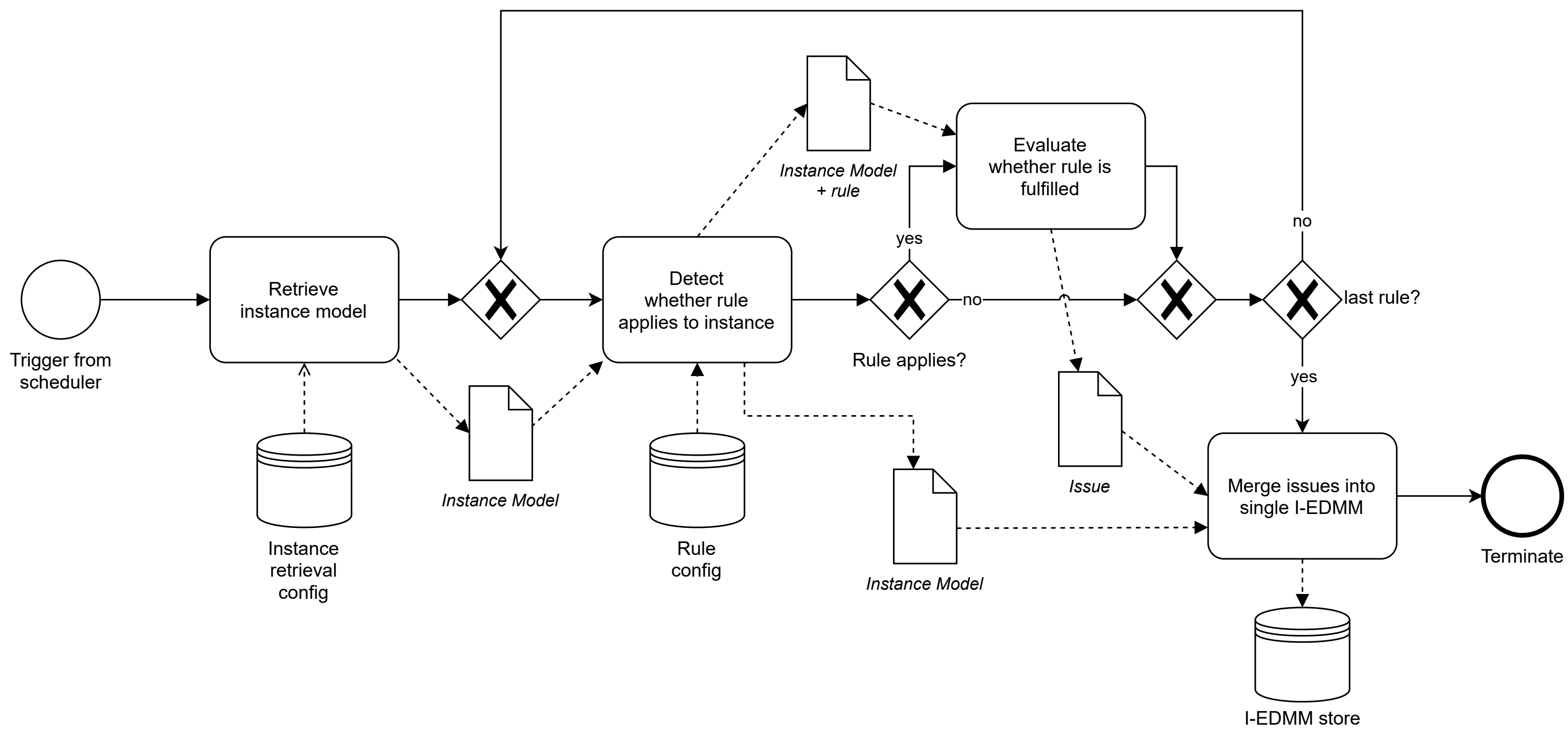

Figure 5.5: Workflow of the framework execution

The plugins, which are required for instance model retrieval, rule detection and rule evaluation, are expected to be made available to the framework by the plugin manager. This means the plugins are installed and registered with the framework prior to usage. The framework can then address the plugins using endpoints the framework is provided with after plugin registration.

Details on the workflows for adding a plugin, and defining a new rule if the rule type already exists, are not explained in detail here. Instead, the activities associated with these workflows will be covered in sufficient detail in the examples section. Following this overview of the framework architecture along with the associated workflows, the following sections will go on to describe the components of the framework in more detail.

5.3.4 Framework Core Components

As was briefly explained in the architecture overview, shown in figure 5.3, the framework core is plugin-based, using compliance rule type plugins, and consists of a rule modeller, a detector, an evaluator, and an orchestrator. The orchestrator is omitted from figures. The framework core's architecture is repeated in figure 5.6.

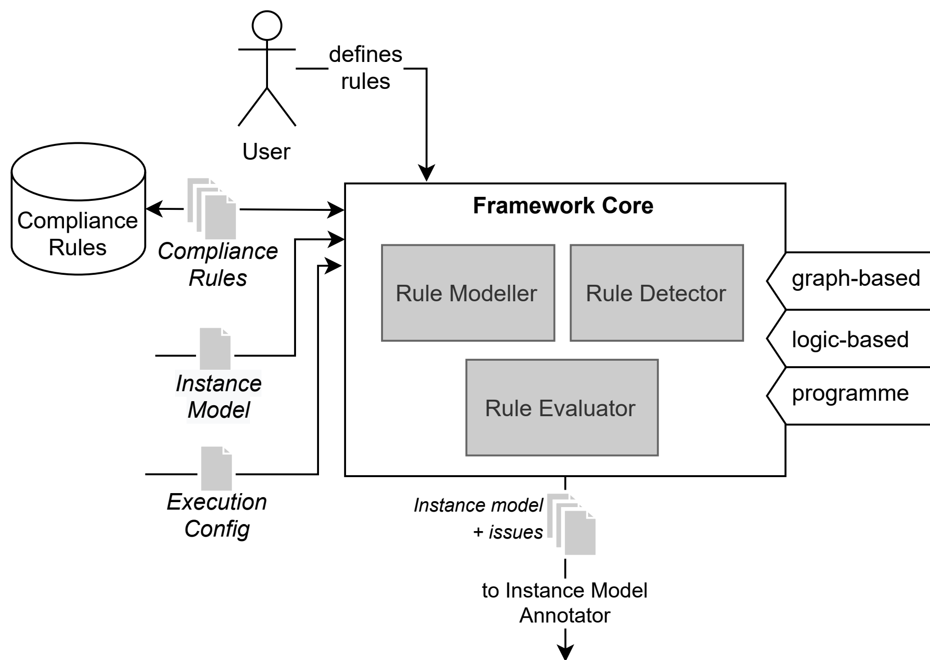

Figure 5.6: Framework Core

5.3.4.1 Compliance Rule Modeller

This section first describes the Compliance Rule Modeller in more detail. The Compliance Rule Modeller provides an interface for any modelling tool that may be made available through a compliance rule type plugin. Plugins can provide any custom tools which users who describe rules

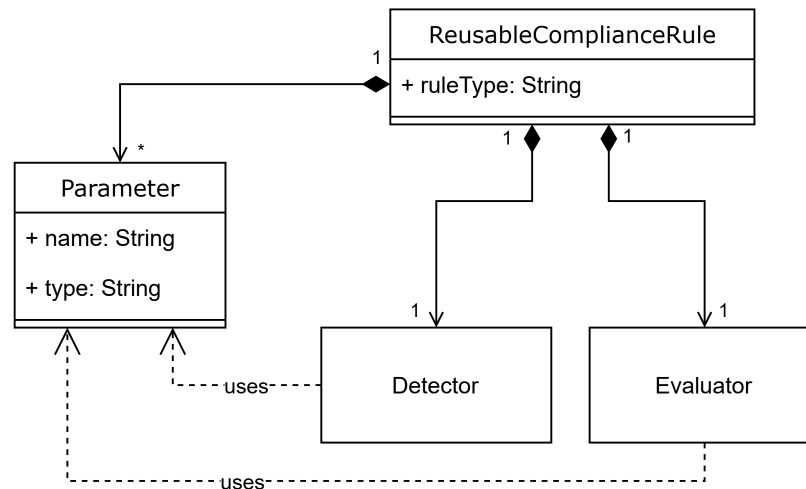

Figure 5.7: Compliance Rule Metamodel

may employ, such as the Winery tool for graph-based compliance rule description[1]. This means the framework itself does not provide any modelling functionality in the rule modeller component. The rule modeller does however provide users with the ability to create, read, update or delete compliance rules. The modeller manages these interactions with the central rule repository.

Before storing the rules, the rule modeller component executes a plugin's validation method, allowing a plugin to ensure that compliance rules follow a structure which is required by the compliance rule type. The validation method validates the rule syntax and returns a boolean to the framework, stating that the rule is either valid (true) or invalid (false). The rule type is recognised using a type label, which must be included in the list of registered, and thus available, compliance rule type plugins. This list is extended whenever additional plugins are installed. Using the result of the validation method, the rule modeller can determine whether to store the rule or not. If the rule is valid, the modeller goes on to store the rule through the compliance rule repository interface. If the rule is invalid, it rejects the rule.

The compliance rule metamodel is shown in figure 5.7. A reusable compliance rule always has a single value describing its rule type. There may be many compliance rules of the same type, however every rule must have exactly one type. Each rule contains one detector and one evaluator. The detector should describe how to detect whether the described rule applies to a provided instance model. What exactly this looks like depends on the plugin that is associated with the rule type, as the plugin associated with the rule type must be able to understand the detector content. A detector could for example be a link to a programme, or a file which provides a description of a rule detection structure. The evaluator should describe the information that is required by the evaluation part of a compliance rule type plugin. This can be similarly structured to the detector, but doesn't have to be. It could for example, additionally, contain information which the associated plugin can evaluate to determine which issue results from a rule violation.

[1] Winery is a topology modelling tool for the TOSCA environment. Details on its compliance rule description capabilities can be found here: `https://winery.readthedocs.io/en/latest/user/features/compliance-checking.html`

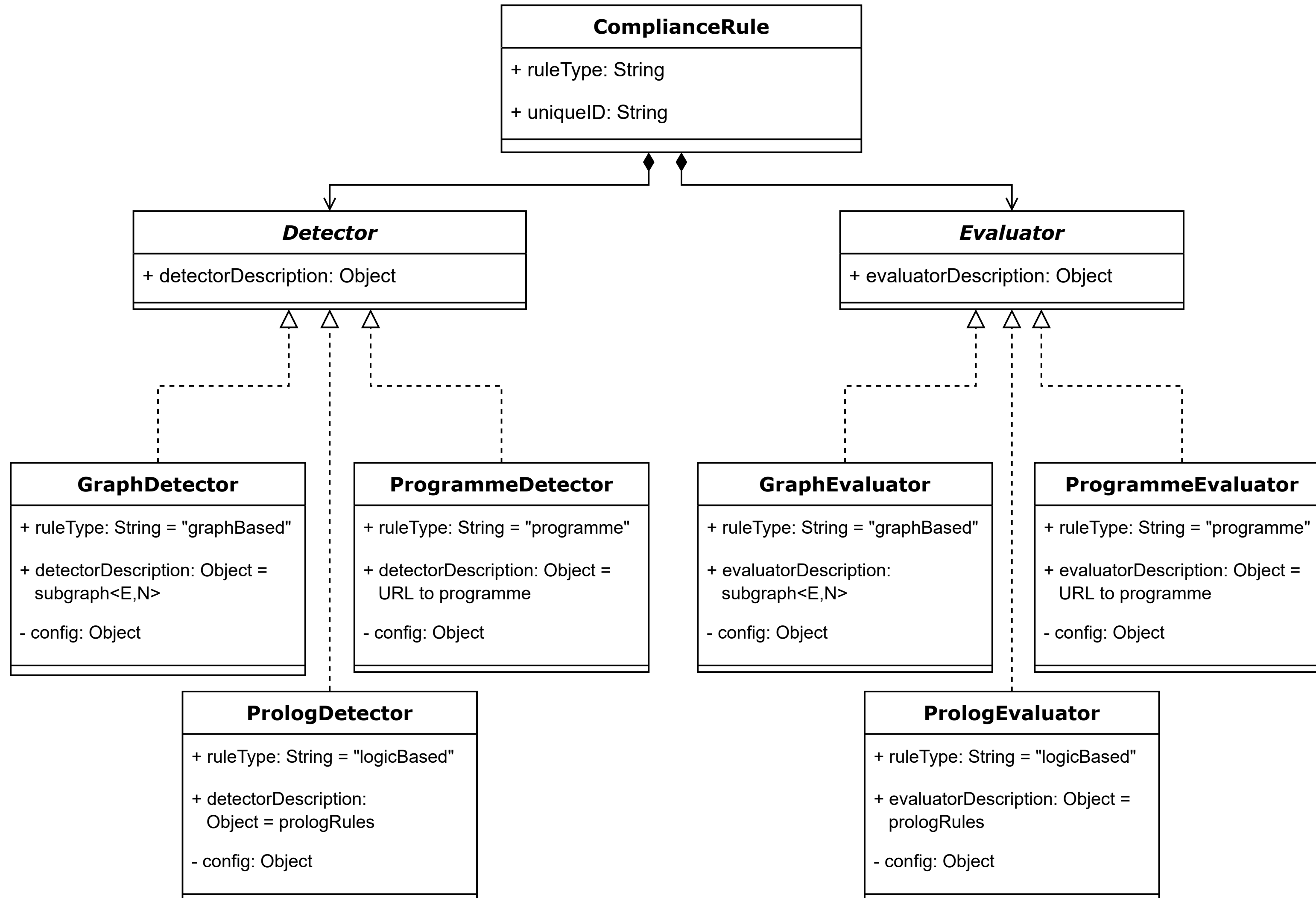

Figure 5.8: Examples Using Compliance Rule Metamodel

These are the elements which each rule must contain. Furthermore, each detector and evaluator of a rule description uses parameters, which are assigned in a rule's configuration, when the rule is bound by a user to an infrastructure instance in an execution. With this approach, the rule description metamodel allows the description of generic, re-usable rules of different types. In figure 5.8, three example implementations of a detector and evaluator are depicted as elements which inherit from the abstract detector and evaluator. These examples will be described in section 5.4.

5.3.4.2 Evaluator and Detector

The next components in the framework's core are the Rule Detector and the Rule Evaluator. These are combined into one section because their design is similar. Another reason for combining these components is that they are two aspects of the same goal: rule evaluation. The goal has been separated into two steps for efficiency reasons. The assumption is made that rule evaluation may be a lot more complex than detection, depending on the size of the instance model, and the evaluation method. Given this assumption, evaluation should not be executed for rules that do not apply to the infrastructure, even though they were selected by the user. Separating the detector allows the user to pick a superset of rules which may apply, without requiring that the user know with certainty which rules actually do apply. Both components are part of the plugin-based framework core, which means the methods of detecting rule applicability and evaluating rule fulfilment are extensible. The plugins for the core are **compliance rule type plugins**, as they extend the rule types which can be used to describe and evaluate compliance rules.

