Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Tracing the Impact of SLO Violations on Business Processes across a Microservice Architecture with the Saga Pattern

Sarah Sophie Stieß

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing Steffen Becker
Supervisor:	Dr. rer. nat. Uwe Breitenbücher, Sandro Speth, M.Sc.
Commenced:	March 31, 2021
Completed:	September 30, 2021

Abstract

Context. Applications in the microservice architecture style consist of many individual services. SLOs describe the quality at which they provide a functionality, as an example their responsetime or availability. Patterns exist to handle recurring problems better. Among these is the saga pattern [Ric18], which deals with transactions distributed across multiple services.
Problem. SLO violations may propagate across the architecture and cause unintended behaviour in the business process. Patterns may hide the cause of a business process' behaviour. As for the saga pattern, an SLO violation may trigger the rollback of a transaction. The process owner notices the rollback. But a rollback is an acceptable behaviour for a transaction, such that they either do not not question the rollback's origin at all or they do question it but cannot find any fault in it. Thus, the connection to the SLO violation remains unidentified.
Objective. This thesis' objective is to expose the impact of SLO violations on a business process in the presence of the saga pattern.
Method. The means to achieve this objective is a notification that informs a user about the SLO violation and its possible impacts on the business process and a modelling language for models that capture all knowledge required to create such notifications. The user received the notification as an issue. An expert survey evaluates the language and the concept and a experiment assures that the modelling language fulfils its purpose. The experiment employs the T2-Project as a reference architecture.
Result. This thesis' results include the aforementioned modelling language, the concept for the notification, and a prototype to calculate impacts. The language connects models of architectures with those of business processes while reusing existing models. The prototype uses models according to the designed language and calculate the impact of SLO violations upon receiving notice about a violation. It also creates issues for these impacts. There is an expert survey and an experiment that attempt to validate this thesis content.
Conclusion. The designed modelling language is of use when connecting existing models of architectures and business processes while also representing an instance of the saga pattern. Models according to the language are useable to calculate the impact of SLO violations on the business process. Notifications about impacts that are not apparent in the business process are helpful.

Kurzfassung

Kontext. Anwendungen im Microservice Architekturstil bestehen aus vielen eigenständigen Services, die lose mit einander interagieren. SLOs beschreiben die Qualität der von ihnen angebotenen Funktionalitäten, zum Beispiel ihre Antwortzeit oder Verfügbarkeit. Patterns existieren um wiederkehrende Problem zu behandeln. Unter ihnen ist das Saga Patten, welches sich mit Transaktionen befasst, die sich über mehrere Services erstrecken.
Problem. SLO Verletzungen propagieren entlang der Architektur und verursachen ungewolltes Verhalten im Business Prozess. Die Verwendung von Patterns kann die Ursachen für das Verhalten eines Business Prozesses verschleiern. Beim Saga Pattern, kann eine SLO Verletzung dazu führen, dass ein Transaktion zurück gerollt wird. Der Besitzer des Prozesses bemerkt das Rollback, hinterfragt jedoch dessen Ursachen nicht, da Rollbacks vorgesehenes Verhalten sind. Der Zusammenhang zu einer SLO Verletzung bleibt unbemerkt.
Ziel. Das Ziel dieser Arbeit ist es, die Auswirkungen von SLO Verletzungen auf Business Prozesse in der Gegenwart von Sagas offen zu legen.
Methode. Die Mittel zum erreichen dieses Ziels sind eine Modellierungssprache und das Konzept einer Nachricht, die den Nutzer über SLO Verletzungen und ihre Auswirkungen auf den Business Prozess informiert. Eine Expertenumfrage evaluiert die Sprache und das Konzept.
Ergebnis. Zu den Ergebnissen dieser Arbeit gehören die zuvor erwähnte Modellierungssprache, ein Konzept für die Nachrichten an den Nutzer und ein Prototyp, der Modelle gemäß der entworfenen Sprache nutzt um die Auswirkungen von SLO Verletzungen zu ermitteln und Nachrichten darüber erzeugt. Dieser Prototyp ist der Nachweis, dass die Modellierungssprache ihren Zweck erfüllt. Eine Expertenumfrage versucht die Inhalte dieser Arbeit zu validieren.
Schlussfolgerung. Die entworfenen Modellierungssprache kann genutzt werden um existierende Modelle der Architektur und des Business Prozesses zu verknüpfen und um Verwendungen des Saga Patterns zu repräsentieren. Modelle gemäß der entworfenen Sprach sind von Nutzen, um die Auswirkungen von SLO Verletzungen auf Business Prozesses zu ermitteln. Nachrichten über Auswirkungen, die im Prozess nicht direkt sichtbar sind, sind hilfreich.

Contents

List of Figures

List of Tables

List of Listings

Glossary

CRUD Create, read, update and delete. 49, 54

Eclipse Layout Kernel An Eclipse project providing a multitude of layouting algorithms to be used in editors.. 33

Ecore Meta-model of the EMF. 8, 9, 29, 30, 31, 30, 31, 32, 53

GraphQL A query language developed by Facebook. 35

Gropius A cross-component issue management system. 3, 5, 10, 18, 19, 20, 22, 23, 28, 29, 30, 31, 32, 33, 35, 36, 37, 39, 40, 41, 50, 51, 53, 55

JSON File format that saves data as key value pairs. Readable for humans and machines alike. xiii, 28, 37, 42, 48, 53, 63, 64

MICROLYZE A framework to recover the architecture of applications in the microservice architecture style. 13, 15, 16

Model Development Tools An Eclipse project concerned with modelling. It provides meta-models and tools to develop them. 9

Sirius A tool for creating graphical concrete syntaxes. Belongs to the Eclipse Foundation. 32, 33, 53, 54

SoLOMON A tool for SLO management. 10, 19, 20, 28, 29, 30, 31, 32, 36, 37, 39, 40, 41, 51, 54

Acronyms

API Application Programming Interface. 5, 19, 30, 35, 36

BPMN Business Process Modeling Language. 5, 8, 9, 13, 14, 19, 20, 22, 23, 26, 27, 30, 31, 36, 47, 53, 54, 55

EMF Eclipse Modelling Framework. xv, 8, 10, 29, 30, 32, 46, 47, 53

GQM Goal Question Metric. 3, 4, 39, 49

SLA Service Level Agreement. 10, 13

SLO Service Level Objective. xv, 1, 2, 3, 4, 5, 10, 11, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 25, 28, 29, 30, 31, 32, 33, 35, 36, 37, 39, 40, 41, 42, 41, 42, 46, 47, 48, 49, 50, 51, 53, 54, 55

UML Unified Modeling Language. 5, 9, 10, 13, 14, 13, 14, 15, 26, 27, 54, 55

1 Introduction

Context. An application in the microservice architecture style consists of many independent services that interact with each other in a loosely coupled manner. Each service is an independent unit that focuses on a small set of functionalities. For functionalities that are outside of its expertise, it must rely on other services. Naturally, each service manages its data itself. The microservice patterns [Ric18] help in dealing with recurring problems regarding microservices. One of these patterns is the saga pattern. It deals with transactions that stretch across multiple microservices.

An Service Level Objective (SLO) describes a guaranteed quality with regard to a subset of a service's attributes, such as a service's availability or response time. One service promises to provide a functionality according to the SLO and other services consume the providing service's interface on the assumption that this interface complies to the guaranteed SLO. If the providing service violates its SLO, a consuming service may run into problems. It may wait for a reply for the duration of the promised response time only and fail if the response time is exceeded or it may be delayed, too. SLO violations propagate through the architecture because of this. Some violations affect the business process and the user as well.

Problem. As a violation propagates from service to service, it might encounter services that implement patterns. The patterns might influence how the violation propagates and in what form it is noticeable in the business process. In fact, a pattern might hide a SLO violation well enough, such that it is no longer identifiable as the cause of the problems it triggers.

As a first example, imagine a microservice architecture, such as the one shown in the lower half of Figure 1.1. It is a web store with the business goal of selling tea. Its business process is depicted in the figures upper half. A customer loads the store's website, places tea in their shopping cart, and proceeds to checkout. Once the order is placed, the store records the order, updates the inventory, and contacts an external service, such as PayPal or the customer's credit institute, to execute the payment. Some services define SLO rules, in Figure 1.1 depicted as a tiny scroll attached to the *provider services'* interfaces. In a real application all services might define SLOs. For the sake of brevity, Figure 1.1 only depicts the SLO at the external payment provider. This service, as an example, promises that its response time is on average below 30 seconds. Focus on the *Payment* and *provider* services and assume that the *payment* service implements a retry when calling the interface of the *provider* service. Now imagine that the provider service takes more than 30 seconds to reply. The payment service trusts in the maximum response time, and after 30 seconds of waiting, it timeouts send its request again. What is visible on the process level, is a delay of the *doPayment* task because the responsible service is busy with retries. It is not apparent that the cause of the delay is the provider service. There is not too much damage done, as the SLO violation is still noticeable in the process level, and someone might be tempted to search for the root cause.

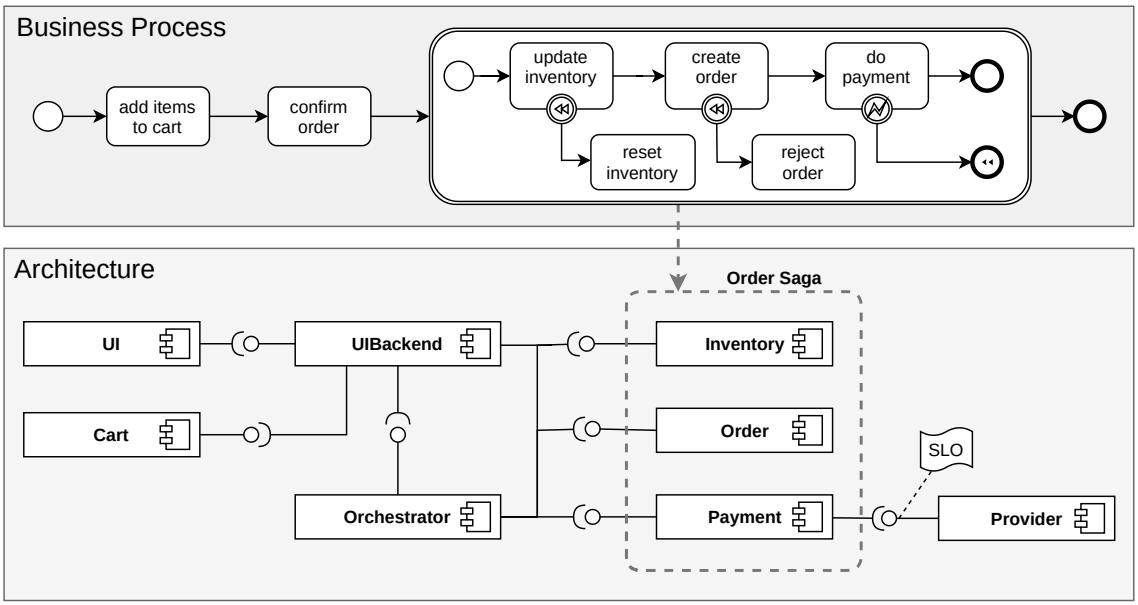


Figure 1.1: A microservice architecture and its business process.

Next, imagine the same microservice architecture, but this time include that it implements the saga pattern. It has a *Order saga* that spans the services *Order*, *Payment* and *Inventory*, as marked with a dashed line in Figure 1.1. Placing an order is now a transaction. Therefore, if any of the order transaction's steps fails, the entire placement of the order must fail, and therefore must roll back.

The SLO rules are the same as before. Again, the provider violates the response time rule. The payment service still trusts in the maximum response time and reports all payments that take longer than 30 seconds as failed. Thus, the transaction rolls back, and the customer's order gets cancelled. Normally, a rollback is perfectly fine. As an example, it may happen because a customer cannot pay. In this case, it is not fine as it resulted from the SLO violation. However, no one will look into it because in the business process they only notice a rollback, but no SLO violation. And even if they do look into it, perhaps because some monitoring system alerted them about the violation, it would be a time-consuming task. They know about the existence of a violation, but they might have no insight into the impact that the violation has on the system.

Objective. This thesis' objective is to inform about the impacts of SLO violations on the business process upon the occurrence of said violations. This includes information about what happened, what effect this incident has on the business process, and how it propagated there. In the scope of this thesis the *what happened* is always an SLO violation and the *effect* is always that the process does not reach its business goal. The *how* is the trace of all the components that the violation impacted before it reached the business process. If an implementation of the saga pattern influences the propagation, this should also be part of the trace.

Figure 1.2 depicts an example for this. The *Provider* service violates one of its SLOs. The *Payment* service consumes an interface that is affected by the violation. It propagates the violation up to the *do payment* task in the business process because it realises that task. Additionally, the *Payment* service participates in a saga. This influences the propagation of the violation, as an example a delay may result in a rollback.

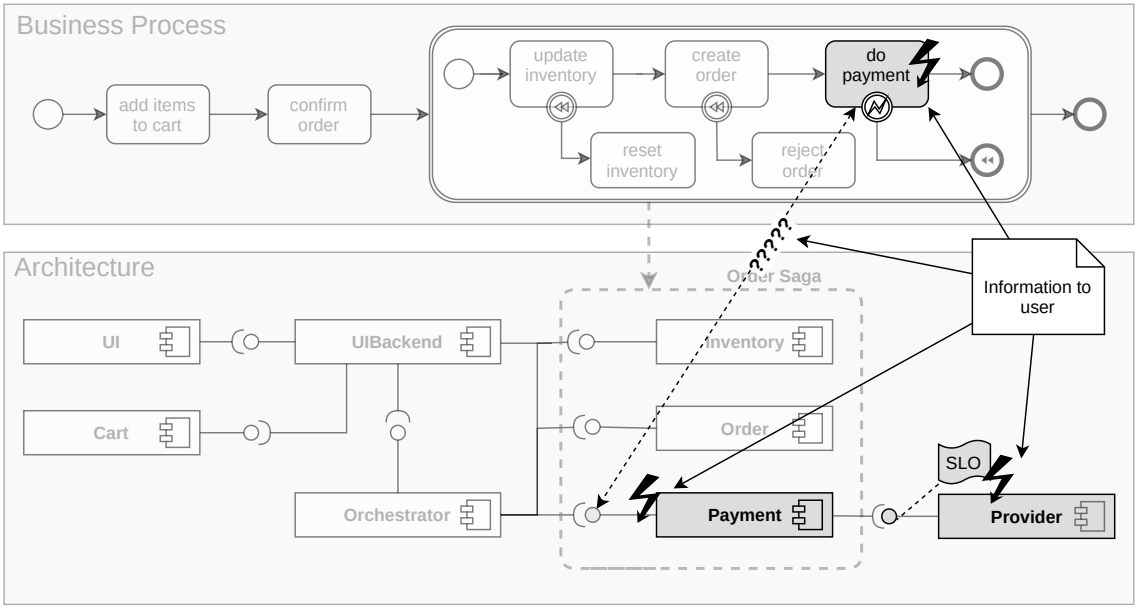


Figure 1.2: A microservice architecture and its business process. An SLO violation occurred. The information provided to the user are highlighted in black.

The path from *Provider* to *Payment* is already in the model depicted in Figure 1.2. However, the path from *Payment* to the business process' task is not yet in there. This connection between architecture and process is contributed by this thesis.

Method. This thesis utilizes a model-based approach. At first it needs a model. If any fitting models already exist it might reuse them. Otherwise it has to start from scratch. Such a model must be described by a modelling language. Thus, this thesis develops a modelling language with a backing meta-model. Because a plethora of languages to model processes or architectures already exist, this thesis seeks to reuse them as much as possible. As part of the language desgin, we decided on which of the existing languages to reuse for thesis' modelling language. Furthermore, we created a prototype[1] that computes the impacts of SLO violations. The computation operates on a model, provided in the formerly mentioned modelling language. In addition to the model, the prototype requires data from a monitoring tool to know about SLO violations that occur. Once it gains knowledge of a new SLO violation, the prototype calculates the impact and creates an cross-component issue at Gropius [SBB20]. Besides that, the prototype is required to test whether the modelling language is, in fact, utilizable to calculate impacts or not.

To evaluate and validate this thesis, we applied a Goal Question Metric (GQM) based approach for which we performed an experiment and an expert survey.

This thesis has the following three contributions.

- A modelling language to model the saga pattern and to connect it to existing models of the architecture and the business process.
- A notification that informs about an SLO violations impact on the business process.

[1] https://github.com/stiesssh/ma-backend

- A prototypical tool that computes the impact based on the provided models and runtime data.

1.1 Thesis Structure

This thesis is structured as follows:

Chapter 2 – Foundations and Related Work: This chapter provides this thesis' foundations and surveys related works. The foundations cover microservices, business processes, SLOs and modelling basics. The related work focuses on analysis related to SLO violations on the one hand and on connections between different kinds of models on the other hand.

Chapter 3 – Design: This chapter describes the requirements of this thesis' concepts and the design of the modelling language.

Chapter 4 – Implementation: This chapter describes the implementation and architecture of the modelling language and the prototype.

Chapter 5 – Evaluation: This chapter covers the evaluation. This thesis' concept is evaluated by applying a GQM based approach with an experiment and an experts survey.

Chapter 6 – Conclusion: This chapter summarizes this thesis, discusses its results and elaborates on future work.

2 Foundations and Related Work

This chapter covers foundations and related work. Section 2.1 introduces the foundations of this thesis, and Section 2.2 describes other works related to the topic of this thesis.

2.1 Foundations

This section introduces the foundations required to understand this thesis. Section 2.1.1 provides insight into microservices, Section 2.1.2 looks into the history of the saga pattern and its incarnation as a microservice pattern, Section 2.1.3 and Section 2.1.4 provides information regarding modelling in general and about the Eclipse Modeling Framework in particular. Section 2.1.5 and Section 2.1.6 provide an overview of business processes and architectures, with a focus on modelling and existing meta-models. They look into Business Process Modeling Language (BPMN), Unified Modeling Language (UML), and the meta-model of Gropius. Section 2.1.7 provides a brief introduction to SLO rules.

2.1.1 Microservices

The microservice architecture style focuses on designing a system as a set of independent services [FOW17; New15]. It is related to the design-approach of service-oriented architecture [New15]. In the microservice architecture style, each service is a small unit that provides a single functionality. The services boundaries align with the boundaries of the realised business functionality. Monoliths tend to be split horizontally into a front-end for the user, a back-end for the domain logic, and a database to store the data. In contrast to that, a microservice architecture is split more vertically. Each service realises one aspect of the domain logic from Application Programming Interface (API) to logic to database. Splitting systems likes this allows for a lot of heterogeneity. Each service may be implemented by a different team, in a different language, using a different technology, et cetera. The only requirement is that the services API should be as technology agnostic as possible, such that it does not enforce its technology on those services who want to access it [FOW17; New15].

All services are supposed to be deployed independently. If a service's implementation changes and the service is subsequently rebuilt and redeployed, then all other services should remain unaffected. This is a benefit, as it allows the developers to redeploy only the changed part of the system. It also implies that services are replaceable as long as they provide the same API. As deploying is a frequent activity in microservice architectures, it is useful to automate it. Another benefit of all that independence is that each service can be scaled independently. However, independent services must communicate via the network, usually either with HTTP calls or with messaging, which makes the communication more complex [FOW17; New15]. It is also noteworthy that each service must manage its data on its own. Otherwise, the benefits of independence come to nought.

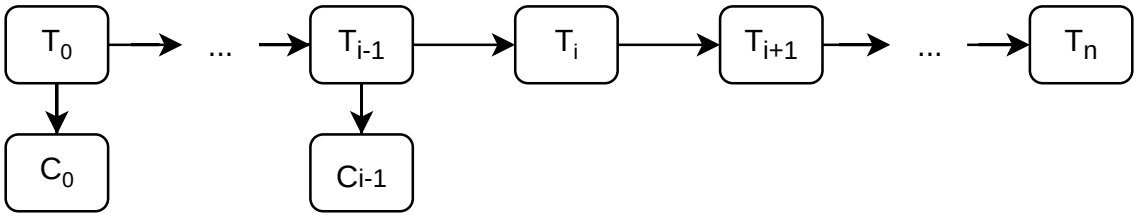


Figure 2.1: A saga consisting of $n + 1$ local steps.

2.1.2 Saga Pattern

The saga pattern as described by Richardson [Ric18] is a microservice pattern. Richardson's sagas are similar to the sagas described by Garcia-Molina and Salem [GS87] but within a different context. In the realm of database management systems, Garcia-Molina and Salem were faced with long-lived transactions locking the database for a long time and delaying all successive transactions. They suggested splitting the long-lived transactions into multiple steps, where each step is a shorter transaction that can commit independently and be interleaved with other short transactions such that the other transactions need not wait for the completion of the long-lived transaction. They called that sequence of shorter transactions a saga. A saga is still one transaction. Thus either all of its steps or none of them must commit. If any of a saga's steps fails, the entire saga, including all other steps that have already been committed must do a rollback.

In contrast to Garcia-Molina and Salem's scenario, there is (ideally) no central database with long-lived transactions in a microservice architecture. Instead each microservice manages its own database (c.f. *database per service* pattern [Ric18]). For example, a webshop might have an *order* service that has a database to store orders and an *inventory* service that has a database to store information about products. When a customer places an order, both databases must update. If one update fails, the other must roll back - a distributed transaction appeared. One way to implement distributed transactions is the two-phase commit protocol (2PC) [ML83]. The 2PC locks all transaction participants until every last of them is committed or any of them failed. In the latter case, those participants that committed already must do a rollback. No matter the outcome, a problem with the 2PC is that all participants are locked throughout the entire distributed transaction, which decreases the Quality of Service [Men02]. This problem is the same that Garcia-Molina and Salem faced concerning the long-lived transactions, and the saga pattern for microservices solves it similarly.

Richardson [Ric18] describes the microservice saga pattern as follows. The services' local transactions are the steps of the saga. As an example, the saga in Figure 2.1 consists of n local transactions. One of them, in this case, T_i is the pivot transaction, which means it is the last step that may fail. If the pivot transaction or any of its predecessors fail, the saga rolls back. Otherwise, it runs to completion. All steps before T_i need a compensation, and all steps after T_i must be retrieable. T_i itself needs neither [Ric18]. As an example, assume a distributed transaction across three services. The first service updates a record, the second service deletes one, and the third service checks a condition. The saga steps may be ordered $T_{update}, T_{check}, T_{delete}$ with T_{check} as the pivot transaction. Subsequently, there must be a compensation to roll back T_{update} at the second service. T_{delete} may be placed after the pivot transaction because it can be retried and will succeed eventually.