See figure 5.6 for the framework core architecture, containing the evaluator and detector. Figure 5.9 depicts the control flow of an execution of the *Orchestrator* in the form of a UML activity diagram [Obj17]. This control flow includes the calls to the Evaluator and Detector. The Orchestrator is triggered by a scheduled execution, coming from the Scheduler. It requires the execution configuration from the Scheduler, as well as the instance model and the configured compliance rules. For this purpose, it must first *retrieve the instance model* using the Instance Model Retriever. This can be called repeatedly if multiple plugins have been selected, until all plugins have been executed and the final instance model is temporarily stored. The orchestrator must pass the retriever configuration part of the execution configuration to the retriever, and make the returned instance model available to all subsequent steps. This step in the control flow is only executed once during an execution, as the instance is not expected to change during a single execution. Next, the orchestrator must *retrieve the rules* associated with the execution configuration, for which it requires a connection to the compliance rule repository.

The orchestrator then iterates through all rules in the rule set. For each rule, it executes the following steps, resulting in multiple executions of the loop if the set contains multiple rules. The loop has as many iterations as the set has rules. In this loop, the orchestrator must first (i) *select the current rule* from the set of rules. Next, the orchestrator must (ii) *identify the correct plugin* for the current rule's type. The correct plugin is identified using a unique identifier of the rule type. The orchestrator then uses the detector component to (iii) *execute the detection method* associated with the rule type plugin, passing it the instance model and the detection segment of the rule description. The plugin returns a boolean back to the framework, stating whether the rule is applicable or not. Using this boolean, the orchestrator (iv) *decides whether or not to call the Rule Evaluator component*. If it finds the current rule to be applicable, it passes the evaluation part of the rule description, as well as the instance model, on to the Rule Evaluator. The plugin's evaluation method, which the evaluator

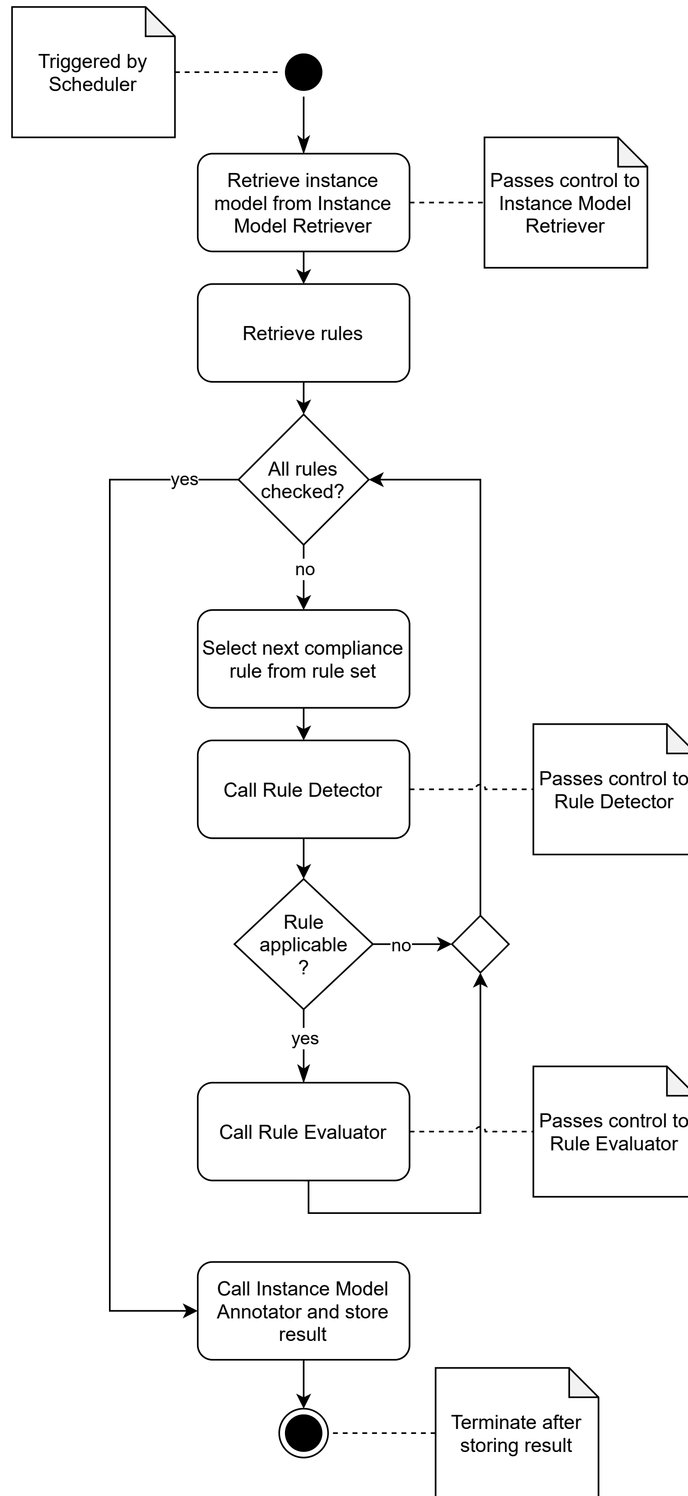

Figure 5.9: Orchestrator Control Flow

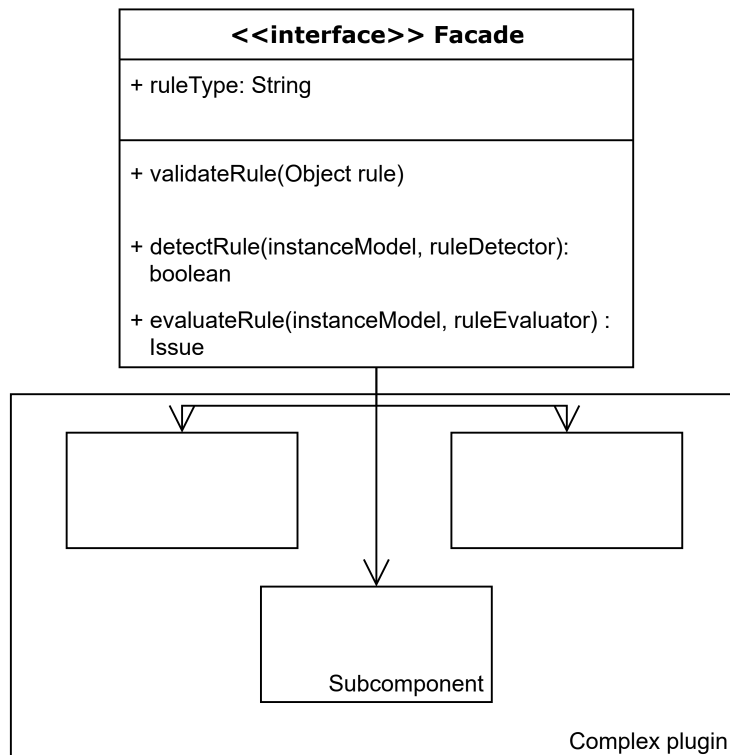

Figure 5.10: Required Public Interface for Compliance Rule Type Plugins

executes, must use the instance model and rule evaluator definition to evaluate whether the rule is fulfilled or not. Once the Rule Evaluator has finished and has returned the plugin's result (as described in the required public interface a plugin must implement), or directly after the detector execution if the rule is not applicable, (v) *control returns to the beginning of the loop*. During loop execution, the orchestrator temporarily stores the rule evaluation and detection results. The loop continues until the orchestrator has reached the end of the rule set. Once all rules have been checked, the orchestrator calls the Instance Model Annotator, which takes care of assembling and storing results. After this, the control flow terminates.

The compliance rule type plugins are the final aspect of this section. Figure 5.10 shows the required public interface of this plugin category. A compliance rule type plugin is always of a unique rule type, which is identified using a unique identifier. The plugin consists of three objects: the validator, the detector and the evaluator. The **validateRule** method must provide a validation method for rule descriptions in the detector, for example validating their syntax. The validation method only takes the rule description as input and returns a boolean, stating whether the rule description is valid (true) or not (false). This is required and used by the Rule Modeller, before the rules can be stored in the central rule repository. The **detectRule** method must provide a detection method for the detector part of a rule. This method takes the detector and an instance model as input, and also returns a boolean. The boolean represents the information whether the rule is applicable to the provided instance model (true) or not (false). The **evaluateRule** method must provide an evaluation method for the evaluator part of a rule. The evaluation method takes the evaluator and an instance model as input. It returns an object consisting of the resulting issue, which includes an issue type, a message, and a set of the identifiers of all affected components in the instance model.

Plugins can either be a generic method for all rules of the associated type, for example for graph- or logic-based rules, or they can be plugins which take a programme location and execute this programme as a rule-specific detection method (for example for the programme rule type that will be covered in section 5.4.2.3). Details such as this are a black box to the framework, as it only knows what it sends to the plugin and what it receives back from the plugin.

In order to make the new plugin available to the framework, it must be installed and registered using the Plugin Manager Component. The plugin manager requires information on how to install the plugin, and will register it using a unique rule type identifier. It then keeps track of the installed, and thus available, plugins. The complexity of evaluation, as well as identification of affected components, is outsourced to the plugins in order to keep the framework as generic as possible. Example detection and evaluation methods a plugin may implement are described in section 5.4.2.

5.3.5 Instance Model Retriever

The next architecture component is the Instance Model Retriever component. Figure 5.11 shows the component, separated from the architecture overview. The Instance Model Retriever is a plugin-based component, which means it can for example be extended using existing research and prototypes in the field of instance retrieval. A number of depicted example plugins are MICROLYZE [KUSM18] (a microservice architecture retrieval tool), the work described by Harzenetter et al. in [HBB+21] (instance model derivation and enrichment), scripts which retrieve required details, and manually providing an instance model. Example retrieval plugins are covered in greater detail in section 5.4.1.

The Instance Model Retriever uses configuration, which was previously provided by a user, in order to execute a retrieval plugin. The Instance Model Retriever is called by the Orchestrator component at the start of an execution, for each plugin configured in an execution configuration. The retriever is passed the required configuration file for the current plugin, which the orchestrator previously identified using the associated ID. Using this configuration, the retriever can execute the associated plugin. The plugin returns a graph-based representation of the infrastructure as an instance model. It is possible for the orchestrator to execute multiple retrieval plugins, if this is configured by the user in the execution configuration. The final instance model, after the last retrieval plugin has finished executing, is stored for the subsequent execution steps.

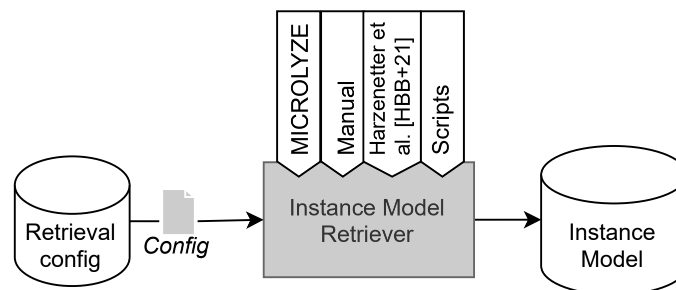

Figure 5.11: Instance Model Retriever Architecture

The future extensibility of infrastructures which the framework can work with is ensured by the plugin-based nature of this component. It can be extended by any plugin which provides an EDMM instance to the framework with the required structure, allowing for the framework to work independently from the many possible component types and infrastructures. It particularly allows the framework to work with component types and infrastructures it cannot know yet. The retriever plugin's required public interface, shown in figure 5.12, only provides a method retrieveInstanceModel, which points to an implementation that allows the retrieval of an instance model. This method must receive the required configuration, as well as an optional parameter that can contain an existing instance model, if the current plugin extends an existing model. The retrieve method returns an instance model to the Instance Model Retriever.

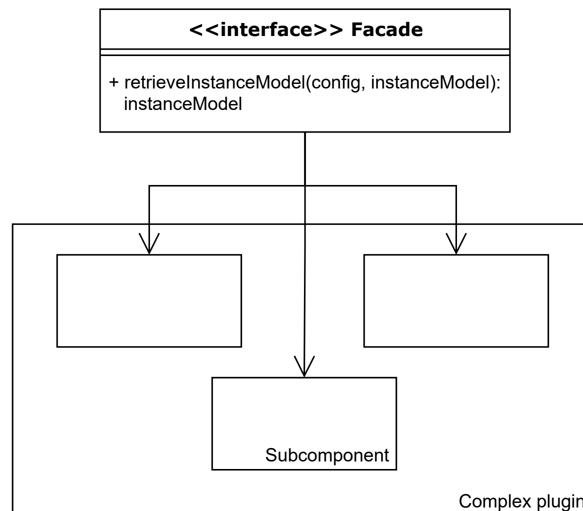

Figure 5.12: Required Public Interface of Retrieval Plugin

5.3.6 Instance Model Annotator

The final component in the framework architecture is the Instance Model Annotator. This component's architecture is repeated in figure 5.13. The control flow of an execution of the annotator is depicted in figure 5.14, as a UML activity diagram [Obj17]. The annotator is executed as the last component in an execution of the framework. It is called by the Orchestrator after all rules have been processed and receives the instance model, as well as a set of issues, where each issue has an issue type, a message and the affected model elements.

The annotator iterates through all issues it receives in the set. If the *set of issues is empty* from the start, the annotator returns the instance model without any issues, allowing the orchestrator to store it. If the *set is not yet empty*, the annotator loops through a number of steps.

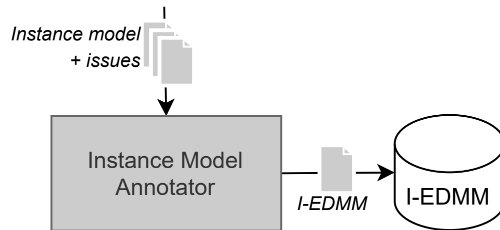

Figure 5.13: Architecture of Instance Model Annotator Component

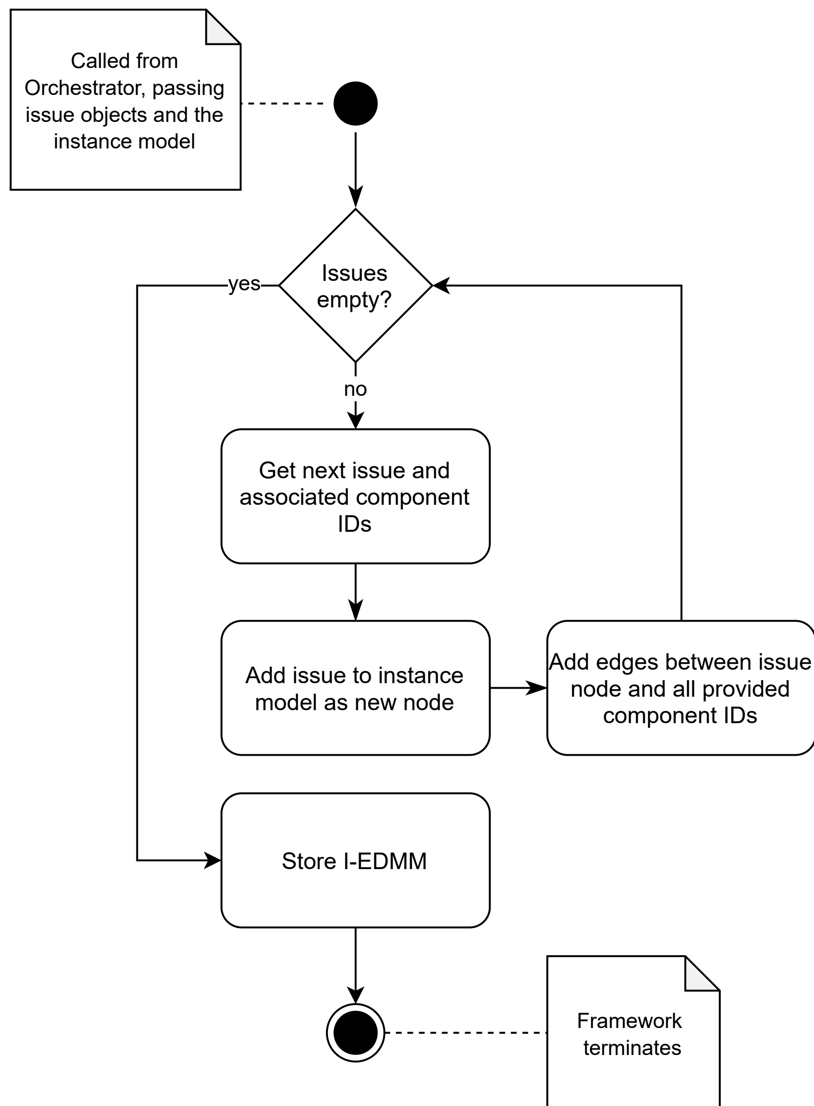

Figure 5.14: Control Flow of Instance Model Annotator

First, (i) the annotator *retrieves the current issue* from the set, together with all associated information. Next, it (ii) *creates an issue node* in the instance model graph. Then the annotator (iii) iterates through all component identifiers that are associated with the current issue. *For each identifier*, it

creates an edge between the issue node and the component with the provided identifier. After this step, the control flow loops back to (iv) *check whether there are any issues left* to be added. This loop iterates as many times as there are issues in the set. By this means, the annotator creates an I-EDMM. Once all issues have been added to the instance model, the annotator returns the I-EDMM to the orchestrator, which stores it in a repository, allowing the user to access it. See figure 5.2 for an example of an annotated I-EDMM model. After this, the framework execution terminates, which also concludes the introduction of all framework components.

This section has presented the framework architecture, as well as the generic structure and control flow of the single components. In order to make the plugin aspect of the framework less abstract, the following section will provide a number of examples for plugins.

5.4 Examples

This section aims to provide a better understanding of the solution concept using concrete examples. It will start with the instance model retrieval and possible plugins for this, as it is assumed that the instance model retrieval is the first plugin-based step, at the beginning of an execution. After this, it will cover three example compliance rule type plugins. These plugins are graph-based compliance rule types, logic-based compliance rule types and programme-based compliance rule types, respectively.

5.4.1 Instance Model Retrieval

For instance model retrieval, there are already multiple existing approaches that are suitable for a variety of application infrastructures. They can be adapted to integrate with the framework as plugins, if necessary by using adapters which translate from the output of an existing prototype to the instance model structure the framework requires.

5.4.1.1 Existing Methods

The **first possible plugin** is a prototypical tool called **MICROLYZE**, which is described in [KUSM18]. MICROLYZE is aimed at recovering microservice architectures in real-time, while components are all up and running. Recovery involves the business, application and hardware layer, as well as the corresponding relationships between them. Particularly the application and hardware layer would be relevant when using MICROLYZE as a plugin for the thesis framework. The tool ultimately aims to allow for better, mostly automatically maintained, documentation of the overall architecture, in contrast to fully manual documentation. The tool iterates through 6 phases. In the first, it rebuilds the current infrastructure that is registered in a service discovery tool, which keeps track of the status of currently running microservice instances. In the second, the information from the service discovery tool and additional monitoring information is used in order to recover hardware-related aspects. This requires the installation of a monitoring agent on each hardware component. The tool also tries to recover dependencies and connections between services in this phase. Phases three to five allow the recovery of business processes and their relations to the microservices. The final phase recognises changes in the IT infrastructure. Depending on

the resulting format of the documented architecture, a plugin for the framework could extract the required information from this architecture that is provided by MICROLYZE, and translate it to the EDMM instance model the framework requires.

A **second possible plugin** is an approach of deriving and enriching instance models of enterprise applications, described by Harzenetter et al. in [HBB+21]. In this approach, they first retrieve instance information of a running application and then derive a standardised instance model. The first step is achieved using an Instance Information Retriever component. It is plugin-based and accesses the APIs of the deployment technologies used to deploy the application. This assumes the application is managed using IaC, and the plugins are dependent on the deployment technology. For the second step, the solution introduces an Instance Model Normalizer component, which interprets the technology-specific information, and derives a normalised instance model of the application. Their normalised instance model is based on TOSCA instance models[2], which is very similar in its approach to the description of instance models using EDMM. The resulting instance model is completed in a third step using the Instance Model Completer component. This step adds additional information which cannot be retrieved from the deployment tool in the first step. The third step is based on a plugin-based information retrieval approach described in [BBKL13]. The remaining steps of their approach are not required for an instance model retrieval plugin of this framework. Since the result is a TOSCA instance model, it can be translated to the structure which the framework requires. This is something the plugin must do before passing its retrieved instance model back to the framework.

A **third possible plugin** is one that allows the user to provide a manually described instance model. This can be created using any tool a user chooses, as long as the resulting structure is correct. Tools for this purpose already exist, such as Winery[3]. In addition to its main purpose, the description of deployment models as a TOSCA topology, and the aforementioned capability of describing graph-based compliance rules, Winery also allows the description of instance models. This can be achieved by creating a TOSCA topology, and filling this model with all relevant, up-to-date runtime information of a concrete infrastructure instance, thus producing an instance model. Models can then be exported, as Winery allows exports in the CSAR-format, which stands for Cloud Service Archive and is a bundle of TOSCA components. The plugin in this case may consist of an installation of Winery, plus an adapter that translates from the format which Winery exports to the framework format.

5.4.1.2 Scripts Plugin

The scripts plugin is an approach that does not yet exist. The idea of this plugin is based on a combination of the approaches described by Binz et al. [BBKL13] and Harzenetter et al. [HBB+21]. This plugin's approach helps implement use-case specific instance retrieval. This means it allows the intelligent extraction of properties with hidden business logic, such as the logRetentionPeriod property, which is not a value that can be read from a component in a general manner but instead has to be computed. The scripts plugin should show that it is possible for the framework to work with adhoc scripts in order to complete an instance model skeleton with missing details. An instance

[2]The definition can be found here: `https://docs.oasis-open.org/tosca/TOSCA-Instance-Model/v1.0/TOSCA-Instance-Model-v1.0.html`

[3]The documentation can be found here: `https://winery.readthedocs.io/en/latest/`

model skeleton is understood to be an instance model that contains all components, relations and properties, but is still missing property values. This has similarities to an approach described by Ntentos et al. in [NZP+21]. They perform architectural reconstruction of application systems based on their source code. In the scripts plugin example, shown in figure 5.15, the instance model skeleton, shown on the left of the big grey box, already includes the currently existing components and relations. It is, however, still missing the values for a number of properties.

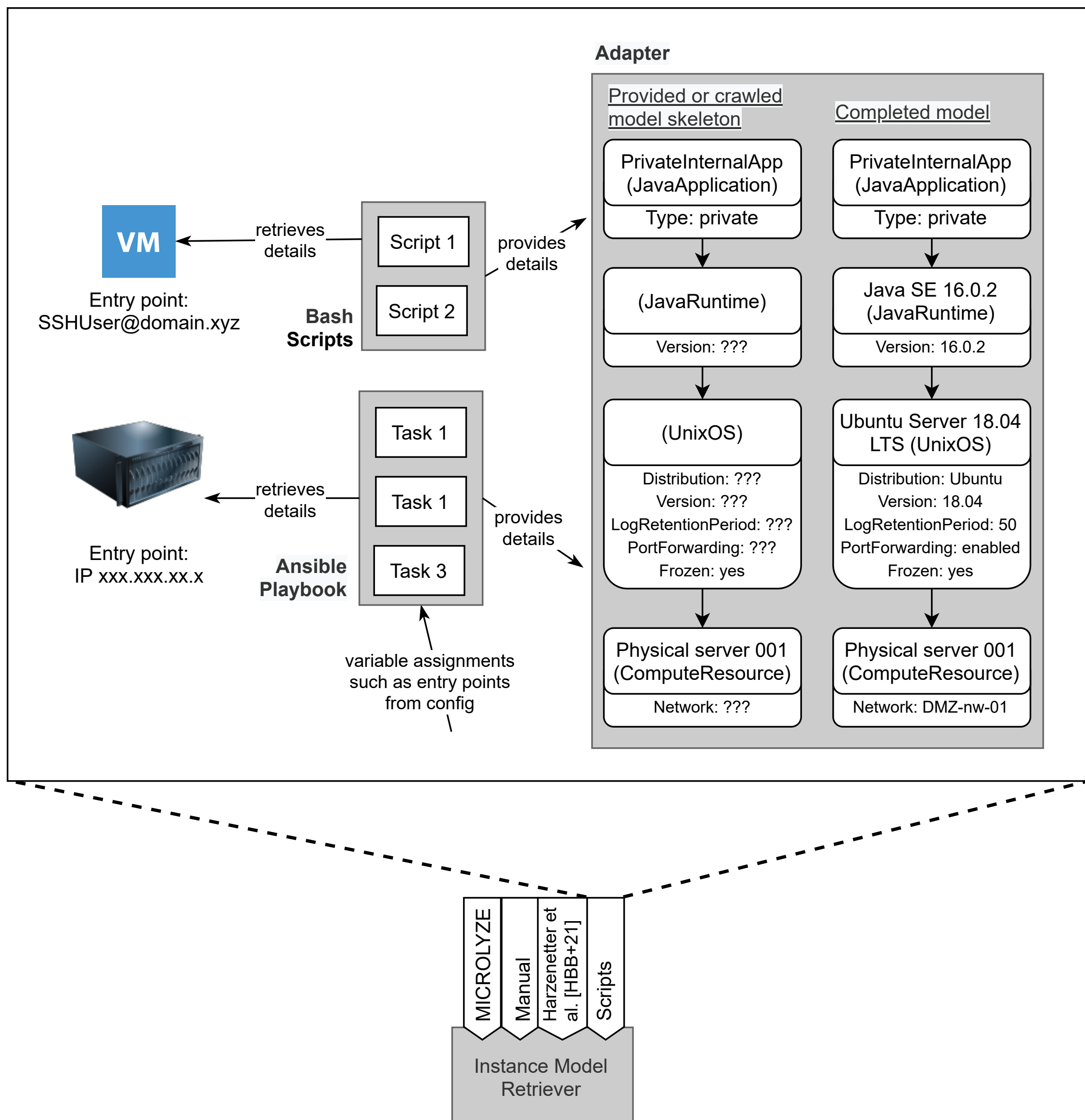

Figure 5.15: Scripts Plugin

The instance model skeleton can be created manually, or using any number of other plugins prior to the scripts plugin. Properties and values, such as the “Frozen” property of the Unix OS component, could be added manually using a manual plugin before the execution of this plugin. Adhoc scripts now allow the retrieval of these use-case specific values, thus allowing a specialised usage of the

framework. Adhoc scripts are understood to be, for example, Ansible[4] tasks, or bash scripts. An Ansible task is a unit of action, which can be executed adhoc or combined into a repeatable format, called a playbook in Ansible.

A scripts plugin could allow execution of such adhoc information retrieval and fill the missing properties into an instance model, where components, relations and property keys were provided, but values were still missing. In a similar way to Ansible tasks, bash scripts could also be used to extract such information from running components. This way, for example, the manual description of instance models could be partially automated, leaving less repetitive manual work to the user in a setting where fully automatic instance retrieval is not feasible. As was already mentioned, this plugin could integrate with any other general-purpose plugin in order to retrieve the skeleton it requires. Depending on the resulting format of such scripts and tasks, as well as the format of the initial instance model skeleton, using an additional adapter would allow translation from the results to the instance model required by the framework. Figure 5.15 depicts the idea behind this process, where an initial instance model must be provided or crawled using another means, and the values with question marks can be filled using existing scripts, which the adapter does. The adapter can then return this completed instance model to the Instance Model Retriever.

This section has covered four possible plugins to the Instance Model Retriever component. First was a plugin for microservice environments which depends on an existing discovery service and the installation of a monitoring agent on components. Second was an extensive solution to crawling and completing instance models, which itself depends on plugins in multiple steps. Third was an approach to manually describing an instance model, and finally a plugin was described which can take and complete a skeleton instance model with existing components and relations, but still lacking many property values. The next section will provide a short comparison of these four plugins.

5.4.1.3 Comparison

This section aims to provide an overview of how the plugins compare, with regards to a number of attributes, easing the selection of an appropriate plugin. The **reusability** attribute represents whether it is possible to re-use the approach for multiple infrastructures, which is tightly coupled to how easily adaptable a plugin is to new infrastructures. The **ease of use** attribute refers to how much effort is required to use the plugin, in form of configuration and potentially learning to use a tool. **Handling of business-specific rules** refers to whether the instance model that results from a given instance retrieval plugin allows the evaluation of business-specific rules, such as the logRetentionPeriod or Frozen properties in motivating scenario 1. The **customisation required** attribute refers to how much configuration and customisation (e.g. in form of additional plugin development) is required for a user to be able to use the plugin for their use-case. Finally, the **effort** attribute requires to how much effort is required when repeating instance model retrieval multiple times, i.e. how much automation the plugin makes possible.

Table 5.1 shows a comparison of the described attributes for each of the retrieval plugin examples that was described. The attribute values for each plugin were derived from reviewing literature, as well as the research in this thesis, however it was not possible to test attributes such as ease of use

[4]Ansible Docs: https://docs.ansible.com/ansible/latest/index.html

through prototypes of each plugin. The MICROLYZE [KUSM18] plugin allows *good reusability* because it is a generic, technology-agnostic approach. It has limitations, for example by being dependent on an existing service registration tool as well as monitoring tools, however it does not specify a particular tool that must be used. Because it is generic, it is assumed *not to be easy to use*, as it requires adaptation to specific use-cases. It can only *partially handle business-specific rules*, because it is not possible to add intelligent retrieval logic that is not based on a service discovery tool or monitoring agent. This means the resulting instance model may be missing properties required for use-case-specific rules. *Customisation is required* for the plugin to work with specific tools, as it does not provide a particular tool that must be used. However, this also means it is easier to integrate with tools that are already in use in a micro-service environment, and doesn't require the installation of additional plugin-specific tools for its use. Once MICROLYZE has been set up, it can be expected to require *very little effort* to maintain (and retrieve) an up-to-date view of the infrastructure instance. A drawback is the requirement of a monitoring agent on hardware for information retrieval, because of which it may not be suitable for some use-cases where this is not permitted. Furthermore, it cannot be used for infrastructures that don't use a service discovery tool.

	Attributes				
Plugins	**Reusability**	**Ease of use**	**Handling of business-specific rules**	**Customisation required**	**Effort**
MICROLYZE [KUSM18]	Good	Medium	Partially	Yes	Low
Harzenetter et al. [HBB+21]	Good	Medium	Partially	Yes	Low
Manual	Bad	Good	Yes	No	High
Scripts	Bad	Depends	Yes	No	Medium

Table 5.1: Comparison of Retrieval Plugins.