Figure 2.2: A simple business process in BPMN.

The saga pattern [Ric18] comes in two flavours, either as *orchestration* with an orchestrator that coordinates all saga steps in a centralized fashion or as *choreography* without any central unit. In both cases, the saga participants communicate over the network. The communication medium must provide exactly-once-delivery to ensure the saga's correctness. Alternatively, the communication medium must provide at-least-once-delivery, and the receivers must be idempotent. Furthermore, a saga requires atomicity with regard to the local transaction and the message published to the other participants. The latter can be ensured with help of the *transactional outboxing* pattern [Ric18] or the *event sourcing* pattern [Ric18]. Transactional outboxing puts messages into a database before publishing them, such that the message publication and the actual transaction may become one transaction, and thereby atomic [Ric18]. Event sourcing persists the state of domain objects as chains of events to an event store. The event store takes over the role of the message broker, thereby making the persisting of the state and the publication of the message atomic [Ric18].

The saga pattern is either defined at the business process level, as an example, as a transactional subprocess (c.f. Section 2.1.5) or the developers implement it at a lower level without the knowledge of the business process people. As an example for the latter case, assume that the developers are faced with a business process as depicted in Figure 2.2. Everything is a task, but the developers do not have a service at hand that provides the required functionality in an atomic fashion. Thus, they unite different services with the saga pattern, such that from the perspective of the business process it looks like one task.

Frameworks

There are frameworks for the saga pattern. Štefanko et al. compared four of them, namely Axon[1], Eventuate Event Sourcing (ES)[2], Eventuate Tram[3], and MicroProfile Long Running Actions (LRA)[4] with regard to performance [ŠCR19]. They designed a microservice architecture with a saga and implemented it using the different frameworks. Then, they tested the applications with different scenarios and measured the processing delay, the total time, and the number of completed requests. They concluded that LRA and Eventuate Tram perform better than Axon and ES. Also, Axon and ES require manual tracking of the saga instances, whereas LRA and Eventuate Tram do not [ŠCR19].

Sailer et al. compared multiple frameworks related to microservices, among theses Eventuate and Axon, with regard to the well-definedness of the interfaces, the independence of the services and other characteristics they thought to be relevant to microservices. According to Sailer et al. Eventuate

[1] https://axoniq.io
[2] https://eventuate.io
[3] https://eventuate.io/abouteventuatetram.html
[4] https://github.com/eclipse/microprofile-lra

as well as Axon focus on a subset of the characteristics and subsequently neglect others [SLW21]. As Eventuate is based on Java Spring it may be combined with Java Spring to make up for its deficiencies.

2.1.3 Modelling Languages

A modelling language is a language to describe a model. A modelling language also needs a meta-model that defines the abstract syntax and the static semantic of the models. Both are independent of a model's representation. Their job is to clarify which models are instances of a meta-model and which are not. The abstract syntax defines the elements and relations of a model. The static semantics are other rules they must comply with. There must also be a concrete syntax that describes how to visualise a model's elements and relations. There might be more than one concrete syntax for one and the abstract syntax [SVC06].

2.1.4 Eclipse Modeling Framework

The Eclipse Modelling Framework (EMF) [SBPM11] is a modelling framework that is part of Eclipse. At its center is Ecore, which is the meta-model for the EMF. Ecore itself is a EMF model and thereby its own meta-model. The EMF provides facilities to define, edit, and serialise models and to generate source code from the models. The generated source code can be edited just like handwritten source code. In addition to source code for the modelled classes, EMF can also generate entire viewers and editors to view and edit a model's instances [SBPM11]. The generated editors are plugins to the Eclipse platform.

2.1.5 Processes Models

According to Weske, a business process is a „set of activities that [...] jointly realise a business goal“ [Wes19]. The Business Process Modeling Language (BPMN) [OMG13] is a language to model business processes. It provides a detailed meta-model that captures many aspects of processes and a graphical notation for the design and visualisation of processes.

Figure 2.3 shows a small excerpt of the BPMN2 meta-model. There is the class `Process` which represents a business process as described above. A process is a `FlowElementContainer` and as such contains flow elements. A `FlowElements` is either a `FlowNode` or a `SequenceFlow`. A sequence flow shows the order of flow nodes. For this thesis, they are of no further interest. Flow nodes further differentiate into the classes `Activity`, `Event` and `Gateway`. Events influence a process with their occurrence, and gateways steer the control flow. For the saga, only the activities are of interest, because that is where actual work is done. A `Task` is a piece of work that one cannot split further. It is atomic. While there are even more types of activities, but this thesis focuses on tasks. There are also elements such as *pools* and *lanes* to organise the business process and data objects to model the data flow.

Figure 2.4 shows a process in the BPMN. It is a process to order some items from a store. The circles with thin outlines are the start events, where the process execution begins, and the circles with thick outlines are the end events, where the execution might end. The rounded rectangles are activities.

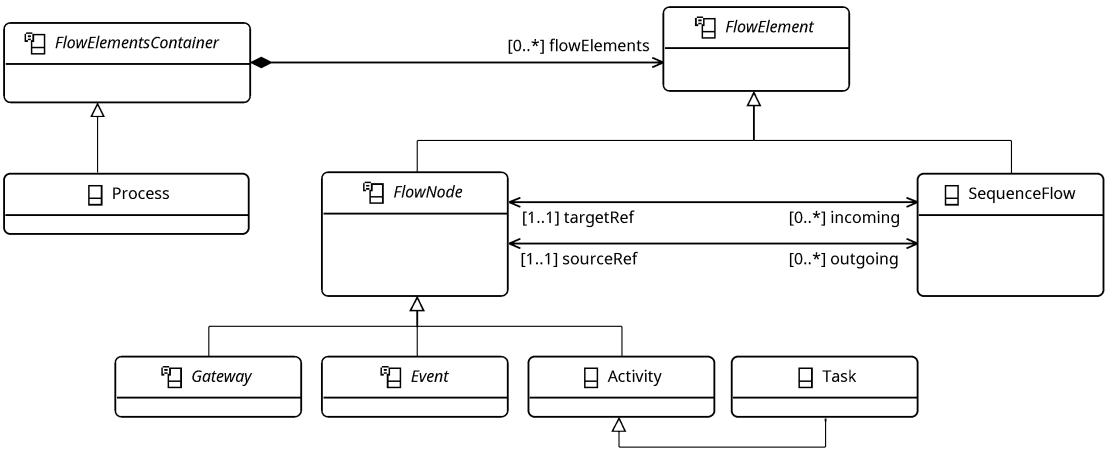


Figure 2.3: Excerpt from the BPMN's meta-model [OMG13] modelled with the EMF.

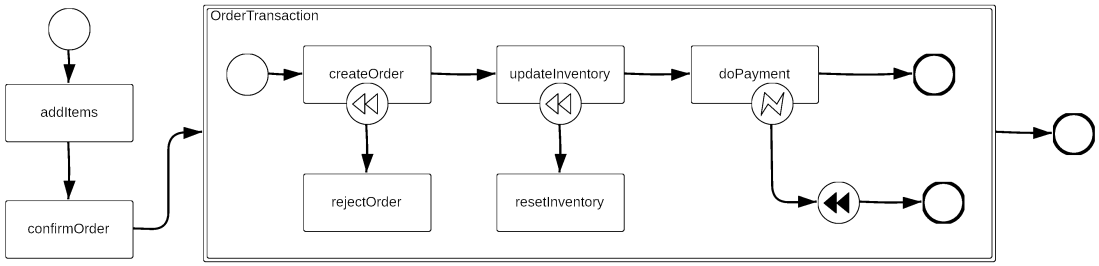


Figure 2.4: A business process with a transaction subprocess.

This example contains normal tasks (*addItems*, *confirmOrder*, *createOrder*, *updateInventory* and *doPayment*), compensations (and) and a transaction subprocess (*OrderTransaction*). Compensations are activities that undo the work of other activities, and the transaction subprocess contains a transaction. Tiny backward arrows mark the compensations as such. Dotted lines and boundary events associate them with their activities. Intermediate events may trigger the compensation. The default handling of compensations is to execute the compensation activities in reverse order of their activities, in case of Figure 2.4 that would be first *resetInventory* and then *rejectOrder*.

The transaction subprocess is denoted by the doubled outline. The outline of a normal subprocess is a single line only. Cancelling the transaction subprocess automatically triggers the compensations.

There is an Ecore model for the BPMN[5]. It is part of the Model Development Tools[6] project maintained by the Eclipse Foundation and available under the Eclipse Public License. The Eclipse BPMN2 Modeller[7] [ecl13] which provides the means to model Business Processes compliant to the BPMN standard uses that meta-model.

[5] https://git.eclipse.org/c/bpmn2/org.eclipse.bpmn2.git/
[6] https://www.eclipse.org/modeling/mdt/
[7] https://www.eclipse.org/bpmn2-modeler/

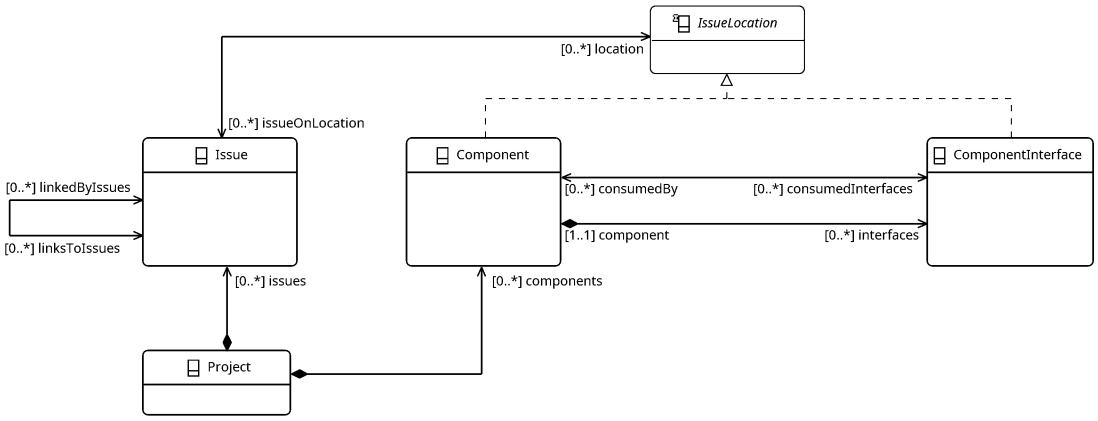


Figure 2.5: Excerpt from Gropius' meta-model [SBB21] modelled with the EMF.

2.1.6 Architecture Models

This section covers architectural description languages. The Unified Modeling Language (UML) [OMG17] is well-known modelling language that may be used to model architectures. It boils down to interfaces and components, where the latter is an encapsulated part of the architecture, and the former is a contract that a component may realise or require. The details, however, get way more complicated.

On the other hand, there is Gropius which is a tool for integrated management of cross-component issues [SBB20; SBB21]. The advantage of Gropius' over UML component diagrams is that its meta-model is very lightweight and far less convoluted. Gropius describes architectures in terms of components and interfaces. It uses the term *ComponentInterface* for interfaces, but for the sake of brevity, this section sticks to the term interface. Figure 2.5 depicts an excerpt from Gropius's meta-model. A component provides and consumes an arbitrary number of interfaces. Each interface is provided by one and consumed by arbitrarily many components. There is no notion of unconnected required interfaces. However, this is not a hindrance to this thesis. It only cares about connected interfaces as unconnected interfaces cannot propagate anything. Thus, Gropius' modelling language is a candidate to be used for architecture description by this thesis. As Gropius' focus is on the management of cross-component issues the meta-model covers more than just components and interfaces. Among the additional things are issues. Issues are attached to zero to many locations, that may be components or interfaces. Issues may link to other issues. For the modelling language of this thesis, issues are not relevant, as it only cares about the model elements concerned with parts of the architecture. However, for the prototype implemented for this thesis they are relevant, as the prototype uses cross-component issue in Gropius to present the notifications to the users.

Gropius' graphical modelling language is akin to UML component diagrams. An example for a simple UML component diagram is depicted in Figure 2.6. The example has four components (*UI*, *UIBackend*, *Inventory* and *Cart*), three required and three provided interfaces. UML depicts components with rectangles with a small component symbol on the right, the provided interfaces with circles, and the required interfaces with open half circles. In the example, *UI* requires an interface provided by *UIBackend*, and *UIBackend* requires one interface that is provided by *Inventory* and another that is provided by *Cart*.

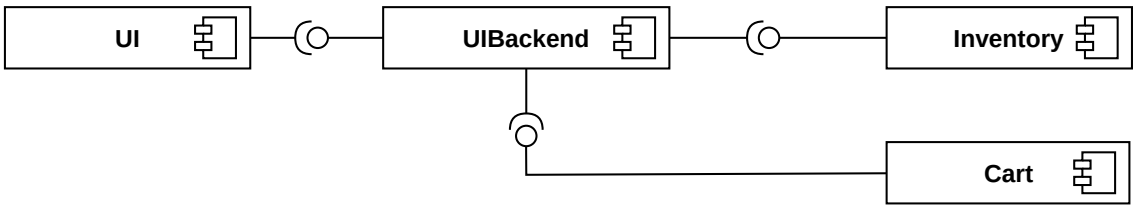


Figure 2.6: A simple UML component diagram.

2.1.7 Service Level Objectives

A Service Level Objective (SLO) [ACD+07] describes the promised quality with regard to a subset of a services' attributes, for example, its availability or response time. SLOs are an important part of a Service Level Agreement (SLA). One service, the provider, promises to provide its service according to the SLO. Another service, the consumer, operates on the assumption that the contracted service complies with the SLO. In addition to the SLOs, the SLA cover contract information and compensations, mostly of monetary nature, that the consumer receives if the provider does not comply with its SLO. In addition to the attribute under observation and the promised quality, an SLO also needs quality indicators. After defining an SLO, someone must monitor the quality indicators, such that the consumer and provider know about violations. Tools such as SoLOMON[8] provide the means to do that. SoLOMON uses monitoring tools to collect metrics from the services and compares the collected values to the defined SLO rules. If a rule is violated it creates a cross-component issue in Gropius.

2.2 Related work

This section discusses other works related to the topics of this thesis and points out the differences. This thesis' topics encompass SLO violations and the analysis of their impact, as wells as models for architectures, business processes, patterns, and transactions and how to connect such different models. Works related to the SLO topic are discussed in Section 2.2.2 and works related to the modeling topics are discussed in Section 2.2.3 to 2.2.5. Before that, Section 2.2.1 describes the literature research methodology.

2.2.1 Literature Research Methodology

The process of collecting related literature began with formulating a set of survey questions, each with a set of keywords. We used google scholar[9] to searched for the keywords. Due to subpar results on the first try, the keywords were refined and searched for once more. We added papers to the literature collection based on the title first and the abstract second. If neither matched the survey question, we discarded the paper. We prioritised papers based on the order in which google scholar provided them after ticking *sort by relevance*. We stopped adding papers after reaching a

[8] https://github.com/ccims/solomon
[9] https://scholar.google.com/

certain level of saturation as any additional paper would be similar to an already selected one. In especially well-fitting literature, we performed backwards and forward snowballing, which yielded more papers. We evaluated these again by title and abstract. We added some additional literature because of personal recommendations.

The literature research has three survey questions:

(SQ1) How do microservice deal with SLO violations and service failures?

(SQ2) What options to connect process models with architecture models have already been explored?

(SQ3) Does a meta-model for the saga pattern exist?

Question **(SQ1)** captures the overall topic of this thesis. To answer this question we used the terms `microservice`, `service oriented architecture`, `service level agreement`, `service level objective`, `quality of service`, `slo violation`, `business process`, `business logic` and `saga pattern`, as well as their various abbreviations, such as `SLA`, `SLO`, `QoS` and so on. We included the term `service oriented architecture`, as the design approach of service-oriented architectures is similar to the microservice architecture style.

Question **(SQ2)** focuses on the connection of models. At first, we combined the terms `bpmn` and `uml`, and as one might expect, these terms are way too general to yield any kind of useful information. On the second try, we combined terms that supposedly capture the business process part, such as `business process model`, `process model` and `bpmn` with terms that supposedly capture the architecture part, such as `architecture model`, `system model`, `UML component diagram`. We also added the terms `connection` and `bridge` and synonyms thereof in an attempt to capture the sentiment of bringing two different types of models together. We omitted the terms `saga`, `saga pattern`, and `transaction` because the third survey question covers them. The number of results was still in the thousands. Finally, we flipped through the first few pages, though most papers did not make it past the title criteria. We discovered most papers for this survey question via snowballing from the few papers that matched the survey question.

Question **(SQ3)** focuses on how to model a saga. We used the terms `pattern`, `design pattern`, `modelling language`, `saga`, `transaction`, `long lived transaction`, `distributed transaction` with and without the term `microservice` for this question. The terms cover patterns as well as transactions because the saga microservice pattern is a pattern but also originates from the saga as it was first identified for long-lived transactions by Garcia-Molina and Salem [GS87]. However, the transaction aspect yielded no helpful results.

2.2.2 SLO Violations, Root Cause Analysis and Impact Analysis

This section is about SLO violations in microservice architectures and service-oriented architectures. An SLO violation yields either some analysis for the root cause or the impact, which is covered in the first part of this section, or it yields the dynamic selection of another, less faulty service, which is covered in the latter part of this section.

The publications concerned with root cause analysis focus on composite services. They want to find out which of a services dependency caused an SLO violation. They drill down into the connections between the services to find the cause. Huang et al. focus on root cause analysis in a multi-domain

context, and Wang et al. construct an impact graph and do a random walk across that graph to identify the service that causes an anomaly. However, as they do *root cause* analysis neither considers any impacts on higher levels such as the overall business process [HZWC05; WXM+18].

The opposite direction to root cause analysis is impact analysis. The *MICROLYZE* framework [KUSM18], covered in more detail in Section 2.2.5, touches that topic as a future work. Authors such as Mohamed and Zulkernine and Hanemann et al. wrote about impact analysis, too.

Hanemann et al. want to reduce the cost of recovery by basing the decision on which action to take on the costs of the SLA violation. To get the costs of a SLA violation they compute the impact a resource failure has. However, they limit resources to things, such as hard drives, and only compute the direct impact of a failure. For them, the propagation beyond the first impact is out of scope [HSS05].

Mohamed and Zulkernine assess system reliability and want to include failure propagation into that assessment. They view the architecture as a set of architectural service routes (ASR). Each route is a sequence of components connected because one component requires an interface provided by another one. They use those routes to calculate a failure's probability to propagate through the system and finally reach the user. They do not specifically consider failures due to SLO violations but failures in general [MZ08].

The publications concerned with dynamic service selection do not care about the impact of an SLO violation on the business process at all, but only about how to find another service to replace the faulty one [VAES12; WBP09]. They assume that the stock of services befitting their needs is large enough such that they always find a replacement. In some cases, however, dynamic service selection is not an option. Some services are too domain-specific such that no alternative services exist or other services that provide a fitting functionality cannot be used due to other restrictions such as compliance issues [DD08].

2.2.3 Connecting different Models

Half of this section's scrutinized publications are old. They date back until 2003, which makes them older than BPMN [OMG07]. That might be the reason why the oldest publications do not consider BPMN.

This thesis wants to connect a process model, which is about behaviour, with an architecture model, which is about structure. Notably, this thesis cares about the behaviour of the entire architecture and thus wants to connect the architecture to some process that describes the behaviour that results from the interplay of the architecture's components. It does not care about behavioural descriptions for each component. As such the latter is not included in this section.

The first two publications originate from the field of requirements engineering. The authors describe the overall behaviour of a system with a business process and try to connect other things such as use cases, classes, and components, to that process. Odeh and Kamm emphasise the importance of UML use case diagrams as the starting point of projects, especially those with object-oriented designs, and propose a transformation from Role Activity Diagram (RAD) to UML use cases. RADs describe business processes. Odeh and Kamm executed a case study in which they modelled a student's enrolment process as a RAD, formulated rules on how to transform the RAD into use cases, and derived the use cases by applying the rules. They concluded that it is indeed possible

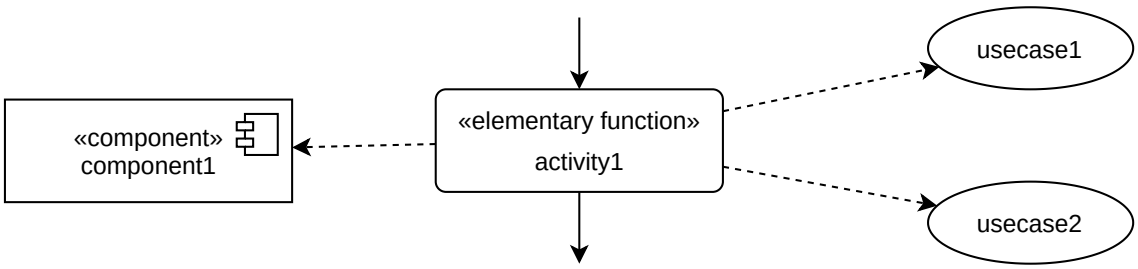


Figure 2.7: Excerpt of an UML activity diagram, whose elements are connected to elements from other types of UML diagrams via dependencies [KL].

to bridge the gap between business process modelling and system modelling even though the task bears some difficulties and ambiguities [OK03]. About ten years later, Cruz et al. try to derive UML use case diagrams from BPMN process models. They argue that derivation improves the conformance of the use case diagrams to the business process. If possible Cruz et al. transform the elements from the process model to elements of the use case diagram. If no matching kind of element exists for the use case diagrams, they represent the process' element as text [CMS14].

Odeh and Kamm and Cruz et al. do indeed connect to different kinds of models. However, they do not leverage knowledge from both models and their connection but derive one from the other. This deviates a lot from this thesis' intention.

Aversano et al. delve deeper into a system's conformance with a business process and have less focus on requirements engineering. Instead, they want good conformance to improve performance. They use the term *alignment* for the connection between business entities and software entities, where one business entity might be realised by multiple or zero software entities. They consider activities, actors, and artefacts as business entities and everything, from package to class to function to variables and so on as a software entity. As such, they provide a fine-grained link from the business level to the architecture level. Once they have connected process and system models, they evaluate the conformance. They use UML activity diagrams for the business process instead of a modelling language specifically designed for *business* processes because their focus is not on executing a business process but on evaluation. The reason that UML activity diagrams come in handier for that sake [AGT16].

Korherr and List shift the focus back to requirement engineering. Like Odeh and Kamm, they view the business process as the starting point for a system's design. But instead of deriving use cases from the process model, they take existing use cases and other artefacts that already exist, such as components, and model how these artefacts contribute to the process. They do so by adding the components and use cases as dependencies to a UML activity diagram that describes the process. As depicted in Figure 2.7 dashed arrows visualise the dependencies. The activities are stereotyped as *«elementary function»* because Korherr and List actually start off with a process model in EPC notation which they transform into a stereotyped UML activity diagram [KL].

Elvesæter et al. want to use BPMN for business process modelling within the *Service oriented architecture Modeling Language* (SoaML), which models service-oriented architectures and consists of a set of UML profiles that extend UML activity diagrams [EPJH10]. They propose a set of rules on how to map the process elements of the BPMN to the architectural elements of the SoaML [EPJH10].

The conclusion to this section is that others have already connected different types of models and done so at different levels of detail. Nothing truly fits the needs of this thesis. However, the mere existence of the aforementioned publications affirms that it is possible and a topic of interest to connect different kinds of models.

2.2.4 Modelling Design Patterns

Mapelsden et al. and Dong and Yang attempt to display design patterns in UML diagrams. Both of them focus on the design patterns by Gamma et al. The literature research did not yield any results specifically for the Microservice Patterns [Ric18], presumably because these patterns, as described by Richardson, are from 2019, which makes them young.

Dong and Yang uses UML Profiles to enhance class and state diagrams such that the usage of patterns in these diagrams is easily discernible [DY03]. They use stereotypes to mark a class as part of a pattern and tagged values to tell the reader more details about the pattern. Dong and Yang work with class diagrams, whereas this thesis would like to work with component diagrams. Also, the visuals are lacking, especially the tagged value notation.

Mapelsden et al. create a new language, the *Design Pattern Modelling Language*, to model patterns. The language consists of two parts. A pattern specification that models a pattern in a general way, and a pattern instantiation that represents the implemented instances of a pattern. They use different shapes such as hexagons and diamonds as well as colours in their concrete syntax, which is far more eye-catching than the textual cues provided by Dong and Yang. Despite the better visuals, Mapelsden et al. do not befit this thesis' needs.

Another approach is UML collaborations. They are for describing how elements of models work together to achieve a common goal [OMG17]. The specification explicitly states that one may use them to model design patterns [OMG17]. The collaboration itself would be used to model the pattern from a static perspective, and *collaborationUses* describe how it maps to actual instances [OMG17].

A takeaway from all publications is that it is important to differentiate between a pattern and its instances and that it is important to differentiate between multiple instances of the same pattern.

2.2.5 *MICROLYZE* and Others

Kleehaus et al. [KUSM18] developed the *MICROLYZE* framework to recover microservice architectures. They state that the architecture of microservice-based systems changes frequently which makes it difficult to keep the architecture documentation up to date. They propose the *MICROLYZE* framework to automate that task. The framework collects static data from the existing documentation and runtime data from monitoring devices and the underlying infrastructure and uses that data to construct an architecture description. Once the framework notices a change in the architecture, it updates the description.

Similar approached were already undertaken by Haitzer and Zdun [HZ12] and Granchelli et al. [GCD+17]. Both are, just like Kleehaus et al., motivated by the ongoing changes in microservice architectures, and combine static and dynamic data to construct architecture descriptions for a running system. However, Haitzer and Zdun as well as Granchelli et al., restrict themselves to the service level, whereas Kleehaus et al. also considers the business level. In that, they are closer to this thesis topic than any other publication. The *MICROLYZE* framework successfully extracts almost all required data. However, it still needs a human's help to create the link to real business activities.

There are two major differences between the *MICROLYZE* framework and this thesis' concept. Firstly, *MICROLYZE* considers the architecture as something that changes whereas this thesis disregards this aspect of architectures. This thesis expects an architecture description as input but does not care about its origin. It could even be an architecture description recovered by the *MICROLYZE* framework. Secondly, the *MICROLYZE* framework does not consider patterns. According to Kleehaus et al., they display patterns in their model, but they stay very superficial in that regard [KUSM18]. Also, SLO violations and their impact on the business level are out of scope for them. Kleehaus et al. mention a failure-impact visualisation as future work, which is very close to this thesis topic but still different, as this thesis includes the influence of patterns onto the failure propagation [KUSM18].

3 Design

This chapter describes the design of the modelling language and the underlying decisions that led to its current incarnation. It starts off with the requirements engineering in Section 3.1, followed by a general description of this thesis' concept in Section 3.2, the description of the modelling language's abstract syntax in Section 3.3 and lastly the description of its concrete syntax in Section 3.4.

3.1 Requirements Engineering

This section describes the requirements engineering process that this thesis applied. After that is lists the gathered requirements, organized into functional requirements in Section 3.1.1 and more detailed use cases Section 3.1.2.

Requirements describe a system's intended behaviour [Som16]. They should be gathered at an early stage of a system's development, and they should be consulted throughout the development process to ensure that the system still meets them. Figure 3.1 shows this thesis' requirements engineering process. It consists of six of the steps described by Sommerville. The first step is the requirements elicitation. This step consisted of brainstorming in solitude and cooperation with this thesis' supervisors. At the end of this step, we established this thesis objective and a set of requirements to achieve this objective. The second step is the requirements analysis. This step consisted of research into existing literature and systems. As a result, we figured out what solution approaches already exist and what is or is not achievable. We updated the elicited requirements and prioritized them by their contribution to this thesis' objective. Requirements that are crucial to the achievement of this thesis' objective are of the highest priority. Requirements that are nice-to-haves

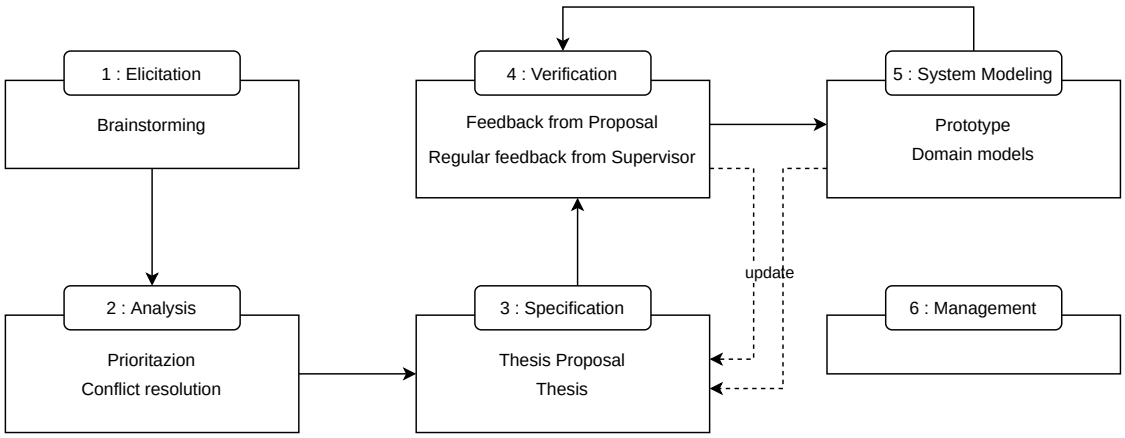


Figure 3.1: This thesis' requirements engineering process.

are of the lowest priority. The third step is the requirements specification, which is where we wrote them down in an organized fashion. This step resulted in a thesis proposal document, which we presented to an audience. The audience feedback became part of the fourth step of the requirements engineering process, which is the requirements verification. Regular meetings and feedback with this thesis' supervisors also contribute to the verification step. Later on, we also placed the requirements in this document. After the first verification of the requirements, we modelled the system as part of the fifth step of the process. We created an initial prototype and various models. They were regularly checked on by the supervisors to verify that they still match the requirements and the requirements still match the supervisor's vision and that we did not misunderstand them. If step four or five caused any changes to the requirements, we updated the specification accordingly.

Another step is the requirements management. We did not perform additional steps to manage the requirements. However, version control systems manage the models from step five and the documents from step four. Therefore, it is still possible to go back in time and to trace how the requirements evolved.

3.1.1 Requirements

This section lists the gathered requirements as user stories. It lists the requirements split into 3 groups. The groups are requirements regarding the concept's output, those regarding its input, and those from the tools perspective.

The following requirements regard the information that the concept eventually provides to its users. They mostly record what the users want to know. Here, the user is the person who observes the system at runtime.

- As a process owner, I want to know about SLO violations that impact the business process such that I can react with strategic actions to minimize the monetary loss.
- As a developer, I want to know about SLO violations that impact the business process such that I can fix the root cause.
- As a user, I want to know which SLO rule was violated. I want to be able to recognize the SLO rule as an example by name or id, and I want to know details about the violation, such as its time of occurrence and its gravity.
- As a user, I want to know how or why (i.e. along which path) an SLO violation propagated from the location it occurred at up to the business process. Such knowledge must encompass the interfaces and components, as well as the sagas and the saga steps, that the violation passed before it reached a task in the business process.
- As a user, I want to know whether and how an impact on the business process is related to the implemented microservice patterns.

The following requirements regard how the user wants to receive the information.

- As a user, I want a cross-component issue in Gropius to inform me about an SLO violations impact.

- As a user, I want a cross-component issue in Gropius that aggregates impacts. If the same SLO rule gets violated multiple times and the impacts are the same for these violations, then I do not want multiple open issues with redundant information but only one open issue that tells me about multiple violations. If all related issues are already closed, I want a new open issue.
- As a user, I want to pick the raw information about the impact up at an API, such that I can process it myself.

The following requirements regard the input of the concept. They capture what information the user is willing to provide to this thesis' concept. They also capture what format this information has. Here, the user is the person who inputs the information that the concept requires.

- As a user, I want to reuse as many existing models as possible such that I do not have the effort of modelling the same thing twice.
- As a user, I want to reuse existing architecture models. As I use Gropius to model the architecture, I want to reuse an architecture model from Gropius.
- As a user, I want to reuse existing business process models. As I use the BPMN to model the business process, I want to reuse a business process model in the BPMN.
- As a user, I want to reuse existing SLO rules. As I use SoLOMON to specify the rules, I want to reuse SLO rules from SoLOMON.
- As a user, I want to reuse existing monitoring tools, such that I have neither the effort of setting up new monitoring tools nor the redundancy of employing two monitoring at the same time. As I use SoLOMON to managed the SLO rules, I already set up Prometheus for monitoring.
- As a user, I want the reused models to be in sync with their source.

The next requirements are about the saga pattern. The record how the user intends on using the saga pattern. These requirements have a great influence on the modeling language. They dictate how the language should be able to represent sagas.

- As a user, I want to use sagas as described by the saga microservice pattern [Ric18].
- As a user, I want to model multiple sagas because my business process is large and has multiple activities that must happen in a transactional context.
- As a developer, I want to implement the saga pattern even if there is no explicit transaction in the business process. I want to do this because there is an atomic activity but during implementation, I realized that I access multiple services to realize that activity. To present this combination of services as atomic I use the saga pattern.
- As a user, I want to model other microservice patterns because I implemented those, too.

The following are requirements from the tools perspective. The tool is referred to as *the tool* and encompasses all components required to realise this thesis' concept. These requirements capture the tool objective and the input it needs to achieve it.

- As the tool, I want to calculate the impacts of an SLO violation. I want to do this based on the models of architectures, business processes, SLO rules, and sagas.

- As the tool, I want to receive alerts about occurring SLO violations, such that I get triggered to start calculating impacts.
- As the tool, I want to be provided with a model that holds knowledge about the architecture, the sagas, the business process, and the SLO rules and also about the connection between saga, architecture, and process.

3.1.2 Use Cases

This section provides the use cases **(UC 1)** to **(UC 3)**. They describe a user's interaction with a tool that fulfills the requirements specified above. The user of the use cases has an online webshop. Their business goal is to sell tea. They modelled the shop's architecture with Gropius, and they have a model of the business process in BPMN. They use SoLOMON to manage the SLO rules and to monitor their shop.

(UC 1): Modelling the system

- The user opens the tool.
- The user imports an architecture model from Gropius.
- The user imports a business process in BPMN.
- The user imports SLO rules from SoLOMON.
- The user adds a saga with multiple steps to the model. They connect each step to its successor, to a task of the process, and an interface from the architecture.
- The user has done his deeds and waits.

(UC 2): Viewing an impact 1

- The user notices that they have not sold any tea in quite some time.
- The user checks Gropius for new issues and finds one about an impact on the business process that an SLO violation caused.
- The user is from management and offers discounts to its customers to make up for the inconvenience.
- The user is a developer. They check the issue's details to fix the violation, either by fixing the service or by using another one.

(UC 3): Viewing an impact 2

- The user is a developer tasked with the ungodly assignment to improve the system.
- The user looks at some cross-component issues at Gropius that report impacts on the business process.
- The user checks the impact paths to identify services that propagated the failure.

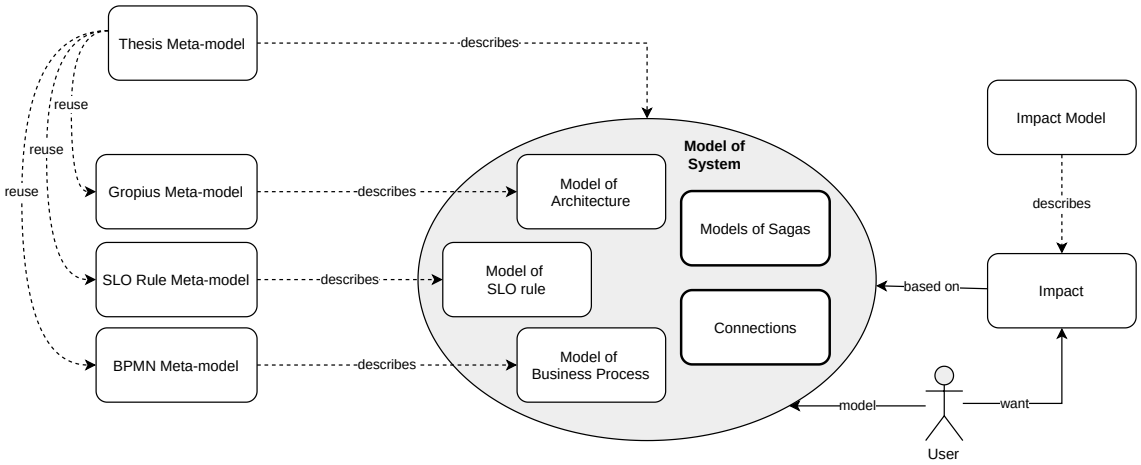


Figure 3.2: Concept of this thesis' meta-model.

3.2 Concept of the modelling language

The objective of this thesis is to trace the impacts of SLO violations across microservice architectures that implement the saga pattern. To achieve this, this thesis designs a meta-model that describes models that capture all information required to calculate the impacts.

Figure 3.2 illustrates this thesis' concept. As a final objective, the user wants to know about impacts. The impacts are calculated based on a model that covers the architecture, the business process, the SLO rules, the sagas, and the connections in between. These five parts form the 'required information' mentioned above, and they must all come together into one model. For lack of any better name, this will be referred to as *model of the system* or *system model*. This very model is the model for which this thesis designs a meta-model.

In the current modelling landscape, meta-models that describe models of architectures, SLO rules, and business processes already exist. An important part of this thesis' concept is to reuse these existing meta-models. There are no meta-models that cover sagas or the connection between elements of the different reused models. Thus, the sagas and the connections are contributed by this thesis. The sagas must be in the model because this thesis is about architectures with that pattern. The connections are important because they bridge the gap between the architecture and the business process. They thereby make the calculation of the impact on the business process possible. Without the connection, one can trace an impact within the process or the architecture, but one cannot cross from architecture to process.

This thesis' meta-model consists of an abstract syntax that describes the new elements, semantics that describe their meaning, and a concrete syntax such that a user can formulate models as dictated by the meta-model. There is also a meta-model for the impacts.

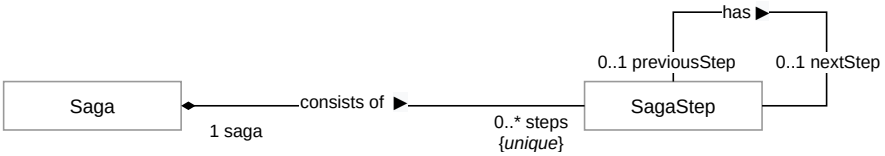


Figure 3.3: Abstract syntax for the saga.

3.3 Abstract Syntax

This section describes the abstract syntax for the system and the impact meta-model. The system model holds the static knowledge about sagas, architecture, business processes, the connections in between, and the SLO rules, and the impacts are about the SLO violations that happen at runtime, and where they impact the system.

This section starts with Section 3.3.1 which covers the saga itself, followed by Section 3.3.2 which recaps the meta-models of architecture and business process, and Section 3.3.3 which covers how to connect the saga, the architecture, and the business processes. After that, Section 3.3.4 covers the SLO violations and the impacts they cause. This section uses `monospace` for class names, attributes and references.

3.3.1 Sagas

As described in Section 2.1.2 a saga is a sequence of steps, where each step is either a local transaction or a compensation. One of the local transactions is the pivot transaction, which means it is the last transaction that might fail. All transactions before the pivot transaction need compensations, and all transactions after the pivot transaction must be retrieable until they succeed eventually.

Sagas are modelled as shown in Figure 3.3. The element `Saga` represents one type of saga, e.g. a *order tea* saga, or a *add new product* saga. The element `SagaStep` represents one of the local transactions that make up the saga. A saga consists of zero to many steps, and each step belongs to exactly one saga. Each step has up to one successor and up to one predecessor. The successor is the step that executes after, and the predecessor is the step that executes before the current step. Both are optional because the last step has no successor and the first step has no predecessor.

3.3.2 Architectures and Business Processes

This thesis' abstract syntax reuses the meta-model of Gropius [SBB21] for the architecture and the meta-model of the BPMN [OMG13] for the business process. This section recaps information about the relevant elements form those meta-models. More detailed descriptions are in Section 2.1.6 and Section 2.1.5.

The relevant elements form the Gropius meta-model are `Component` and `ComponentInterface`. The class `Component` represents a component of the architecture, and the class `ComponentInterface` represents the interfaces the components provide or consume.

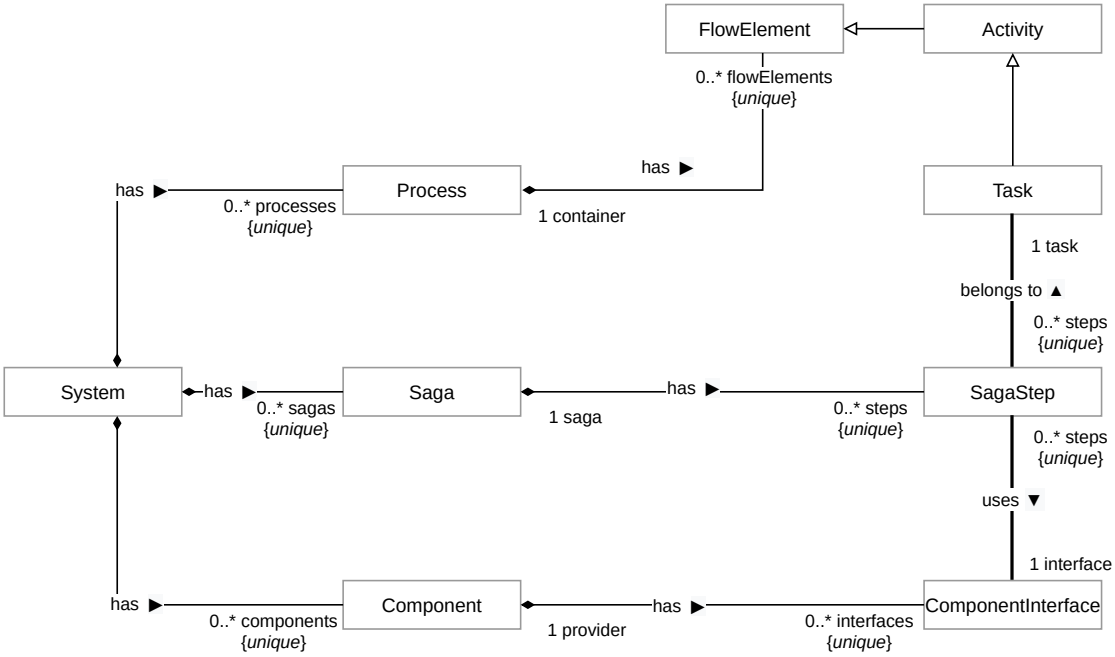


Figure 3.4: Abstract syntax of the system meta-model.

The relevant elements from the BPMN meta-model are `Process` and `Task`. The class `Process` represents a business process and the class `Task` represents a atomic piece of work. A process contains, among others, of tasks.

3.3.3 Connections

This section is about the additional connections that bridge the gap between architecture and business process and thereby make the calculation of the impact on the business process possible. Without these connections, it is only possible to trace an impact either withing the process or within the architecture, but it is impossible to cross from architecture to process. This section is not concerned with the relationships between elements imported from the same meta-model because these are already sufficiently defined by the imported meta-model itself.

As explained in Section 3.3.2, this thesis' meta-model uses the Gropius meta-model for the architecture and BPMN meta-model for the business process. It also needs a meta-model for sagas. For this purpose, it uses the meta-model described in Section 3.3.1.

Figure 3.4 shows the relevant elements from all three meta-models and the connections between them. The elements `Component` and `ComponentInterface` originate from the Gropius meta-model [SBB21]. The elements `Process`, `FlowElement`, `Activity` and `Task` originate from the BPMN and the elements `Saga` and `SagaStep` originate from the saga meta-model. All relations between elements of the same meta-model already exist in their respective meta-model. Only the connections between elements of different meta-models are new. The element `System` is new as well. It represents everything from the architecture down below up to the business processes on top. It has an arbitrary number of components, processes, and sagas each.