The plugin based on Harzenetter et al.'s work [HBB+21] provides *good reusability*, as it is agnostic to both use-cases and technologies, allowing the re-use of the core logic. This, however, entails that it requires adaptation to an infrastructure. Furthermore, the fact that it requires plugins itself adds an additional layer of complexity, albeit also flexibility. Contrary to the MICROLYZE plugin, it does not require certain service discovery technologies to be used in order to function. It does require an IaC deployment technology to have been used though, as it retrieves instance information from the deployment technology APIs. This plugin is most likely *not easy to use*, without existing plugins for the infrastructure at hand. Given existing plugins, it probably becomes easier to use. It can *partially handle business-specific rules*, especially through the enrichment process that allows manual refinement of an instance model. However, it does not allow the automatic retrieval of information that cannot be automatically mapped to properties of TOSCA templates. *Customisation is required*, possibly even in the form of plugin development. However, in consequence, the *effort for repeated retrieval is low*, once the generic approach has been customised to fit a specific use-case.

The manual plugin is one that requires *a lot of effort*, as every change to the infrastructure must be manually adjusted in the instance model. This means the approach is *not at all reusable*, as the retrieval is not at all automated using this approach. However, it is *easy to use* as it does not require any form of automatic access to components, although manually retrieving necessary information is a high-effort task. *No customisation is required*, as the approach is use-case specific. This means it is possible to *evaluate all business-specific rules* a user has defined, as a manually described instance model can contain any properties the rule requires. This plugin allows a low-effort initial usage of the framework, for example if automatic retrieval is currently not feasible, and changes do not occur often. However, it does not allow any form of automation for future framework executions.

The scripts plugin in itself is reusable, but the scripts it uses have *bad reusability*, as they are assumed to be tightly coupled to a certain instance of an infrastructure. *Ease of use depends on the scripts* that are used, for example it allows users to use a scripting language they are familiar with, however this does not mean it is overall easy to use. The execution of intelligent business logic, to retrieve use-case specific values, allows the handling of *any business-specific rules* the user requires. This plugin *does not require any form of customisation*, as it is already tightly coupled to an infrastructure and use-case specific rules. It requires *medium effort* for repeated instance model retrieval, as it may still require manual steps in order to provide the instance model skeleton, however it does not require the manual retrieval of all information that is retrieved by scripts. Thus, it allows partial automation for use cases where full automation is not feasible, whilst not requiring a fully manual input of the instance model either. Moving on from instance model retrieval plugins, the next section will describe example compliance rule type plugins.

5.4.2 Compliance Rule Type Plugins

The second dimension of plugins that the framework concept builds on is the *compliance rule type plugins*, which provide methods of detecting rule applicability, evaluating their fulfilment, and validating a rule description's schema. The following subsections will describe three examples of potential compliance rule type plugins to the framework.

5.4.2.1 Graph-based Compliance Rule Type Plugin

The *graph-based* compliance rule type plugin is based on the works of Zimmerman et al. [ZBKL18] and Krieger et al. [KBKL18], both of which represent their deployment models as graphs. The rules described using their approach can be described as graphs with nodes and edges. As the approach in this thesis separates detector and evaluator, the approach of Krieger et al. is better suited. This plugin is used to evaluate **CR 2**, as an example, on the infrastructure described in scenario 1 in subsection 4.1.3.1. This rule states that all instances of PrivateInternalApp must be hosted on a JRE of exactly version 15.

Compliance rule description: As required by the framework and explained in subsection 5.3.4.1, the compliance rule must be described using a rule type, rule detector and rule evaluator. The rule type for this plugin would be identified as "graph-based". The detector and evaluator must each consist of one graph. The first graph must act as a detector and is labelled as the detector. The

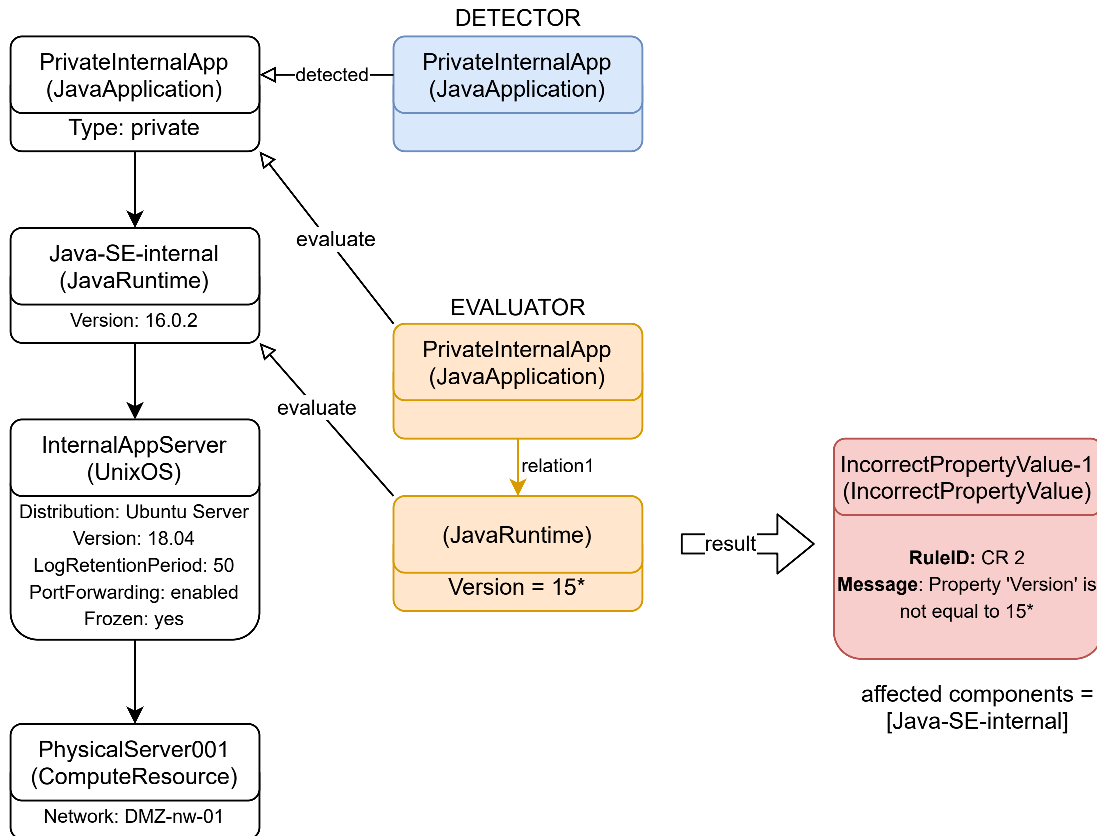

Figure 5.16: CR 2 applied to PrivateInternalApp stack of scenario 1

second graph must describe the required structure and be labelled as the evaluator. Figure 5.16 shows the PrivateInternalApp stack from scenario 1 and both detector as well as evaluator parts of **CR 2**.

Detector and detection process: The detector is shown in blue in figure 5.16. It is a graph with a single node that represents a component in an instance model. This component is of the type JavaApplication and is an instance of PrivateInternalApp. It has no edges (i.e. relations). The name of the component could be a parameter, allowing this rule to be re-usable for any application-specific detector. The detector is what is passed to the graph-based plugin together with the instance model.

The plugin which can correctly interpret this detector, in order to detect whether this component exists in the instance model or not, uses a subgraph isomorphism method for this purpose. Provided with two graphs A and B, it can analyse whether the graph B is completely contained in a subgraph A' of graph A, in a way that the graphs A' and B are isomorphic. The detector in this example will detect all instances of PrivateInternalApp in a given instance model. In this case, the component depicted in the detector is also contained in the graph that represents the instance model. For this reason, passing both detector and instance model, represented as graphs, to a method which checks for subgraph isomorphism, will return confirmation that the detector graph is a subgraph of the

instance model. In this way, the detector ensures that, in the next step, this rule is evaluated, as it is deemed to be applicable. With this information, the detection plugin in this example can return "true" to the framework and the framework decides on the next step using the returned result. If no subgraph A' is found that is isomorphic to B, then the rule does not apply to the current instance model, and the plugin has to return "false".

Evaluator and evaluation process: The evaluator is shown in yellow in figure 5.16. It is also a graph, this time with two nodes, a property associated with the node of type JavaRuntime, and a directed edge of type hostedOn between the two nodes. The evaluator states that any instance of PrivateInternalApp must be hosted on a node of type JavaRuntime with a property "Version" that is equal to any subversion of version 15. The name PrivateInternalApp could be a bound parameter, as could the name and value of the property that must be evaluated for equality. For evaluation, the plugin is passed the instance model and the evaluator portion of the rule.

In order to process the evaluator, the evaluation method of the graph-based plugin must again employ a method of checking for subgraph isomorphism. Furthermore, it must provide operators which can compare the values of graph elements and properties. In this example, an equality operator is required, as well a method of evaluating regular expressions, in order to discern that, for example, version 15.0.2 would be a valid version. The plugin would have to provide a number of operators, consisting at the very least of an equality operator, a comparison operator (greater than, less than), and a regular expression processor.

This plugin can share the subgraph isomorphism method as well as the operators between both the detection and the evaluation methods, meaning it doesn't have to provide one method for each. However the plugin does have to process the results of the subgraph isomorphism method differently depending on the current step, so it must be aware of whether it is evaluating or detecting. In the case of evaluation, if the subgraph isomorphism method returns "true", the instance model is compliant and does not have any issues.

Having received the results of evaluating the rule, the plugin must produce a result in the format required by the framework. If there is an isomorphic subgraph, meaning the evaluator is contained in the instance model and the instance model is thus compliant with the current rule, then the plugin doesn't return anything, meaning nothing will be annotated into the graph as nodes and edges. However, if there is no isomorphic subgraph, this means the evaluator is not contained in the instance model. In this case, the instance model is not compliant with the rule. The plugin must now return the issue title associated with this rule, which in this case is IncorrectPropertyValue, a message, which states that the property "Version" is not equal to 15*, a list of the affected components, and the rule ID. The affected component in this case is the component with the incorrect property value, so it is only Java-SE-internal, and the rule ID is **CR 2**. The names are assumed to be the unique identifiers of the components, and the information that these components are affected in the case of rule violation must be provided in the evaluator of the rule description.

The evaluator, in this example, is a whitelisting rule. This means it describes a structure that is specifically required, all other structures are prohibited. Attempting to evaluate a blacklisting rule (describing prohibited structures) in the described way would result in false positives and false negatives. The user that describes the rule models must be aware of this. The final result of detection and evaluation using this plugin is that **CR 2** is applicable to the first motivating scenario but not

fulfilled, because an instance of the application which goes by the name of PrivateInternalApp is running in the given infrastructure but hosted on a JavaRuntime version 16.0.2. This results in an issue title and message, as well as a set of affected model elements.

5.4.2.2 Logic-based Compliance Rule Type Plugin

The *logic-based* compliance rule type plugin is based on the work of Saatkamp et al. in [SBKL19]. Their prototype[5] allows automated issue detection in TOSCA topologies, namely in TOSCA deployment models. For this plugin, the approach is modified to evaluate instance models instead of deployment models, and detector logic must be added. The original approach is particularly aimed at automatically detecting rule violations which are caused by restructuring of deployments, for example when modifying infrastructure to work in a distributed way. However, the reason behind arising issues can equally be of a different nature. Saatkamp et al.'s approach formalises issue detection using the logic programming language Prolog. In this example, parts of the related example in [SBKL19] are re-used, and applied to motivating scenario 1. The logic-based approach is used to detect and evaluate **CR 5**, which states that all communication channels between components must be encrypted.

Compliance rule description: The compliance rule descriptions for this plugin must be provided in Prolog. Prolog is based on horn clauses of first-order logic, thus enabling the representation of knowledge as facts and rules. Facts describe specific circumstances, meaning objects and their relationships. Based on facts, queries can be used to either prove a fact is true or infer new knowledge. Rules simplify complex queries, they are extended facts with conditions that have to be satisfied for the rule to be fulfilled. Conditions of a rule can be linked using AND (,) or OR (;) operators. Using Prolog allows the evaluation of a fact base using the conditions of the rules.

As required by the framework and explained in subsection 5.3.4.1, the compliance rule must be described using a rule type, a rule detector and an evaluator. The rule type for this plugin would be identified as "logic-based". The detector and evaluator must each consist of Prolog rules. The first rule must act as a detector and is labelled as the detector. The second rule must describe the properties that fulfil the rule, i.e. the required structure, and this part of the compliance rule is labelled as the evaluator. This is a slight modification of the approach in [SBKL19], as their rules describe the forbidden structures.

Detector and detection process: The logic evaluation method of the *logic-based* compliance rule type plugin allows the evaluation of any logic clause written in Prolog. Using the logic-based approach, the instance model which the plugin receives must first be transformed to a suitable format. The transformation into a suitable format is the formalisation of the instance model, allowing it to be interpreted by a logic-based programming language such as Prolog. It requires an adapter within the plugin, which translates from the framework's representation of an instance model to a Prolog set of facts.

[5] The prototype is available here: `https://github.com/OpenTOSCA/ToPS`

The instance model represents the fact base for this example execution. Listing 5.1 shows an excerpt of a Prolog description of the uppermost components of the right two stacks of scenario 1, those being a component WebApp of the type WebApplication, which connects to a component UserData of type RelationalDB. This excerpt is the required portion for **CR 5**.

```
component(WebApp).
component(UserData).
relation(WebApp, UserData, connectionID1).
relationOfType(connectionID1, connectsTo).
componentOfType(WebApp, WebApplication).
componentOfType(UserData, RelationalDB).
...
```

Listing 5.1: Instance Model Description in Prolog

A detector in this plugin is also a set of rules that the fact base, i.e. the instance model, must fulfil. For **CR 5**, the rules required in a detector are depicted in listing 5.2. `communicationExists` for components C1 and C2 evaluates to true when there exists a relation R between C1 and C2 AND the relation R is of type connectsTo. This applies to components PrivateInternalApp, SupportService, UserData and WebApp in the instance model, but the focus is put on the relation between WebApp and UserData.

```
communicationExists(C1, C2) :-
relation(C1, C2, R),
relationOfType(R, connectsTo).
```

Listing 5.2: Detector Description in Prolog

Since the detector clause of the compliance rule description evaluates to true for the instance model that the evaluation method of this plugin receives, the plugin returns “true” to the framework. The framework now knows to call the evaluation method, passing it the evaluator and the instance model in the framework-specific representation.

Evaluator and evaluation process: For evaluation, the logic evaluator method of the *logic-based* compliance rule type plugin can be reused, as it does not care what rules it is evaluating. The plugin must however be aware of whether it is calling the logic evaluator for detection or for evaluation. The plugin’s evaluation method receives the evaluator part of the rule description from the framework when it is called.

The evaluator method of the plugin again requires a translation of the instance model as shown in listing 5.1. The plugin could try to optimise this by storing a state containing detection history and the formalised instance model associated with the rule. However, the plugin does not know which execution of the framework is currently running. The consequence of this is that the plugin would run the risk of evaluating an incorrect status of the infrastructure, as it might have stored an old version of the instance model. This is not an acceptable risk and means translating the instance model twice is an acceptable overhead.