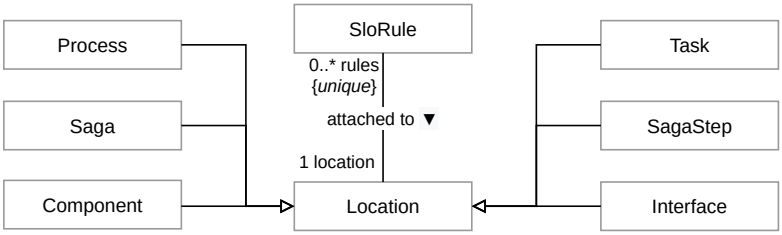


Figure 3.5: SLO rules may be defined for anything.

First of all, the connection must be between the steps, tasks, and interfaces because these have the highest level of detail. Relations between the components, processes, and sagas result in a loss of information. While knowing, as an example, which task was impacted by an SLO violation also carries information about which process is impacted (as each task belongs to one process only) the opposite direction does not work because each process contains multiple tasks. The same applies to components and their provided interfaces and to sagas and their steps. Only knowing which 'bigger' thing a violation impacts does not provide knowledge about which 'smaller' things it impacts. Concerning interfaces, this thesis simplifies all interfaces to CRUD interfaces. In the case of more complex interfaces, it is not enough to model which interface a step is realized by, but it would be necessary to model the distinct operations of the interface.

A saga step belongs to exactly one task, whereas a task may have an arbitrary number of steps. Tasks are atomic. Saga steps are local transactions. They are atomic, too. However, the atomicities are from different points of view. For the task, the point of view is the process. Within the process, a task is always atomic. That is what makes it a task. However, the point of view of the saga step considers the underlying architecture as well, which might be finer-grained than the tasks. As an example, there might be a *do payment* task. But the actual implementation might not be that simple. In fact, there might be different interfaces, such as *increaseBalance* and *decreaseBalance* and only in orchestration can they do the payment. To achieve atomicity at the process level the different interfaces are bundled into one saga. As a result, multiple saga steps partake to realise one task. A saga step that realises multiple tasks does not make any sense because the tasks are of coarser granularity.

A saga step uses exactly one interface whereas an interface may be used by an arbitrary number of steps. A step that uses more than one interface is not a local transaction anymore thus the restriction to one interface only. The interface, however, does not care about who uses it how often. A direct relation between tasks and interfaces is not part of the meta-model because it does not provide additional insight. All insight it might provide is already contained in the relations between interface, respectively task, and step.

Lastly, the meta-model must contain the SLO rules because they are the starting point of each impacts trace. Optimally an SLO rule can be attached anywhere, just as depicted in Figure 3.5. The element `SloRule` represents an SLO rule and the element `Location` may be anything to which an SLO rule may be attached. An SLO rule may attach to exactly one location. Each location has arbitrary many SLO rules attached. Like this, the impact calculation can start at the SLO rule, hop from there to any element of the system and then hop along the connections between the elements up to the business process.

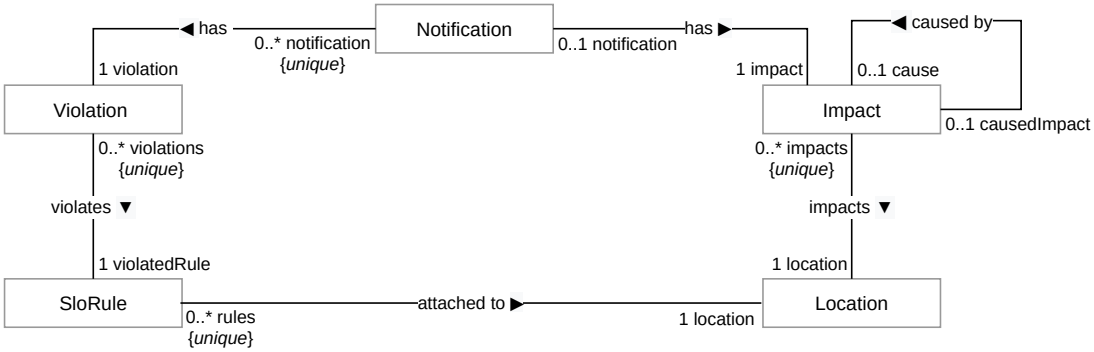


Figure 3.6: Abstract Syntax for the notification

3.3.4 Violations and Impacts

This section covers SLO violations and impacts. Whereas the system models are static, as they are created beforehand and do not change at runtime, new instances of impacts pop up during runtime whenever an SLO rule is violated.

Figure 3.6 shows the impact meta-model. The element `Impact` represent a change in behaviour, that may happen at any element of the system model (c.f. Figure 3.4). The element `Violation` represents the violation of an SLO rule. The meaning of the element `SloRule` is as described in the previous section. The element `Location` represents anything that might be affected by an impact. It might be any of the system model's elements, just as depicted in Figure 3.5. Each location may be impacted by arbitrarily many impacts or by none at all. An SLO rule may be violated arbitrarily often or not at all. An impact has exactly one `location` and up to one `cause`. The location is where the impact happens, and the cause is another impact that causes the current one. If an impact has no cause, it is the initial impact that was caused by the SLO violation. A violation violates exactly one SLO rule. The element `Notification` represents the information that reaches the user. It has exactly one impact and violation. The violation is the impact's root cause. If an impact belongs to a notification, this implies that it has not caused any other impacts and that it reached the business process. A violation may belong to arbitrarily many notifications because one violation might be the root cause of multiple impacts on different elements of a business process. In that case, each impact would get its own notification and the current violation would be the root cause of all of them.

With the meta-model as depicted in Figure 3.6, it is now possible to describe the chain of impacts that lead from an SLO violation anywhere in the system, to an impact on the business process.

3.4 Concrete Syntax

This section described the concrete syntax. It covers a concrete syntax for the system meta-model in Section 3.4.1 and a concrete syntax for the impacts in Section 3.4.2. The former is a graphical notation that uses elements from existing notations. The latter is a textual notation. The graphical notations was designed with Moody's "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering" [Moo09] in mind.

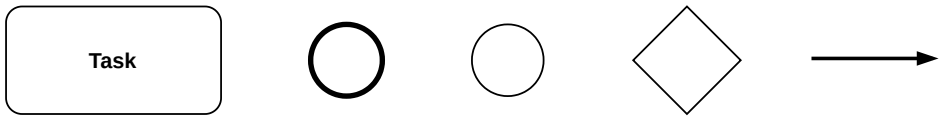


Figure 3.7: An excerpt of the BPMN. Form left to right: a task, an end event, a start even, a gateway and a control flow.

3.4.1 Saga

This section describes a concrete syntax for the system meta-model (c.f. Section 3.3.3). It is a graphical notation.

As stated by Moody it is important that a graphical notation's symbols are sufficiently distinct, such that a reader has fewer difficulties with telling different symbols apart. Moody lists eight visual variables: horizontal and vertical position, shape, brightness, size, orientation, texture, and colour. This concrete syntax relies on shape and position and omits colour coding at all costs.

This concrete syntax reuses notations from other models. For architectures and processes, a plethora of notations already exists. It is reasonable to reuse notations that are already in use to lessen a user's effort in understanding the concrete syntax of this thesis' meta-model. A well-known notation for business processes is the BPMN [OMG13], and a well-known one for architectural elements are UML component diagrams [OMG17]. Excerpts of the BPMN and of UML component diagrams are shown in Figure 3.7 and Figure 3.8. They only show notational elements relevant to this thesis. The BPMN depicts tasks as rectangles with rounded corners, start events as circles, end events as circles with a thicker outline, and control flow with arrows, diamonds, and border events. It employs swim lanes. These are rectangles with a small separate area on the left-hand side to depict a processes' affiliation. UML component diagrams provide multiple notations for components. One of them is depicted in Figure 3.8. This notation depicts a component as a rectangle with sharp corners, a tiny component icon on the right-hand side, and the stereotype «component» above the component's name. This black box view on components suffices for this thesis. None of the other notation for components will be included in the system meta-models concrete syntax because they are redundant (c.f. Symbol Redundancy [Moo09]). UML component diagrams depict interfaces as lollipops, where the actual lollipop is for provided interfaces, and the cavity is for the consumed interfaces. There are different ways to connect the lollipop to the cavity. This thesis reuses the symbolism that sticks the lollipop directly into the cavity.

Both notations use rectangles and solid lines. However, the BPMN's rectangles have rounded corners, whereas the rectangles of the UML component diagrams have sharp corners, which makes them sufficiently different. The solid lines pose no problem either because the elements they are connected to make it very clear, whether a line belongs to the processes or the architecture. Thus, the relevant parts of the BPMN and the UML component diagrams can be reused without any customizations.

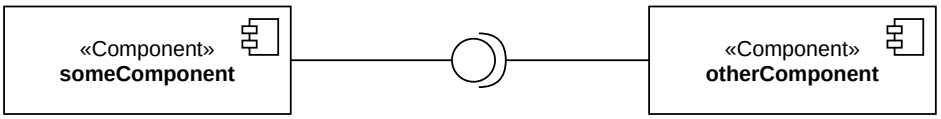


Figure 3.8: An excerpt of the notation for UML component diagrams. Form left to right: a component, a provided interface, which is already connected to a required interface, the required interface and another component.

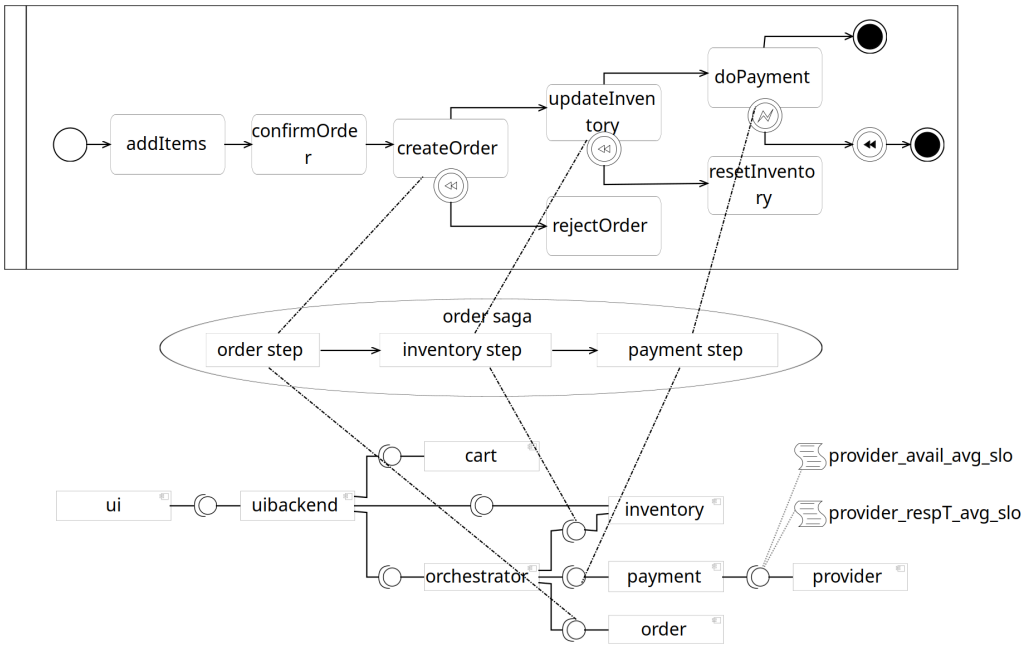


Figure 3.9: An instance of the system meta-model, notated in the concrete syntax described in this section.

The next challenge is to find symbols for those elements that are not yet present in other models, namely the saga, its steps, and the additional connections. As mentioned before, these symbols must be distinct from the other symbols of the notation.

Figure 3.9 shows the concrete syntax for a saga model. It displays the process in the upper third of the figure, the architecture in the lower third, and the saga in between. Sagas are depicted as an ellipse with solid outlines. Located inside the saga ellipse are the saga's name and steps. The steps are depicted as rectangles. A step's relation to its successor is depicted by a solid line. The line end that connects to the successor is decorated with an arrowhead. The opposite end is undecorated. UML collaborations [OMG17] inspired this notation. It is a reasonable choice to use an ellipse because circular shapes are not yet in use. Everything within the ellipse looks separated from its surroundings, therefore using sharp-cornered rectangles for the saga steps does not collide with the sharp-cornered rectangles of the components. In addition, the component rectangles have the tiny component icons, which the step rectangles do not have. Choosing a notation for the relations

between saga steps and interfaces respectively tasks poses a challenge. Most types of lines are already in use by either the BPMN, the UML or both of them such that reusing them for additional concepts might cause confusion about their semantics. Solid lines are not an option because BPMN already uses them for the control flow. Additionally, UML component diagrams use them for associations between components and interfaces. Before, using solid lines for the process and the architecture was acceptable because the context clarified what they represent. However, the relations between steps, interfaces, and tasks, cross the border between the three parts of the meta-model, such that there is no context to deviate the meaning from anymore. Dashed lines are not an option because BPMN uses them for message flow. Messages are not part of this concrete syntax, but users familiar with the BPMN might still get confused. Also, dashed lines are already in use for various kinds of associations in UML. Lastly, dotted lines are already in use for data association in BPMN. As all simple styles are already in use, and colour coding is not an option this thesis wants to rely on, the relations between saga steps and interfaces respectively tasks are depicted with lines that alternate dashes with dots. The lines connect a step rectangle to a task rectangle or an interface lollipop. They are undecorated.

3.4.2 Notification

This section describes a concrete syntax for the notification as introduced in Section 3.3.4. The notification consists of the impact path and the root causes, with the latter being SLO violations.

The impact path has a textual syntax in JSON, such that humans and machines alike can understand it. The JSON schema is depicted in Appendix B.

It has the top level elements `impactlocation`, `vioaltedrule` and `impactpath`. The `impactlocation` is the impacted business process, the `rootcause` is the violated SLO rule and the `impactpath` is the path from the location of the violated SLO rule up to the impacted task of business process. The syntax displays this impact path flattened into a list because in JSON that is easier to read than a tree structure.

Each impact is represented by the `id`, the `name` and the `type` of its location and by its cause. The representation includes the type in case the user does not know the ids and names in their system by heart. When knowing the type it is most likely easier to understand the path even when the user does not remember each detail about the system. The cause is only present if the impact on the current location has one. If so, `cause` links to the location from which the violation propagated to this location. Like this, all information about an impact's path are conveyed by its concrete syntax.

The root causes are represented as cross-component issues in Gropius. SoLOMON created these issues with relevant information about the SLO violation. This thesis uses the issues created by SoLOMON as they are, thereby reusing their concrete syntax for the violations.

The notification as a whole is represented as a cross-component issue in Gropius as well. The issue combines the concrete syntax of the impact path and root causes by placing the impact's JSON in its body and adding the issues of the root causes as linked issues.

4 Implementation

This chapter describes the implementation of this thesis' concept. It commences with an overview of the prototype's architecture in Section 4.1, followed by a description of the models' realizations as Ecore models in Section 4.2 and a description of the editor's implementation in Section 4.3. After that, it continues with a description of the back-end's implementation in Section 4.4 and finishes with the description of the cross-component issue in Gropius created by the back-end in Section 4.5.

4.1 Architecture Overview

This section describes the prototype's architecture. The prototype is implemented according to a subset of the requirements and design decisions gathered in Chapter 3.

Figure 4.1 shows a schematic overview of the prototype's architecture. The prototype depends on Gropius [SBB20] and on the SoLOMON tool[1]. Gropius provides the architecture and manages the issues, and SoLOMON provides the SLO rules and monitors the microservice. The prototype consists of a back-end and a front-end. The sources for both can be found on GitHub[2]. The front-end is implemented as an Eclipse plugin using EMF and runs locally on the user's machine. The back-end is a service. From here onwards this thesis refers to the front-end as *the editor*. In case this is ambiguous it uses the term *this thesis' editor*. The back-end is mostly either *the back-end* or *the prototype*. The editor as well as the back-end rely on the meta-models of the modelling language. The meta-models are Ecore models.

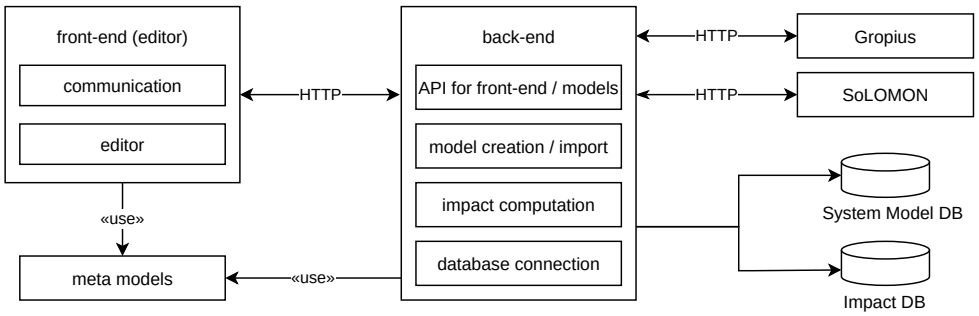


Figure 4.1: Overview over the prototypes architecture.

[1] https://github.com/ccims/solomon

[2] back-end: https://github.com/stiesssh/ma-backend, front-end and models: https://github.com/stiesssh/ma-models

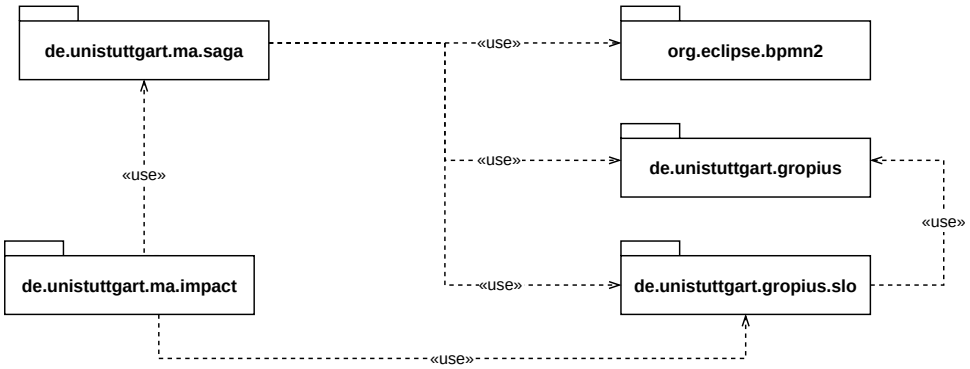


Figure 4.2: Package structure of the models.

The back-end is responsible for computing impacts, creating and importing models, and persisting the impacts and the system models to their respective databases. It communicates with Gropius to get the model of the architecture and to create issues, and it communicates with SoLOMON to get the SLO rules and alerts about violations. It also communicates with the editor to pass it newly created models or to receive updates for existing ones. The editor is responsible for the editing. The user opens the editor to specifies which architecture, process, and SLO rules to import. Once the editor displays the imported parts, the user models a saga in-between process and architecture, and the editor passes the updated model back to the back-end. The editor has no other responsibilities. The user must look at their issue management system to view the issues about the impacts.

4.2 Models

This section covers the implementation of the meta-models as Ecore models. It displays multiple figures of models that we exported from a EMF-based modelling tool. The images exported from the tool do not contain all informations that are in the models. As an example the attributes of the multiplicities are not displayed. Unless stated otherwise, all missing attributes are as described in Section 3.3.

The Ecore models are organized as depicted in Figure 4.2. The package `de.unistuttgart.saga` contains the system model, which is the first of this thesis' objectives. It depends on the Gropius meta-model from `de.unistuttgart.gropius`, the meta-model for the SLO rules from `de.unistuttgart.gropius.slo` and on the BPMN2 meta-model from `org.eclipse.bpmn2`. The used BPMN2 meta-model is provided by the Eclipse Foundation[3] and is available at their git repository[4]. This thesis created the Ecore models for Gropius and the SLO rules anew because they did not yet exist as Ecore models. It created the Ecore model for Gropius according to its API schema[5] and the Ecore model for the SLO rules according to the model provided by SoLOMON[6]. The Ecore model

[3] `httpofs://www.eclipse.org/`

[4] `https://git.eclipse.org/c/bpmn2/org.eclipse.bpmn2.git/`

[5] `https://github.com/ccims/ccims-backend-gql/blob/master/schema/schema.graphql`
commit `739e2f9e76805204ea23c6f295469abc6fcd8656`

[6] `https://github.com/ccims/solomon-models`
commit `ea01537144520233640fc83e739ac5b7d02ff9ec`

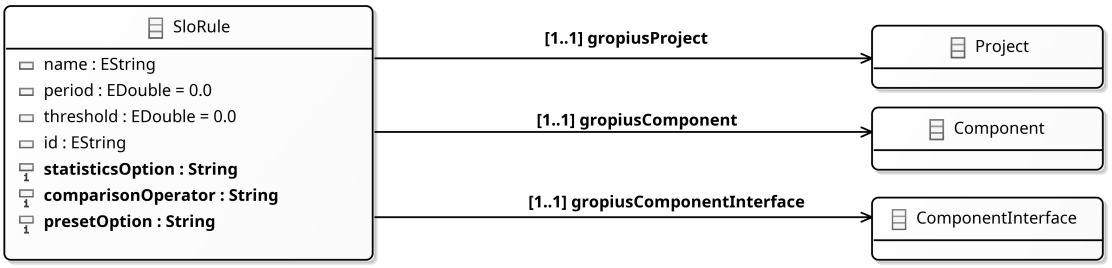


Figure 4.3: Ecore model for SLO rule from SoLOMON.

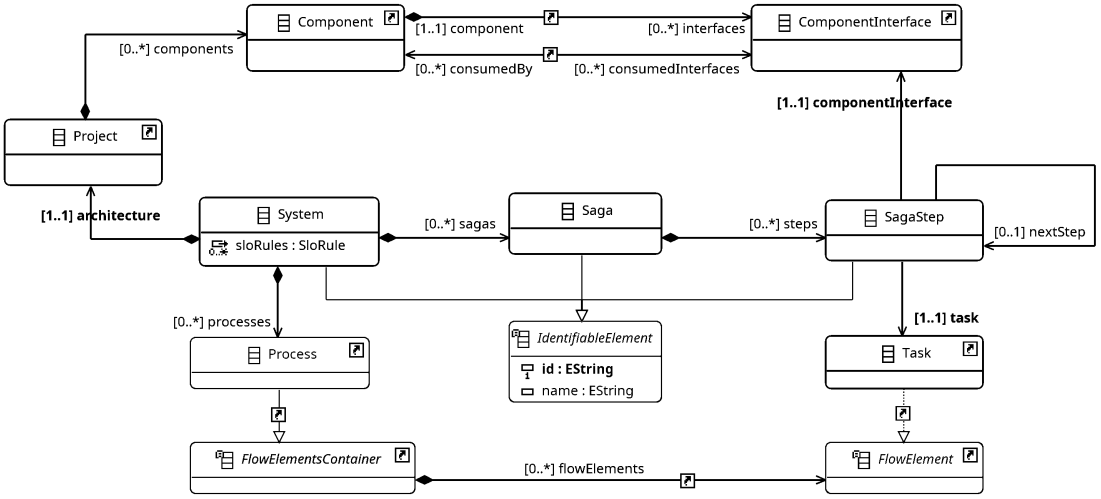


Figure 4.4: Ecore model for the system.

for the SLO rules is shown in Figure 4.3. It deviates from the SoLOMON model with regard to the interface reference. The current SoLOMON can only manage SLO rules at components. However, in the future, it might also be able to manage SLO rules at interfaces. As this thesis is in favour of SLO rules at interfaces, the meta-model already includes this. If at some point in time an official Ecore model of Gropius or SoLOMON emerges, they should replace these substitute meta-models. The relevant elements of the Gropius meta-model were already introduced in Section 2.1.6. A more complete figure of the Gropius substitute may be found in Appendix C.