The Prolog clauses in listing 5.3 are the evaluator part of the description of **CR 5**, which the framework passes to the plugin and the plugin can use to evaluate the instance model. In this case, the evaluator part describes the required structure in the form of a logic formula. The formula evaluates to true when the rule is fulfilled, and false if it is violated. The plugin knows this and uses this to communicate the correct information back to the framework.

```
1   encryptedCommunication(C1,C2) :-
2   relationOfType(R, connectsTo),
3   relation(C1, C2, R),
4   property(R, encrypted, true).
```

Listing 5.3: Evaluator Description in Prolog

This means the formula `encryptedCommunication` between components C1 and C2 is fulfilled if the set of clauses is fulfilled. The clauses state there must be a relation R of type connectsTo in the instance model AND this relation R must connect components C1 (source) and C2 (destination) AND the relation R must have a property "encrypted" with a value of "true". If any of these clauses, which are linked with AND operators, are not fulfilled, then there exists a problem which results in the issue type "ConnectionIssue". This must be returned to the framework as an issue linking to the unique identifiers of the components C1, C2 and relation R. In the motivating scenario, it means the plugin would return the rule ID **CR 5**, the rule type "ConnectionIssue", the message "encryptedCommunication = false", as well as the set [WebApp, UserData, connectionID1], which states that the detected issue affects all model elements included in this set. `connectionID1` is assumed to be the unique identifier of the relation between WebApp and UserData.

5.4.2.3 Programme-based Compliance Rule Type Plugin

The *programme* compliance rule type plugin allows the user to provide a custom rule detection and evaluation method, in form of an existing script or programme that evaluates a use-case-specific rule. This allows for existing programmes to integrate with the framework, if the framework user does not wish to define compliance rules anew. This section will use the programme compliance rule type plugin in order to evaluate **CR 1** on motivating scenario 1. The rule states that all web applications must be hosted in the same region as connected databases.

Compliance rule description: As required by the framework and explained in subsection 5.3.4.1, the compliance rule must be described using a rule type, a detector and an evaluator. The rule type would be identified as "programme-based". The detector and evaluator must each consist of a URL, which points to the location of a downloadable programme. The plugin must know how to compile and execute these programmes. The programme stored at the location of the first URL must act as a detector and is labelled as the detector. The programme stored at the location of the second URL must act as an evaluator, which returns when a rule is fulfilled or not, as well as the resulting issue. It is labelled as the evaluator.

Detector and detection process: The detection method part of the *programme* compliance rule type plugin parses the URL, pulls the programme, compiles it and then executes it. The programme must always take the instance model as an input, however it doesn't need to take any rule detector as input, as the programme is the rule itself. The detector programme must return whether the rule is applicable (true) or not (false).

In order to detect whether **CR 1** applies to motivating scenario 1, a provided programme can iterate through all edges of the instance model. As soon as the programme detects one edge of the type "connectsTo", it has to check whether the source node (i.e. component) is of type WebApplication and the destination is of type Database, or vice versa. RelationalDB is assumed to be a more refined type of type Database, which means it inherits from the more generic type and is also detected. If these properties apply to any of the edges, the programme can stop execution and return true. If it has reached the end of the set of edges and hasn't detected any edges with said properties, the programme can return false. The plugin passes this information back to the framework, which can now decide whether or not the rule must be passed on to the evaluator.

Evaluator and evaluation process: The evaluation method part of the *programme* compliance rule type plugin also parses the URL, pulls the programme, compiles and executes it. This programme must also take the instance model as input, however, the result the programme returns will be different. The result must conform to framework requirements and provide both an issue type, message and set of affected model elements.

In order to evaluate whether **CR 1** is fulfilled or not, a provided programme can check the properties of all relations, as explained above in the detector programme. Once it has determined the components that are affected, the programme can traverse all "hostedOn" relations until it finds a component with the property called "Region". Having found this property on two components that are indirectly connected, the programme can compare them for equality.

In the provided example, for the UserData component, it would traverse the graph until it reaches the AmazonRDS component, which has the region "us-west-2". For the WebApp component, the programme would traverse the graph until it reaches the Instance2-WebApp component, which has the region "us-east-2". Comparing these two values, the programme could conclude that the values are not equal. It would then return the rule ID **CR 1**, an issue type "IncorrectPropertyValue", a message stating that a database and web application do not have the same value for a property "region", as well as a set of affected model elements. This set would be [AmazonRDS, Instance2-Webapp], because they are in two different regions and host a Database and WebApplication, respectively. The plugin can now pass this information back to the framework, and the framework can hold onto it until all rules have been evaluated.

5.4.2.4 Annotation

All previously detected issues can now be passed from the evaluator to the annotator. The issues are all of the same format, and using the provided IDs together with the associated issue type and message, the annotator can create a new issue element. This new element is then provided with a label for the rule ID and message. Finally, the annotator can create edges from this issue component to every component of which the ID is included in the set of affected components.

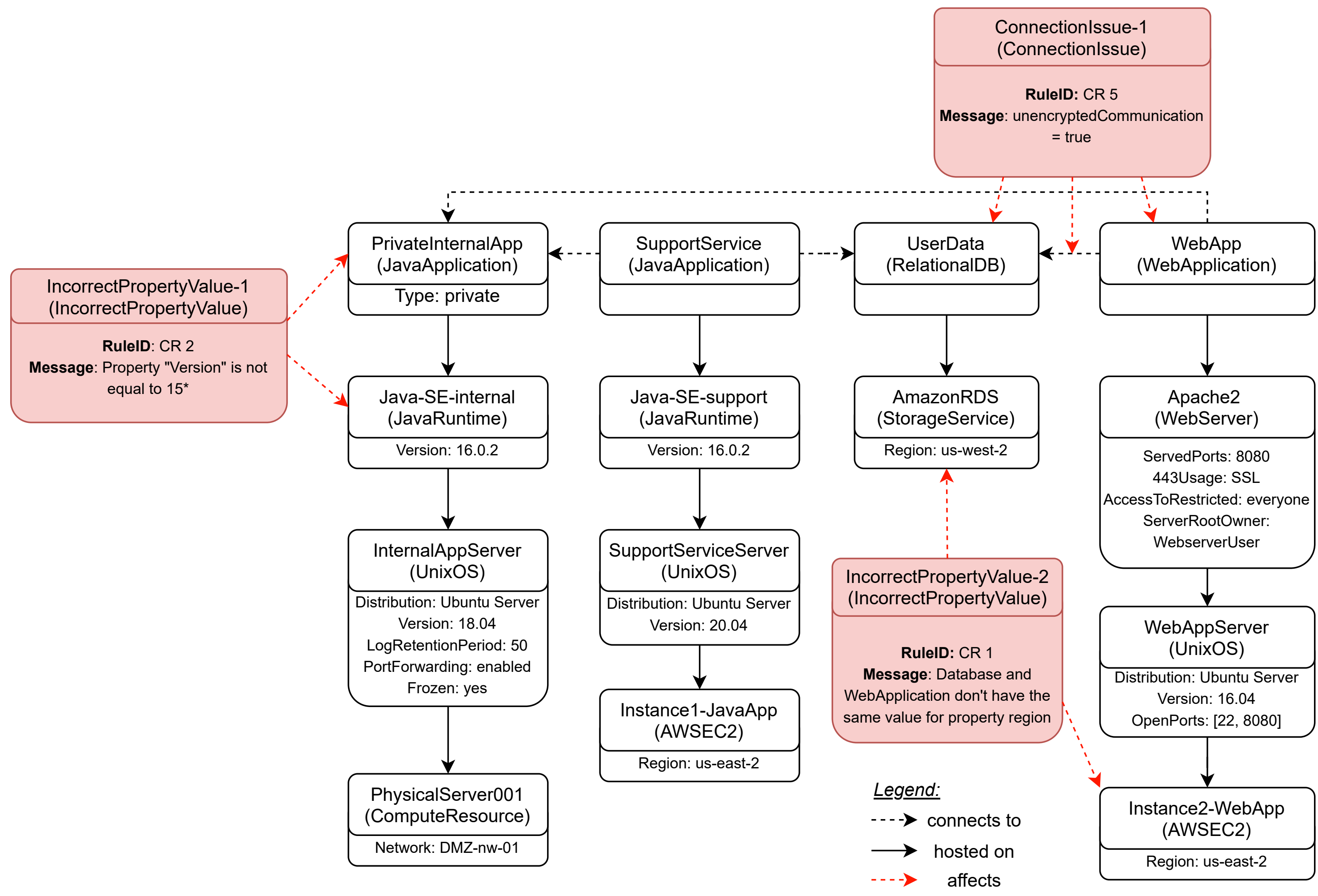

Figure 5.17: The I-EDMM resulting from example rule evaluations using three plugins

By executing these steps for each issue, the annotator produces an I-EDMM in a plugin-independent way, without requiring detailed knowledge about the rules, issues or instance. The I-EDMM that results from the three rules evaluated in the examples is shown in figure 5.17.

5.4.3 Compliant Example

This subsection will briefly show and explain what the instance models of motivating scenarios 1 and 2 look like, after all rules have been detected, evaluated and enforced. This means the resulting infrastructure status, and thus instance model, is fully compliant. Reaching the closest possible compliance status to fully compliant that is possible is the overall goal. The rules not covered in the previous examples can assume to have been evaluated using any of the presented plugins.

5.4.3.1 Scenario 1

In figure 5.18, the instance model for scenario 1 now shows the same application system as in figure 4.2, however all the compliance rules have been correctly enforced. Thus this instance model represents a compliant application system. To understand what was modified in order to make scenario 1 compliant, each rule is evaluated in order of definition.

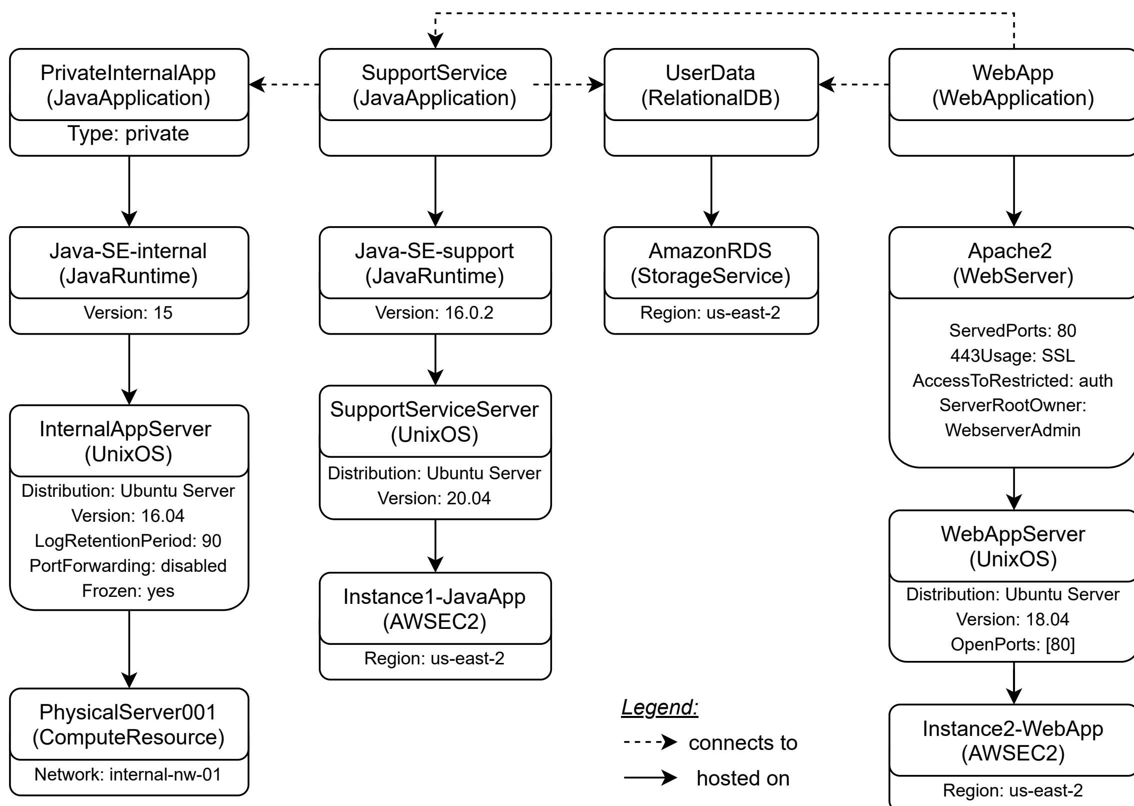

Figure 5.18: (Scenario 1) The application infrastructure without compliance violations.

CR 1 applies to scenario 1 because it contains components of type WebApplication that connects to a component of type RelationalDB, which is a subtype of the Database type. The rule states that both of these components must be hosted in the same region, which was not the case and the database has now been moved so it is located in the same region as the web application.

CR 2 applies to the PrivateInternalApp component. It was initially not compliant, as the application was hosted on a JRE with version 16.0.2. It has now been modified so Java-SE-internal is running JRE version 15.

CR 3 applies to the WebServer type component, as well as the underlying UnixOS component, both of which were serving the incorrect ports. Both have now been corrected so that the web server is serving data on port 80 instead of 8080, and the operating system has this port (but no others) open too.

CR 4 is applicable to the running instance of the PrivateInternalApp again. Initially, there was a direct connection from the WebApp instance to the PrivateInternalApp instance. However, the rule states that only instances of the SupportService app may connect to this application. The connection from the WebApp instance has been removed from the PrivateInternalApp and instead been rerouted to the SupportService, which is supposed to be the destination of this connection.

CR 5, the final rule, is applicable to all components in the upper layer, as they all communicate with each other. Initially, these connections were assumed to be unencrypted (which has been omitted from the diagram). Encryption has now been enforced for communication between these components, which means this rule is now fulfilled.

5.4.3.2 Scenario 2

Figure 5.19 shows an instance model after the compliance violations in figure 4.3 have been identified using the two described rules, and enforced.

The database Database1 has now been migrated from the territory US to the territory DE, and connected to network "internal-nw-01" instead of network "DMZ-nw-01", in order to comply with **CR 6**. Furthermore, the diagnostic level has been modified to capture level 2, and the database discovery has been disabled in order to fulfil **CR 7**. This makes the current infrastructure instance comply to all existing, applicable rules. There were no violations of the compliance rules described in scenario 1, although any one of them could equally apply to scenario 2.