Figure 4.4 shows the system meta-model as an Ecore model. It imports the classes `Project`, `Component` and `ComponentInterface` from the Gropius meta-model and the classes `Process` and `Task` from BPMN2 meta-model. A tiny arrow symbol in the upper right corner marks the imported classes. The same icon marks imported relations. Except for the element `project`, the meta-model corresponds to the one described in Section 3.3.3. The `project` is part of the model because SoLOMON's SLO rules require them. As described in Section 3.3.3, the `SagaStep` is the point of connection between architecture and process. The references from steps to interfaces, and the ones from steps to tasks, are unidirectional. They originate at the steps. Thus, neither tasks nor

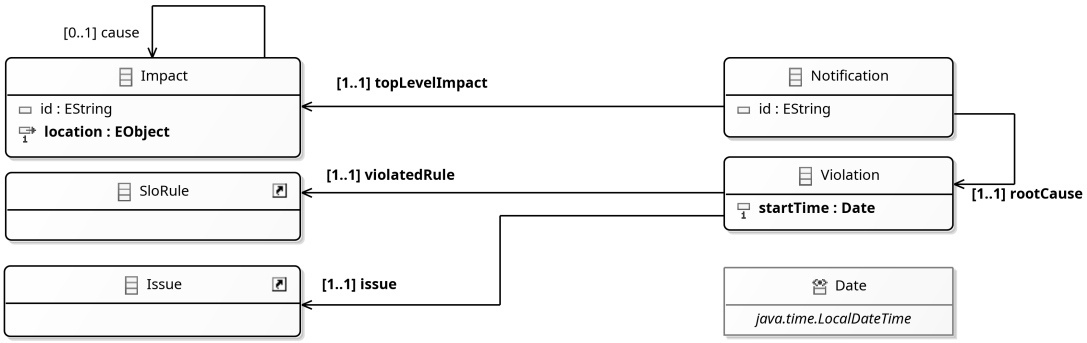


Figure 4.5: Ecore model for the notification.

interfaces know about them. The connection is realised like this because the imported classes cannot be changed, which also means that no additional references can be added. If needed, a reference's opposite direction can be computed at runtime.

For the sake of identification, all model elements posses the attributes `name` and `id`. The `id` exists to uniquely identify each element and `name` exist such that it is easier for humans to tell them apart. As for the imported classes, they already have names and ids. The classes `Saga`, `SagaStep`, and `System` extend the abstract class `IdentifiableElement` from which they inherit these attributes.

The class `System` is the top-level element that contains all others. It exists because with the EMF everything needs to be contained.

Figure 4.5 shows the notification's meta-model as an Ecore model. It consists of the classes `Notification`, `Impact`, `Violation` and `SloRule`, just as described in Section 3.4.2 and of the imported classes `SloRule` and `Issue`. The tiny arrow symbol in the classes' upper right corner indicates an import. The `SloRule` is imported from `de.unistuttgart.gropius.slo`. Contrary to the intended design, the imported rules restrict the locations that have SLO rules to components only. It is like this because the SoLOMON tool works this way. An `Impact`, however, may impact everything, just as intended. The model realizes this by creating the `location` references with type `EObject`, as this is the only common ancestor of all model elements that might be an impact's location.

In addition to the initial design, the model contains a reference between the classes `Violation` and `Issue`. It imports the `Issue` from `de.unistuttgart.gropius`. This reference expresses that for each violation, there already exists an issue in Gropius. In theory, the SoLOMON tool creates the issues regarding the violations and forwards the issues as part of the alerts to other tools, such as this thesis back-end.

These models are the foundation for the rest of the implementation. The back-end uses classes generated from these models, and a large part of this thesis' editor is generated from them.

4.3 Model Editor

This section covers the implementation of this thesis' editor. The editor is an Eclipse plugin that consists of the basic editor generated from the meta-models (c.f Section 4.2) and a Sirius[7] plugin for better design. Figure 4.6 provides an overview of the editor's structure. It omits dependencies to other packages for the sake of brevity.

The generated editor provides basic functionalities to create and edit model instances in a tree view. The tree view is ugly and does not comply with the concrete syntax conjured in Section 3.4.1. Also, the models created with the generated editor contain only the top-level element and nothing else. This thesis' editor changes this behaviour. Instead of starting with an empty system (`System` is the top-level element of the meta-model, c.f. Section 4.2) this thesis' editor imports the architecture, the business process, and the SLO rules right away, such that a newly created model already contains these elements. To achieve this, this thesis' editor has some additions to the generated model creation wizard. It prompts the user for the input displayed in Figure 4.8. Once the user presses *finish*, this thesis' editor requests the back-end to create a new model and import the specified existing models into it. The back-end does this and returns an instance of the saga meta-model to the editor. That model already includes the imported elements and only misses the saga. This is important, as one of this thesis' requirements is the reuse of existing models. The package `saga.presentation` contains the generated parts of the editor, and the extension to the creation wizard The package `importer` contains the classes to communicate with the back-end.

The Sirius plugin realizes the concrete syntax as described in Section 3.4.1. It is a separate Eclipse plugin that requires the generated editor plugin. It consists of the two parts `action` and `description`. Located in `description` is the actual design of the editor and `action` contains some helpers used by that design. Figure 4.7 shows the editor pane as dictated by the design. The figure does not show the other panes because they are not influenced by this thesis. The current pane displays the process on top, the saga in the middle, the architecture below, and the SLO rules attached to their respective locations. The design uses the *Layered ELK Algorithm* from the Eclipse Layout Kernel[8] to layout the diagram. If the resulting layout is not to the user's liking, the diagram elements may be dragged to other positions, individually or in groups. After creation, the model would only contain the architecture, the SLO rules, and the process. Figure 4.7 shows the state of the editor pane after a user modelled a saga into it.

The plugin provides the tools that are displayed in the tab *Saga Tools* in Figure 4.7, to edit the model. *Create new Saga* adds a new saga to the model. *Create new Saga Step* adds a new saga step to the most recently added saga. This tool cannot be used if no saga exists. *Connect to Task* sets a saga step's task reference to the selected task, *Connect to Interface* sets a saga step's interface reference to the selected interface and *Connect to next Step* sets a saga step's nextStep reference to the selected step. The design also provides edition tools to reconnect the outgoing edges of the steps by dragging the edge's ends and an edition tool to assign steps to other sagas. Architecture and process are not editable because they would get out of sync with the source they were imported from. The tool *Send to Backend*, as listed under the tab *Actions* sends the model to the back-end. It is realised as an `External Java Action` that resides in the `action` package.

[7] https://www.eclipse.org/sirius/
[8] https://www.eclipse.org/elk/

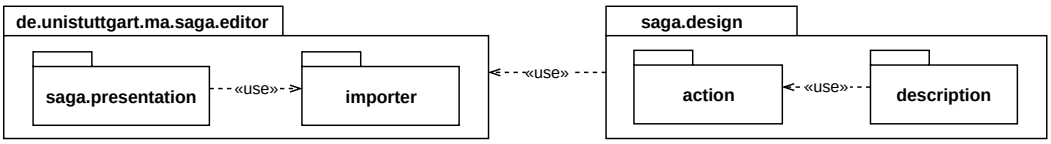


Figure 4.6: Package structure of the editor.

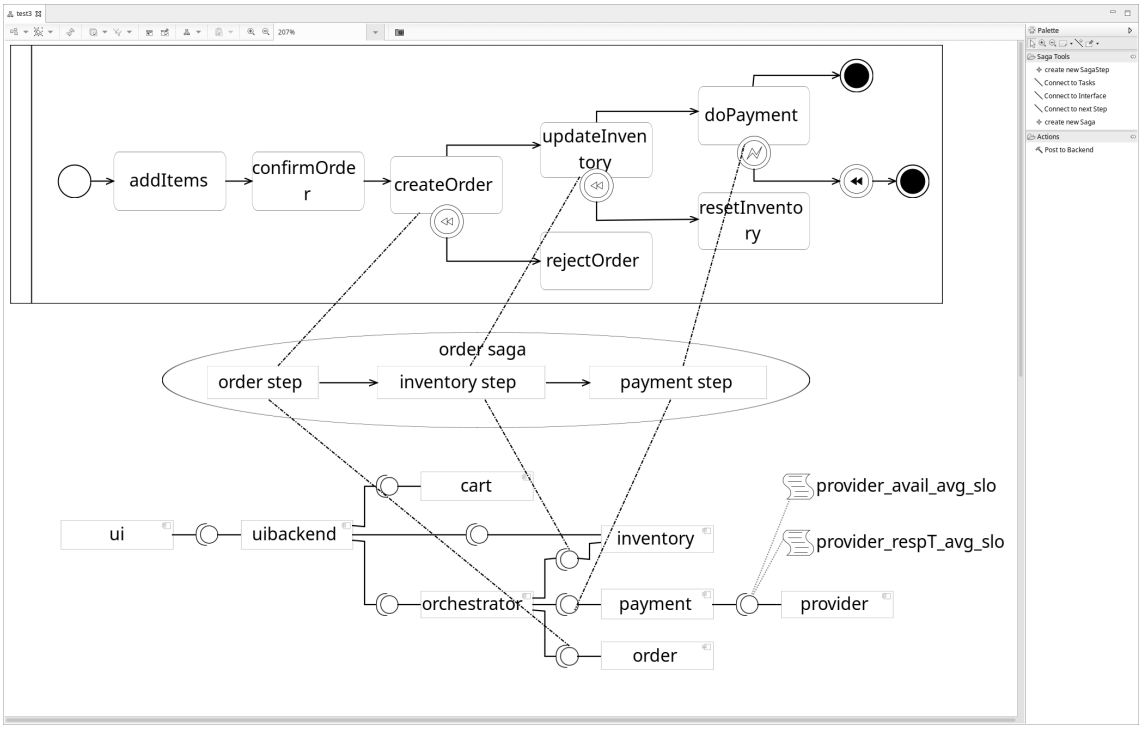


Figure 4.7: Editor pane displaying a model and some tools on the right.

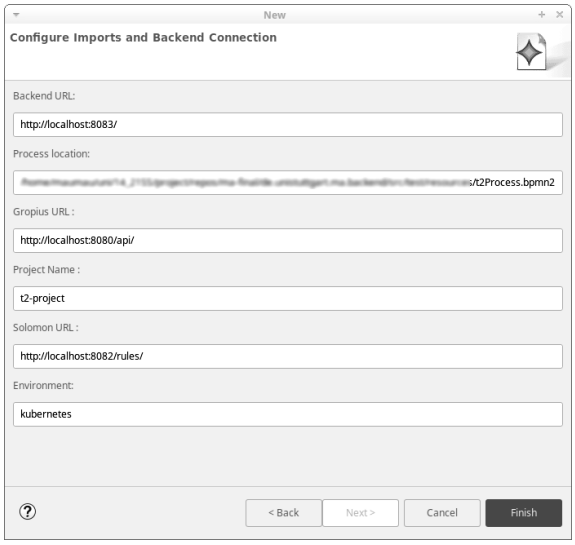


Figure 4.8: Addition to the editor's Model Creation Wizard Dialog.

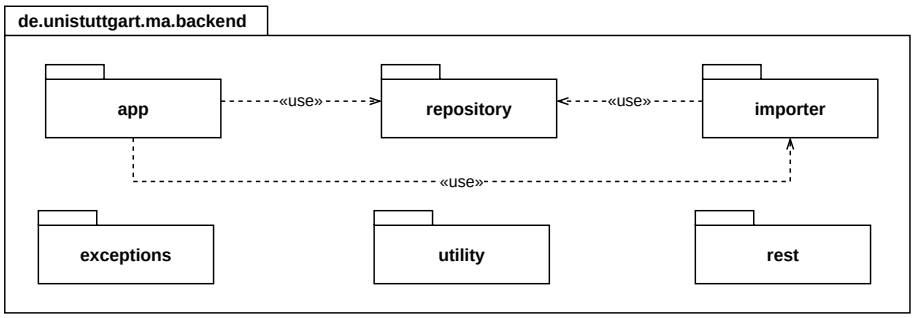


Figure 4.9: Overview over the package structure of the back-end.

4.4 Back-end

The back-end creates the models and generates the notifications. To achieve the former, it imports existing models, transforms them if necessary, and merges them. For the latter, it listens for incoming alerts, calculates the impacts upon receiving an alert, puts all gathered information into a notification, and reports the notification as a cross-component issue in Gropius. The back-end is a Spring Boot[9] application in Java.

Figure 4.9 displays the back-end's package structure. It consists of six packages. The packages `app`, `repository` and `importer` hold the majority of the back-end's logic. The other three packages hold classes of supportive nature. The package `exceptions` contains domain specify exceptions, the package `utility` contains helpers used when creating a notification and the package `rest` contains classes to parse the body of http request and replies.

The the back-end depend on the packages `de.unistuttgart.slo`, `de.unistuttgart.gropius`, `de.unistuttgart.ma.saga`, `de.unistuttgart.ma.impact`, `org.eclipse.bpmn2` and on the package `de.unistuttgart.gropius.api`. The first five contain the meta-models as described in Section 4.2. Almost all parts of the back-end use these, therefore the remainder of this section only mentions them if they are especially important. The last one is generated from Gropius' GraphQL schema. The back-end needs it to communicate with the Gropius API.

The `app` package, as depicted in Figure 4.10, contains the controller classes with the API endpoints and the services that do the actual work. The `AlertController` awaits incoming alerts from a monitoring tool. It exposes an HTTP endpoint to which the monitoring should send its alerts. Upon receiving and alert, the `AlertController` uses the `CreateNotificationService` to calculate the impacts and put them into a notification and then it uses the `CreateIssueService` to create a cross-component issue in Gropius from the notification. `CreateIssueService` depends on the `de.unistuttgart.gropius.api` package to create the issues. The `ModelController` exposes endpoints to create, get and update models. It delegates all work to the `ModelService`. That service uses the importers defined in the `importer` package to import SLO rules, architecture and process and save the resulting model in the `SystemRepository`.

[9] https://spring.io/projects/spring-boot

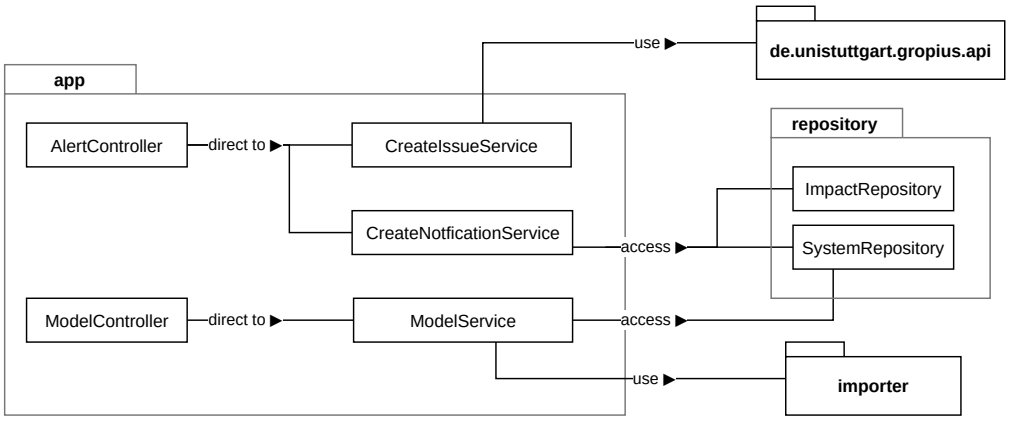


Figure 4.10: Details of the back-end's `app` and `repository` packages.

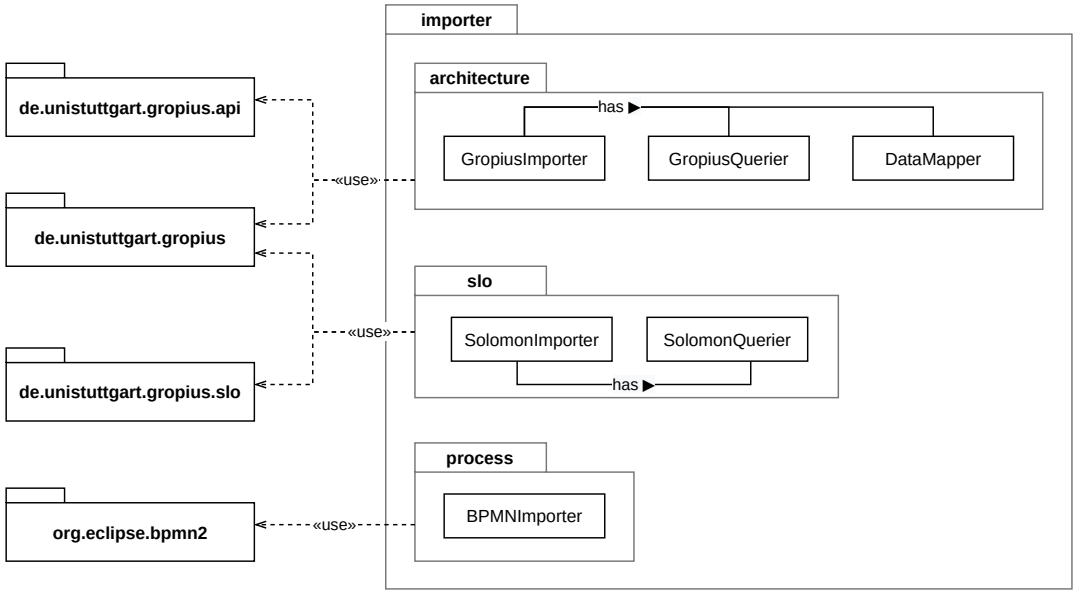


Figure 4.11: Details of the back-end's `importer` package.

The `repository` package, which is also depicted in Figure 4.10, contains the interfaces to the database. The interface `SystemRepository` persists system models and the interface `ImpactRepository` is for persisting impacts. The `SystemRepository` is accessed by the `ModelService` and by the `CreateNotificationService`. The former accesses it to save, update and get models, whereas the latter only gets models, as it needs them to calculate impacts. The `CreateNotificationService` also accesses the `ImpactRepository`, to save all impacts it calculated.

The `importer` package contains all importers. It is organized into the subpackages `architecture`, `slo` and `process`. They are depicted in Figure 4.11. The package `architecture` depends on the packages `de.unistuttgart.gropius` and `de.unistuttgart.gropius.api` and contains the importer `GropiusImporter` that imports an architecture model from Gropius. The importer utilizes the `GropiusQuerier` to query the Gropius API for an architecture model. Then it utilizes the `DataMapper` to transform the query result into a model instance of the Gropius meta-model. The package `slo` depends on the packages `de.unistuttgart.gropius.slo` and `de.unistuttgart.gropius`. It is

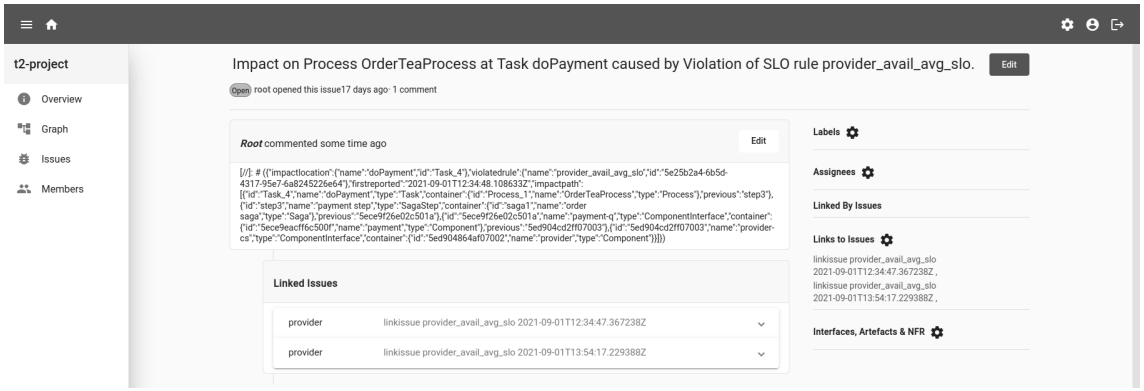


Figure 4.12: Cross-component issue in Gropius created for a notification.

structured similarly to the `architecture` package, with and importer (`SolomonImporter`) and a querier (`SolomonQuerier`). This importer uses the querier to query the API of the SoLOMON tool for SLO rules and then transform them into instances of the SLO meta-model. The package `process` depends on the package `org.eclipse.bpmn2` and contains the importer to import a BPMN process. It requires no transformation as it assumes that the models to import already are instances of the BPMN2 meta-model.

4.5 Cross-Component Issue in Gropius

This section describes the cross-component issue in Gropius that the back-end creates for each notification. Figure 4.12 shows a screenshot of the issue as displayed by the Gropius front-end and Listing 4.1 lists the content of the issue's body in a more readable fashion. The issue title informs the user right away about the problem. In this, that a violation of the SLO rule `provider_avail_avg_slo` impacts the task *doPayment* of business process *OrderTeaProcess*. As displayed on the right side and the bottom of the Gropius front-end, the issue links to two others, namely those that the SoLOMON tool created for the initial SLO violations. The user must go there to get more details about the violations. The issue body contains a JSON representation of the notification. It is hidden within a markdown comment, and thus wrapped into `[//]: # (` and `)`. The Gropius front-end version from which the screenshot in Figure 4.12 originates does not support markdown, therefore, the JSON is visible anyway. Newer versions do support markdown. The JSON complies to the schema described in Section 3.4.2.

Listing 4.1 JSON representation of the impact as contained in an issues body.

```
{
  "impactlocation": {"name": "doPayment", "id": "Task_4"},
  "violatedrule": {"name": "provider_avail_avg_slo", "id": "5e25b2a4-6b5d-4317-95e7-6
a8245226e64"},
  "impactpath": [{
      "id": "Task_4", "name": "doPayment", "type": "Task",
      "container": {"id": "Process_1", "name": "OrderTeaProcess", "type": "Process"},
      "cause": "step3"
    }, {
      "id": "step3", "name": "payment step", "type": "SagaStep",
      "container": {"id": "saga1", "name": "order saga", "type": "Saga"},
      "cause": "5ece9f26e02c501a"
    }, {
      "id": "5ece9f26e02c501a", "name": "payment-q", "type": "ComponentInterface",
      "container": {"id": "5ece9eacff6c500f", "name": "payment", "type": "Component"},
      "cause": "5ed904cd2ff07003"
    }, {
      "id": "5ed904cd2ff07003", "name": "provider-cs", "type": "ComponentInterface",
      "container": {"id": "5ed904864af07002", "name": "provider", "type": "Component"}
  }]
}
```

5 Evaluation

This chapter covers the evaluation of this thesis' concept. The evaluation bases on the Goal Question Metric (GQM) approach [SBCR02]. Section 5.1 describes the application of the GQM approach to this thesis. Section 5.2 und Section 5.3 elaborate on the execution and on the results of the evaluation. Section 5.4 discusses the results and Section 5.5 discusses the threats to the evaluation's validity.

5.1 Applied GQM Approach

The evaluation of this thesis' concept is base on the GQM approach [SBCR02]. The GQM approach defines metrics in a top-down fashion. At first, it identifies goals. Then it defines questions to evaluate whether this thesis reached its goals. Lastly, it adds metrics that answer the questions, at least one for each.

The overall goal of this thesis is **(G1)** to trace how SLO violations that occur in a microservice architecture that implements the saga pattern, impact business processes and to present the gathered information to the user. From a user's perspective, this trace should improve their understanding of the caused impacts, such that they can more easily react. The following questions refine **(G1)**:

(Q1.1) Do developers face problems with regard to SLO violations in microservice architectures that implement the saga pattern?

(Q1.2) Does this thesis' concept help in solving any of these problems? If so, to what degree?

These questions capture whether a tool according to this thesis' concept is wanted or needed and whether it would improve the current situation. Simple numbers can hardly capture the answers to these questions. The answers require experience and knowledge of the real world. Experts usually provide such things. Thus, an expert survey is the metric of choice. Section 5.3 describes the survey.

In addition to the overall thesis goal, we define the goal **(G2)** which is to calculate the impacts that an SLO violation that occurs in a microservice architecture that implements the saga pattern has on the business process, based on this thesis' modelling language. The question that refines **(G2)** is:

(Q2.1) Can a model in this thesis' modelling language be used to calculate impacts on the business process?

The metric for this question is the ratio of correctly created cross-component issues in Gropius, which an experiment captures best. Section 5.2 describes the experiment.

5.2 Experiment

The experiment consists of multiples steps, two for preparation and then the actual execution. The first step is to deploy a reference architecture, as described in Section 5.2.1. The second step is to set up Gropius, SoLOMON and this thesis' prototype, which is described in Section 5.2.2. The third step is to create the models, and the fourth step is to trigger different SLO violations and to observe the created issues. This is described in Section 5.2.3.

5.2.1 Reference Architecture

The experiment utilizes the t2-project[1] as a reference architecture. The t2-project is a webshop with a microservice architecture that implements the saga pattern. As depicted in Figure 5.1, it consists of seven services and an additional eighth service that pretends to be an external service provider. The eighth service is the service `CreditInstitute`.

The services have the following responsibilities. `UI` provides a web frontend to the user and `UIBackend` is an API Gateway [Ric18]. The other services realise the actual business logic of the shop. `Cart` manages the shopping carts, `Order` records all orders and their status, `Inventory` manages the product stocks and `Payment` manages the money transfer. The service `Orchestrator` orchestrates the saga. It has no additional responsibilities. The t2-project's shop depends on an extern component for money transfer because it is not a credit institute but a simple shop that focuses on selling tea. In real life it would depend on PayPal[2], Klarna[3] or similar payment providers. In the t2-project the service `CreditInstitut` takes this role. `CreditInstitut`'s response time and failure rate are configurable at runtime, which makes it easy to trigger SLO violations.

For the experiment, we deployed the t2-project on a Kubernetes cluster and generated load with the approach proposed in the t2-project's documentation.

5.2.2 Tool Setup

Figure 5.2 depicts the tools used in this experiment and their setup. The reference architecture is deployed on a Kubernetes cluster and monitored with Prometheus. There is a `prometheus-blackbox-exporter` on the cluster to probe for availability. The exporter is deployed from the Prometheus community helm charts[5]. In theory a full `kube-prometheus-stack` and the SoLOMON tool itself should also run on the cluster. Due to technical difficulties, it was not possible to get SoLOMON's full range of functionality to work. The SLO rule management worked, but the monitoring did not. The service `fakesolomon` is a workaround for that problem. For now, SoLOMON, Gropius and this thesis' prototype, as well as the `fakesolomon`, a Prometheus instance and a Prometheus alert-manager run locally and watch the cluster from afar.

[1] https://t2-documentation.readthedocs.io/en/latest/index.html
[2] https://www.paypal.com/
[3] https://www.klarna.com/
[5] https://github.com/prometheus-community/helm-charts

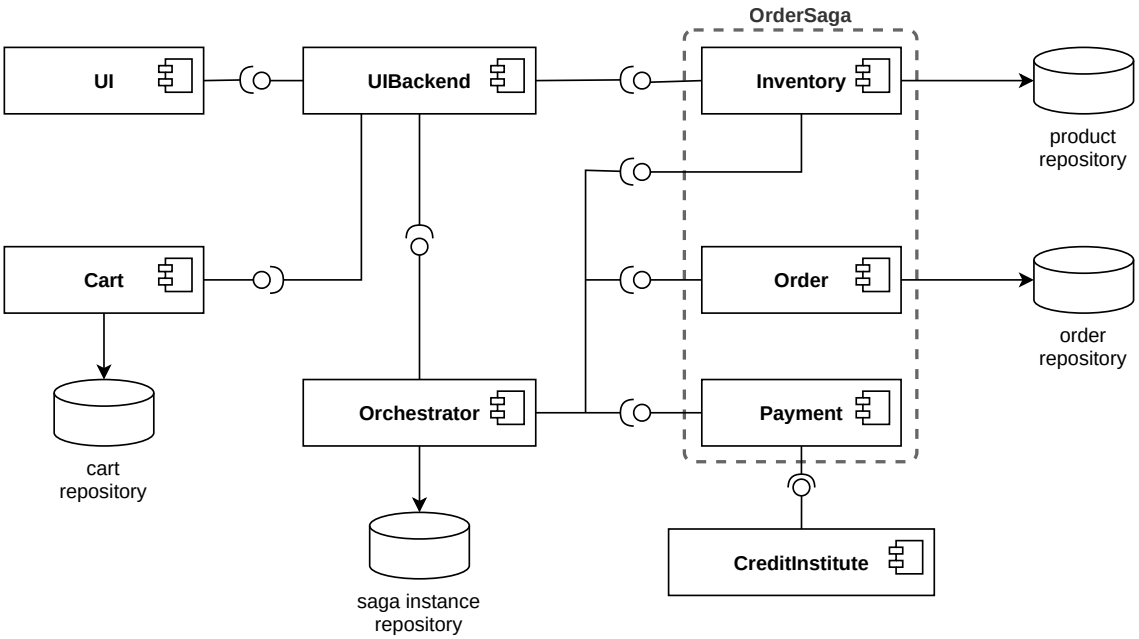


Figure 5.1: Services of the t2-project microservice reference architecture[4].

Once all tools are up and running, it is time to define the required models. We created the models according to the t2-project's documentation. We modelled the architecture with Gropius, as depicted in Figure 5.3, the business process with the Eclipse BPMN2 Modeller, as depicted in Figure 5.4, and the saga with this thesis' editor, as depicted in Figure 5.5. The SLO rules are modelled with SoLOMON, as depicted in Figure 5.6. The rule *provider_avail_avg_slo* expresses that the availability, averaged over 60 seconds, should always be over 90%, and the rule *provider_respT_avg_slo* expresses that the response time, averaged over 10 seconds should always be lower than 10 seconds. Both rules are attached to the `Provider` service.

At last, we defined Prometheus alerting rules for the experiment. They are listed in Appendix A. They match the SLO rules defined with SoLOMON. They are annotated with the unique ids of SoLOMON's SLO rules, such that fake alerts triggered with these rules can be correlated to the actual SLO rules from SoLOMON. If SoLOMON worked correctly, it would handle the alerting itself, and the additional alerting rules would not be necessary.

A list of the used versions of the tools and so on is in Appendix A.

5.2.3 Execution

The experiment consists of three treatments **(E1)** to **(E3)** as depicted in Figure 5.7. It applies each treatment twice, once for the response time SLO and once for the availability SLO. The purpose of **(E1)** is to show that the tool creates a single correct issue upon receiving a single alert. The purpose of **(E2)** is to show that the tool does not create a new issue if an open issue with the same root cause and impact location already exists. Instead, the tool should link the existing impact issue to the

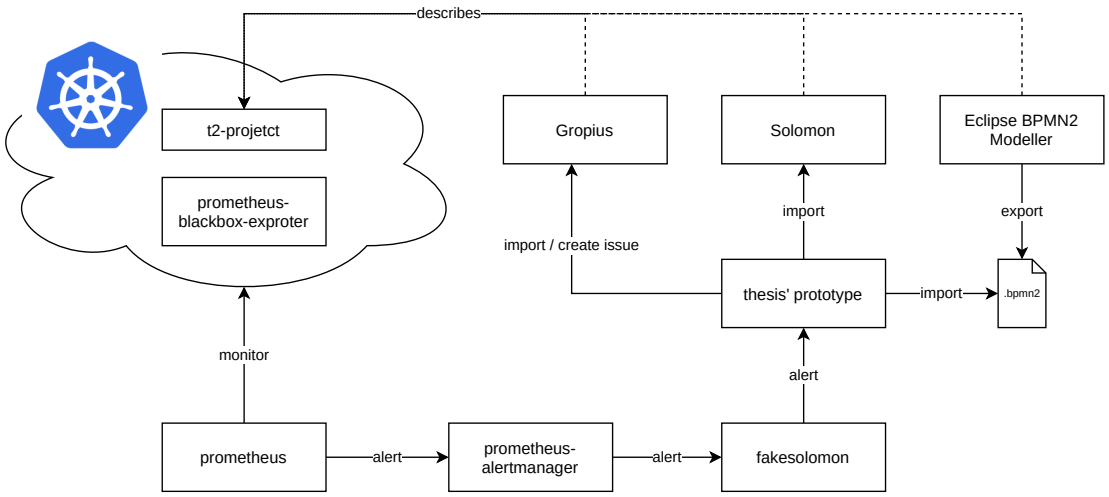


Figure 5.2: Experiment setup with tools and reference architecture.

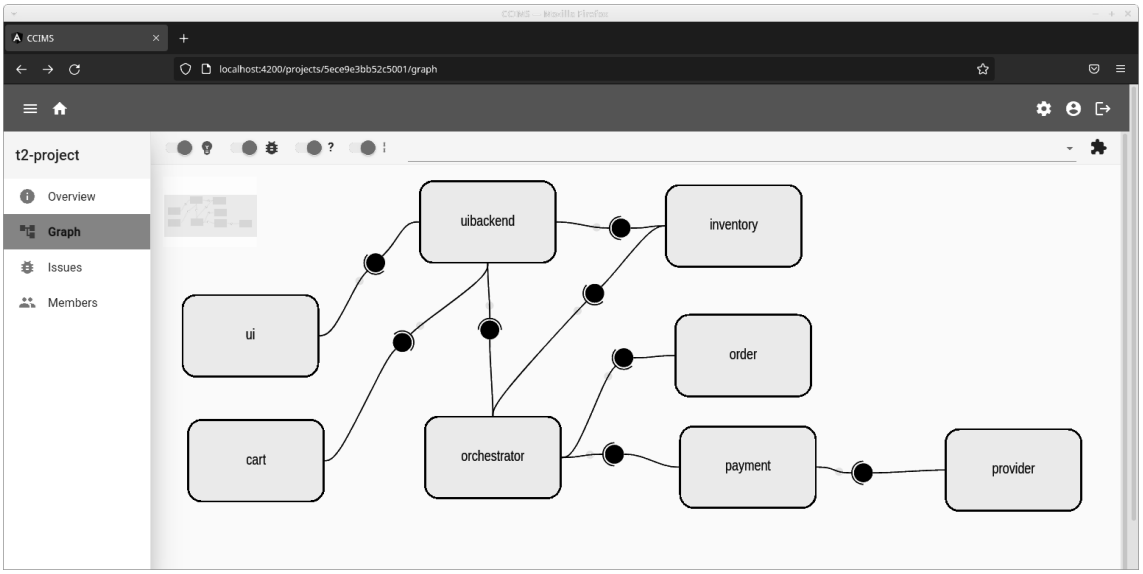


Figure 5.3: Architecture for the experiment, modelled with Gropius.

new violation issue. The purpose of **(E3)** is to show that the tool again creates a new issue if the existing issue that matches the root cause and impact location is already closed. Table 5.1 provides an overview.

For the experiment, we triggered a violation of the availability SLO by deleting the Kubernetes service of the `Provider`. To trigger a violation of the response time SLO, we increased the response time of the `Provider` via the interfaces it provides for that very purpose.

With the given reference architecture all created issues should have the same impact chain, which Listing 5.1 lists. The violation happens at the `Provider`'s interface. Thus, the first impact is on that interface. From there it passes to the service `Payment`, which passes it to the step *paymentStep* of the saga *orderSaga*, and from there it reaches the task *doPayment* of the *orderTea* process. The JSON representation from Listing 5.1 is supposed to be part of the bodies of the created issues.

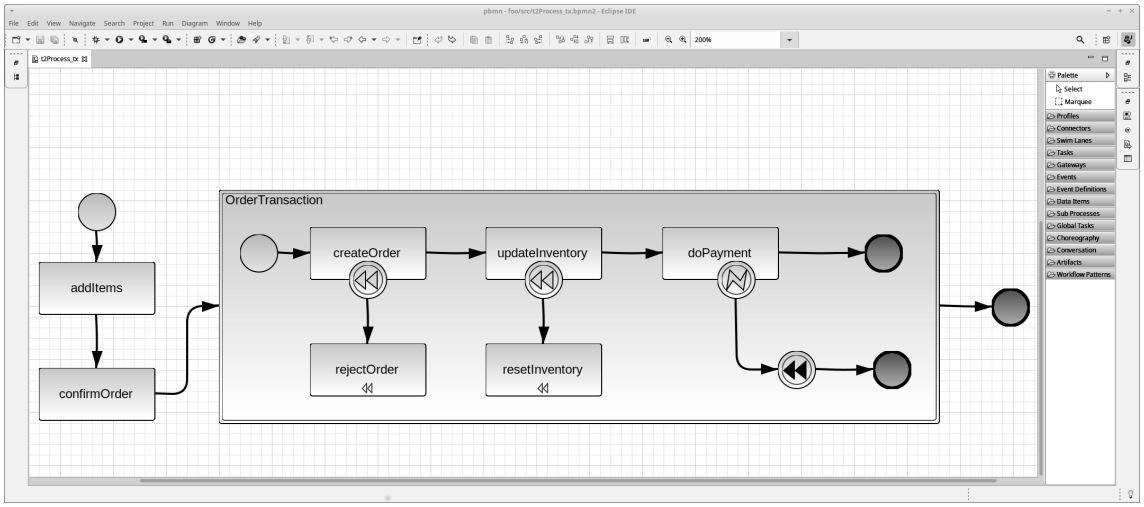


Figure 5.4: Business process for the experiment, modelled with the Eclipse BPMN2 modeller.

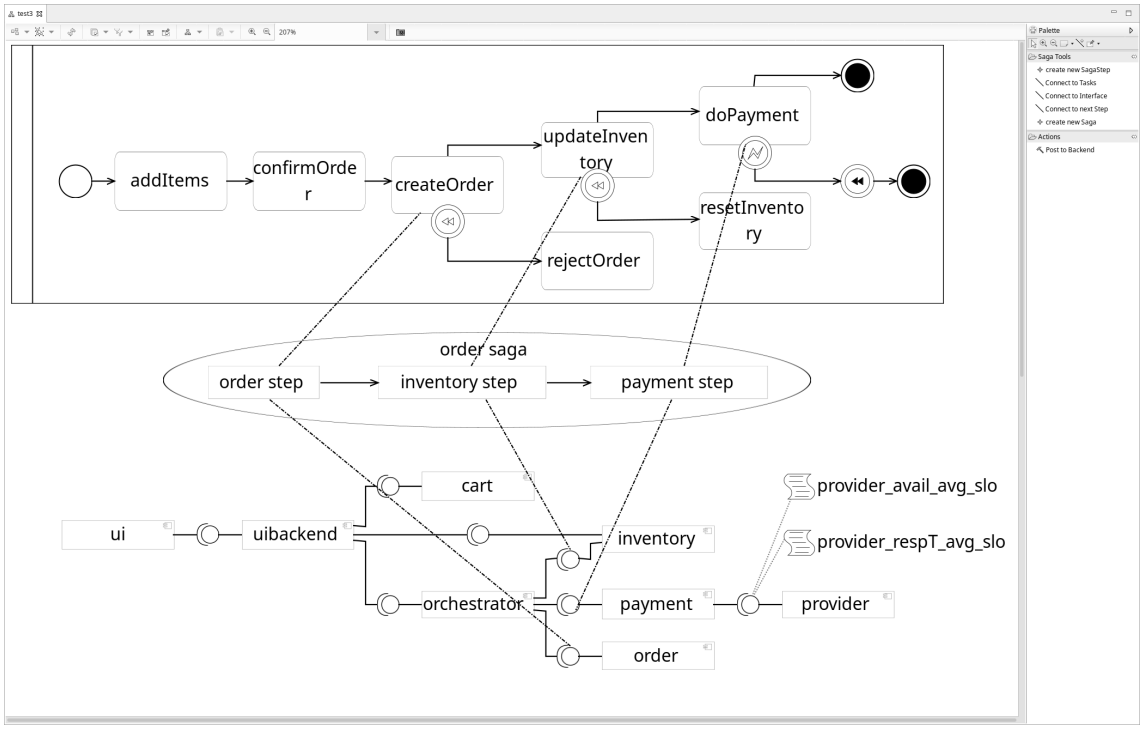
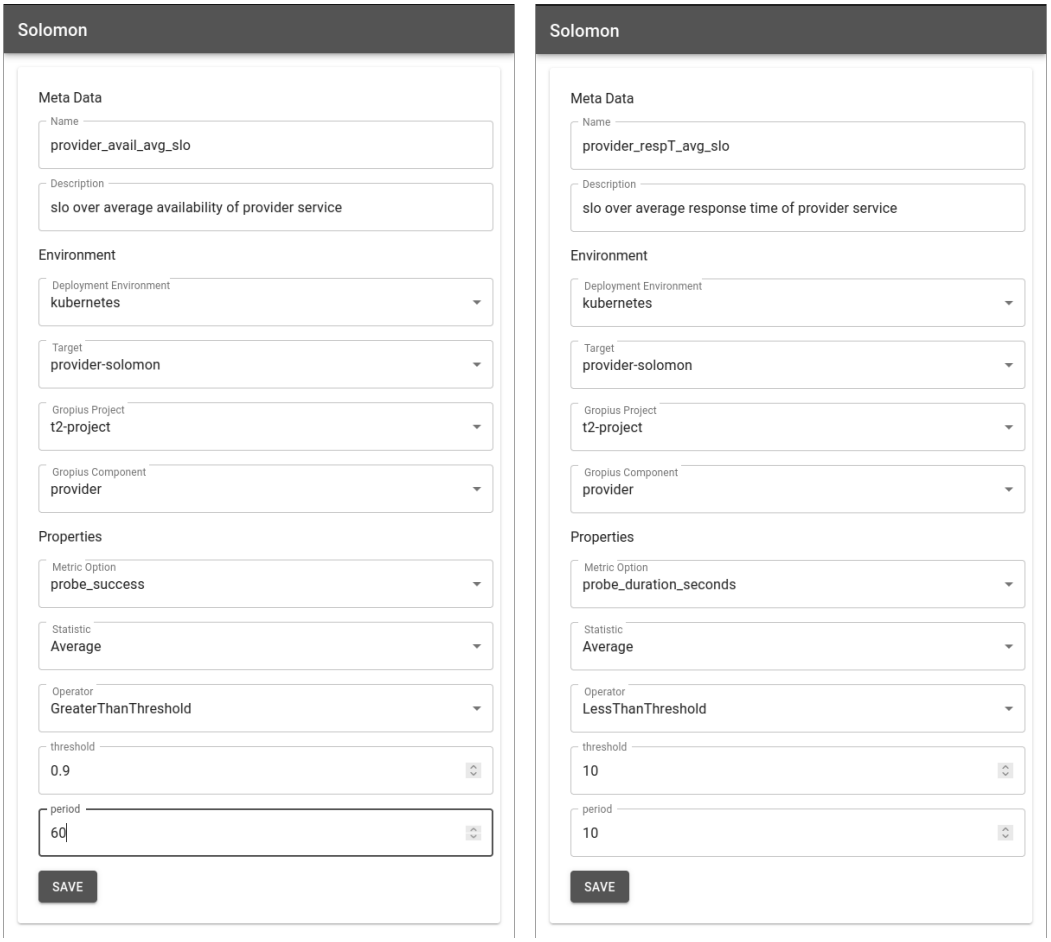
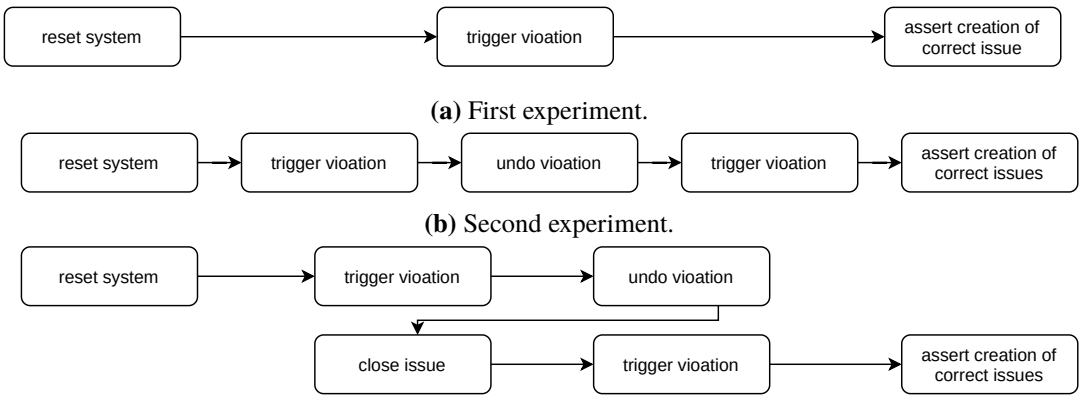


Figure 5.5: Saga of the t2-project, modelled with this thesis' editor.



(a) Availability SLO rule. **(b)** Response time SLO rule.

Figure 5.6: SLO rules for the experiment, modelled with SoLOMON.



(a) First experiment.

(b) Second experiment.

(c) Third Experiment.

Figure 5.7: Executions of the experiment.

	covered case	expected behaviour
E1	no other issue exists	Creation of **two** new issues: **one** issue for the violation (created by fakesolomon), and **one** for the impact (created by this thesis' prototype), linked to violation issue only.
E2	an open issue already exists	Creation of **three** new issues: **two** issues for the violations (created by fakesolomon), and **one** issue for the impact (created by this thesis' prototype), linked to both violation issues.
E3	a closed issue already exists	Creation of **two** new issues in addition to the closed one: **one** issue for the violation (created by fakesolomon), and **one** for the impact (created by this thesis' prototype), linked to the new violation issue only.

Table 5.1: Cases covered by the experiment and expected behaviour.

Listing 5.1 Impact path as JSON.

```
{
  [...]
  "impactpath": [{
      "id": "Task_4", "name": "doPayment", "type": "Task",
      "container": {"id": "Process_1", "name": "OrderTeaProcess", "type": "Process"},
      "previous": "step3"
    }, {
      "id": "step3", "name": "payment step", "type": "SagaStep",
      "container": {"id": "saga1", "name": "order saga", "type": "Saga"},
      "previous": "5ece9f26e02c501a"
    }, {
      "id": "5ece9f26e02c501a", "name": "payment-q", "type": "ComponentInterface",
      "container": {"id": "5ece9eacff6c500f", "name": "payment", "type": "Component"},
      "previous": "5ed904cd2ff07003"
    }, {
      "id": "5ed904cd2ff07003", "name": "provider-cs", "type": "ComponentInterface",
      "container": {"id": "5ed904864af07002", "name": "provider", "type": "Component"}
  }]
}
```

5.2.4 Result

All treatments produced the expected results. For **(E1)** the back-end created two issues, one for response time SLO's violation and one for the availability SLO's violation. Each issue links to the respective issues about the actual violation of the SLO rules. For **(E2)** the back-end created two issues, one for the violation of the response time SLO and one for the availability SLO's violation. Each issue links to two others, which are the respective issues about the actual violation of the SLO rules. For **(E3)** the back-end created four issues, two for the violation of the response time SLO and two for the availability SLO's violation. Each issue links to the respective issues about the actual violation of the SLO that caused it.

5.3 Expert Survey

The purpose of this expert survey is to evaluate the modelling language. Section 5.3.1 describes the execution of the survey, including the instructions and questions given to the participants. Section 5.3.2 describes the survey's results.

5.3.1 Execution

The system was setup in the same manner as for the experiment (c.f Section 5.2.1 and Section 5.2.2), except for the modelling of the saga. We prepared the architecture, the SLO rules and the business process but did not model the saga, as that was supposed to be part of the participants' tasks.

We contacted the participants via email, providing them with a brief overview of the survey and requesting their aid. The email was written in English and German, such that as many of the contacted persons as possible could understand it. The participants were offered two options for participation. The first option was via direct interaction, for which we scheduled a call or met the participants in person. With this option, the participants viewed the survey setup and this thesis' editor via screen sharing and interacted with it personally. The remote interaction reduced the participants' effort because they needed not set up the system themselves. It also ensured that all participants worked with a correct and identical system setup. The second option was to provide the participants with screenshots of the editors and the issues, and with written descriptions of what this thesis' concept is doing. We preferred the first option over the second as it gave us, as the surveyors, the possibility to observe the participants interaction with the editor in real life. It also gave the participants the possibility to inquire about things they did not understand, which in turn provided us with additional information about areas in which this thesis is lacking.

We provided each participant with an introduction and two tasks. The introduction covered this thesis' goal and its foundations, such that the participants know the context of the survey. After the introduction, the participant received these tasks:

(T1) Add a saga to the already opened model in the editor.

(T2) Look at an issue created by this thesis' prototype and try to understand it.

The two tasks confront the participants with the two main contributions of this thesis. Task **(T1)** confronts the participants with the modelling language, and task **(T2)** confronts them with the impact of an SLO violation, or rather this thesis' representation thereof. For **(T1)** the participants also received an introduction to the modelling language and the editor. For **(T2)** they received an additional explanation about what lead to the creation of the issue. The participants had the opportunity to ask questions at any time. We accompanied each participant through the tasks but did only intervene or help when requested. We took notes about the participants' interaction with the system. After performing the tasks, the participants had to answer the questions listed in Table 5.2.

Questions 1.1 to 1.3, 2 and 3 attempt to capture the participants' prior experiences. They also attempt to capture existing problems with regard to microservice patterns and SLOs, thereby targeting **(Q1.1)**. The questions 4.1 to 4.3 and 5 to 7 target **(Q1.2)**, as they inquire whether a tool akin to this thesis' concept could bring any improvements over the current situation.

No.	Question
	Do you have prior experience with...
1.1.	... microservice patterns in general?
1.2.	... the saga pattern, or other ways to realize transactions across multiple service?
1.3.	... EMF-based modelling editors?
2.	Have microservice patterns ever caused you any problems?
3.	Your business process deviates from its expected behaviour (e.g. a transaction repeatedly rolls back, or some task is very much delayed). How would you normally notice this and how do you react? What do you do to get the processes back on track?
	Would an issues, as created by this thesis' prototype,...
4.1.	... make you realise sooner that something is going wrong?
4.2.	... help to understand what is wrong?
4.3.	... help when fixing the problem?
5.	What is your opinion on the modelling language?
6.	What is your opinion on the created issues?
7.	Would you use a tool akin to this thesis' prototype? What are your reasons?

Table 5.2: Questions of the expert survey.

We tested the survey on two participants. The test participant underwent a similar procedure as the final participants. The difference was that aside from evaluating the language and this thesis' concept, we also requested feedback on the survey itself. We refined the survey and the questionnaire based on the feedback of those participants.

5.3.2 Results

In general, the experts are convinced that connections between architectures and business processes are useful. Additionally, they mostly agreed that an issue about an SLO violation's impact on the business process is useful. However, most of them doubt either the relevance and necessity of the saga pattern or the restriction to the saga pattern only.

In the end, six experts participated in the survey. They came from academic and industrial backgrounds and had different amounts of familiarity and experiences with this thesis' foundations. All participants had prior experiences with microservices, and all except for one had already been in contact with microservice patterns. About half of them had prior encounters with the saga pattern, and about half of them had prior encounters with EMF-based modelling editors. The halves are not the same. Some participants had encounters with both topics, only one of them, or neither.

The following paragraph first describes the insights gained from our observations of the participant behaviour during tasks **(T1)** and **(T2)**, followed by a description of the participants answers to the questionnaire. Due to timing issues, some experts did not make it to **(T2)**.

Regarding **(T1)**, that is the interaction with thesis editor, all participants completed it successfully. They identified the process and the architecture and added the saga in between. Some struggled with the order of clicking when using a creation tool or when connecting two model elements with a connection tool. Those experts that already had experiences with EMF-based modelling editors had

fewer problems than those that did not. At the same time, the editor was of great help, as it forbids most illegal actions. Some experts struggled with the notation of the business process. Some of them because they were not familiar with the BPMN and others because they were very familiar with the notation and got confused by the prototype's imperfect application thereof.

All experts understood the gist of the created issues and their general structure. They understood that the root of the problem was the SLO violation, and they identified the impact path as such. However, most had complaints about some details regarding the representation of the path. Some got confused by the ids in the JSON because the ids do not have a unified format. Others got confused by the chosen tags. They mistook the top-level impact for the location of the root cause or read the impact path upside down. Only after they read the entire path and identified which impact impacted which location, these misunderstandings cleared up.

As for existing problems with microservice patterns, the experts only ever encountered problems because of misunderstandings about a patterns design and difficulties with their implementation or because of their application in the wrong context. As for existing strategies to handle deviations from a business process excepted behaviour, most experts would resort to manual debugging. If they had monitoring, they would rely on the monitoring to inform them about a problem and also use the monitored data when scrutinizing the services.

As for potential improvements due to the issues created by this thesis' prototype, the experts agreed that issues are helpful. The participants stated that it is likely that the issues trigger a fast reaction by the concerned personnel. However some participants pointed out, that too many issues concerning the same problem might be more hindrances than help. They also stated that the issue's content helps in understanding how an SLO violation propagates. This in turn helps in figuring out the root cause of an impact which again will help in fixing the initial problem. Some participants noted that an issue's usefulness might greatly depend on the reader's expertise and knowledge about the affected system. Other participants criticised that it is not clear whom the issue is supposed to help. It covers bits of the architecture and bits of the process and also bits about patterns such that in the end, everyone might assume that it is someone else's problem. In addition, some participants suggested to split the trace into two parts, one to cover the SLO violation's propagation from the architecture to the process, and another to cover its propagation from the component where it initially happened to the component from which it goes up to the process. The reason behind that split being, that the former traces vertically and crosses the border between different models whereas the latter traces horizontally within one and the same model. Furthermore, the former is an abstract connection that exists due to on expert's knowledge, whereas the latter is more tangible. The participants' opinion on the impact information's representation as JSON in the issue's body is divided. Some liked it because JSON is easy to read and understand, while others did not because JSON is difficult to read and understand. Some state that every representation would be fine, as long as it is standardised such that it may be integrated into other tools.

Regarding the modelling language, the participants appreciated its looks and that the editor is of great help when using it, as it restricts a user to legal actions only. Some also appreciated the design decision to exclude any option of editing the imported models. The majority was convinced that any model that provides insight into the connections between architecture and business process are useful in many ways. The usefulness of the explicit inclusion of the saga pattern, however, is questionable. A point of critique was the concept behind the mapping between the saga pattern and the process. If the process is very fine-grained, modelling a saga with all of its steps is almost redundant, as each saga step maps to exactly one task. Plus, using the saga as the point of connection

eliminates the possibility to connect tasks and interfaces directly. The latter can be considered a feature request. Another point of critique was the responsibility, as in who is supposed to have enough knowledge about all parts of the system to connect them successfully. In real life, this knowledge is distributed among many persons and many departments and the creation of the model would require their collaboration.

Concerning acceptance, some participants find this thesis' prototype overall useful but would only use it in complex scenarios where the connections between process and architecture are not as trivial as in the example they witnessed during the survey. Others are unsure whether such a tool would fit with existing ones. Some noted, that it would be difficult to use the tool supporting this thesis' concept in the industry as the industry lacks explicitly defined SLOs and monitoring thereof.

The expert survey also revealed some new desired features or backed features, that were considered during this thesis but did not make it into the final contribution. Multiple participants requested to go beyond the saga pattern by allowing direct connections between tasks and interfaces. They also requested the option to add additional information to the connections. Some participants requested to drop the CRUD assumption that was imposed on all interfaces and to model them in more detail. In real life, this is relevant because some SLO rules concern a certain operation only and not the entire interface. Some participants recommended to aggregate issues as to not overwhelm the user with a multitude of issues about the same problem. In part, this is already implemented as this thesis' prototype aggregates issues with the same root cause and impact path. However for large systems where a SLO violation at one component may impact many other components and tasks, it might desirable to further aggregate the issues. As an example, if the impact path of one issue is a subpath of another issue, one could aggregate theses two issues, as it is likely, that the SLO violation that caused the issue with the shorter impact path was caused by the violation that caused the issue with the longer impact path.

5.4 Discussion

This section discusses the results of the experiment and the expert survey. Its structure follows the GQM plan described in Section 5.1. The hypotheses for **(G1)** are:

(H1.1) There are problems with regard to SLO violations in microservice architectures that implement the saga patterns.

(H1.2) This thesis' concept helps in solving problems with regards to SLO violations in microservice architectures that implement the saga patterns.

The hypothesis for **(G2)** is:

(H2.1) A model in this thesis' modelling language can be used to calculate impacts on the business process.

This section discusses the acceptance or rejection of the hypotheses **(H1.1)** and **(H1.2)** based on the results of the expert survey described in Section 5.3.2. After that, it discusses the acceptance or rejection of the hypothesis **(H2.1)** based on the results of the experiment described in Section 5.2.4.

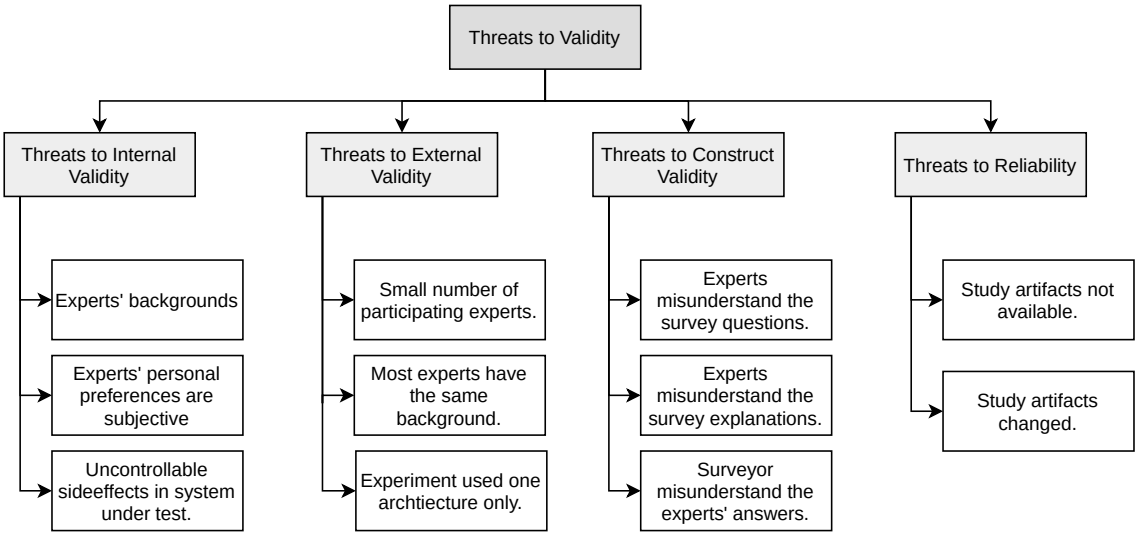


Figure 5.8: Threats to validity.

None of the survey's participants has ever had any problems in conjunction with SLO violation and the saga pattern. This might be due to the fact that only about half of the participants had heard of the saga pattern before and none of them had ever actually implement it. Still, we have no reason to accept the hypothesis **(H1.1)** and must reject it. As we already rejected **(H1.1)** we must also reject **(H1.2)**. If there is no problem, there is also no problem to be solved.

The experiment uses this thesis' prototype to calculate the impacts of SLO violation based on a model provided in this thesis modelling language. It checked the impacts' correctness by looking at the created cross-component issues in Gropius. The created issues were correct. Thus, hypothesis **(H2.1)** is accepted.

In conclusion, this thesis contributed something that works, but no one wants it. Yet, the concept is still appreciated. It would be even more appreciated without the saga pattern.

5.5 Threats to Validity

This section addresses the threats to this thesis validity. Runeson and Höst [RH09] differentiate between four aspects of validity: internal validity, external validity, construct validity and reliability. Figure 5.8 shows an overview over the identified threats, grouped by validity aspect. The Sections 5.5.1 to 5.5.4 discuss each aspect's threats and possible actions to diminish them.

5.5.1 Internal Validity

Internal validity is about the observed relations [RH09]. Things that were not considered in the study but still influence the relation between the variables under test threaten it. For the expert survey, a major threat are the experts themselves. They are human beings that have subjective

opinions. Also, the fact that this is only some thesis might bias them, such that they do not take the survey seriously. There is no countermeasure to the experts' humanity. To ensure the experts' seriousness, the survey relies on experts known to and trusted by us or our supervisor.

A threat to the experiment's internal validity is the used architecture. The experiment uses the t2-project references architecture to ensure a controllable environment that bears as few unwanted influences as possible. The t2-project has only a few services and additionally offers the possibility to trigger SLO violation on demand.

5.5.2 External Validity

External validity is about the generalisability of the result [RH09]. In the case of the expert survey, the number of participants was small, and in addition, all experts had a similar background. Thus, the study covers only a fraction of the entire population and leaves many groups unrepresented. This derogates the generalisability. The same argument applies to the experiment as well, as we executed it on one architecture only.

5.5.3 Construct Validity

Construct validity is about whether a study matches the intention of the executor [RH09]. The expert survey relied upon real human beings with a mind of their own. As such, the most imminent threat to the survey's construct validity is the human participants misunderstanding or misinterpreting the survey questions or any of the given instructions and explanations. As the survey consists of open questions, which means the participants could answer freely, it is also possible that the equally human surveyors misinterpret the participants' answers. To counter this, we did a trial run of the survey and afterwards consulted with the trial's participants about their understanding of the given instructions and questions. We also prepared a slide set for the introduction such that we do neither lose our train of thought nor forget about things that are important to mention.

For the experiment, the used measure appropriately represents the intention to grasp the prototype's correctness. Yet, there remain other threats to the experiment's construct validity, mostly the use of a fake instead of the real SoLOMON tool.

5.5.4 Reliability

Reliability is about a study's repeatability [RH09]. This means, if other researchers execute the same study, they ought to get the same results. The reliability is threatened if the survey's artefacts change or are no longer available to other researchers. The latter is countered by the description of the survey questions and the survey's instructions in this thesis and by making this thesis' prototype and editor available on GitHub. Furthermore, Gropius, SoLOMON and Prometheus are also publicly available. This thesis explicitly states the versions of all tools to counter possible changes in them.

6 Conclusion

This chapter concludes this thesis. Section 6.1 is a summary of what this thesis achieved by which means, and Section 6.2 is an outlook onto future work.

6.1 Summary

This thesis addressed the topic of tracing the impact of SLO violations on business processes across microservice architectures that implement the saga pattern. Its final goal became to calculate the violation's impacts and to present them to a user. Calculation, in this case, means stepping from the location of the SLO violation through the architecture and the pattern up to the process, thereby getting the path along which the violation may propagate to the business process, in other words, the trace. To achieve this goal, we designed a modelling language, which is the heart of this thesis and its first contribution. It reuses exiting meta-models as much as possible and adds new elements only where necessary. It imports the Gropius meta-model for the architecture and the BPMN meta-model for the business process. As such the actual contribution is the connections between the elements of the already existing meta-models. With this thesis' language it is possible to create a holistic model of architecture, business process and saga pattern, and to connect these elements such that the model provides all information necessary to calculate an SLO violation's impact on the business process. The language's meta-model is implemented using the EMF. A byproduct of this thesis is an Ecore version of the Gropius meta-model, as the format of the existing implementation was not compatible with the EMF. Furthermore, this thesis contributes a graphical editor to view and create instances of this thesis' meta-model. The editor is generated from the meta-model, and the graphical representation is refined with Sirius.

The second contribution of this thesis is the notification about the impacts of an SLO violation. The notification consists of the impact's root cause, which is the SLO violation itself, and the violation's trace. The trace is the path along which the SLO violation propagates to the business process. The first chain link of the trace is the impact on the interface, at which the SLO violation occurred, and the last chain link is the impact on the business process. In addition, this contribution includes a prototype to calculated the chain of impacts, based on a model of the system described in this thesis' modelling language. As the concept of this thesis is supposed to fit in with Gropius, this thesis realises the notifications as cross-component issue in Gropius. The trace is represented in JSON format, hidden within the markdown body of the issue. Links to the issues about the actual SLO violations represent the root cause of the impact.

Thirdly this thesis contributes an expert survey and an experiment. The expert survey evaluated the thesis contributions from a user's perspective.

The experiment proved that models in this thesis' modelling language can be used to trace an SLO violation's impact from the interface at which it happened up to its impact on the business process. However, despite the fact that it works, the provided solution is rather naive as it reports all impacts that might happen, which likely result in a lot of false positives.

6.2 Future Work

At first, there are some possible improvements regarding the graphical editor. The current implementation supports only the most basic elements of the BPMN. The support for the UML component diagram-like notation of the architecture is also incomplete. An editor that supports the full range of both notations might be a good improvement. However, at this point, it might be worth thinking about merging this thesis entire editor into an already existing editor to leverage that editors capabilities concerning the completeness of the graphical notations. Also, this thesis editor provides only the bare minimum of editing options. Everything beyond adding and editing sagas is missing. For this thesis that was a conscious decision, as editing the imported models requires updating the sources or else they get out of sync. However, it might be worth considering as it saves the users from switching to and fro between different editors for the business process, the architecture and the saga. Based on the feedback received from the experts, it might also be an improvement to drop Eclipse and Sirius all together and to recreate the editor from scratch and host it as a web application. Apparently, people do not like to install things on their machines.

Secondly, there are also ways to improve the notification. For now, the notification consists of the SLO violation that is the root cause and the trace from the location of the violation up to the process. This means the notification is only related to *one* SLO violations, namely its root cause. However, SLO violations do not exist in isolation. Thinking back to the t2-project (c.f. Figure 5.1), there might be one response time SLO rule at `CreditInstitute`'s and another one at `Payment`'s provided interface. Both are violated. It is very likely that these violations are related to each other, as an example because `CreditInstitute` violated its SLO rule so hard that due to the resulting delay, `Payment` violated its SLO rule as well. Despite the very likely relation of these two violations, this thesis would generate two entirely independent notifications. Both notifications would report impacts to the same task of the business process, and the user might still figure out that the notifications are related by themselves. However, it would be way more helpful if the notifications include the relations right away. At best, each impact of the trace would link to other SLO violations that happened at the impact's location. For example, the trace of the violation of the SLO rule at `CreditInstitute` passes through the `Payment` component and thus would link to the SLO violation at `Payment`.

Thirdly, one may improve the prototype. The prototype can handle violations at components and interfaces only, and the violations at interfaces are not yet tested beyond unit tests. This is because the current SoLOMON only allows for SLO rules at components. However, this thesis' concept allows for SLO violations everywhere, and at some point, the implementation should do that as well.

There are also some ways to improve this thesis meta-model. The first one is to remove the assumption of CRUD interfaces and to move on to more complex interfaces. This requires modelling interfaces at the operation level, such that the user can model exactly which operation of an interface is part of a saga.

The second way is to expand the range of patterns. The current meta-model only supports the saga pattern, as this was the focus of this thesis. However, it would be interesting to analyse how other patterns propagate SLO violations as well. For example, a retry might propagate the violation of an availability SLO as a delay because the component implementing the pattern retries until it gets an answer. Thus, what reached the business process is not an unavailable service, but only a very late answer.

The third way is to add adapters for all elements imported from other meta-models. This change bears by far the most possibilities for improvement. For now, this thesis meta-model imports elements from the Gropius and the BPMN meta-model as they are, which means that it can only use but not change them. However, once this thesis meta-model employs adapters, it may add new attributes and relations to the imported elements by adding them to the adapters. Thereby creating the opportunity to directly connect the architectural elements to the process elements, which was up to now not possible. All connections had to detour over the saga. For this thesis, as the saga pattern is its focus, that is all good. But the entire concept would be more useful if it could also cover cases without patterns. Or instead of adapters, one could plug a connector between the business process and the architecture, that is either nothing but a plain *realised by* respectively *realises* relationship between tasks and interfaces, or an entire pattern, such as the saga pattern.

Another advantage of adapters is that it is easier to import new meta-models or to swap the imported meta-models. If a user refuses to describe their business process with the BPMN and would rather like to stick to UML activity diagrams, one must only touch the adapters, such that they can also handle activities. The actual connections that are now realised between the adapters remain untouched.

On another note, adapters might also be useful when refining the interfaces to operations, as mentioned above. Without adapters, one must import a meta-model for the architecture that already supports the modelling of operations, whereas with adapters one might use any meta-model and model the operations at the adapters.

Another weakness of this thesis is that it reports all impacts an SLO violation might have. It cannot differentiate between a failure and a delay or between a failure or delay that propagates and one that does not. This might result in a high number of issues with a lot of false positives. To enact such differentiation, one must know about each component's behaviour upon a delay or failure. As example, there is a services `S` that consumes to different interface, `InterfaceA` and `InterfaceB`. However `S` consumes `InterfaceA` for personal reasons and only `InterfaceB` influences the result of the service. Thus, an SLO violation at `InterfaceA` may not propagate through `S`, as its result is still correct. And if a tool knew about this behaviour, it would be able to not report violations that do not propagate to the business process.

One more suggestion is this: This thesis' prototype operates in a predictive fashion, which is okay because that is all it can do with the information it is provided with (the models and the alerts). Though if the prototype is provided with execution traces, for example, recorded with Jaeger[1] or similar tools, it might not only report that there may be an impact on the business process but it might also report, which past or current transactions were or are affected by an SLO violation. Just in case anyone would be interested in that, e.g. to compensate their customers.

[1] https://www.jaegertracing.io/

Concerning all things mentioned above, the overall point is: the more detailed information you put into the model, the more detailed is the result you get.

Bibliography

[ACD+07] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu. *Web Services Agreement Specification (WS-Agreement)*. Tech. rep. Open Grid Forum, Mar. 2007. URL: https://www.ogf.org/documents/GFD.107.pdf (cit. on p. 11).

[AGT16] L. Aversano, C. Grasso, M. Tortorella. "Managing the alignment between business processes and software systems". In: *Information and Software Technology* 72 (2016), pp. 171–188. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.12.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584915002189 (cit. on p. 14).

[CMS14] E. F. Cruz, R. J. Machado, M. Y. Santos. "From Business Process Models to Use Case Models: A Systematic Approach". In: *Advances in Enterprise Engineering VIII*. Ed. by D. Aveiro, J. Tribolet, D. Gouveia. Cham: Springer International Publishing, 2014, pp. 167–181. ISBN: 978-3-319-06505-2 (cit. on p. 14).

[DD08] D. Dyachuk, R. Deters. "Ensuring Service Level Agreements for Service Workflows". In: *2008 IEEE International Conference on Services Computing*. Vol. 2. 2008, pp. 333–340. DOI: 10.1109/SCC.2008.117 (cit. on p. 13).

[DY03] J. Dong, S. Yang. "Visualizing design patterns with a UML profile". In: *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*. 2003, pp. 123–125. DOI: 10.1109/HCC.2003.1260215 (cit. on p. 15).

[ecl13] eclipse. *Eclipse BPMN2 Modeler User Guide (Version 1.0.1)*. Tech. rep. 2013. URL: https://www.eclipse.org/bpmn2-modeler/documentation/BPMN2ModelerUserGuide-1.0.1.pdf (cit. on p. 9).

[EPJH10] B. Elvesæter, D. Panfilenko, S. Jacobi, C. Hahn. "Aligning Business and IT Models in Service-Oriented Architectures Using BPMN and SoaML". In: *Proceedings of the First International Workshop on Model-Driven Interoperability*. MDI '10. Oslo, Norway: Association for Computing Machinery, 2010, pp. 61–68. ISBN: 9781450302920. DOI: 10.1145/1866272.1866281. URL: https://doi.org/10.1145/1866272.1866281 (cit. on p. 15).

[FOW17] *Microservices Resource Guide*. 2017. M. Fowler. URL: http://martinfowler.com/microservices (cit. on p. 5).

[GCD+17] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, A. Di Salle. "Towards Recovering the Software Architecture of Microservice-Based Systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Apr. 2017, pp. 46–53. DOI: 10.1109/ICSAW.2017.48 (cit. on p. 16).

[GHJV15] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. Deutsch. Ed. by M. Feilen, K. Lorenzen. 1. Aufl. Frechen: mitp, 2015, 472 Seiten. ISBN: 978-3-8266-9700-5 (cit. on p. 15).

[GS87] H. Garcia-Molina, K. Salem. "Sagas". In: *SIGMOD Rec.* 16.3 (Dec. 1987), pp. 249–259. ISSN: 0163-5808. DOI: 10.1145/38714.38742. URL: https://doi.org/10.1145/38714.38742 (cit. on pp. 6, 12).

[HSS05] A. Hanemann, D. Schmitz, M. Sailer. "A framework for failure impact analysis and recovery with respect to service level agreements". In: *2005 IEEE International Conference on Services Computing (SCC'05) Vol-1*. Vol. 2. 2005, 49–56 vol.2. DOI: 10.1109/SCC.2005.10 (cit. on p. 13).

[HZ12] T. Haitzer, U. Zdun. "DSL-Based Support for Semi-Automated Architectural Component Model Abstraction throughout the Software Lifecycle". In: *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '12. Bertinoro, Italy: Association for Computing Machinery, 2012, pp. 61–70. ISBN: 9781450313469. DOI: 10.1145/2304696.2304709. URL: https://doi.org/10.1145/2304696.2304709 (cit. on p. 16).

[HZWC05] X. Huang, S. Zou, W. Wang, S. Cheng. "MDFM: Multi-domain Fault Management for Internet Services". In: *Management of Multimedia Networks and Services*. Ed. by J. Dalmau Royo, G. Hasegawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 121–132 (cit. on pp. 12, 13).

[KL] B. Korherr, B. List. "Aligning Business Processes and Software Connecting the UML 2 Profile for Event Driven Process Chains with Use Cases and Components." In: Citeseer (cit. on p. 14).

[KUSM18] M. Kleehaus, Ö. Uludag, P. Schäfer, F. Matthes. "MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments". In: June 2018, pp. 148–162. ISBN: 978-3-319-92900-2. DOI: 10.1007/978-3-319-92901-9_14 (cit. on pp. 13, 15, 16).

[Men02] D. Menasce. "QoS issues in Web services". In: *IEEE Internet Computing* 6.6 (2002), pp. 72–75. DOI: 10.1109/MIC.2002.1067740 (cit. on p. 6).

[MHG02] D. Mapelsden, J. Hosking, J. Grundy. "Design Pattern Modelling and Instantiation using DPML". In: (Sept. 2002) (cit. on p. 15).

[ML83] C. Mohan, B. Lindsay. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions". In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. PODC '83. Montreal, Quebec, Canada: Association for Computing Machinery, 1983, pp. 76–88. ISBN: 0897911105. DOI: 10.1145/800221.806711. URL: https://doi.org/10.1145/800221.806711 (cit. on p. 6).

[Moo09] D. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779. DOI: 10.1109/TSE.2009.67 (cit. on pp. 25, 26).

[MZ08] A. Mohamed, M. Zulkernine. “On Failure Propagation in Component-Based Software Systems”. In: *2008 The Eighth International Conference on Quality Software*. 2008, pp. 402–411. DOI: 10.1109/QSIC.2008.46 (cit. on p. 13).

[New15] S. Newman. *Building Microservices*. 1st. O’Reilly Media, Inc., 2015. ISBN: 1491950358 (cit. on p. 5).

[OK03] M. Odeh, R. Kamm. “Bridging the gap between business models and system models”. In: *Information and Software Technology* 45.15 (2003). Special Issue on Modelling Organisational Processes, pp. 1053–1060. ISSN: 0950-5849. DOI: https://doi.org/10.1016/S0950-5849(03)00133-2. URL: https://www.sciencedirect.com/science/article/pii/S0950584903001332 (cit. on pp. 13, 14).

[OMG07] OMG. *Business Process Modeling Notation (BPMN) Specification, Version 1.0*. Tech. rep. Object Management Group, Mar. 2007. URL: http://www.omg.org/spec/BPMN/2.0.2 (cit. on p. 13).

[OMG13] OMG. *Business Process Model and Notation (BPMN), Version 2.0.2*. Tech. rep. Object Management Group, Dec. 2013. URL: http://www.omg.org/spec/BPMN/2.0.2 (cit. on pp. 8, 9, 22, 26).

[OMG17] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5.1*. Tech. rep. Object Management Group, Aug. 2017. URL: http://www.omg.org/spec/UML/2.5.1 (cit. on pp. 10, 15, 26, 27).

[RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. English. In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8 (cit. on pp. 50, 51).

[Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. ISBN: 9781617294549 (cit. on pp. iii, 1, 6, 7, 15, 19, 40).

[SBB20] S. Speth, U. Breitenbücher, S. Becker. “Gropius—A Tool for Managing Cross-component Issues”. In: *Communications in Computer and Information Science*. Ed. by H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Camporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, U. Zdun. Vol. 1269. Springer. Springer, 2020, pp. 82–94 (cit. on pp. 3, 10, 29).

[SBB21] S. Speth, S. Becker, U. Breitenbücher. “Cross-Component Issue Metamodel and Modelling Language”. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. INSTICC. SciTePress, May 2021, pp. 304–311. ISBN: 978-989-758-510-4. DOI: 10.5220/0010497703040311. URL: https://www.scitepress.org/PublicationsDetail.aspx?ID=Vtgn8x557mU=&t=1 (cit. on pp. 10, 22, 23).

[SBCR02] R. van Solingen (Revision), V. Basili (Original article 1994 ed.), G. Caldiera (Original article 1994 ed.), H. D. Rombach (Original article 1994 ed.) “Goal Question Metric (GQM) Approach”. In: *Encyclopedia of Software Engineering*. American Cancer Society, 2002. ISBN: 9780471028956. DOI: https://doi.org/10.1002/0471028959.sof142. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471028959.sof142. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof142 (cit. on p. 39).

[SBPM11] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *Eclipse modeling framework: EMF*. Englisch. Ed. by D. S. Steinberg. 2. ed., rev. and updated, 2. printing. The eclipse series. UB Vaihingen. Upper Saddle River, NJ ; Munich [u.a.]: Addison-Wesley, 2011, XXIX, 704 Seiten. ISBN: 9780321331885 (cit. on p. 8).

[ŠCR19] M. Štefanko, O. Chaloupka, B. Rossi. "The Saga Pattern in a Reactive Microservices Environment". In: Jan. 2019, pp. 483–490. DOI: 10.5220/0007918704830490 (cit. on p. 7).

[SLW21] I. Sailer, R. Lichtenthäler, G. Wirtz. "An Evaluation of Frameworks for Microservices Development". In: *Advances in Service-Oriented and Cloud Computing*. Ed. by C. Zirpins, I. Paraskakis, V. Andrikopoulos, N. Kratzke, C. Pahl, N. El Ioini, A. S. Andreou, G. Feuerlicht, W. Lamersdorf, G. Ortiz, W.-J. Van den Heuvel, J. Soldani, M. Villari, G. Casale, P. Plebani. Cham: Springer International Publishing, 2021, pp. 90–102. ISBN: 978-3-030-71906-7 (cit. on pp. 7, 8).

[Som16] I. Sommerville. *Software engineering*. Englisch. Ed. by I. Sommerville. 10. ed., global ed. Always learning. UB Vaihingen. Boston ; Munich [u.a.]: Pearson, 2016, 810 Seiten. ISBN: 9781292096131 (cit. on p. 17).

[SVC06] T. Stahl, M. Voelter, K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley and Sons, Inc., 2006. ISBN: 0470025700 (cit. on p. 8).

[VAES12] J. Vizcarrondo, J. Aguilar, E. Exposito, A. Subias. "ARMISCOM: Autonomic reflective middleware for management service composition". In: *2012 Global Information Infrastructure and Networking Symposium (GIIS)*. 2012, pp. 1–8. DOI: 10.1109/GIIS.2012.6466760 (cit. on p. 13).

[WBP09] M. Wang, K. Y. Bandara, C. Pahl. "Integrated Constraint Violation Handling for Dynamic Service Composition". In: *2009 IEEE International Conference on Services Computing*. 2009, pp. 168–175. DOI: 10.1109/SCC.2009.31 (cit. on p. 13).

[Wes19] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Englisch. 1 Online-Ressource (XVII, 417 p. 311 illus). Berlin, Heidelberg, 2019. URL: https://doi.org/10.1007/978-3-662-59432-2 (cit. on p. 8).

[WXM+18] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, P. Chen. "CloudRanger: Root Cause Identification for Cloud Native Systems". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2018, pp. 492–502. DOI: 10.1109/CCGRID.2018.00076 (cit. on p. 13).

All links were last followed on September 27, 2021.

A Experiment Supplementals

```
groups:
- name: provider-alerts
  rules:
  - alert: provider_avail_avg_slo
    expr: 'avg_over_time(probe_success[1m]) <= 0.9'
    labels:
      severity: error
    annotations:
      summary: "Bad Availability on {{ $labels.instance }} /  {{ $labels.target }}"
      value: "{{ $value }}"
      solomonId: "5e25b2a4-6b5d-4317-95e7-6a8245226e64"

  - alert: provider_respT_avg_slo
    expr: 'avg_over_time(http_server_requests_seconds[10s]) >= 10'
    for: 10s
    annotations:
      summary: "High request latency on {{ $labels.instance }}"
      value: "{{ $value }}"
      solomonId: "67cc8ce2-3f78-4122-ad36-d9640efdde40"
```

Listing A.1: Alerts for Prometheus used in the experiment.

Tool	Repository / Version (git commit)
Gropius Back-end	https://github.com/ccims/ccims-backend-gql 123381d602a53241d270256b471874578cd621be
Gropius Front-end	https://github.com/ccims/ccims-frontend 80b5a43e20a4c1d7b96b462ade2abb0bef1ace16
SoLOMON	https://github.com/ccims/solomon d1861e603a122b23c959511a35a5c6668deadb7f
Thesis' prototype	https://github.com/stiesssh/ma-backend f87dc5ccf7875461eb9e8769331fb09c58f54b04
Thesis' editor	https://github.com/stiesssh/ma-models cbfe1a01bbffb31ae6525c78e07668d8995c61da
fakesolomon	https://github.com/stiesssh/ma-fakesolomon ac7d339aa01b16e736ea9dadc7dd73e698281948

Table A.1: Repositories used by the experiment.

Tool	Version
Prometheus	`2.26.0`
Alertmanager	`0.21.0`
prometheus-community/prometheus-blackbox-exporter helm chart	`5.0.3`

Table A.2: Tools used by the experiment.

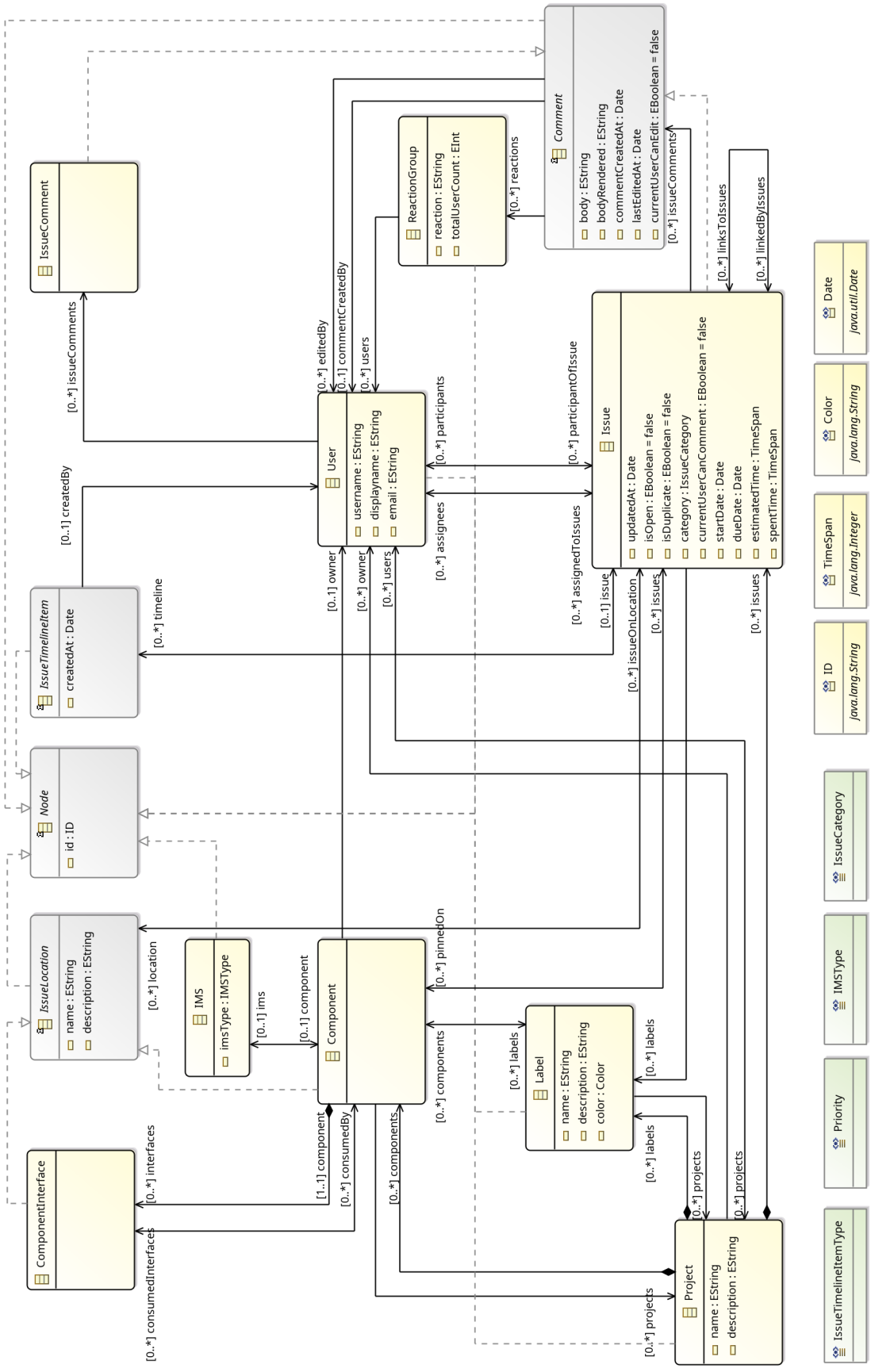
B JSON Schema

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "notification",
    "description": "a notification",
    "type": "object",
    "properties": {
        "impactlocation" : {
            "title": "impactlocation",
            "description": "location of the top level impact",
              "type" : "object",
            "properties": {
                "name" : {
                    "description": "name of the impacted location",
                      "type" : "string"
                },
                  "id" : {
                      "description": "id of the impacted location",
                      "type" : "string"
                }
            },
            "required": ["name", "id"]
          },
        "violatedrule" : {
            "title": "violatedrule",
            "description": "violated slo rule",
              "type" : "object",
            "properties": {
                "name" : {
                    "description": "name of violated slorule",
                      "type" : "string"
                },
                  "id" : {
                      "description": "id of the violated slorule",
                      "type" : "string"
                }
            },
            "required": ["name", "id"]
          },
        "impactpath": {
            "type": "array",
            "items": {
                "title": "location",
                "description": "location that is part of the impact path",
                "type": "object",
```

```
                "properties": {
                    "id" : {
                        "description": "id of the location",
                          "type" : "string"
                    },
                    "name" : {
                        "description": "name of the location",
                          "type" : "string"
                    },
                    "type" : {
                        "description": "type of the location",
                          "type" : "string"
                    },
                    "container" : {
                        "title": "container",
                        "description": "container of the location",
                        "type": "object",
                        "properties": {
                            "id" : {
                                "description": "id of the location container",
                                  "type" : "string"
                            },
                            "name" : {
                                "description": "name of the location container",
                                  "type" : "string"
                            },
                            "type" : {
                                "description": "type of the location container",
                                  "type" : "string"
                            }
                        },
                        "required": ["name", "id", "type"]
                    },
                      "cause" : {
                          "description": "id of location that caused this location to be
impacted",
                          "type" : "string"
                    }
                },
                "required": ["name", "id", "type", "container"]
            },
            "uniqueItems": true
        }
    },
    "required": ["impactlocation", "violatedrule", "impactpath"]
}
```

Listing B.1: JSON schema for impacts.

C Gropius Substitute



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