The notion of compliance is simplified in the motivating scenario by multiple assumptions that have been made for easier comprehensibility. First, it is assumed that there are no conflicts between different compliance rules. Furthermore, this motivating scenario does not reflect the fact that enforcing one compliance issue may lead to other compliance issues, as a consequence of changes to the system. The closest example in the first scenario would be the fact that some stacks require outdated versions of components, such as the left stack with the "Frozen" property on the OS component, whereas in the general case, components should always have the most recent version. Another example is that certain settings required for a secure OS, such as disabling port forwarding, might be required for functionality reasons in other settings, such as network connectivity of containerised applications. However, given the way that the compliance rules have been formulated for this particular infrastructure instance, conflicts have been avoided by defining the rules in a very

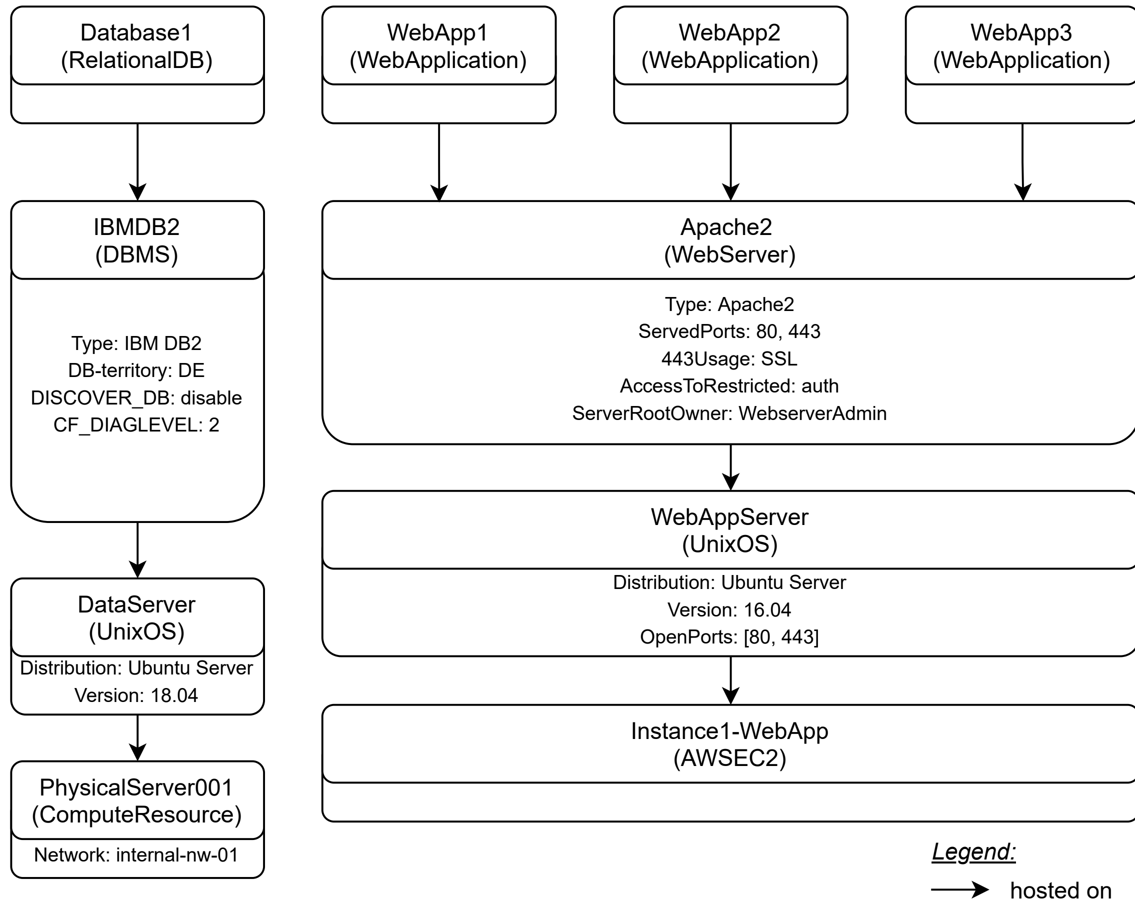

Figure 5.19: (Scenario 2) Set of infrastructure components without compliance violations.

detailed, use-case-specific way. This may not always be possible or even recommendable, which could in consequence lead to a negative change in the number of fulfilled rules overall, where one was actually aiming to fulfil more rules.

In this chapter, the extension of EDMM, denominated as I-EDMM, has been covered. This is the result of the framework's execution. Next, the overall framework architecture and its main components, as well as a compliance rule description metamodel, are described. Finally, this chapter described a number of plugins using examples, and thus evaluated example rules to find the violated rules in the motivating scenarios. The next chapter will present the prototype that has been implemented.

6 Proof of Concept

After having described the solution concept, this chapter goes on to describe the prototype which was implemented as a proof of concept. This prototype will be used to verify the solution concept's feasibility in section 6.2. Verification will include challenging the solution concept by reviewing potential problems with the concept, made visible during the implementation of the prototype.

6.1 Prototype

The prototype[1] focuses on implementing the core components of the solution concept in a way that allows rules to be added and evaluated in a pre-defined way. The prototype allows a set of rules to be checked, and returns the result in a single file that represents an I-EDMM.

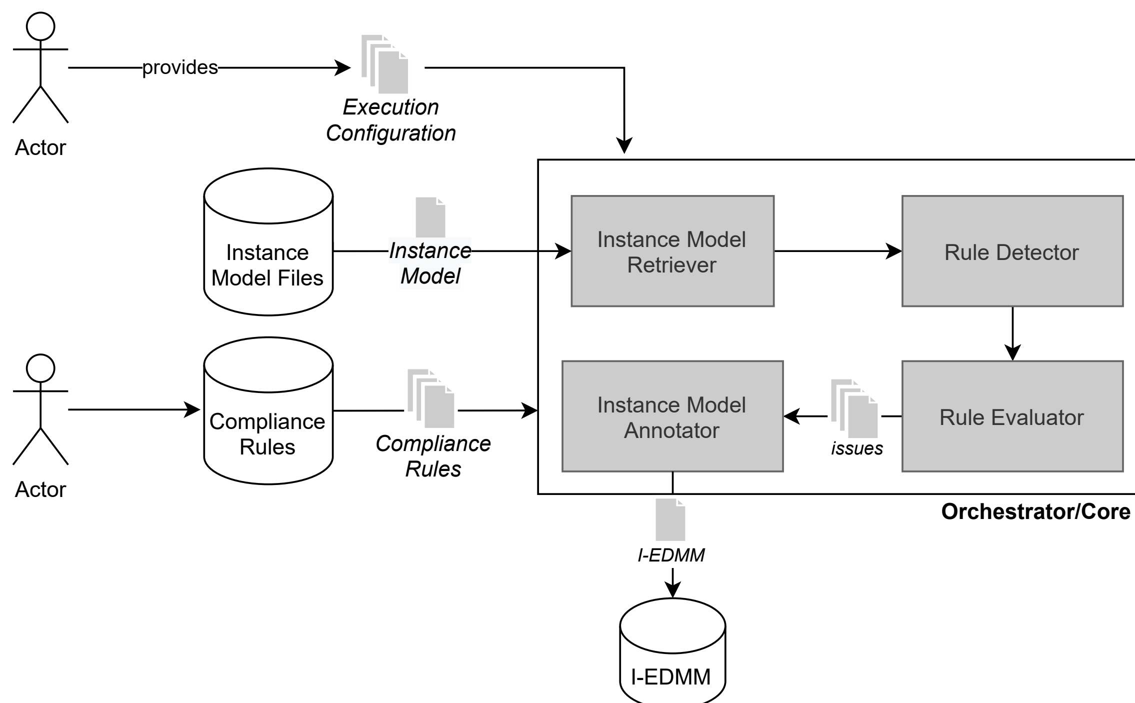

Figure 6.1: Architecture of prototype

[1]Made available here: https://github.com/ElH1/masterthesis-prototype

The prototype follows the idea of a *programme-based compliance rule type plugin* as described in section 5.4.2.3, albeit with modifications. As an example rule, **CR 1** was chosen and implemented. **CR 1** states that whenever a web application connects to a database, both must be hosted in the same region. It was described along with motivating scenario 1 in section 4.1.3.1. Describing this rule in terms of an EDMM instance model, this means that whenever a component of type WebApplication is connected to a component of type Database, both must be hosted on components that have a property “region”, and the value of the “region” properties must be the same for both components. This rule also applies to derivatives of said component types, such as the component type RelationalDB in motivating scenario 1. The prototype uses Java to implement the orchestrator, detector, evaluator, instance model retriever and annotator.

Figure 6.1 shows the architecture of the prototype. The orchestrator creates an instance of each component required for the evaluation of a rule. It uses the execution configuration provided by the user to retrieve a single instance model, iterate through all rules provided in a rule description and decide, for each rule, whether the rule is applicable. If yes, it triggers rule evaluation. The orchestrator collects the issues returned by each rule evaluation, and finally uses the instance model annotator in order to combine and store the results in an I-EDMM.

Figure 6.2 shows the classes of the prototype in a UML class diagram. The `Main` class represents the orchestrator component. It creates and uses each of the other components as required, shown using a composition relation between the `Main` class and the components, as the components are destroyed at the latest when the `Main` instance is destroyed upon termination. The instance model retriever component provides a `retrieveInstance` method that takes the path where an instance model is stored and returns the model in an object which can be used by the framework. The rule detector component provides a `detectRule` method that takes a Java object containing the instance model, and a description of the rule’s location, which is assumed to be a single class in the prototype. The `detectRule` method returns a boolean indicating whether or not the rule is applicable. The rule evaluator component provides an `evaluateRule` method which takes the same arguments as the `detectRule` method. The `evaluateRule` method returns a Java object containing the result of the evaluation, i.e. the issue, which is stored by the `Main` class until framework execution is finished. Finally, the instance model annotator component provides two methods. The first, `annotateModel`, takes an array of Java objects, containing all the results that have been collected during rule detection and evaluation, and returns a Java object that is an I-EDMM. The second, `saveToFile`, allows the storage of the I-EDMM to a file. The next sections will describe each component and step in more detail.

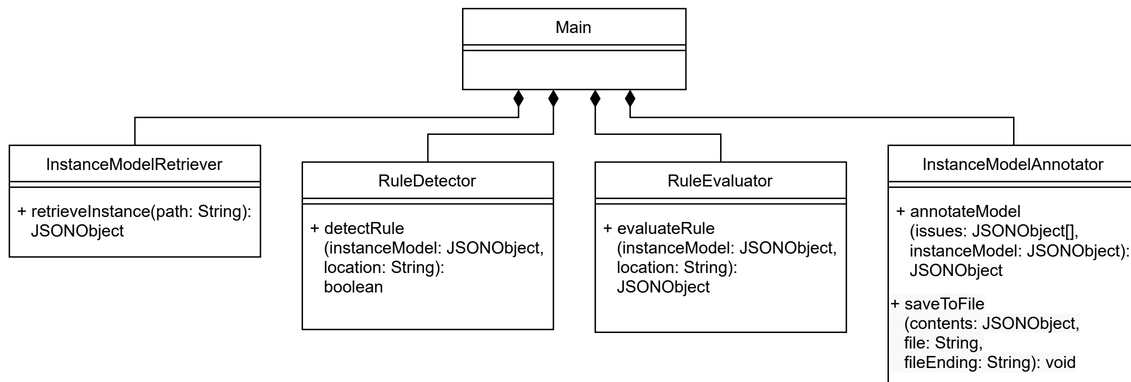

Figure 6.2: Class diagram of prototype

6.1.1 Rule Description and Rule Definition

Rule descriptions in the prototype are provided in a file that must have keys for the rule type, rule detector (as a path to a directory), and rule evaluator (as a path to a directory). Execution configuration is described in the form of execution arguments which a user passes to the framework upon execution. The execution configuration must include (i) a *path to the instance model file*, and (ii) a *path to the rule description file*. In the solution concept, the rule description is what is used by the framework, and parts of which are passed from the framework to the plugin. In the prototype, the rule description can include multiple rule implementations in one location, which does not fully correspond with the metamodel. However, this does allow for any number of rules to be evaluated in one execution, without a large number of execution arguments. In a future implementation, it would be possible to create separate rule descriptions by exchanging the paths to directories in the rule description with paths to files. A list of rule descriptions could then be passed in an execution configuration file. The rule description stands in contrast to the rule definition. The description uses the compliance rule metamodel, and provides information for the framework on where to find (path) and how to execute (rule type) the rules, whereas the definition contains the actual logic (programme files) required to evaluate the compliance of an instance model.

New rule definitions can be added to the framework prototype in the form of Java programmes, allowing the framework to compile and execute them. For the prototype, it is assumed that a programme consists of a single class, and that dependencies used in the classes are provided in a lib folder. Using the rule path provided in the description file, the prototype can find the Java classes, compile them, instantiate objects of the classes and call their methods. In the future, this can be extended to allow the pulling of a rule from a remote repository, provided as a URL.

A rule's definition must implement the interface described in listing 6.1 and be in a package `compliance.rules`. It must implement a method called `detectRule` that takes an instance model, and this method must return a boolean. This boolean must be true if the rule applies to the instance model, and it must be false if the rule does not apply. Furthermore, the rule definition must implement a method called `evaluateRule`. This method takes the same parameters as `detectRule`, but must instead evaluate whether the rule is fulfilled or not, and return the result to the framework in an

object. If the rule is fulfilled, `evaluateRule` must return null. If the rule is not fulfilled, `evaluateRule` must return an issue that has an issue type, a rule ID and a message. The user can decide how to implement the logic.

```
package compliance.rules;

interface complianceRules {

    /* detect whether rule applies to instanceModel */
    public boolean detectRule(JSONObject instanceModel);

    /* detect whether rule is fulfilled in instanceModel */
    public JSONObject evaluateRule(JSONObject instanceModel);

}
```

Listing 6.1: Pseudocode for interface that rules must implement in prototype

An instance model can be described in a file using the specifications[2] for an EDMM deployment model. In the prototype, the instance model is described as a JSON file and manually provided.

6.1.2 Orchestrator

The orchestrator component is implemented in the prototype as a `Main` class. It coordinates the execution, receives the user's execution configuration, retrieves rule description as well as definitions and calls the necessary rule components. Listing 6.2 shows pseudocode of the implementation.

```
/* copies all files at locations in rule description into framework's classpath */
copyFiles(from, to);
/* compiles in framework's own classpath */
compileFiles(to);
/* get the instance model and instantiate components */
InstanceModelRetriever getModel = new InstanceModelRetriever();
Object instanceModel = getModel.getInstance(instanceModelPath);
RuleDetector detector = new RuleDetector();
RuleEvaluator evaluator = new RuleEvaluator();
/* for all rules that were in directory of description, execute detection and evaluation */
for (File child : directoryListing) {
    String ruleClassPath = "compliance.rules." + child.getName();
    /* check if rule applies to instance model */
    boolean applicable = detector.detectRule(instanceModel, ruleClassPath);
    /* if the rule applies, evaluate the rule and annotate the instance model */
    if (applicable) {
         result = evaluator.evaluateRule(instanceModel, ruleClassPath);
         if(result != null) {
            issues[ctr] = result;
        } else {
            issues[ctr] = null;
            System.out.println("rule is fulfilled with name " + child.getName());
```

[2] https://github.com/UST-EDMM/spec-yaml

```
23          }
24      } else {
25          System.out.println("rule not applicable with name " + child.getName());
26          issues[ctr] = null;
27      }
28      ctr++;
29  }
30
31  /* after all rules have been evaluated + their results collected, annotate the instance model
    with this information */
32  InstanceModelAnnotator annotator = new InstanceModelAnnotator();
33  Object iedmm = annotator.annotateModel(issues, instanceModel);
34  if (iedmm != null) {
35      annotator.saveToFile(iedmm, path, filename, fileEnding);
36  }
```

Listing 6.2: Pseudocode of prototype orchestrator

The orchestrator (i) first *copies the files* from the location in the description to the framework's execution environment. Next, it (ii) *compiles the programmes*, so they become executable on the current execution platform. Then, it (iii) *creates instances* of components, and uses the instance model retriever's retrieval method to *get the instance model* from a stored file, parsing it into an object. Then, (iv) for each rule found in the provided directory, the orchestrator calls the detector's *detection method*, to evaluate whether the rule is applicable or not. The detector is passed the previously retrieved instance model and location of the detection implementation. If the rule is not applicable, (v) the orchestrator prints this information to the console. If the rule is applicable, (vi) the orchestrator calls the evaluator's *evaluation method*, in order to check whether the rule is fulfilled or not. If the evaluator returns null, the orchestrator prints the information to console that the rule is fulfilled. Otherwise, the orchestrator stores the result as an item in a Java object array, so the result can be added to the I-EDMM later. After all rules have been processed, (vii) the orchestrator uses an *instance model annotator* to modify the instance model by adding the results to it in the form of issues, thus modifying an instance model into an I-EDMM. Finally, the I-EDMM is saved to a single file.

6.1.3 Instance Model Retriever

Going through the framework components in order of execution, the first is the instance model retriever component. In the prototype, it retrieves the instance model from a file located on disk at a location provided in the execution configuration. It does not care how this model is created, i.e. how this file is created and put at the given location. In the prototype, the instance model is manually supplied and describes motivating scenario 1 from section 4.1.3.1. The file follows the EDMM metamodel, and is parsed into a Java object that represents an EDMM instance.

6.1.4 Detector and Evaluator

In the prototype, the detector and evaluator work in a similar way. They both allow the dynamic loading of a provided class at runtime, depending on the description and definitions provided by the user. This means the framework does not need to know, at compile time, where the rule is located, and can be given this information by the user at runtime. The provided classes are expected to contain the rule definitions.

Detector: Listing 6.3 shows the pseudocode of an implementation of the detector component. It uses Java reflection[3] to retrieve a class at runtime, provided it has been correctly defined by the user, and correctly loaded by the orchestrator. Reflection allows the dynamic loading of classes at runtime, without requiring knowledge of the class before. The detector instantiates an object of the class, calls the rule's `detectRule` method, and stores the result. This result is passed back to the `Main` method for further handling.

```
/* set the args to identify the method signature */
Class[] argclassesDetector = new Class[1];
argclassesDetector[0] = Object.class;

/* get the provided rule's class and detection method */
Class detector = Class.forName(classLocation);
Method detect = detector.getMethod("detectRule", argclassesDetector);

/* create new instance of class */
Object newObject = detector.getDeclaredConstructor().newInstance();

/* invoke detection method */
boolean detectionResult = (boolean) detect.invoke(newObject, instanceModel);
return detectionResult;
```

Listing 6.3: Pseudocode of detection component

Evaluator: This component is always created, but only called by the `Main` class if the detector detects the current rule to be applicable. Listing 6.4 shows the portions of the evaluation method which are different to the detector. The only differences lie in the name of the method that is retrieved, this being `evaluateRule` for the evaluator, and in the result, which is now an object containing information on the evaluation that took place.

```
 /* Get evaluateRule method of provided rule */
 Method evaluate = evaluator.getMethod("evaluateRule", argclassesDetector);

/* invoke evaluation method */
Object evaluationResult = (Object) evaluate.invoke(newObject, instanceModel);

return evaluationResult;
```

Listing 6.4: Pseudocode of evaluation component

[3] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html

This approach outsources all business logic to the rules the user provides, which follows the logic that the user knows the business logic, and the framework cannot know the business logic. The framework must know the interface details, such as the return type of the rules, in order to correctly cast the returned object type. This is automatically enforced by the interface, allowing the framework to work with any correct implementation of the `complianceRules` interface. From a user's perspective, the approach for both detection and evaluation only requires that rules correctly implement the provided rule interface, in order to allow the framework to use the methods.

6.1.5 Instance Model Annotator

The instance model annotator takes a Java object array containing a set of objects. This array describes the results of the rule evaluations. The annotator adds them to the instance model provided to it by the orchestrator, as well as a new relation type, and a number of issue types. In this way, the annotator creates an I-EDMM.

The elements the annotator adds to the instance model are (i) a *relation type "affects"*, which is used to connect the results of rule evaluations (in the form of issues) to the model elements that are affected by the rules. (ii) a *description of the issue types* in the model. The annotator always adds the issue types "base_issue" and "compliance_issue". Beyond these two issue types, it adds whichever types occur in the detected issues. Finally, it (iii) adds the *resulting issues* of rule evaluations (issue type, rule ID and message). It creates a new entry, i.e. node or model element, for each rule evaluation result where the rule was applicable, and violated. All other results are ignored, as they do not belong in an I-EDMM.

After creating the I-EDMM by annotating the instance model with the described elements, the annotator provides the orchestrator with a method of storing this I-EDMM to file. The stored file can then be retrieved either by a user in order to see the results, or potentially by further processing programmes, such as a programme for rule enforcement.

The prototype thus successfully implements prototypical versions of an orchestrator, an instance model retriever that gets the instance model from a file, a detector, an evaluator and an instance model annotator. The next section will show the concept's feasibility by mapping these implemented components to the solution concept from chapter 5.

6.2 Verification

This section will discuss the verification of the prototype. The method for this will be a comparison of components in figure 5.3 (concept architecture) and components in figure 6.1 (prototype architecture). The prototype successfully implements the core components from the solution concept, as is shown in this section.

The prototype implements the instance model retriever component of the solution concept by retrieving an instance model from a file. This component is, however, still lacking a method of parsing the retrieved file into a Java object that represents an EDMM-based instance model, which would be the next step in an implementation. There already exists a parser for EDMM

descriptions in YAML[4]. It should be possible to re-use this for the retriever component, resulting in an EntityGraph object that the framework can use. Plugins, which can be executed just like the detector and evaluator execute provided classes, could create a file the framework's retriever component can then use, which means the retriever would not require much modification.

The prototype also implements a detector and evaluator, which are, similar to the concept components, already plugin-based in a rudimentary form. This is achieved by allowing the execution of any rules in the form of a Java programme, without knowing anything about the classes (i.e. programmes, or plugins) before runtime. Copying and compilation of the provided files can be seen as a form of plugin installation, although the prototype installs rule definitions, not plugins. However, detector and evaluator could execute other detection and evaluation methods in the same way, such as a programme that utilises subgraph isomorphism, for example. Both evaluator and detector take their part of a compliance rule description, as well as an instance model, and return the results required of them (validity and issues, respectively), which is as it is foreseen in the solution concept.

An instance model annotator is also implemented in the prototype. Similar to the foreseen functionality of this component in the concept, the instance model annotator takes an instance model and a set of issues, using this information to create and store an I-EDMM. Instead of storing the information on a central repository, it is stored locally, however this could be modified to a method of writing to a repository location. Either method makes the data available to the user that executed the framework.

Finally, the prototype implements an orchestrator. The concept foresees that the orchestrator manages the execution of framework components, as required, as well as the data transfer between components. This is exactly what the prototypical orchestrator does, by executing only the required framework components, storing intermediate values and passing parameters to components as required. In the prototype, the orchestrator also does parts of the tasks that belong to other components in the concept, such as the plugin manager.

The missing components in the prototype are the rule modeller, scheduler, configuration manager and the plugin-capability. The rule modeller could be implemented by providing an interface to the user that allows them to store rules centrally. Given a situation where the framework might be executed remotely, for example, it could provide an endpoint for the user to upload their classes, whilst making the internals transparent to the user. The scheduler could be implemented by using an existing scheduler tool for the operating system that hosts the framework, for example, CRON[5] on a Linux-based system. A schedule definition could then provide the bash command to execute the framework, along with the location of a user-provided configuration. Furthermore, the configuration manager could be implemented by allowing configuration to be uploaded in a pre-defined file format, which the configuration manager can then import to the framework when required. As an alternative, the configuration manager could directly provide a graphical user interface that allows the user to select possible configurations, which the framework then stores internally, thus making the method of storage transparent to the user. Finally, the plugin-capability can be implemented using the demonstrated functionality of copying and compiling programmes at runtime. This,

[4]The implementation can be found here: https://github.com/UST-EDMM/edmm/tree/master/edmm-core/src/main/java/io/github/edmm/core/parser

[5]https://man7.org/linux/man-pages/man8/cron.8.html

combined with Java reflection, allows the framework to install and execute programmes which it does not know prior to runtime. This approach could be extended to allow for any type of Java programme to be added as a plugin that extends the framework functionality.

With this, the current chapter has successfully shown the feasibility of the solution concept, that being a model-based approach which assists in the management of compliance at runtime. It does this by allowing the evaluation of rules against a model of the infrastructure, using a set of plugins that make the framework generic. The next chapter will now go on to evaluate how well the solution concept fulfils posed requirements.

7 Evaluation

In this chapter, the solution concept is evaluated using the requirements that were collected. The solution concept suggests a framework architecture for aiding in the management of structural compliance of running infrastructure, which consists of a set of infrastructure components. The full set of requirements from section 4.1.2 will be compared to the framework functionalities, in order to analyse to what extent the framework fulfils requirements, and to challenge its feasibility. This entails showing up potential weaknesses of the solution concept, and what issues may arise when implementing the concept.

Req 1 stated that it must be possible to add a compliance rule. The framework allows users to add new rule descriptions through the rule modeller component, making the rules available to the framework from a central repository. Rule configuration is possible through the configuration manager of the framework. Through the decoupling of instance model retrieval and rule description, as well as the possibility to use parameters in the rules through the configuration manager, the reusability of the rules is ensured. This allows users to evaluate rules on different infrastructures and infrastructure components. Thus, the framework fulfils this first requirement, allowing the user to describe new rules, and to re-use them for different scenarios. However, several conditions must be fulfilled for the framework to successfully add a rule. A plugin of the rule type the user is attempting to add must exist, as subsequent usage of the rule will otherwise fail. If the rule type the user is attempting to add does not correspond to an existing, installed plugin, the user will not be able to add the rule. This means the user will have to install a suitable plugin, that can handle the rule type. In addition, the framework does not provide any means of avoiding duplicate rules or ensuring their generic definition, since it assigns a new ID to each rule and does not know the semantics of the rule that is being added. In this sense, it is in the user's responsibility to be aware of already existent rule definitions and to describe rules in a generic, re-usable fashion that doesn't map directly to a concrete infrastructure.

Req 2 stated that it must be possible to extend the methods of compliance rule description. The framework makes this possible using compliance rule type plugins. The user can choose how they wish to describe their rules, either by installing existing plugins, or their own custom plugin, both of which is made possible through the plugin manager component. A compliance rule type plugin defines its own methods for validating a rule's schema, detecting its applicability and evaluating its fulfilment. This way, none of these aspects are pre-defined by the framework. Furthermore, the compliance rule metamodel presented in figure 5.7 provides a generic method of describing rules, regardless of the rule type. This is a novelty compared to other compliance rule modelling approaches described in chapter 3, as all of those approaches use a specific, pre-defined modelling technique. The examples described in section 5.4.2 show that the generic metamodel is applicable to a variety of existing approaches. Although this thesis could not find an example, it may be possible that an approach exists which does not fully fit into the compliance rule metamodel. For example, such an approach may not be able to separate detector and evaluator. In such a case, it would be possible (albeit not recommended) to circumvent certain aspects of the framework, such as the

detector, by returning a static value (e.g. “true”). Using a circumvention such as this, it should still be possible to use the framework with approaches that are not fully compatible. A further aspect, which may be of concern, is that the approach suggested in this thesis has the prerequisite that an associated plugin must exist in order to be able to evaluate rules of a certain type. This may present a certain hurdle, as a qualified person must first develop such plugins, so as to allow people without developer skills, such as security experts, to use the framework.

Req 3 stated that it must be possible for users to extend the methods of retrieving the current infrastructure status. In the context of this requirement, it was first necessary to determine how the current infrastructure status would be described. For this purpose, a representation of instance models as graphs was chosen, using EDMM. This provides a generic method of describing a topology, including that of a set of infrastructure components. Using the properties which can be associated with components and relations, an arbitrary set of information that is required about the infrastructure at hand can be made available to the framework. Multiple plugins can be combined to allow incremental creation of an instance model, for example to populate said properties using domain-specific scripts. What the framework does restrict the user and plugin developers in is how the instance model must be described and passed to the framework, in order to allow the remainder of the framework to remain generic. If necessary, a plugin developer may have to provide an adapter in an instance model retrieval plugin, which translates from a source format (e.g. CSV) to the format the framework requires. If adapters are required, providing them lies in the responsibility of the plugin developer. A user is able to choose from existing instance model retrieval plugins in the plugin manager, or to add new instance model retrieval plugins through the plugin manager. This way, the framework allows for users to extend the methods of retrieving the status of a running infrastructure. It does not limit the user to a pre-defined set of tools, e.g. requiring that monitoring agents are installed on running components, thus allowing the framework to be used with any infrastructure. The downside to this is that it requires users to first develop a plugin, if they have specific requirements that cannot be fulfilled by existing plugins. Similar to the limitations of **Req 2**, this may be a hurdle in using the framework, but is required in order to allow the framework to be generic. Furthermore, it allows users to extend the framework in a way that suits their specific needs, as well as the infrastructure they are working with.

Req 4 stated that it must be possible to re-use certain aspects of the framework, namely the compliance rule descriptions, retrieval methods and evaluation methods. The framework achieves this by providing a configuration manager, and plugin-based components. It is assumed that rules are defined in a generic way, allowing the assignment of values to variable parameters in the rules during configuration, which allows their reuse on multiple infrastructures. This generic description of rules is in no way enforced by the framework, which means users could define domain-specific rules. The same goes for the retrieval plugins, which can be defined in a domain-specific way. By doing this, users would force themselves to describe new rules and write new instance retrieval plugins every time something changes. Enforcing the generic nature of framework extensions and rules does not lie in the framework’s responsibility, as it is only required that it can be used in a generic way, not that it must only be used in such a way. This allows users to use the framework for their business-specific requirements. After adding a rule, the rule does not become directly available, it must first be configured and associated with an execution configuration, which includes a schedule. This successfully decouples rule description from binding rules to concrete running infrastructures and their components. Both retrieval methods and evaluation methods can be added to the framework as plugins, namely as instance model retrieval plugins and compliance rule type plugins, respectively. These can also be configured and associated with an execution configuration

using the configuration manager. Thus, the framework allows for the re-use of plugins as well as compliance rules, by separating their logic from a binding to specific use-cases in form of configuration, and so fulfils this requirement.

Req 5 stated that it must be possible to repeat rule evaluation regularly, without requiring a manual trigger every time. The framework allows this through the configuration manager and scheduler. The configuration manager allows storage of configuration files, thus allowing users to store all information the framework requires for an execution. The scheduler, which is an always-on component, can then use an execution configuration in order to automatically trigger a framework execution at a time defined by the user. This allows repeated rule evaluation through automatic triggers, that have been manually set up, and also allows re-use of configuration files, as they are stored by the configuration manager for future retrieval.

Req 6 stated that it must be possible to evaluate whether rules are fulfilled or not. The framework achieves this, in a generic way, using the evaluator, which belongs to the plugin-based framework core. Through compliance rule type plugins, the framework allows a user to evaluate rules in ways required by them, or by choosing existing methods. In order to make the framework execution more efficient, it was decided to separate the detector and evaluator. During the prototype implementation, it was found that the current approach of passing a boolean from the detection method of the plugin to the detector component of the framework could lead to duplicate logic in the evaluator. As shown by example rule definitions in the prototype, it was required to re-implement and re-execute parts of the detection process in the evaluator. A generic framework cannot know in advance what information would need to be passed between detector and evaluator in order to avoid this re-implementation. Avoiding this would require an intermediate storage, as well as management logic to prevent accidentally using an incorrect version of data. A further insight from the prototype is that the assumption which led to the separation of detector and evaluator is correct, as the evaluator of an example rule definition in the prototype is more complex than the detector. It uses a recursive helper method to traverse the graph, whereas the detector only requires loops. The framework allows a user to see the results of each rule which was evaluated, as well as details on issues which were found, in the form of an I-EDMM. In this way, the user gains enough information in order to rectify compliance violations if necessary, and this information should be usable for automated rule enforcement in the future.

8 Conclusion and Outlook

In the scope of this thesis, research was conducted on if and how it is possible to provide a generic approach that aids compliance management automation of heterogeneous application infrastructures. The first two research questions (RQ1 and RQ2), asked how runtime structural compliance rules can be described, and how these descriptions can be made generic. In response to these questions, the thesis presents a compliance rule description metamodel, which has multiple extension points. These are the rule type, a detector element, and an evaluator element. The detector and evaluator use parameters, in order to allow the re-use of the rule descriptions. These parameters are instantiated to a concrete value when a rule is used. This structure allows the metamodel to be applied to a variety of rule description methods, and the parameters allow for a rule description to be generic, so it can be re-used. Examples are given for each of the metamodel elements, based on multiple existing approaches to describing compliance rules. The examples describe rules either as graphs, logic clauses or programmes.

Finally, the thesis proposed an approach to answer research question three (RQ3). This research question asks if and how a framework can be developed that automates compliance management at runtime for various rule types and various infrastructures, allowing extensibility in the future. The solution concept this thesis provides uses models of the rules and the infrastructure state to evaluate the compliance status. The concept is plugin-based in order to allow extensibility in the area of rule description and evaluation types, as well as in the area of infrastructure state retrieval. Using this approach, which was verified with the help of a prototype, the thesis successfully shows that it is possible to manage compliance of heterogeneous application infrastructures at runtime in a generic, extensible way. Although the processes of rule applicability detection and fulfilment evaluation were separated for efficiency reasons in the concept, this separation was also found to have a disadvantage. Due to this approach taken in the concept, where the detector only returns a boolean, data cannot be passed between the component responsible for detection, and that responsible for evaluation. This means some logic may be repeated in the evaluation process, for example in order to find a starting point for traversing the graph that represents the infrastructure state. However, since it is assumed that the detector is a relatively fast executing component, and the evaluator may be slow in execution, this was found to be a negligible disadvantage. This could be optimised in the future by using intermediate storage for data transfer. In order to allow the generic description of detected compliance violations, the thesis additionally provided the I-EDMM. It allows for a graph-based description of violations, describing what is violated and which components are affected by this violation. The method of modelling the results taken by this work's approach should facilitate the use of the evaluation results in further business process activities, such as the enforcement of compliance, and the final documentation of the overall process execution.

Over the course of this thesis, multiple new questions arose, which could be of interest for future research. One such research question that arose was how one could automate the execution of responses to detected compliance violations. Such an approach should take into consideration a fact

which this thesis has simplified for research purposes, namely that conflicts may occur between multiple compliance rules. When executing automated responses, it would be necessary to take into consideration how to resolve such conflicts.

Visualisation of the current infrastructure state after instance model retrieval, or after compliance evaluation, was another idea that came up. One could use the EDMM instance model or resulting I-EDMM to graphically represent the current infrastructure state for a user, for example colour-coding (red, green) the compliance status of components, and showing the user how components relate to each other, as well as their current properties. Furthermore, one could also represent the degree of compliance, for example using the number of rules that are fulfilled or violated, allowing a user to see the effect of changes on an overall compliance status of an infrastructure.

A further idea would be to allow integration into an issue management system such as that proposed by Speth et al. [SBB20], which would allow compliance issues to be tracked together with other issues that may occur across services or applications, for example during development.

Furthermore, the efficiency of the solution approach could potentially be improved by allowing for change tracking of the instance models between retrievals. If no change occurs on any of the components between two consecutive executions of the framework, and no new rules have been added, there would be no reason to re-execute detection and evaluation processes for each rule. One could instead instantly return the previous I-EDMM. However, this would require more storage, and version tracking. It is not clear whether such a change tracking approach would be any more efficient, overall, than re-executing detection and evaluation in each framework execution. Thus, it would be necessary to weigh up the efficiency of change tracking and state evaluation against just detecting and evaluating rules again.

A final aspect that could be of interest is how to extract a generic compliance rule description from existing, specific (i.e. bound) definitions. This would allow for automatic integration of use-case specific rules into a generic framework such as the one this thesis proposes, and remove the need for use-case specific plugins in cases where there are use-case specific rule definitions.

Bibliography

[AAE16] N. Alshuqayran, N. Ali, R. Evans. "A systematic mapping study in microservice architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15 (cit. on p. 1).

[ABE+15] A. Awad, A. Barnawi, A. Elgammal, R. Elshawi, A. Almalaise, S. Sakr. "Runtime Detection of Business Process Compliance Violations: An Approach Based on Anti Patterns". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1203–1210. ISBN: 9781450331968. DOI: 10.1145/2695664.2699488. URL: https://doi.org/10.1145/2695664.2699488 (cit. on p. 14).

[AKL+09] T. Anstett, D. Karastoyanova, F. Leymann, R. Mietzner, G. Monakova, D. Schleicher, S. Strauch. "MC-Cube: mastering customizable compliance in the cloud". In: *Service-Oriented Computing*. Springer, 2009, pp. 592–606 (cit. on p. 13).

[Awa10] A. Awad. "A compliance management framework for business process models". PhD thesis. University of Potsdam, Dec. 2010. URL: http://opus.kobv.de/ubp/volltexte/2010/4922/ (cit. on p. 14).

[BAE+15] A. Barnawi, A. Awad, A. Elgammal, R. El Shawi, A. Almalaise, S. Sakr. "Runtime self-monitoring approach of business process compliance in cloud environments". In: *Cluster Computing* (Dec. 2015). DOI: 10.1007/s10586-015-0494-0 (cit. on p. 14).

[BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. "Integrated Cloud Application Provisioning: Interconnecting Service-centric and Script-centric Management Technologies". In: *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS 2013)*. Springer, 2013. DOI: 10.1007/978-3-642-41030-7_9 (cit. on p. 1).

[BBKL13] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "Automated Discovery and Maintenance of Enterprise Topology Graphs". In: *Proceedings of the 6th IEEE International Conference on ServiceOriented Computing & Applications (SOCA 2013)*. IEEE Computer Society, Dec. 2013, pp. 126–134. DOI: 10.1109/SOCA.2013.29 (cit. on pp. 16, 48).

[Bra97] S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels (RFC2119)*. 1997. URL: https://datatracker.ietf.org/doc/html/rfc2119 (visited on 09/08/2021) (cit. on p. 21).

[Bri17] Y. Brikman. *Terraform: Up and Running, First Edit*. 2017 (cit. on p. 6).

[CMV+10] R. Cloutier, G. Muller, D. Verma, R. Nilchiani, E. Hole, M. Bone. "The concept of reference architectures". In: *Systems Engineering* 13.1 (2010), pp. 14–27 (cit. on p. 12).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications". In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 22–27. URL: https://www.iaas.uni-stuttgart.de/publications/INPROC-2017-12-Declarative-vs-Imperative-Modeling-Patterns.pdf (cit. on p. 17).

[FBKL17] M. P. Fischer, U. Breitenbücher, K. Képes, F. Leymann. "Towards an Approach for Automatically Checking Compliance Rules in Deployment Models". In: *Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*. Xpert Publishing Services (XPS), 2017, pp. 150–153. ISBN: 978-1-61208-582-1. URL: https://www.iaas.uni-stuttgart.de/publications/INPROC-2017-47-Towards-an-Approach-for-Automatically-Checking-Compliance-Rules-in-Deployment-Models.pdf (cit. on pp. 15, 16).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (cit. on p. 12).

[GDP] GDPR. *General Data Protection Regulation*. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1532348683434&uri=CELEX:02016R0679-20160504 (cit. on p. 9).

[GKN08] D. Ganesan, T. Keuler, Y. Nishimura. "Architecture Compliance Checking at Runtime: An Industry Experience Report". In: *2008 The Eighth International Conference on Quality Software*. 2008, pp. 347–356. DOI: 10.1109/QSIC.2008.45 (cit. on p. 15).

[GPGN16] J. García-Galán, L. Pasquale, G. Grispos, B. Nuseibeh. "Towards Adaptive Compliance". In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, May 2016. DOI: 10.1145/2897053.2897070 (cit. on p. 13).

[HBB+21] L. Harzenetter., T. Binz., U. Breitenbücher., F. Leymann., M. Wurster. "Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models". In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER,* INSTICC. SciTePress, 2021, pp. 99–110. ISBN: 978-989-758-510-4. DOI: 10.5220/0010477900990110 (cit. on pp. 16, 44, 48, 51).

[HN05] H. R. Hansen, G. Neumann. "Wirtschaftsinformatik 1-Grundlagen und Anwendungen". In: *Stuttgart, Lucius & Lucius Verlagsgesellschaft* (2005) (cit. on p. 19).

[KBF+20] C. Krieger, U. Breitenbücher, M. Falkenthal, F. Leymann, V. Yussupov, U. Zdun. "Monitoring Behavioral Compliance with Architectural Patterns Based on Complex Event Processing". In: *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC 2020)*. Springer International Publishing, Mar. 2020, pp. 125–140. DOI: 10.1007/978-3-030-44769-4_10 (cit. on pp. 11, 16).

[KBKL18] C. Krieger, U. Breitenbücher, K. Képes, F. Leymann. "An Approach to Automatically Check the Compliance of Declarative Deployment Models". In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC2018)*.

IBM Research Division, 2018, pp. 76–89. URL: https://www.iaas.uni-stuttgart.de/publications/INPROC-2018-42-An-Approach-to-Automatically-Check-the-Compliance-of-Declarative-Deployment-Models.pdf (cit. on pp. 1, 15, 16, 19, 52).

[KKR+13] F. Koetter, M. Kochanowski, T. Renner, C. Fehling, F. Leymann. "Unifying compliance management in adaptive environments through variability descriptors (short paper)". In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE. 2013, pp. 214–219. DOI: 10.1109/SOCA.2013.23 (cit. on pp. 9, 10, 17).

[KKW+14] F. Koetter, M. Kochanowski, A. Weisbecker, C. Fehling, F. Leymann. "Integrating compliance requirements across business and IT". In: *2014 IEEE 18th international enterprise distributed object computing conference*. IEEE. 2014, pp. 218–225. DOI: http://dx.doi.org/10.1109/EDOC.2014.37 (cit. on pp. 4, 11, 17).

[KS21] D.-K. Kipker, D. E. Scholz. "Das IT-Sicherheitsgesetz 2.0". In: *Datenschutz und Datensicherheit-DuD* 45.1 (2021), pp. 40–45. DOI: https://doi.org/10.1007/s11623-020-1387-9 (cit. on p. 9).

[KUSM18] M. Kleehaus, Ö. Uludağ, P. Schäfer, F. Matthes. "MICROLYZE: a framework for recovering the software architecture in microservice-based environments". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2018, pp. 148–162 (cit. on pp. 16, 44, 47, 51).

[Low11] L. Lowis. "Automatisierte Compliance-Prüfung von Geschäftsprozessen". PhD thesis. Albert-Ludwigs-Universität Freiburg, 2011. URL: https://freidok.uni-freiburg.de/data/8292 (cit. on pp. 10, 11).

[LSK09] D. Liginlal, I. Sim, L. Khansa. "How significant is human error as a cause of privacy breaches? An empirical study and a framework for error management". In: *computers & security* 28.3-4 (2009), pp. 215–228 (cit. on p. 20).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing". In: (2011) (cit. on p. 8).

[MMT+14] E. Metalidou, C. Marinagi, P. Trivellas, N. Eberhagen, C. Skourlas, G. Giannakopoulos. "The human factor of information security: Unintentional damage perspective". In: *Procedia-Social and Behavioral Sciences* 147 (2014), pp. 424–428 (cit. on p. 20).

[Mor20] K. Morris. *Infrastructure as Code*. O'Reilly Media, 2020 (cit. on pp. 1, 3, 6, 7).

[NZP+21] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, W. Hasselbring. "Detector-based component model abstraction for microservice-based systems". In: *Computing* (2021), pp. 1–31 (cit. on p. 49).

[Obj17] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML) Version 2.5.1*. Dec. 5, 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 09/03/2021) (cit. on pp. 6, 41, 45).

[OGP03] D. Oppenheimer, A. Ganapathi, D. A. Patterson. "Why do Internet services fail, and what can be done about it?" In: *USENIX symposium on internet technologies and systems*. Vol. 67. Seattle, WA. 2003 (cit. on pp. 19, 20).

[Olu14] H. S. Oluwatosin. "Client-server model". In: *IOSRJ Comput. Eng* 16.1 (2014), pp. 2278–8727 (cit. on p. 12).

[SBB20] S. Speth, U. Breitenbücher, S. Becker. "Gropius—A Tool for Managing Cross-component Issues". In: *European Conference on Software Architecture*. Springer. 2020, pp. 82–94 (cit. on p. 78).

[SBKL19] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns". In: *SICS Software-Intensive Cyber-Physical Systems* 34.2 (2019), pp. 85–97 (cit. on pp. 16, 55).

[SFG+11] D. Schleicher, C. Fehling, S. Grohe, F. Leymann, A. Nowak, P. Schneider, D. Schumm. "Compliance domains: A means to model data-restrictions in cloud environments". In: *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*. IEEE. 2011, pp. 257–266 (cit. on p. 14).

[SGL+11] D. Schleicher, S. Grohe, F. Leymann, P. Schneider, D. Schumm, T. Wolf. "An approach to combine data-related and control-flow-related compliance rules". In: *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2011, pp. 1–8 (cit. on p. 13).

[SLSW10] D. Schleicher, F. Leymann, D. Schumm, M. Weidmann. "Compliance scopes: Extending the BPMN 2.0 meta model to specify compliance requirements". In: *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2010, pp. 1–8 (cit. on p. 14).

[Spe19] S. Speth. "Issue Management for Multi-Project, Multi-Team Microservice Architectures". MA thesis. University of Stuttgart, Nov. 25, 2019 (cit. on p. 17).

[Ste19] J. Sterne. "Plug-in". In: *Encyclopedia Britannica*. Oct. 3, 2019. URL: https://www.britannica.com/technology/plug-in (cit. on p. 34).

[sum21] sumo logic. *What is Application Infrastructure?* Aug. 11, 2021. URL: https://www.sumologic.com/glossary/application-infrastructure/ (visited on 08/11/2021) (cit. on p. 3).

[tec21] techwithtech. *IT Architecture vs. IT Infrastructure: How Do They Differ?* Aug. 4, 2021. URL: https://techwithtech.com/it-architecture-vs-it-infrastructure/ (visited on 08/11/2021) (cit. on p. 4).

[TRH+15] Y. Takhma, T. Rachid, H. Harroud, M. R. Abid, N. Assem. "Third-party source code compliance using early static code analysis". In: *2015 International Conference on Collaboration Technologies and Systems (CTS)*. 2015, pp. 132–139. DOI: 10.1109/CTS.2015.7210413 (cit. on p. 11).

[WBF+20] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. "The essential deployment metamodel: a systematic review of deployment automation technologies". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 63–75. DOI: https://doi.org/10.1007/s00450-019-00412-x (cit. on pp. 4, 5).

[ZBKL18] M. Zimmermann, U. Breitenbücher, C. Krieger, F. Leymann. "Deployment Enforcement Rules for TOSCA-based Applications". English. In: *Proceedings of The Twelfth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2018)*. Ed. by G. Yee, S. Rass, S. Schauer, M. Latzenhofer.

Xpert Publishing Services, Sept. 2018, pp. 114–121. ISBN: 9781612086613. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2018-32&engl=1 (cit. on pp. 15, 16, 52).

All links were last followed on 6th October 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature