Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Identifying Autoscaling Antipatterns at Design Time using LLM-based Explanation Generation

Jonas Lammert

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Floriment Klinaku, M.Sc., Sarah Stieß, M.Sc.
Commenced:	October 1, 2024
Completed:	April 1, 2025

Abstract

Context. Cloud-based systems offer unique elasticity in the number of computational resources they provide. When modeling these systems, architects face the challenge of engineering elasticity to determine how to manage resource allocation. Too few resources lead to unreliable service, while too many result in unnecessary expenses. To address this, autoscaling policies can be employed to automatically adjust resource availability based on predefined conditions.
Problem. Engineering elasticity is a complex task. When designing autoscaling policies, many antipatterns [SEK+23] can emerge, which are difficult to detect at design time. A system is needed that integrates simulation data to guide architects toward problematic components in their model that may lead to antipatterns.
Objective. To facilitate this process, previous work [Hah23] proposed the development of a graphical editor that provides visual feedback from simulation data. Building on this foundation, we enhanced the system with textual explanations generated by LLMs to guide developers toward a deeper understanding of the SPD-model and aid in identifying antipatterns.
Method. We extended existing implementations of a graphical editor for SPD-models that provide visual feedback on the design. A new feature was introduced that leverages large language models to generate textual explanations as feedback. We conceptualized different types of explanations a user might seek when working with SPD-models. Additionally, we designed templates for generating prompts that request these explanations from the LLM. Finally, we evaluated the quality of the generated explanations to assess their potential in helping developers identify and resolve antipatterns at design time while gaining a deeper understanding of the SPD-model.
Result. We delivered a functional prototype that enhances previous work with our newly implemented explanation feature. Additionally, we evaluated the quality of the generated explanations, justifying their integration into the editor. We demonstrated that the explanations are of sufficient quality to assist architects in understanding SPD-models and identifying antipatterns at design time, although there is still room for improvement.
Conclusion. Experts rated most kinds of explanations with a promising rate of usefulness. This implies that architects could benefit from the explanations in identifying antipatterns, creating fixes, and gain an overall deeper understanding of the policies. However, the explanations also have problems with imprecisions and lack of conciseness in some cases. Fine-tuning the prompts, contriving new types of explanations, and advancements in LLM-technology could help reduce these problems.

Kurzfassung

Kontext Cloudbasierte Systeme bieten eine einzigartige Elastizität in der Anzahl der bereitgestellten Rechenressourcen. Bei der Modellierung dieser Systeme stehen Architekten vor der Herausforderung, die Elastizität so zu gestalten, dass die Ressourcenzuweisung effizient verwaltet wird. Zu wenige Ressourcen führen zu einem unzuverlässigen Dienst, während zu viele unnötige Kosten verursachen. Um dies zu adressieren, können Autoskalierungs Policies eingesetzt werden, die die Ressourcenzuweisung basierend auf vordefinierten Bedingungen automatisch anpassen.
Problemstellung Die Gestaltung der Elastizität ist eine komplexe Aufgabe. Beim Entwurf von Autoskalierung Policies können viele Antipatterns [SEK+23] auftreten, die zur Entwurfszeit schwer zu erkennen sind. Es wird ein System benötigt, das Simulationsdaten integriert, um Architekten auf problematische Komponenten in ihrem Modell hinzuweisen, die zu Antipatterns führen könnten.
Ziele Um diesen Prozess zu erleichtern, wurde in früheren Arbeiten [Hah23] die Entwicklung eines grafischen Editors vorgeschlagen, der visuelles Feedback aus Simulationsdaten liefert. Aufbauend auf dieser Grundlage erweitern wir das System um textuelle Erklärungen, die von LLMs generiert werden, um Entwicklern ein tieferes Verständnis des SPD-Modells zu ermöglichen und sie bei der Identifikation von Antipatterns zu unterstützen.
Vorgehensweise Wir haben bestehende Implementierungen eines grafischen Editors für SPD-Modelle erweitert, der visuelles Feedback zum Entwurf liefert. Eine neue Funktion wurde eingeführt, die Large Language Modelle nutzt, um textuelle Erklärungen als Feedback zu generieren. Wir haben verschiedene Arten von Erklärungen konzipiert, die ein Nutzer beim Arbeiten mit SPD-Modellen benötigen könnte. Zusätzlich haben wir Vorlagen für die Erstellung von Eingabeaufforderungen (Prompts) entwickelt, die das LLM zur Generierung dieser Erklärungen verwenden kann. Abschließend haben wir die Qualität der generierten Erklärungen evaluiert, um deren Potenzial zur Unterstützung von Entwicklern bei der Identifikation und Behebung von Antipatterns zur Entwurfszeit sowie zur Vertiefung des Verständnisses des SPD-Modells zu bewerten.
Ergebnisse Wir haben einen funktionalen Prototyp entwickelt, der frühere Arbeiten durch unsere neu implementierte Erklärungsfunktion erweitert. Zusätzlich haben wir die Qualität der generierten Erklärungen evaluiert und ihre Integration in den Editor gerechtfertigt. Wir konnten zeigen, dass die Erklärungen eine ausreichende Qualität aufweisen, um Architekten bei der Analyse von SPD-Modellen und der Identifikation von Antipatterns zur Entwurfszeit zu unterstützen, auch wenn weiterhin Verbesserungsmöglichkeiten bestehen.
Fazit Die Experten bewerteten die meisten Erklärungen mit einer vielversprechenden Nützlichkeitsrate. Dies bedeutet, dass Architekten von den Erklärungen profitieren könnten, wenn es darum geht, Antipatterns zu identifizieren, Korrekturen zu erstellen und ein insgesamt tieferes Verständnis der Policies zu erlangen. Allerdings haben die Erklärungen auch Probleme mit Ungenauigkeiten und mangelnder Prägnanz in manchen Fällen. Eine Feinabstimmung der Prompts, die Entwicklung neuer Arten von Erklärungen und Fortschritte in der LLM-Technologie könnten dazu beitragen, diese Probleme zu verringern.

Contents

List of Figures

List of Tables

Listings

Acronyms

LLM Large Language Model. iii, v, 1, 6, 7, 8, 9, 10, 13, 17, 20, 21, 22, 23, 27, 31, 32, 33, 34, 37, 39, 41, 42, 43, 45, 46, 47

PCM Palladio Component Model. ix, 21, 22, 30, 31, 32, 33, 34

SLI Service Level Indicator. 7, 41

SLO Service Level Objective. 7, 13, 17, 18, 19, 21, 22, 24, 26, 29, 38, 40, 41, 42, 45

SPD Scaling Policy Definition. iii, v, ix, 1, 3, 4, 5, 18, 21, 22, 26, 29, 30, 31, 32, 33, 34, 37, 38, 43, 45, 47

1 Introduction

In modern times, cloud systems have become commonplace. As the demand for computation resources ever increases, cloud systems offer an elegant solution, because they are, in theory, capable of handling workloads of every size. Because of how easy it is to add or remove resources, cloud-based systems are ideal for handling the highly fluctuating workloads of, for example, online services. However, this confronts architects with a new challenge: how to manage the amount of resources available? To let software architects design such elasticity, autoscaling policies were created [KHB21]. An architect can define certain triggers, circumstances on which a policy should scale the resources. For example, the CPU usage exceeds a certain threshold. If these fire, the scaling policy will automatically adjust the available resources.

This is a strong tool for designing systems that can adapt to fluctuating workloads. However, these policies are not easy to design. There are several antipatterns that can occur if the architect is not careful. These antipatterns were defined by Martin Straesser et al. [SEK+23], and their existence will lead to incorrect scaling. For example, if the policy scales down some available resources, this could happen too fast, which would lead to insufficient resources to run the service. This will lead to our service working slowly, or failing completely, which inevitably will lead to dissatisfied customers that we are very likely to lose. On the other hand, if the policy scales down too slowly, we will end up with a lot more resources than we need. These resources are not free, so we would generate a lot of costs we could have avoided. As these examples illustrate, we should be motivated to avoid these antipatterns to the best of our abilities.

The contribution of this thesis is to design a system that can give feedback and explanations for scaling policies, to help architects to spot, identify and potentially fix occurring antipatterns at design time.

Work on this has been done by Hahn [Hah23], who envisioned an enhancement to a graphical editor for scaling policies created by Summerer [Sum22]. The editor was extended with a new view that is able to display feedback from simulations onto the SPD-model, the graphical representations of the scaling policies. For example, critical regions where antipatterns may have occurred are marked in red, and the user is able to open graphs that describe how this element of the policy acted during the simulation. We aim to enhance this concept further, giving a new kind of feedback. We want to enhance the prototype by integrating an LLM-based explanation component. The user should be able to get a textual explanation of why a certain element was marked as a critical region. For example, why did a certain trigger not fire? They should also be provided with explanations on every aspect of the SPD-model that may cause questions. We believe this will help architects to gain a deeper understanding of their SPD-models and identify antipatterns more easily.

We want to evaluate if LLMs are capable of generating explanations of a suitable quality to be helpful for users. For this we will introduce methods and prompt templates to prompt LLMs to generate the desired kinds of explanations. Then we let experts rate them based on quality criteria, to understand if they would be suitable for the task. We will also provide the first implementation of a prototype that integrates the explanations directly into the graphical editor. In the end, we expect to show that the feedback features of our new editor will have the potential to help architects to find and identify antipatterns at design time.

Thesis Structure

The thesis is structured as followed:

Chapter 2 – Foundations and Related Work: In this chapter we summarize the foundations that are relevant to our topic. This includes previous work on the graphical editor we want to enhance, as well as concepts we need to understand for our new approach. We also briefly introduce some related works in the field of explanation generation.

Chapter 3 – Requirements Engineering This chapter describes our process of requirement engineering for enhancements to the editor.

Chapter 4 – Concept: In this chapter we work out the concepts for enhancing the editor with a focus on the new feature and prompt engineering.

Chapter 5 – Implementation: In this chapter we describe the implementation of our first prototype that incorporates the explanation generation feature and some of the suggested enhancements to previous work. We also briefly discuss the limitations this prototype has.

Chapter 6 – Evaluation: This chapter describes the process, the results and the discussion of the quality evaluation we conducted on the explanations.

Chapter 7 – Conclusion: This chapter concludes the thesis and provides an outlook on future work.

2 Foundations and Related Work

In this section we will introduce the necessary foundations that are required to understand the thesis.

2.1 Foundations

Scaling Policy Definition Language

The Scaling Policy Definition (SPD) language is a language to model scaling policies at an architectural level [KKSB23][Sli]. It can be used to define scaling policies, which contain all information on how a certain target is scaled.

A scaling policy consists of a trigger that fires when certain conditions are fulfilled (e.g. CPU utilization exceeds a defined threshold), an adjustment type, which determines how the target group is scaled when a trigger is fired, and a finite number of constraints, which could, for example, be a cooldown period. Every scaling policy references a target group, which is a structure of elements where the adjustments defined in the policy are carried out. Those target groups can also have constraints. For example, the number of running instances can have upper/lower bounds. Figure 2.1 shows the overview of the meta-model of the SPD language.

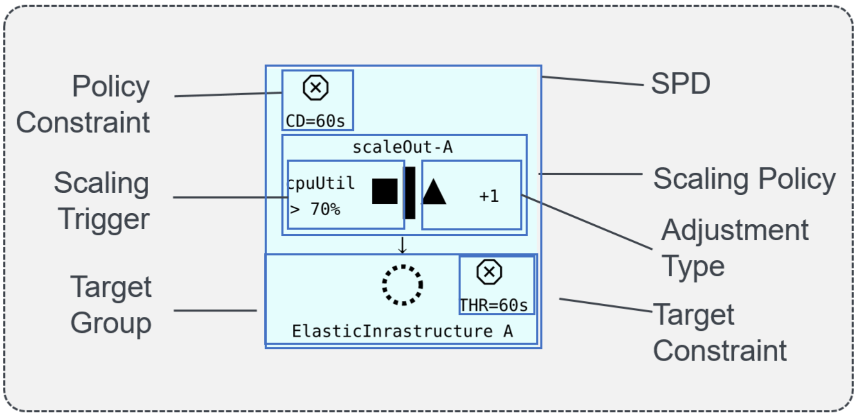


Figure 2.1: Conceptual overview of the meta-model and the key elements behind the SPD language. [KKSB23]

Palladio

Palladio [RBB+11][Pal] is a software architecture simulation approach with which the user can analyze software systems at model level for all kinds of criteria. We will use it to run simulations on cloud systems we create the scaling policies for. With Palladio, we can simulate user interactions on a software architecture without the need to implement it first. We only need to model the architecture and dictate the desired user behavior to be able to run simulations. Palladio can then output all sorts of simulation data, like Ram-usage or HDD-usage during the course of the simulation.

Slingshot

Slingshot [KKB21][Sli] is an approach that enhances the Palladio simulator with the ability to engineer and simulate elasticity policies for cloud-native applications. It is based on event-driven architecture [BD10] which makes it more easily extensible and allows users to define scaling policies via the SPD-language and simulate their impact on the cloud system.

Eclipse Sirius

The tool we use to develop the editor in this thesis is Eclipse Sirius [Sir], a framework built on the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF). Sirius enables developers to create custom graphical modeling workbenches. Building such a workbench requires two steps. First, the metamodel has to be defined. It establishes the structural foundation of the editor and is specified in an Ecore file. This file follows an XML-based schema and defines the core elements, relationships, and constraints of the model. It dictates how components interact and ensures consistency across instances of the model.

Next we need to define viewpoints. They are defined in an odesign file (an XML-like format), determine the Graphical User Interface (GUI) and associated functionalities of the editor. A single odesign file can house multiple viewpoints, enabling different UI configurations while maintaining a shared underlying metamodel. Within a viewpoint, the layout and behavior of the policy template are specified, including the placement of model elements in the palette—a UI component that allows users to drag and drop elements into the model canvas. Sirius provides a tree-based configuration editor, allowing developers to structure model elements hierarchically and define additional behaviors.

For data retrieval and model queries, Sirius utilizes the Acceleo Query Language (AQL), which allows dynamic interaction with model elements. Additionally, developers can extend Sirius functionality using Java by implementing service methods in a Services.java file. These methods can be invoked within the modeling environment to enhance element behavior and introduce custom logic.

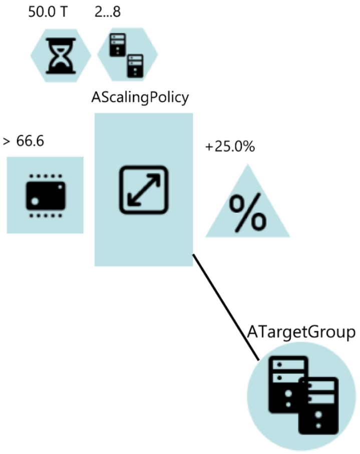


Figure 2.2: Example of a scaling policy created in the graphical editor of Summerer.

Graphical Editor by Summerer

In his Bachelor Thesis *"Graphical Editors for Defining Scaling Policies Analysable Using Simulations"* [Sum22], Summerer designs and implements a graphical editor for SPDs. In the editor, one can define target groups and assign scaling policies to them. These policies come as a template with placeholders for a scaling trigger (in square form), adjustment types (in triangle form) and constraints (in hexagon form), which the user can fill with the desired type via drag and drop.
A finished scaling policy consists of exactly one trigger, one adjustment and zero or more constraints. An example of such a policy can be seen in Figure 2.2. Here the trigger fires once the RAM-usage exceeds 66.6%, the RAM is then increased by 25% of its current value. The policy has two constraints. First, a cooldown, which prevents the trigger from firing more than once every 50 steps. Second, a constraint that limits the size of the target group to between 2 and 8 instances.

Enhancements by Hahn

In his Bachelor Thesis "*Enriching Graphical Editors for Scaling Policy Definitions with Feedback from Simulations"* [Hah23], Hahn aims to enhance the editor by making it able to utilize simulation data generated by Palladio simulations to provide visual feedback.

He created a new view for the editor called Simulation Results View (SRV), where this visual feedback is provided. Additionally, the user can click certain elements to open graphs generated from the simulation data. These could visualize simulation results like utilization, restraint behavior, or trigger firing behavior. These visualizations are built to help the user identify antipatterns.

For example, in Figure 2.3 the triggers that do not fire are marked with the color red. The user can click on these triggers to open several graphs, as in Figure 2.4. These can then be analyzed to identify which antipattern occurred.

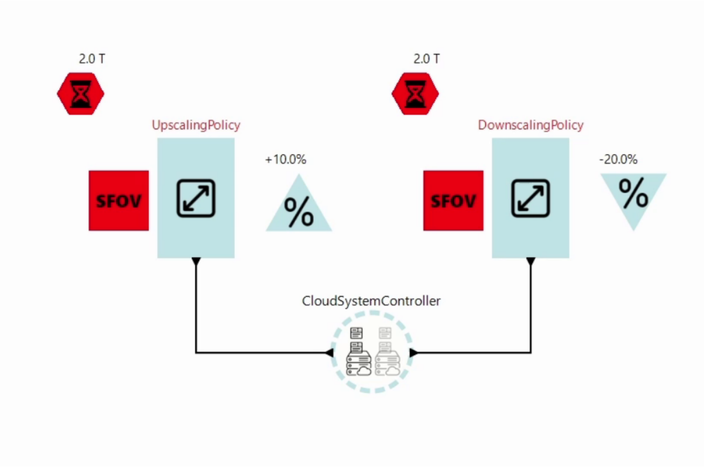


Figure 2.3: Example for marking triggers that do not fire.

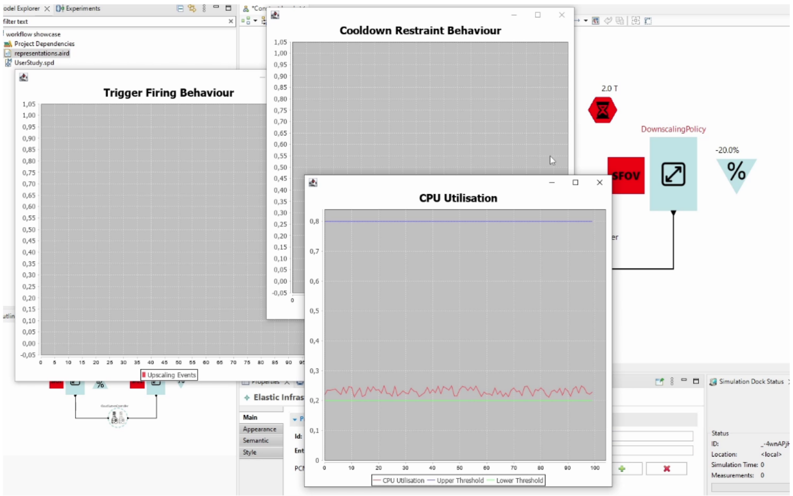


Figure 2.4: The Simulation Results View offers several graphs to analyze.

Integrating explanations to Palladio

Another previous work our thesis is build upon is the integration of explanation into the Palladio tool chain by Haas [Haa23]. He implemented an extension to Palladio's Slingshot simulator that can be dynamically configured to give explanations in different forms as required. In contrast to the approach we offer, these explanations are not generated by an LLM. Instead, it provides a framework with which experts can define cases, which are event chains or patterns that require explanations. If the occurrence of such a case is monitored, it takes the occurring case and its computed values and creates a machine-readable explanation. This explanation will then be transmitted to the interpreter. In Haas implementation, he had a simple console-based interpreter that turns the explanation into a human-readable output. It will give freely definable but fixed explanation texts inside the console. This explanation text, just as the cases, needs to be predefined by experts, compared to our approach,

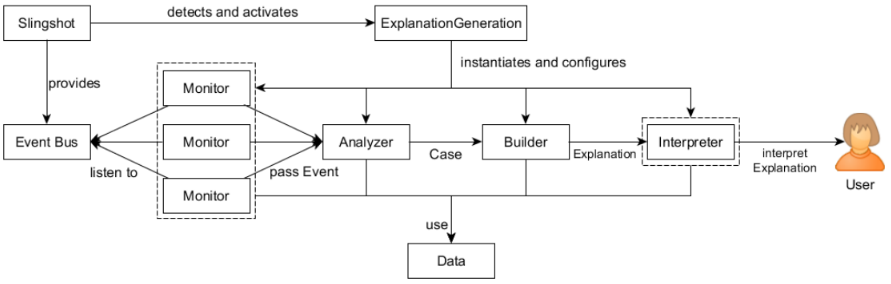


Figure 2.5: Architectural overview of the explanation generation component by Haas.

where we leave the analysis, and the formulation of the explanations to the LLM. To get a better grasp of Haas' explanation generation component we included his architecture overview in Figure 2.5.

Antipatterns

Straesser et al. [SEK+23] put together a collection of antipatterns that can occur in autoscaling policies. The paper mentions seven possible antipatterns and explains potential reasons for each of them. These are:

- 'No reaction', no scaling occurs even when load increases or decreases.
- 'Wrong steady state provisioning', the load stays constant, and no scaling occurs, but there are still Service Level Objective violations occurring.
- 'Rapid upscaling', Service Level Objective violations occur because too many instances are added on load increase and the load stays constant afterwards.
- 'Slow upscaling', autoscaler does not add enough instances fast enough, which leads to Service Level Objective violations.
- 'Rapid downscaling', load decreases and stays constant afterward but the autoscaler removes too many instances to fast.
- 'Slow downscaling', load decreases, but the autoscaler does not remove enough instances, leading to unnecessary costs.

A failure to avoid these antipatterns will often lead to Service Level Objective (SLO) violations.

Service Level Objective (SLO)

Service Level Objectives are the constraints a (cloud) service has to fulfill to be considered properly working [JWNS]. These constraints are defined for metrics such as failure rates or throughput of a service, the so-called Service Level Indicator (SLI). They are defined inside Service Level Agreements (SLA), along with consequences for meeting or missing them (mostly financial in nature).

Neural Networks

We will use Large Language Model (LLM) to generate our textual explanations. To be able to grasp that concept it is important to get a general understanding of what Neural Networks are and how they work. A neural network is a computational model inspired by the human brain, designed to recognize patterns and learn from data [Nie15]. It consists of artificial neurons, which process and transmit information through interconnected layers. These neurons are organized into three types of layers:

There is an input layer, hidden layers, and an output layer. When data enters a neural network, it first reaches the input layer, which simply forwards the raw information to the hidden layers. The hidden layers contain multiple neurons that apply mathematical transformations to uncover meaningful patterns in the data. Finally, the output layer provides the network's final prediction.

The neural network achieves its capability of learning by being able to continuously adjusting the aforementioned weight and bias parameters during its training, until the desired outcome is archived. In this case weights are numerical values that represent the strength of connections between neurons in different layers. They do control how much influence a particular input has on the next layer. Biases are constant values that are added to each neuron once it receives inputs from all other neurons connected to it. They allow more flexibility in learning. Knowing about these two parameters we can now describe the output of a neuron in the hidden layer with the following formula:

$z = \sum(x_i * w_i) + b$

where x_i are the inputs, w_i the respective weights, and b is the bias.

This weighted sum is then passed through an activation function, which introduces non-linearity, allowing the network to learn complex patterns beyond simple linear relationships. Since the computations within the hidden layers are not directly observable, they are called hidden layers, contributing to the perception of neural networks as black boxes.

Large Language Model

Building from the last section, we can consider Large Language Model (LLM) to be very large neural networks trained on vast amounts of text. These models adjust millions or even billions of weights and biases—referred to as parameters in this context—allowing them to generate coherent, context-aware responses [Wol23]. These models are advanced AI systems capable of understanding, generating, and manipulating human language. Modern LLMs can generate text that is often indistinguishable from human writing.

The key to achieving this impressive capability lies in the training data and the training process. For example, GPT (Generative Pre-trained Transformer) models are trained on massive, diverse datasets—including books, articles, and scientific papers—collected from various sources on the internet. This data spans a wide range of topics, writing styles, and knowledge domains.

To understand how an LLM transforms this data into a system that can generate human-like responses, we can imagine the model analyzing large text corpora. It examines the context in which words (or more precisely, tokens—words broken into smaller components) appear, learning statistical relationships between them. From these relationships, the model constructs a high-dimensional semantic space, where each token is represented as a numerical vector (embedding vector). Words or phrases that appear in similar contexts are positioned closer together in this space, while unrelated words are farther apart.

Knowing the concept of these vectors, we can now understand how the LLM takes a user's input, the so-called prompt, and generates a response in an almost human-like language. First, the prompt will be tokenized , meaning split into parts that the LLM can then convert into the embedding vectors we discussed before. Depending on the tokenizer used, a token can be a word, a part of a word, or even a character. After the tokenized prompt is converted into embedding vectors, it is further processed via multiple layers of a transformer model. In this step, mechanisms like self-attention and positional encoding help determine word relationships and sequence structure.

After all this processing is done, it is finally time to generate the output. To explain it in simple terms, the model does generate its responses by predicting which token is most likely to succeed the last token. This probability is learned from the training data by recognizing patterns in token sequences and their positions in the embedding space. To avoid repetitive or robotic responses, LLMs don't always select the highest-probability token. Instead, sampling techniques like top-k sampling or nucleus sampling (top-p) allow for controlled randomness, improving fluency and creativity.

The model continues generating tokens until it reaches a stopping criterion. This could either be reaching a special end-of-sequence (EOS) token, reaching the predefined maximum length, or encountering repetitive patterns. Finally, the output is converted back into human-readable text, cleaned up, and, if necessary, formatted based on what is requested in the prompt.

2.2 Related work

In this section we will introduce some related works that also used LLM-based explanation generation. Although they had different use cases, we can still take inspiration from how the implementation was realized.

LLM-based Explanation Generation

A lot of work is being done investigating how Large Language Models can be used to generate explanations for different purposes. There is, for example, the entire field of XAI, explainable artificial intelligence, which aims to provide AI-systems that have the ability to explain individual steps of their decision process that lead to their output. Just as in our project, many of these systems provide their generated explanations in natural language, mimicking how humans tend to explain their decisions. This is said to heighten efficiency and coverage (in terms of audience) when analyzing these explanations [CMM+23].

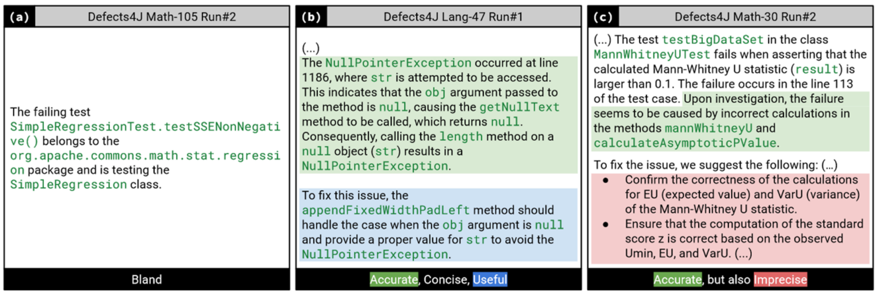
There are also various other use cases, where the LLM generation of explanation is explored. Reif et al. [RQWK24] introduces a system that uses LLMs to automatically analyze unstructured datasets. With its ability to summarize the datasets and identify outliers and interesting slices of data, it eases the process of understanding the unstructured data. Lubos et al. [LTF+24] use LLM explanation generation to explain why a recommender system recommends products to a user via natural language explanations.

Another big related field of study is using LLM-based explanation generation for code understanding. The ability of LLMs to explain code is well known [MTM+22], and its use is studied in educational and practical contexts. Jury et al. [JLL+24] used LLM to generate worked examples in an introductory programming course. These worked examples illustrate the process for solving a problem step-by-step by providing explanations for every step individually. The provided program is able to generate explanations for every step and also single out keywords inside them, which the user can click to receive more detailed explanations regarding that keyword in its context. In an evaluation conducted with students in a computer classroom, it was concluded that the students found the generated explanations overall valuable, especially with the additional interactive component, the clickable keywords. Another work that uses LLMs to help with code understanding is the work done by Nam et al. [NMH+24], which builds a prototype of an in-IDE LLM information support tool called GILT (Generation-based Information-support with LLM Technology). The user can highlight parts of the code and then trigger the tool. GILT provides a generated summary of the selected code. The user can then request deeper explanations, generated with preset prompts, for example on domain-specific concepts, or usage examples. We can use this prototype as inspiration. In our program, the user should also be able to highlight specific parts of the policy and request explanations about them.

The subfield of LLM-based explanation generation our work is most related to is using it for fault localization. There are several works that investigate the usage of LLMs to generate explanations on where and how a fault occurred and how to potentially fix it nowadays. For example, Das et al. [DBC21] introduced a system that can generate explanations for the cause of an unexpected failure of a robot's plan execution. They are in natural language and aim to be written such that even non-experts should be able why errors happened and how the problem can potentially be fixed. There are, of course, many works that investigate the usage of LLM-based explanation generation to explain faults in software. Leinonen et al. [LHS+23] uses LLMs to generate explanations for programming error messages, which are often unclear, especially for novices. Their evaluation showed that their approach is feasible and shows promise. However they did not find much success with suggesting correct fixes. These were only correct 47% of the time. Similarly, Liu et.al [LTM+24] used LLMs to automatically analyze online logs, which helps comprehend a program's status and take appropriate action.

One work that is of special interest to us is an evaluation of LLM-based explainable fault localization by Kang et al. [KAY24]. They proposed an LLM-based fault localization technique called AutoFL, that can generate explanations of a bug in a software, along with a suggested fault location. Especially interesting for us is how they rated explanation quality. The explanations were rated on four criteria:

- Accuracy: the explanation describes why the bug occurs in detail.
- Imprecision: the explanation contains at least one inaccurate statement.
- Conciseness: explanations are succinct and do not contain extraneous content.
- Useful: the explanation correctly explains how the bug can be fixed.



(a) Defects4J Math-105 Run#2

The failing test `SimpleRegressionTest.testSSENonNegative()` belongs to the `org.apache.commons.math.stat.regression` package and is testing the `SimpleRegression` class.

Bland

(b) Defects4J Lang-47 Run#1

(...)
The `NullPointerException` occurred at line 1186, where `str` is attempted to be accessed. This indicates that the `obj` argument passed to the method is `null`, causing the `getNullText` method to be called, which returns `null`. Consequently, calling the `length` method on a `null` object (`str`) results in a `NullPointerException`.

To fix this issue, the `appendFixedWidthPadLeft` method should handle the case when the `obj` argument is `null` and provide a proper value for `str` to avoid the `NullPointerException`.

Accurate, Concise, Useful

(c) Defects4J Math-30 Run#2

(...) The test `testBigDataSet` in the class `MannWhitneyUTest` fails when asserting that the calculated Mann-Whitney U statistic (`result`) is larger than 0.1. The failure occurs in the line 113 of the test case. Upon investigation, the failure seems to be caused by incorrect calculations in the methods `mannWhitneyU` and `calculateAsymptoticPValue`.

To fix the issue, we suggest the following: (...)

- Confirm the correctness of the calculations for EU (expected value) and VarU (variance) of the Mann-Whitney U statistic.
- Ensure that the computation of the standard score z is correct based on the observed Umin, EU, and VarU. (...)

Accurate, but also Imprecise

Figure 2.6: Example of how the evaluation criteria of Kang et al. [KAY24] were applied to different explanations.

They would also rate explanations as bland, if they described the test and the covered method but did not offer additional analysis. To see an example of how these criteria were applied to the explanations, see Figure 2.6. These criteria can be useful for evaluating the quality of our explanations. We need to aim for explanations that are accurate, useful, and concise, while being as free of imprecisions as possible.

2.2.1 Literature Research Methodology

We conducted our literature research mainly using Google Scholar [Goo], to find relevant papers for our topic. If the paper was considered useful, we also did a forward- and backward search. Meaning we looked at the works' bibliography and at later papers that reference it, to potentially find further work that is related to our topic. We used the following search terms:

- explanation generation
- llm explanation generation
- explanation prompts
- llm fault localization
- llm fault explanation
- llm explanation evaluation

When a paper's title seemed promising, we first scanned over the abstract and introduction, to verify if the work is related and relevant to our topic. If they still appeared promising, we further judged them according to their key chapters like results and conclusion.

3 Requirements Engineering

This section addresses the process of gathering and prioritizing the requirements our prototype should fulfill.

3.1 Requirements Source

Because we are building atop of two previous works, which were both evaluated, we already have a solid source for requirements there. Both works collected user feedback in their evaluation study. Participants in these studies included students, members of the SQA department of the University of Stuttgart, and even experts from the industry. All these participants can be said to have an interest in the finished prototype, so we will consider them our stakeholders in the following, and use their feedback and ideas to extract requirements for the editor and especially the feedback view component.

To collect requirements for the generation explanation feature, we consider works we discussed in section 2.2 as inspiration. Their discussion on what explanations should entail and how they can be presented is used to extract requirements for our new component.

3.2 Requirements definition

After extracting the requirements from user feedback and literature, we divided them into 3 categories: requirements for the visual feedback, requirements for the simulation result graphs, and requirements for the LLM-generated textual explanations. In this work we do not consider non-functional requirements, like, for example, performance or reliability. This would exceed the scope of this thesis, in which we only want to build a prototype to evaluate the concepts. However, if one ever wants to advance our prototype into a full-fledged product, non-functional requirements should also be considered.

For the visual feedback of the simulation result view, we extracted the following requirements from the user study:

- Req.1: The prototype should be able to display graphical elements (e.g. exclamation marks) next to problematic elements in addition to coloring.
- Req.2: The prototype should use a more colorblind-friendly color-scheme for its visualizations.
- Req.3: The prototype should be able to visualize the extent of an SLO violation.
- Req.4: The prototype should be able to gray out policies that were not triggered.

- Req.5: The prototype should be able to mark elements where no problems exist as non-problematic.
- Req.6: Users should be able to access a summary view.
- Req.7: During the simulation the prototype should be able to provide dynamic visualizations, so the user can see the visualizations change.
- Req.8: The prototype should provide a timeline, with which the user can retrace how the visualizations have changed.

For the simulation result graphs, our extracted requirements are as follows:

- Req.9: Users should be able to toggle manually which data of which elements are added to the graphs.
- Req.10: Graphs should have better representation of data (e.g. dotted lines, transparent bars for cooldown periods)
- Req.11: Users should be provided with a concise list of all possible result graphs.
- Req.12: The prototype should allow users to pin windows.
- Req.13: The prototype should be able to automatically analyze the graph (where did the antipattern occur? Where is the problem source?)

Finally, these are the requirements we got for the textual explanations feature:

- Req.14: The prototype should be able to give a textual explanation of what the modeled scaling policy does.
- Req.15: The explanation should give tips on how the antipattern could be fixed.
- Req.16: In addition to the visual feedback, the prototype should also textually explain which parts of the policy is problematic.
- Req.17: The explanations should be concise and not contain any extraneous content.
- Req.18: The explanation should indicate at what point in time the problem occurred during the simulation.
- Req.19: The user should be able to ask further questions if the initial explanation did not suffice

3.3 Requirements Prioritization

To decide which requirements are most important to fulfill, we decided to create an online form and send it to people who may be interested in the prototype to let them rank the requirements. Inspired by DuMouchels approach of measuring satisfaction [DuM93] we let them rank the requirements on a scale from -2 to 2. The numbers mean the following:

-2 ”I dislike it (I do not want it to be fulfilled)”

-1 ”I tolerate it”

0 ”I am neutral”

1 ”I expect it”

2 ”I like it (above expectation)”

We also took the opportunity to ask the participants for their input on how they want to request the explanations and how they should be displayed. For a detailed discussion of these questions, see Chapter 4.

To compile all the answers to this requirements’ prioritization survey, we set up a *Zenodo* repository [Lam25]. We will now discuss these answers in detail.

Our survey was answered by eight people. To better interpret the results, we asked them for their current job title or occupation, how they would rate their experience with self-adaptive systems, and whether they had previously used autoscalers; if so, which ones. Of the participants, three were master’s students in computer science, three were researchers/research-assistants, and two were industry professionals working as a software architect and an embedded software engineer, respectively. Participants rated their experience with self-adaptive systems and autoscaling on a scale from 1 to 5, where 1 means no experience, 2 beginner-level experience, 3 intermediate-level experience, 4 advanced-level experience, and 5 expert-level experience, indicating they could train others. We managed to get answers from people with a wide variety of experience. We got at least one participant for every experience level, including two at the beginner level and two at expert level. Five participants had used autoscalers before. The technologies mentioned include AWS Auto Scaling Groups, Kubernetes (specifically the Horizontal Pod Autoscaler), HPA, VPA, KEDA, CAUS and Microsoft Azure’s built-in autoscaler for SaaS services. One participant also used a previous prototype of the autoscaling tool we are developing.

The requirements for the visual feedback of the simulation result view (Req.1–Req.8) were ranked as follows: The highest-ranked requirements were Req.6 and Req.3, with the same mean score of 1.38, followed by Req.4, with a mean score of 1.25. Req.2, with a mean score of 1.13, was the final requirement to achieve a score of 1 or higher. Next, we had Req.8 with a mean score of 0.75, followed by Req.1 and Req.7 with the same mean score of 0,5. The lowest rated requirement, and the only one with a negative mean score, was Req.5, with a mean score of -0.25. One participant mentioned that the timeline of Req.8 could be very useful if it includes where violations occurred, and how and why a resource was scaled. So the timeline should be less about how the visualization changed, but more about what happened in the simulation.

The requirements for the simulation result graphs (Req.9-Req.13) were ranked as follows: Both Req.9 and Req.13 were rated the highest with a mean score of 1.38, followed by Req.10 and Req.12, both with a mean score of exactly 1. The only requirement that scored lower than 1 was Req.11, which had a mean score of 0.53. One participant suggested that the prototype should also allow users to change the colors of the data representation, which would be especially useful for color-blind users. Another participant stated how much they would dislike multiple windows being opened. Instead, they suggested that everything could be done in one window, with toggles and drop-down menus. They would also prefer to have multiple graphs in one window simultaneously.

Finally, the requirements for the textual explanations feature (Req.14-Req.19) were ranked as follows: The highest ranking requirement was Req.17 with a mean score of 1.5, followed by Req.18 with a mean score of 1.38 and both Req.15 and Req. 16 with the same mean score of 1.25. Req.14 got a mean score of exactly 1. Only one of the requirements was rated lower than 1, Req.19, with a mean score of 0.63.

To summarize our results, we will categorize the requirements into four categories: undesired, which includes all requirements with a mean score below 0; nice to have, which includes requirements with a mean score greater than 0 but less than 1; important, which includes requirements with a mean score between 1 and 1.3; and essential, which includes requirements with a mean score above 1.3. Looking at the mean scores above, we categorize the requirements as follows:

- undesired: Req.5
- nice to have: Req.1, Req.7, Req.8, Req.11, Req.19
- important: Req.2, Req.4, Req.10, Req.12, Req.14, Req.15, Req.16
- essential: Req.3, Req.6, Req.9, Req.13, Req.17, Req.18

For the implementation, this means that we will prioritize the essential requirements. We also aim to implement all the important requirements, provided that technical expertise and time constraints allow it. If time permits, we may implement some of the nice-to-have requirements, but these will most likely be left for future work. The one requirement categorized as undesired will, of course, not be implemented.

4 Concept

In this chapter, we develop the concept for our new prototype. Aside from discussing some minor changes to the editor, we mostly focus on the new explanation generation feature and how the prompts for it are designed. We also design the kinds of explanations we seek in this chapter, by analyzing some use cases and imagining which information the user may seek in that situation.

4.1 Changing the Editor

Hahn's previous concept incorporated visual feedback within the editor and provided the option to open simulation graphs in separate windows. In our concept, we plan to retain the visual feedback while relying more on Palladio's existing features for simulation graphs. Developing an entirely new method for generating simulation graphs would be redundant. Moreover, Palladio's experiment view already includes useful features that address several requirements outlined in Section 3.

RQ10 is addressed; Palladio's experimentation view allows users to select their preferred graph type. If they are unsatisfied with the initial representation, they can simply choose another. This also addresses RQ12. Starting a new experiment run does not delete the simulation results of the previous run. Users can keep old simulation graphs open and easily compare them with the results of the new run.

By integrating these existing Palladio features, we establish a new workflow, as illustrated in Figure 4.1.

In this new workflow, we first model the scaling policies as usual. Next, monitors must be created to specify which system variables should be measured by Palladio and what simulation result data will be recorded. This step could potentially be automated, as the policy model already suggests which specific measuring points would be relevant for simulation. For instance, if the architect defines a trigger that fires when CPU utilization exceeds a certain threshold, it is highly likely that they want to measure CPU utilization during the simulation. Once the necessary monitors are set up, the simulation is executed.

Similar to Hahn's concept, the user then opens the simulation result view in the editor. To satisfy RQ3, parts of the policies that may have led to SLO violations are highlighted using color coding. The severity of an SLO violation is visually represented by color intensity: the deeper the color, the more severe the violation. If no elements are marked, the policy does not cause any SLO violations, and the architect can finalize the design. If problematic elements are marked, the user can double-click them to prompt the program to request a textual explanation from the LLM. The explanation will describe why the SLO violation occurred and suggest how it can be resolved. The user can then open corresponding graphs in the Palladio experiment tab to analyze the results and verify the accuracy of the textual explanation. If necessary, they can open additional graphs for

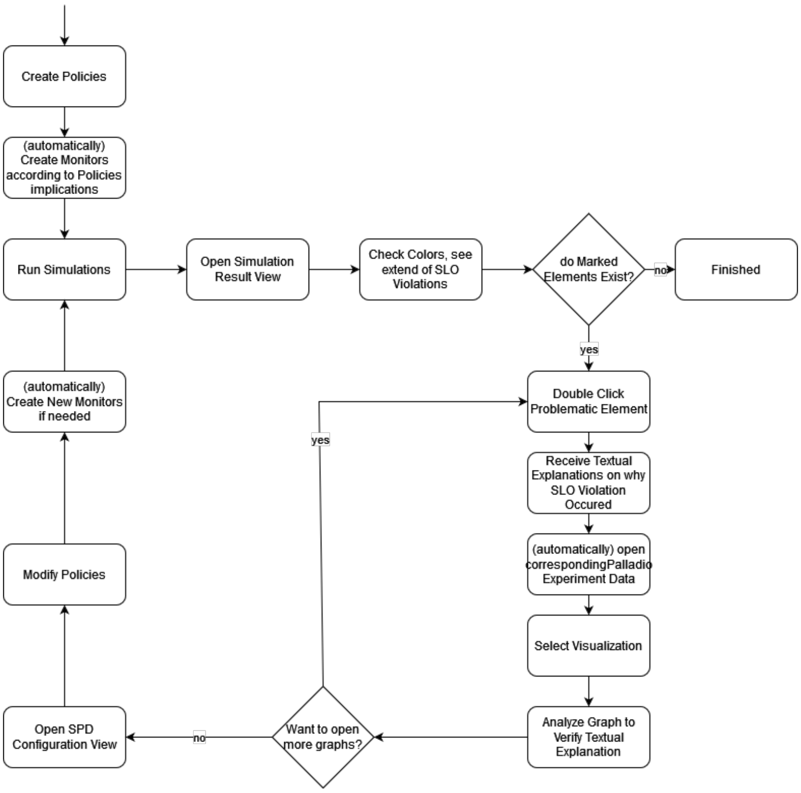

Create Policies
(automatically) Create Monitors according to Policies implications
Run Simulations
Open Simulation Result View
Check Colors, see extend of SLO Violations
do Marked Elements Exist?
no
Finished
yes
Double Click Problematic Element
Receive Textual Explanations on why SLO Violation Occured
(automatically) open correspondingPalladio Experiment Data
Select Visualization
Analyze Graph to Verify Textual Explanation
Want to open more graphs?
yes
no
Open SPD Configuration View
Modify Policies
(automatically) Create New Monitors if needed

Figure 4.1: The new workflow using our suggested concept.

further investigation. If changes are needed, the user returns to the SPD modeling view to modify the policy and attempt to fix the issue. The process of running simulations and analyzing results in the simulation result view continues until no more marked elements exist, ensuring that the policy meets the required SLOs.

4.2 Prompt Design

This section will outline the process of how we designed the prompt we request the explanations with, in detail.

4.2.1 Kinds of Explanations

To provide a comprehensive understanding of scaling policies, it is essential to consider the specific information users seek in different scenarios. The following use cases outline key concerns and the necessary explanations required to address them.

Use Case 1: Successful scaling without errors. When scaling functions as expected and the feedback view reports no errors, users may seek information on the following aspects: The purpose and functionality of the applied policies. Which specific policy was triggered, when it was activated, and why? Detailed explanations regarding scaling policies and their associated trigger behaviors.

Use Case 2: SLO violation with identified problematic elements. In instances where an SLO violation occurs and the feedback view highlights problematic elements, users require deeper insights beyond those outlined in Use Case 1. These include: Understanding which policies were triggered, which were not, and the reasons behind these outcomes. Identifying which kind of SLO violation occurred when and to what extent. Explanation of why certain elements were marked as problematic. Guidance on resolving antipatterns that lead to the SLO violations, including potential corrective actions. Verification of whether the correct metric is being measured, and if not, recommendations on the appropriate metric to use.

To address these concerns effectively, a structured explanation should be provided for each marked element. This explanation should do the following: Describe the reason for the element being flagged as problematic. Identify any antipatterns that may have contributed to the issue. Specify which SLO was violated and to what extent. Offer actionable recommendations for resolving the issue.

Use Case 3: Testing individual policies during policy creation. When users test individual policies during their creation process, they require insights that extend upon Use Case 1. Specifically, they need to understand: The defined SLO and the necessary policies required to achieve it. An overview of which policies are already applied and which are still needed to meet the SLO requirements. Potential risks posed by existing policies, particularly if they introduce antipatterns that may lead to unintended SLO violations.

Analyzing these use cases led us to four types of explanations the user can request, depending on which part of the policy they seek information for:

1. Scaling Policy Explanations – Explains the functionality of the policy and the conditions under which it operates.
2. Trigger Behavior Explanations – Details when and why a specific policy trigger was activated and helps users understand the logic and timing behind trigger events.
3. Problem & Solution Explanations – Clarifies why a specific component of a policy has been flagged as problematic, identifies the relevant antipattern and explains the underlying cause, and provides concrete suggestions for resolving the issue.
4. Target Group & SLO Explanations – Describes the SLO the target group should adhere to, outlines which scaling policies are currently attached and identifies any gaps, and assesses potential risks from existing policies that may lead to SLO violations.

By structuring explanations in this manner, users can gain a clear understanding of scaling policies, their impact on system behavior, and the necessary steps to maintain compliance with SLOs.

4.2.2 Prompt Structure

In general a well-structured prompt consists of four key components, according to Giray et. al. [Gir23]

1. Instruction – A clear directive that guides the model's behavior and specifies the task it should perform, ensuring alignment with the desired outcome.
2. Context – Relevant background information or external details that enhance the model's understanding, enabling it to generate more precise and contextually appropriate responses.
3. Input Data – The core question or problem statement that the model is expected to process and respond to, serving as the foundation of the prompt.
4. Output Indicator – A specification of the desired response format, such as a brief answer, a paragraph, or another structured output, which helps refine the model's response accordingly.

In the following, we will construct prompts for all our explanation types, component by component.

Instruction

To ensure that the LLM generates precise and relevant responses, the instruction component of each prompt must clearly define the task to be executed. To optimize instruction clarity and improve the effectiveness of the model's responses, we designed the instruction part of our prompt as followed:

Each prompt begins with a short preamble that establishes the context of the task and communicates the overall purpose of the interaction. The preamble serves to orient the LLM, ensuring it understands both its role and the broader problem domain.

The preamble we have designed is as follows:

"Your task is to assist self-adaptive system architects in designing scaling policies for elastic cloud systems that assure that the system always has enough resources to provide its service, but never too many, which would lead to unnecessary costs, by giving explanations and/or feedback on their scaling policy designs."

This preamble already conveys several critical pieces of information to the LLM. Its role is clearly defined, the LLM functions as an assistant to a self-adaptive system architect. Additionally, we also already clarify the objective of the user, to design scaling policies for elastic cloud systems. The general purpose of scaling policies, maintaining sufficient resources while avoiding excess capacity and unnecessary costs, is also conveyed. As well as a general description of the task, the LLM will provide explanations and feedback, though the specific nature of each explanation must be further specified within the instruction.

Aside from the preamble, each type of explanation requires more detailed instructions to ensure the response aligns with the specific needs of the user. These additional instructions are structured as follows:

For the Scaling Policy Explanation, we want it to describe the policy's impact on system resources, its activation conditions, and the adjustments it enforces. Therefore, we add the following to the instruction: *"Explain how the policy would affect the target resource during the simulation, under which circumstances would it trigger and to which kind of scaling would it lead?"*

Trigger Behavior Explanation should contain information on when and why a trigger event occurred, as well as how the system adapted. To specify the task for this kind of explanation, we add *"Explain the trigger behavior of this policy, how often and when it was triggered and in which way was the target resource adjusted."* to the instruction part.

For the Problem & Solution Explanation to evaluate potential anomalies within a policy, we include the following to the instruction: *"After simulating our system, we want to analyze our policy for anomalies, describe if this policy could have led to an anomaly in the simulation and, if yes, which antipattern could have occurred in the design process of the scaling policy, to which service level objective violation it led in the simulation and to what extent it was violated."* This ensures that the explanation identifies potential flaws, diagnoses the underlying causes, and assesses the impact of the issue on the system's service level objectives.

Finally, for the Target Group & SLO Explanation to examine how policies interact with a given target group and its associated SLOs, the following is added to the instruction: *"For this target group, explain which Service Level Objectives are defined, which scaling policies, if any, are attached to it and how they would affect the target group if triggered (could they potentially lead to problems?), and which scaling policies may be necessary in addition to upholding the Service Level Objectives."* This instruction is also useful for policies that are still under development, as it can provide guidance on missing policies necessary to meet SLOs. However, it is equally valuable for evaluating completed policy sets, as it can highlight whether any attached policies could introduce potential issues.

Through empirical testing, we determined that separating the prompt into two distinct components—a preamble and a task-specific instruction—significantly improves both the readability of the prompt and the quality of the LLM's responses. We put the preamble at the start of every prompt, while the task-specific instruction is best placed after the input data. This approach ensures a logical flow within the prompt, making it easier for the model to process the request effectively.

Context

For the LLM to generate meaningful and accurate explanations, it must have access to sufficient information about the SPD-model it is expected to explain, as well as the corresponding PCM. Ensuring the model receives this information in a structured manner improves the clarity and reliability of its responses.

To guide the LLM toward generating useful explanations, we determined that it must have a comprehensive understanding of the PCM. For our prototype, we opted to provide handcrafted textual descriptions of the relevant PCM components, leaving the automation of these descriptions or alternative methods for conveying this information for future work. Through experimentation, we identified the following essential elements that should be included in a PCM description: A brief description of the modeled system, including its purpose and how users interact with it. A description of the resources subject to scaling, along with their specifications (e.g., CPU processing rate), as found in the Resource Environment Model. A list of all methods that utilize the resource, including their resource demand, as specified in the Repository Model and its Service Effect Specification (SEFF) Models. An example of what such a PCM description can look like is included in chapter 5. This description is necessary for every type of explanation.

In addition to the PCM description, the LLM also requires specific details about the SPD-model it is analyzing. At a minimum, it must have access to the parameters of the policy being explained. This requirement applies to all types of explanations, while certain explanation types—such as Target Group & SLO explanations—require this information for every policy associated with a given target group. To standardize the information provided to the LLM, we developed a policy description template, which can be automatically populated by the program:

"The architect has defined a scaling policy as followed: The policy [POLICYNAME] [increases|reduces] the resource [TARGETGROUP] by [ADJUSTMENTVALUE] if [TRIGGERCONDITION] holds. It is restricted by [CONSTRAINTTYPE] of [CONSTRAINTVALUE]."

This structured description ensures that the LLM receives all relevant parameters required to accurately describe the function and behavior of each policy.

Certain explanation types require further details beyond the basic policy description: Problem & Solution Explanations and Target Group & SLO Explanations necessitate information regarding the SLOs of the system. The latter also require details about the target group itself, including its operational constraints. These constraints can be conveyed using the following template:

"The target group for our scaling policies is [TARGETGROUP]. The group has the following constraints: At least [MINSIZE] instances must always be running. Not more than [MAXSIZE] instances are allowed to run at the same time."

For simplicity, our prototype does not include additional constraints such as thrashing constraints; however, it is straightforward to extend this template to accommodate such constraints if needed.

For Problem & Solution Explanations, it is crucial that the LLM correctly identifies and names known antipatterns that may occur in scaling policy design. During testing, we observed that without explicit guidance, the LLM sometimes created inaccurate or misleading terms for antipatterns, even when it could describe them correctly. To prevent this, we explicitly list all the possible antipatterns in the prompt:

"There are seven possible antipatterns: No reaction (neither up nor downscaling performed), Wrong steady state provisioning (after scaling action, no more scaling actions but SLO violations) Jitter (No steady state is reached, repeated scaling), Rapid upscaling, rapid sownscaling, slow upscaling, slow downscaling."

Including this predefined set of antipatterns led the LLM to correctly reference them in testing, improving the reliability of its responses.

Input data

Not all explanation types require input data beyond the SPD itself. Specifically, Scaling Policy Explanations and Target Group & SLO Explanations focus solely on the structural aspects of the SPD. These explanations do not require simulation data, as they describe the policies as defined without

Explanation Type	**Context**	**Input Data?**
Scaling Policy Explanation	Parameters of own Policy	No
Trigger Behavior Explanation	Parameters of own Policy	Yes
Problem & Solution Explanation	Parameters of own Policy, SLOs, kinds of antipatterns	Yes
Target Group & SLO Explanations	Target Group Restrictions SLOs, Parameters of all connected Policies	No

Table 4.1: Overview which explanation types need which context and if they need simulation data as input.

considering their real-time behavior. Trigger Behavior Explanations and Problem & Solution Explanations, however, rely on simulation data to assess how policies function in simulation. The LLM requires this data to analyze policy execution, detect trigger behavior, and identify potential scaling anomalies or antipatterns.

To enable the LLM to effectively describe trigger behavior and recognize potential antipatterns, we provide it with two key types of data: Simulation data, ideally data that includes metrics directly linked to the policy's trigger condition. For instance, if a policy is configured to activate when average CPU utilization exceeds a threshold, then providing the LLM with CPU utilization measurements from the simulation is ideal. We also observed that the LLM can still generate reasonable explanations using indirectly related metrics, such as response time. Even when CPU utilization data was unavailable, the LLM inferred possible trigger conditions based on response time variations, demonstrating its ability to analyze secondary indicators.

Additionally, to assess how the system responded to triggers, we provide data on the number of active instances in the elastic infrastructure over time. This information allows the LLM to determine when upscaling or downscaling actions occurred, enabling it to analyze whether policies triggered as expected and whether scaling behavior aligned with the system's requirements.

One should also include a brief description of the data format within the prompt. This is essential to ensure the LLM correctly interprets the provided data. For example, simulation data is frequently represented as tuples (x, y), where x represents the timestamp of the measurement, and y represents the measured value (e.g., CPU utilization percentage, response time in milliseconds, or the number of running instances). Clearly specifying this structure in the prompt helps the LLM accurately parse and analyze the data.

Depending on the use case, we can imagine that the simulation data can get very large. This, paired with the fact that LLMs usually have token limits, begs the question if we may need to do some preprocessing, and what limitations our approach may face. We will discuss this in more detail in chapter 5.

As we can see, the prompts differ heavily in which context they need, and if they require input data, depending on the kind of explanation we want. Table 4.1 gives an overview of which context is required and if input data is needed, for every kind of explanation.

Output Indicator

To enhance the usability of our prototype, our explanations must be concise and to the point. Given that they will be integrated into an editor interface, potentially within a pop-up window, space is limited. Also, users should not have to sift through unnecessary details to find relevant information. To enforce brevity and maintain clarity, we include the following instructions at the end of every prompt: *”Answer in natural language, keep the answer short and concise, keep it under 400 characters.”* Through experimentation, we determined that 400 characters is a suitable length; long enough to convey all necessary details while short enough to prevent excessive or redundant information. For Target Group and SLO explanations, we increased the limit to 600 characters, because it needs to include explanations for more than one scaling policy.

Problem & Solution Explanations require a clearer structure to remain comprehensible. Without guidance, responses can become disorganized. To address this, we provide explicit formatting instructions:

> *”Structure your answer as followed: Explanation on which SLO violation occurred at which time and to what extent, explanation of which antipattern could be the cause and why the policy design led to it, suggestion on how to fix the problem. Answer in natural language, keep the answer short and concise, keep it under 400 characters.”*

This structured approach ensures that the explanation remains logical, easy to follow, and informative, covering the problem, its cause, and a potential solution within a compact format.

4.2.3 Prompt Templates

In the end, we produced 4 templates for our 4 different explanation types. They can be seen in Listings 4.1, 4.2, 4.3, and 4.4.

```
Your task is to assist self-adaptive system architects in designing scaling policies for
elastic cloud systems that assure that the system always has enough resources to provide its
service, but never too many, which would lead to unnecessary costs, by giving explanations and
/or feedback on their scaling policy designs.

[PCM DESCRIPTION]

The architect has defined scaling polices as followed:
The policy [POLICYNAME] [increases|reduces] the resource [TARGETGROUP] by [ADJUSTMENTVALUE] if
 [TRIGGERCONDITION] holds. It is restricted by [CONSTRAINTTYPE] of [CONSTRAINTVALUE].

Explain how the policy would affect the target resource during the simulation, under which
circumstances would it trigger and to which kind of scaling would it lead?

Answer in natural language, keep the answer short and concise, keep it under 400 characters.
```

Listing 4.1: Scaling Policy Explanation Prompt Template

```
  Your task is to assist self-adaptive system architects in designing scaling policies for
elastic cloud systems that assure that the system always has enough resources to provide its
service, but never too many, which would lead to unnecessary costs, by giving explanations and
/or feedback on their scaling policy designs.
```

```
  [PCM DESCRIPTION]

  The architect has defined scaling policys as followed:
  The policy [POLICYNAME] [increases|reduces] the resource [TARGETGROUP] by [ADJUSTMENTVALUE]
if [TRIGGERCONDITION] holds. It is restricted by [CONSTRAINTTYPE] of [CONSTRAINTVALUE].

  Explain the trigger behavior of this policy, how often and when it was triggered and in
which way was the target resource adjusted.

  [DATA FORMAT DESCRIPTION]

  [INPUT DATA]

  Answer in natural language, keep the answer short and concise, keep it under 400 characters.
```

Listing 4.2: Trigger Behavior Explanation Prompt Template

```
  Your task is to assist self-adaptive system architects in designing scaling policies for
elastic cloud systems that assure that the system always has enough resources to provide its
service, but never too many, which would lead to unnecessary costs, by giving explanations and
/or feedback on their scaling policy designs.

  [PCM DESCRIPTION]

  The architect has defined scaling policys as followed:
  The policy [POLICYNAME] [increases|reduces] the resource [TARGETGROUP] by [ADJUSTMENTVALUE]
if [TRIGGERCONDITION] holds. It is restricted by [CONSTRAINTTYPE] of [CONSTRAINTVALUE].

  After simulating our system, we want to analyze our policy for anomalies, describing if this
 policy could have led to an anomaly in the simulation and, if yes, which antipattern could
have occurred in the design process of the scaling policy, to which service level objective
violation it led in the simulation and to what extent it was violated.
  There are seven possible antipatterns: No reaction (neither up nor downscaling performed),
Wrong steady state provisioning (after scaling action, no more scaling actions but SLO
violations), Jitter (No steady state is reached, repeated scaling), Rapid upscaling, rapid
sownscaling, slow upscaling, slow downscaling.

  [DATA FORMAT DESCRIPTION]

  [INPUT DATA]

  Structure your answer as followed: Explanation on which SLO violation occurred at which time
 and to what extent, explanation of which antipattern could be the cause and why the policy
design led to it, suggestion on how to fix the problem. Answer in natural language, keep the
answer short and concise, keep it under 400 characters.
```

Listing 4.3: Problem & Solution Explanation Prompt Template

```
  Your task is to assist self-adaptive system architects in designing scaling policies for
elastic cloud systems that assure that the system always has enough resources to provide its
service, but never too many, which would lead to unnecessary costs, by giving explanations and
/or feedback on their scaling policy designs.
```

```
  [PCM DESCRIPTION]

  The target group for our scaling policies is [TARGETGROUP], the group has the following
constraint:
  (At least [MINSIZE] instances must always be running.)?
  (Not more than [MAXSIZE] instances are allowed to run at the same time.)?
  The following Scaling Policies are already attached to the target group:
  (The policy [POLICYNAME] [increases|reduces] the resource [TARGETGROUP] by [ADJUSTMENTVALUE]
 if [TRIGGERCONDITION] holds. It is restricted by [CONSTRAINTTYPE] of [CONSTRAINTVALUE].)*

  For this target group, explain which Service Level Objectives are defined, which scaling
policies, if any, are attached to it and how they would affect the target group if triggered (
could they potentially lead to problems?), and which scaling policies may be necessary in
addition to upholding the Service Level Objectives.

  Answer in natural language, keep the answer short and concise, keep it under 400 characters.
```

Listing 4.4: Target Group & SLO Explanation Prompt Template

We used these templates for our evaluation. After conducting it and gathering feedback, we also came up with a few ideas with which we could still fine-tune them. These will be discussed in Chapter 6.

4.3 Integrating the Explanations

As mentioned in Section 3, we took the opportunity during the survey to gather participant input on the design of the generated explanation feature. Specifically, we asked two questions: how they would prefer to request explanations and where they would like to receive them.

To request textual explanations, we provided three options: Highlighting components of interest, asking questions directly via a chatbot, or automatically including explanations with result graphs.

Three participants preferred requesting explanations by highlighting components of interest, while another three preferred having explanations included with the result graphs. One participant favored asking questions directly via a chatbot, and another suggested implementing all three options simultaneously.

Since we decided to rely more on Palladio's existing features for result graphs—which unfortunately do not allow explanations to be displayed within the same window—we opted for the highlighting approach. In this method, users can request explanations by clicking on specific components they want to understand.

To further illustrate this interaction, refer to the use case diagram in Figure 4.2. The diagram shows that users can click on any part of the SPD to receive an explanation: Clicking on a trigger after a simulation run prompts the Trigger Behavior Explanation. Clicking on a scaling policy prompts the Scaling Policy Explanation. Clicking on a target group prompts the Target Group & SLO Explanation. Clicking on a marked element after a simulation prompts the Problem & Solution Explanation.

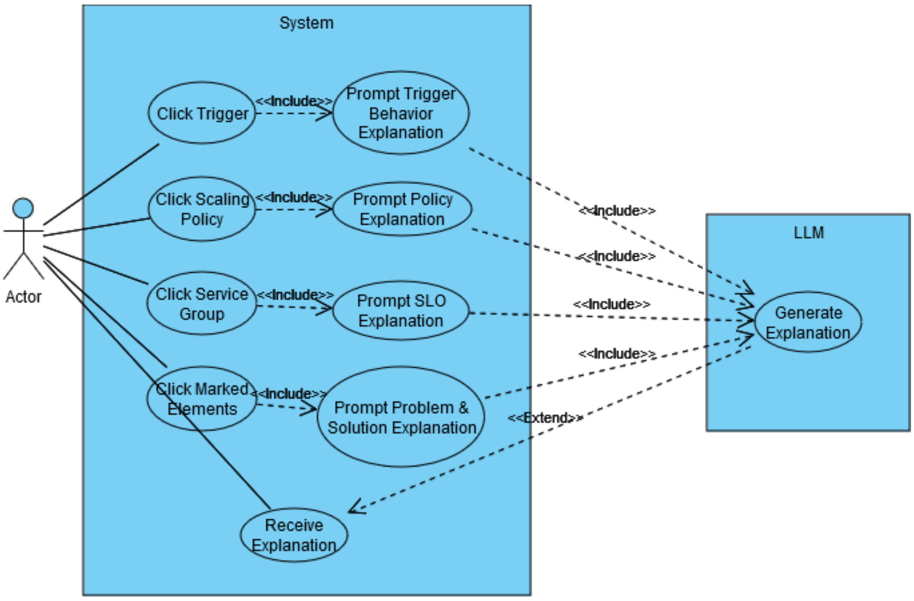


Figure 4.2: Use case diagram for our prototype.

Once the user selects a component, the system sends a corresponding prompt to an LLM, which generates an explanation that is then presented to the user.

5 Implementation

This chapter will describe how we went about implementing a prototype of the concept described in Chapter 4[1]. As mentioned in Chapter 2, we have already inherited the implementation of a prototype of the graphical editor by Summerer which was realized using Eclipse Sirius, and further enhanced by Hahn. Thanks to these enhancements, the editor was already in an almost usable state. Only slight modifications to adjust to the slightly evolved SPD metamodel were necessary to make the prototype compatible with the newest configuration of Slingshot. Analyzing Hahn's implementation of his feedback concept also helped us figure out how to utilize many of the features provided by Sirius. For example, how to call a method from the Services Java class, which will be important later. To develop the prototype, we used the Obeo Designer [Des], which is optimized for the creation of modeling workbenches and includes Sirius, and also has good compatibility with Palladio and Slingshot.

Aside from abandoning Hahn's feedback graph feature in favor of relying more on Palladio's analysis features that are already present, we left the implementation of his visual feedback concept as it is for the most part. There may have been some more room for improvement on that part, but we decided to prioritize the implementation of our generated explanation feature more. The only extension we did on the visual feedback feature was to enable the prototype, which was previously only able to work with dummy data, to use actual Palladio simulation data for its feedback. How this data was accessed will be discussed further down the line, when we need it for the explanation generation feature.

We did not implement the essential requirement 3 worked out in Chapter 3 (The prototype should be able to visualize the extent of an SLO violation) in this prototype. However, it is easy to imagine how it could be done. Right now it is checked if an anomaly exists for a specific element of the SPD. If yes a flag is raised (anomaly boolean set to true). In the editors' odesign file, where its specifications are defined, we declare that if this flag is set for an element, Sirius will display an alternative image for said element. In this prototype it is the same image, colored red, to imply that a problem exists. Instead of a boolean, we could use a double here, that represents the percentage of points in the simulation where the SLO is violated. Depending on how high that percentage is, we could use deeper tones of e.g. red, to imply that the violation is more severe. It is important to note that Sirius does not support recoloring of elements, so for every shade of color we want we would have to create a new image.

---

[1] The source code of the prototype at the time of this thesis submission, including an installation and usage tutorial, can be found here: `https://git.rss.iste.uni-stuttgart.de/slingshot/spd-graphical-editor/-/tree/v0.2?ref_type=tags`

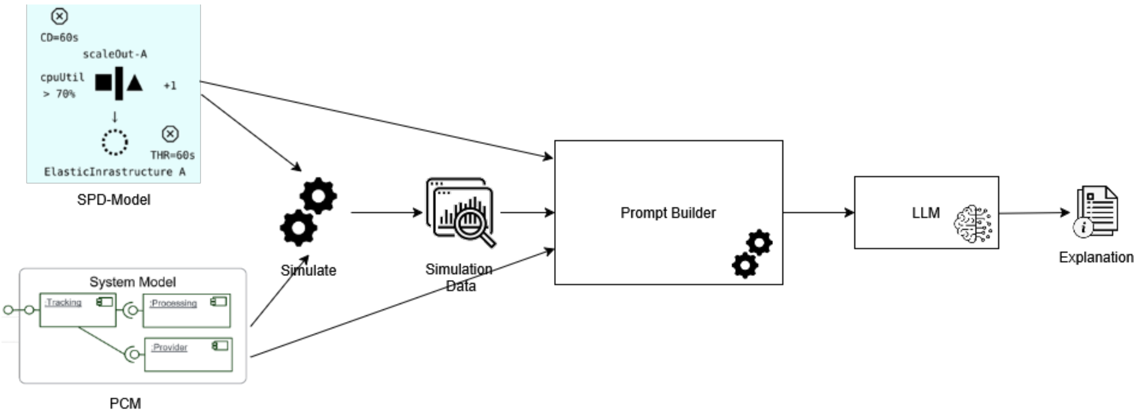


Figure 5.1: Explanation Generation Component - Conceptual Overview

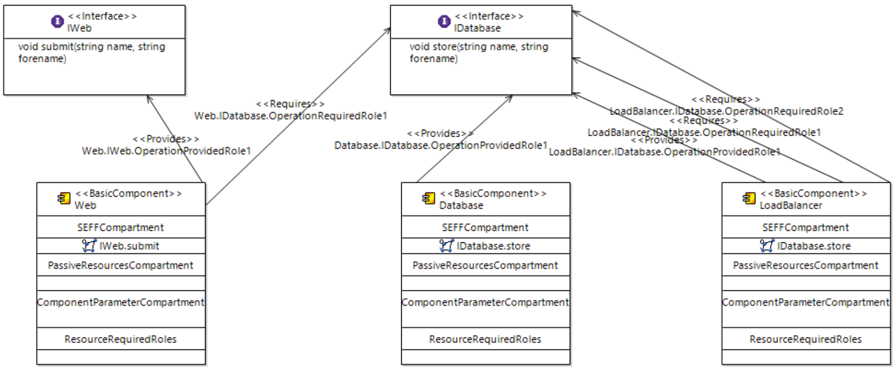

<<Interface>> IWeb
void submit(string name, string forename)
<<Interface>> IDatabase
void store(string name, string forename)
<<Requires>> Web.IDatabase.OperationRequiredRole1
<<Provides>> Web.IWeb.OperationProvidedRole1
<<Provides>> Database.IDatabase.OperationProvidedRole1
<<Requires>> LoadBalancer.IDatabase.OperationRequiredRole2
<<Requires>> LoadBalancer.IDatabase.OperationRequiredRole1
<<Provides>> LoadBalancer.IDatabase.OperationProvidedRole1
<<BasicComponent>> Web
<<BasicComponent>> Database
<<BasicComponent>> LoadBalancer
SEFFCompartment
IWeb.submit
IDatabase.store
IDatabase.store
PassiveResourcesCompartment
ComponentParameterCompartment
ResourceRequiredRoles

Figure 5.2: Repository Model of the example PCM

5.1 Explanation Generation Feature

This section is all about the explanation generation feature of our prototype, which is the main implementation focus of this thesis. For this reason, we want to discuss it in detail. Consider Figure 5.1, an overview of this components concept. In the following we will describe it part by part in detail.

5.1.1 Example PCM

As mentioned in Chapter 4, automatically parsing all the relevant information of a PCM for our SPD-model and generating a description to use in the prompt goes beyond the scope of this thesis. Instead, we settled on building our first prototype around a pre-built PCM that is kept relatively simple. This allows us to focus more on the explainability of the SPD later.

The system we model is a very simple web service, where the user can submit their first and last name via a web interface. The system stores this data in a DBserver via a database interface. To keep it simple, in this model, only the operation to store the data in a DBserver has a resource demand. It requires 10 CPU workunits in 30% of cases, 20 CPU workunits in 40% of cases and 30 CPU workunits in 30% of cases.

Figure 5.2 shows the repository model of this PCM. Here we can see the different components that are deployed within the system and how they connect to the interfaces. The load balancer component is responsible for assigning which database component will carry out which store operation. This will be important once we start scaling out the DBServers.

Figure 5.3 shows the resource environment of our system. The DBServers are the target group to be scaled. Each has a CPU with a process rate of 10 work units per second. The system always starts out with 2 DBServer instances. This is important because we need at least this much to implement the load balancer component we discussed before. The SPD can add or remove instances of these DBServers during the simulation depending on its configuration. For the linking resource that connects the web server with the DBServers, we set latency to zero and throughput to a higher amount than we ever needed to avoid having to deal with network delays in the simulation.

To conduct a simulation in Palladio, we also need to define user behavior, which can be done by creating a usage model. In our example PCM, we define a very simple usage behavior: the user enters the system, executes the submit function, waits 10 seconds (defined as the workload's think time), and repeats the process. This workload can be modified depending on which antipattern we want to provoke. Palladio even allows us to specify usage evolution, enabling the simulation of dynamic workloads that vary over time. Defining these is important for provoking the occurrence of certain antipatterns in the simulation data. However, we did not include information about the usage model in the prompts used for the evaluation step. Later experiments incorporating this context into the prompt did not result in significant changes in the generated explanations. The explanations did not reference the workload and were comparable in content to those generated without this additional context.

The complete PCM further consists of an allocation diagram and an assembly diagram. However, because they do not contain information the LLM necessarily needs to generate the explanations of the SPD components, we will refrain from explaining them in detail. In the end, we created the following description of the PCM, that we will add to every prompt:

> *"In the current system, a user can input their information via a web interface, which is then stored in a DBServer via a database interface. The resource to be scaled is the DBServer. The CPU of one DBServer can process an amount of 10 workunits per second. The operation to store data in the database has a resource demand of 10 workunits in 30% of cases, 20 workunits in 40% of cases and 30 workunits in 30% of cases."*

Delivering this description to the prompt builder is represented with the arrow from the PCM to the prompt builder in Figure 5.1.

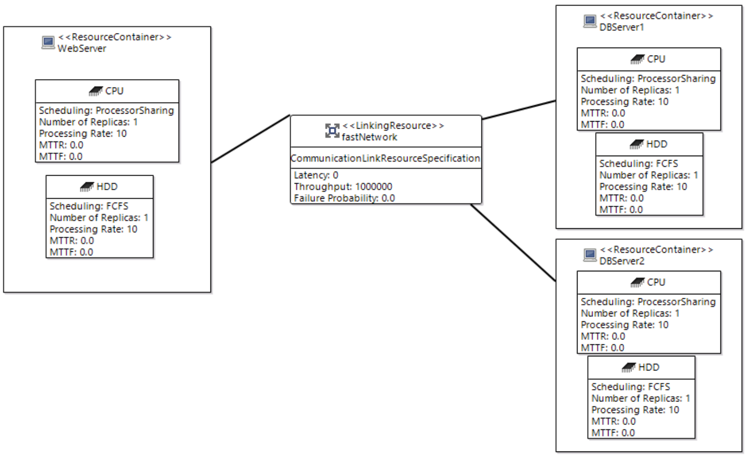


Figure 5.3: Resource Environment of the example PCM

5.1.2 Acquiring SPD Component Information via the Graphical Editor

Sirius supports adding interactivity to finished models, such as handling double-click events. We will leverage this functionality to implement our concept. Specifically, when a user double-clicks an element for which they want an explanation, the system will initiate the process of constructing a prompt and requesting an explanation from the LLM. To achieve this, Sirius allows us to call methods from a specialized Services.java class within these double-click events. The challenge lies in accessing the attributes of the clicked node so they can be used in the prompt builder. For example, if a user double-clicks a scaling policy, we need to extract relevant attributes such as constraints, adjustment values, and trigger conditions. When defining a method in the Services.java class to be called through model interaction, it must accept at least one parameter—typically an EMF type like an EObject (a modeled object). When this method is invoked via an AQL call in a double-click event, Sirius provides the clicked element as a parameter. However, this element is only its graphical representation (a DNode). To retrieve the actual data it represents, we must first access its semantic object. In Sirius, all representation elements maintain a reference to their corresponding semantic model. This relationship can be accessed through the DSemanticDecorator, allowing us to obtain the semantic object. Once retrieved, we can access all its attributes and references. For instance, in the case of a scaling policy, attributes such as cooldown and adjustment value can be accessed directly, while references enable retrieval of related elements like trigger conditions.

Finally, we store all extracted values in a map for easy lookup and pass them to the prompt builder. This process is visually represented by the arrow from the SPD-model to the prompt builder in Figure 5.1.

5.1.3 Simulation Data

Palladio, in addition to Slingshot, allows us to simulate our system with attention to elasticity enabled through the scaling policies defined in the SPD-model. In figure 5.1, this is represented via the arrows from SPD-Model and PCM to simulate. This simulation data is temporarily saved in repositories inside local memory and can be accessed using the EDP2Plugin class provided by Palladio.

To get the data we need, we first have to set up monitors, with which we can determine which measurements should be taken. For our prototype we decided to work with response time, from which the LLM should be able to draw conclusions about how much the system is utilized. We also want data on how many instances are running in our target group, so the LLM can work out which up- or downscaling actions have taken place. To create these monitors we first have to specify measuring points. For the response time it is the usage scenario, and for the number of instances it is the elastic infrastructure. We can monitor the response time and the number of resource containers based on the respective measuring points. This gives us the desired simulation data that we can then deliver to the prompt builder. This is represented by the arrows from simulate to simulation data to prompt builder in Figure 5.1.

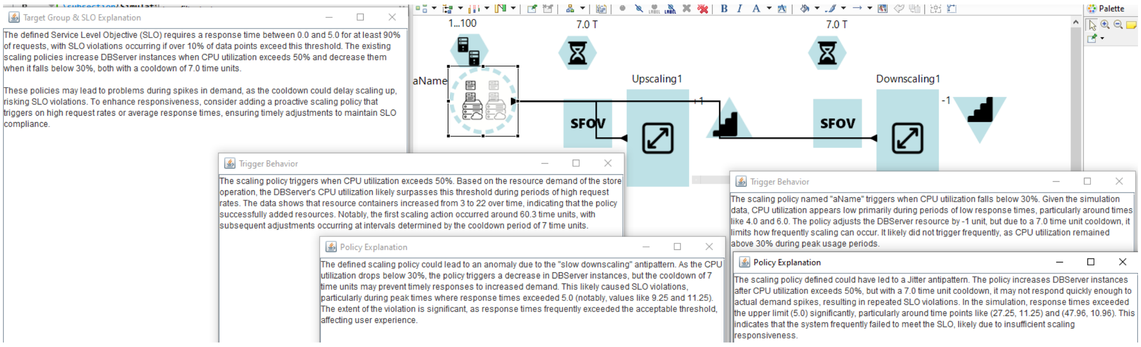
This begs the question about limitations, how big can our simulation data get so that the prompt will still be accepted by the LLM. Because we did not have the means to afford our own OpenAi API key, we did use the demo key LangChain4j provides for demonstration purposes. This demo key is restricted to the gpt-4o-mini model and has a quota. Although we could not find information on how high this quota is, we assume that our token use is more limited than if we had our own API key. In our experiments we got reliable answers to our prompts with a maximum simulation time of up to 500 simulated time units, which gives an output of approximately 300 data tuples. If we had our own OpenAI API key however, depending on the model used, the limit of a request can be up to 128,000 tokens, according to OpenAIs FAQ. Given that Palladio simulation data is typically in the form of a tuple (x,y), where x is the point in time and y the simulation data, and assuming that a floating point number takes up around 2 tokens, we could fit approximately 32000 tuples within the token limit of an OpenAI API call. Considering that many models nowadays also support analyzing CSV files, we can imagine transmitting our simulation data in that way. The upload limit of a CSV file for OpenAI models is approximately 50MB. Assuming a typical numeric value takes around 6-10 bytes, this would allow us to upload files containing several million data points. However, because most models are priced on a per token basis, we may still want to reduce the size of our simulation data if it is very big. We can easily imagine ways to do this. For example, only consider data points in a certain range of time points where scaling has happened. For testing our prototype, we only worked with short simulations, so we delivered the simulation data to the prompt builder raw.

5.1.4 Building the Prompt

After receiving the information about the SPD-Model, the PCM and the simulation data, the prompt builder has enough input to build the prompt according to the templates created in Chapter 4. This is realized via a string builder that fills the placeholder in the template with whichever information

```
  Your task is to assist self-adaptive system architects in designing scaling policies for
elastic cloud systems that assure that the system always has enough resources to provide its
service, but never too many, which would lead to unnecessary costs, by giving explanations and
/or feedback on their scaling policy designs.

  In the current system, a user can input their information via a web interface, which is then
 stored in a DBServer via a database interface.
  The resource to be scaled is the DBServer. The CPU of one DBServer can process an amount of
10 workunits per second.
  The operation to store data into the database has a resource demand of 10 workunits in 30%
of cases, 20 workunits in 40% of cases and 30 workunits in 30% of cases.

  The architect has defined scaling policys as followed:
  The policy aName increases the resource DBServer by 1 units, if "CPUUtilization GreaterThan
50.0" holds. It is restricted by a cooldown of 7.0 time units.

  Explain how the policy would affect the target resource on the system's runtime, under which
 circumstances would it trigger and to which kind of scaling would it lead?

  Answer in natural language, keep the answer short and concise, keep it under 400 characters.
```

Listing 5.1: Scaling Policy Explanation Prompt Example



The defined Service Level Objective (SLO) requires a response time between 0.0 and 5.0 for at least 90% of requests, with SLO violations occurring if over 10% of data points exceed this threshold. The existing scaling policies increase DBServer instances when CPU utilization exceeds 50% and decrease them when it falls below 30%, both with a cooldown of 7.0 time units.

These policies may lead to problems during spikes in demand, as the cooldown could delay scaling up, risking SLO violations. To enhance responsiveness, consider adding a proactive scaling policy that triggers on high request rates or average response times, ensuring timely adjustments to maintain SLO compliance.

The scaling policy triggers when CPU utilization exceeds 50%. Based on the resource demand of the store operation, the DBServer's CPU utilization likely surpasses this threshold during periods of high request rates. The data shows that resource containers increased from 3 to 22 over time, indicating that the policy successfully added resources. Notably, the first scaling action occurred around 60.3 time units, with subsequent adjustments occurring at intervals determined by the cooldown period of 7 time units.

The defined scaling policy could lead to an anomaly due to the "slow downscaling" antipattern. As the CPU utilization drops below 30%, the policy triggers a decrease in DBServer instances, but the cooldown of 7 time units may prevent timely responses to increased demand. This likely caused SLO violations, particularly during peak times where response times exceeded 5.0 (notably, values like 9.25 and 11.25). The extent of the violation is significant, as response times frequently exceeded the acceptable threshold, affecting user experience.

The scaling policy named "aName" triggers when CPU utilization falls below 30%. Given the simulation data, CPU utilization appears low primarily during periods of low response times, particularly around times like 4.0 and 6.0. The policy adjusts the DBServer resource by -1 unit, but due to a 7.0 time unit cooldown, it limits how frequently scaling can occur. It likely did not trigger frequently, as CPU utilization remained above 30% during peak usage periods.

The scaling policy defined could have led to a Jitter antipattern. The policy increases DBServer instances after CPU utilization exceeds 50%, but with a 7.0 time unit cooldown, it may not respond quickly enough to actual demand spikes, resulting in repeated SLO violations. In the simulation, response times exceeded the upper limit (5.0) significantly, particularly around time points like (27.25, 11.25) and (47.96, 10.96). This indicates that the system frequently failed to meet the SLO, likely due to insufficient scaling responsiveness.

Figure 5.4: The prototype provides explanations for every part of the SPD-Model.

it is provided about the SPD-Model, and the input data. We hardcoded the description of the PCM inside the prototype, because, as mentioned, automating this description exceeds the scope of this thesis.

As an example, for a scaling policy aName that adds an instance if the CPU utilization exceeds 50%, with a cooldown of 7 seconds, the prompt builder would create a prompt for requesting the Scaling Policy Explanation, like in listing 5.1.

Using LangChain4j, this prompt will then be sent to an LLM, which will generate an explanation that the user can then receive. This operation is represented by the arrows from prompt builder to LLM, to explanation in Figure 5.1. We present the explanation inside a pop-up JFrame in our prototype. Figure 5.4 shows what it would look like if we had a SPD-Model with an upscaling and a downscaling policy and request explanations for every part of it after a simulation.

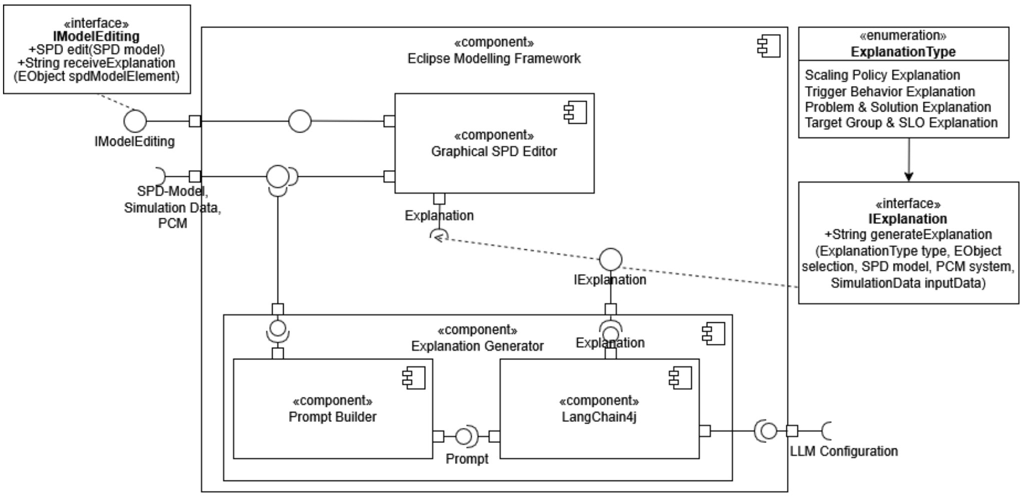


Figure 5.5: Component diagram of the explanation generation feature.

5.1.5 The Finished Component

To round out the implementation chapter and provide more clarity on the new feature, we created a component diagram, shown in Figure 5.5.

Our main component in this UML Component diagram is the Eclipse Modeling Framework, which serves as the IDE on which our graphical editor runs. It provides an interface, IModelEditing, that allows the user to modify the SPD model and request explanations before and after a simulation is conducted.

To achieve this, it requires the SPD model to be modified, as well as the PCM that corresponds to this model—both provided via user input. Additionally, for visual feedback, it needs simulation data, which is supplied by the Slingshot simulator.

If the user requests an explanation, it is generated by the explanation generator and then provided to the graphical editor. The explanation generator receives the same information about the SPD model, the PCM, and the simulation data. It uses this data to construct a prompt inside the prompt builder component, which is then processed by LangChain4j. LangChain4j sends the prompt to an external LLM and retrieves the generated explanation. This explanation is returned to the IExplanation interface of the explanation generator, which subsequently provides it to the graphical editor for display to the user.

Figure 5.6 shows the sequence diagram of a successful run of this feature, illustrating the interactions between the components in detail.

The user configures the PCM and the SPD model in the graphical editors, which run within the Eclipse Modeling Framework IDE. These models can be displayed graphically to the user. The user can then start a simulation. To do this, the EMF provides its models to Slingshot, which uses them to simulate the system and return simulation data. In the graphical SPD editor, this data is used to provide visual feedback.

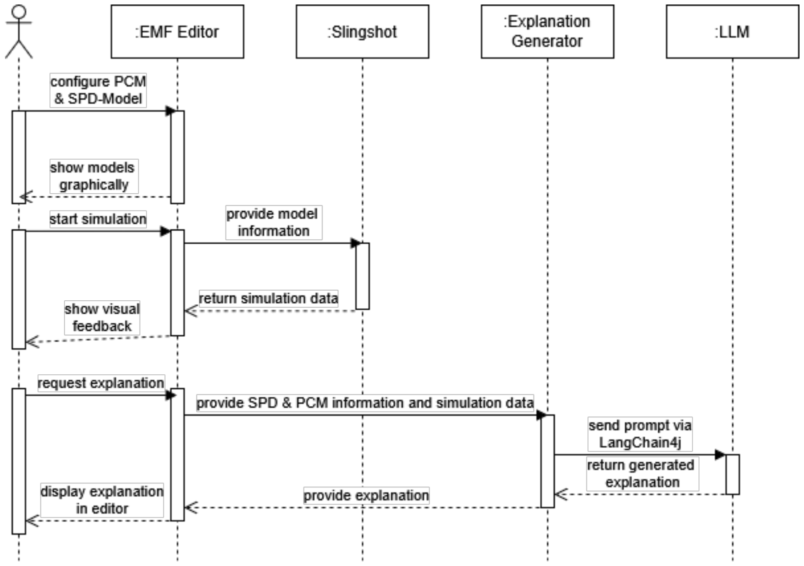

:EMF Editor
:Slingshot
:Explanation Generator
:LLM
configure PCM & SPD-Model
show models graphically
start simulation
provide model information
return simulation data
show visual feedback
request explanation
provide SPD & PCM information and simulation data
send prompt via LangChain4j
return generated explanation
provide explanation
display explanation in editor

Figure 5.6: Sequence diagram of a successful run of the explanation generation feature.

If the user wants a deeper explanation of any part of the SPD model, they can request textual explanations by clicking on areas of interest. In this case, the editor provides relevant information about the SPD model, the PCM, and the previously generated simulation data to the explanation generator. The explanation generator uses this information to construct a prompt, which is then sent to an LLM via LangChain4j. The LLM generates the desired explanation and returns it to the explanation generator, which then sends it back to the editor. The explanation is then displayed directly within the graphical editor's interface.

6 Evaluation

We focus our evaluation on assessing the quality of explanations. Specifically, how accurate, concise, and useful are the explanations generated by the LLM? Do they tend to contain imprecisions? Are certain types of explanations perceived as more useful than others? To answer these questions, we chose to first evaluate a larger number of explanations rather than immediately conducting a traditional user study on our prototype. This approach allows us to concentrate entirely on evaluating the explanations themselves, whereas a user study would place greater emphasis on their presentation and the overall usability of the program. Moreover, due to time constraints, we were unable to conduct both studies. As a result, the implementation described in Chapter 5 remains unevaluated for now. We leave its assessment, or that of a more advanced implementation, for future work. In this chapter, we present our evaluation of explanation quality. Table 6.1 outlines our evaluation plan, designed according to the Goal Question Metric (GQM) paradigm [CR94].

6.1 Study Design

To evaluate the quality of the explanations, we asked two members of the Software Quality and Architecture (SQA) institute of the University of Stuttgart. Both are experts on self-adaptive system architecture and working with SPDs. They were tasked to rate 47 explanations according to 5 quality criteria. In the following sections, we describe the study design, the types of explanations generated and their creation process, as well as the evaluation criteria in detail.

Goal	**Validate the quality of the explanations generated using the prompt design.**
Question 1	How many of the explanations contain correct information related to the problem?
Metric 1.1	Percentage of explanations rated as accurate.
Hypothesis 1	A significant percentage of explanations will contain correct information.
Question 2	How many of the explanations are useful to fix problems/identify antipatterns?
Metric 2.1	Percentage of explanations rated as useful.
Hypothesis 2	Many explanations which contain correct information will also be useful.
Question 3	How many explanations do abide to format constrictions and expectations of conciseness?
Metric 3.1	Percentage of explanations rated as concise.
Hypothesis 3	Most explanations will meet the constrictions given by the output indicator.

Table 6.1: GQM plan overview

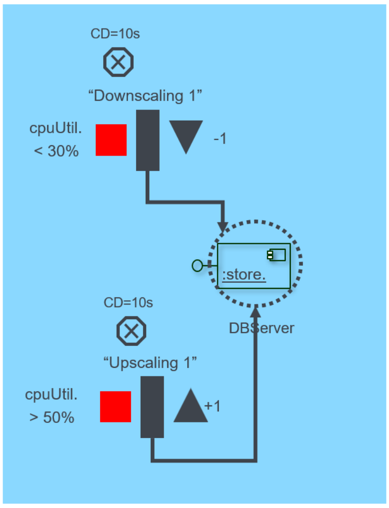


Figure 6.1: An SPD-Model that lead to the Jitter antipattern in the simulation.

6.1.1 Generating the Explanations

As mentioned in Chapter 4 we have 4 kinds of explanations: Scaling Policy Explanations, Trigger Behavior Explanations, Problem & Solution explanations and Target Group & SLO Explanations. To ensure a balanced distribution, we aimed to include a diverse mix of all explanation types, allowing us to later compare which types were rated more or less favorably. To achieve this, we designed SPD-models and generated explanations for each part of them.

To further increase the range of explanation generation, we designed our SPD-models as such, that they would lead to one of the antipatterns described in Chapter 2. We design such a model for every antipattern, which leads us to 7 different SPD-models. All of them have an upscaling and a downscaling policy, connected to a single target group.

With two explanations per policy (Scaling Policy Explanation and Trigger Behavior Explanation), and one explanation per target group, we already have 5 explanations per SPD-model. Additionally, every SPD-model has one or two Problem & Solution Explanations. We only generate them for policies where actual anomalies occur. Only for the SPD-models that lead to the Jitter and the No Reaction antipattern, both scaling policies show anomalies. In all others, only one policy is to blame.

Finally, we added 4 incomplete SPD-models to the pool, so we can also have a few Target Group & SLO Explanations that give tips on how to complete an unfinished SPD-model to rate.

As an example, Figure 6.1 shows a policy that leads to the Jitter antipattern in the simulation. Once it reaches a CPU utilization threshold of 50% it scales up, but thanks to the high downscaling threshold of the downscaling policy, it will immediately scale down again, which will lead to the CPU utilization exceeding the upscaling threshold again, and so on. This way, no steady state can be reached.

To see the antipatterns reflected in the simulation data, we need certain workloads occurring. To have full control over the CPU utilization during the simulation, we created dummy simulation data by hand inside a Microsoft Excel spreadsheet. We created the data in the form of tuples (x,y,a,b), where x = Point in time, y = CPU utilization, a = number of upscaling operations up to point x, and b = number of downscaling operations up to point x. We delivered this dummy data to the LLM as input data. The rest of the prompt was created as described in Chapter 4.

Our prototype has the aforementioned restriction of only using the demo API key of LangChain4j. This restricts it to the usage of the GPT-4o mini model, which is not the most sophisticated, and also has a quota. For this reason, we decided against using it to generate the explanations. If we truly want to assess the potential of the explanations, we should use a more state-of-the-art model. We decided to generate our explanations via the ChatGPT web interface, which grants us access to GPT-4o. To compare these two models, GPT-4o mini is described as a ”small” model with around 8 billion parameters. The number of parameters in GPT-4o is not officially known, but it is estimated to be around 1.8 trillion [AYF+24]. Using the web interface has its own constraint, the size restriction for a message is 4096 tokens. However, for the size of our dummy data, this was always more than sufficient. Our experiments even suggest that this limit may exceed the message size allowed by LangChain4j’s demo API key. However, we can not state this for certain, because we could not find detailed information about its restrictions.

Another drawback of using the web interface instead of the API is that we cannot directly influence the model’s temperature. In the context of LLMs, the temperature determines whether the output is more deterministic or more unpredictable. As we know, when an LLM generates text, it predicts the next token based on a probability distribution. If we set the temperature parameter high, the distribution becomes flatter, meaning tokens with lower probabilities have a higher chance of being selected. On the other hand, if we set it low, the distribution becomes steeper, meaning the model will choose the highest-probability token more often. In effect, setting the temperature lower could eliminate extraneous content and prevent the model from hallucinating, which introduces imprecisions. On the other hand, setting it higher could make the output more ’creative’, potentially leading to more helpful explanations, especially for troubleshooting tips. Experimenting with different temperature settings to assess their impact on the quality of explanations could be interesting. However, due to our lack of access to a proper API key, we were unable to explore this. Perhaps future work could investigate this aspect further. How high the temperature is for the ChatGPT web interface is not entirely clear. According to the model itself, it is around 0.7, which is the standard configuration for most chatbots and conversational AIs. With this setting, the LLM provides natural, engaging responses, which work well for our use case since we want explanations in understandable, natural language.

Once all explanations were generated, we compiled them into a Microsoft Form, enabling our two experts to easily rate them based on the five evaluation criteria, which we describe in the following section.

6.1.2 Evaluation Criteria

Inspired by the work of Kang et al.[KAY24], we worked out 5 criteria, on which the explanations would be rated.

Expert 1	Accurate	Imprecise	Concise	Useful	Bland	Total
All Explanations	80,85%	19,15%	44,68%	61%	12,77%	47
Problem & Solution Explanation	77,78%	33,33%	33,33%	77,78%	0%	9
Trigger Behavior Explanation	69,23%	23,08%	30,77%	53,85%	15,38%	13
Scaling Policy Explanation	71,43%	14,29%	35,71%	42,86%	14,29%	14
TargetGroup & SLO Explanation	81,81%	0%	72,72%	72,72%	9,09%	11

Expert 2	Accurate	Imprecise	Concise	Useful	Bland	Total
All Explanations	85,11%	55,32%	72,34%	57,45%	55,19%	47
Problem & Solution Explanation	55,56%	100%	66,67%	77,78%	55,56%	9
Trigger Behavior Explanation	100%	23,08%	69,23%	61,54%	30,77%	13
Scaling Policy Explanation	71,43%	64,29%	50%	14,29%	71,43%	14
TargetGroup & SLO Explanation	90,91%	54,55%	81,81%	81,81%	36,36%	11

Table 6.2: Explanation Quality Evaluation Results

- Accuracy: describes policy part in detail, for problem explanation, names the antipattern and describes its cause correctly.
- Imprecision: the explanation contains at least one inaccurate statement.
- Conciseness: explanations are succinct, do not contain extraneous content.
- Usefulness: the explanation correctly explains how policy can be improved/problems can be fixed.
- Blandness: explanation correctly describes respective part of the policy but does not offer additional analysis.

These criteria are binary, meaning an explanation either meets the criterion or it does not (e.g., it is either accurate or inaccurate). In the evaluation form, we present our experts with a multiple-choice question for each explanation, listing all five criteria. They can check a box if they believe the explanation meets a given criterion or leave it unchecked if they think it does not.

6.2 Results

We compiled every explanation and its rating in our *Zenodo* repository. All the prompts used to generate the explanations can also be found there [Lam25]. The results of the explanation quality evaluation can be seen in Table 6.2, which shows the percentage of explanations that met each criterion.

We present the results for all explanations combined, as well as for each type of explanation separately. Additionally, we show the ratings given by each expert individually, as there were significant discrepancies between them. With a Cohen's κ coefficient [McH12] of only 0,202, we can only conclude a slight agreement between the experts. To clarify why, we invited them both again after the evaluation, to discuss some of their ratings. We will talk about this discussion and analyze the results deeper in the following section.

6.3 Discussion

We discussed the discrepancies in the two experts' ratings further by selecting 5 explanations that reflect the most significant disagreement trends, as seen in Table 6.2. Each expert was asked to justify their ratings and explain their reasoning. In two cases, the experts reached a consensus after further discussion. In both instances, one expert had overlooked an inaccuracy; upon having it pointed out, they agreed with the other expert's rating. While such minor oversights could explain some disagreements, they did not account for all of them. Further discussions revealed that the experts interpreted the rating criteria slightly differently.

For expert 1, an explanation they would rate as imprecise could not be useful anymore, while expert 2 would still sometimes rate explanations with imprecisions as such, if at least parts of it contain useful information. Expert 1 was stricter with conciseness. Any information that did not contribute to the overall usefulness of the explanation would prevent a conciseness rating. For expert 2, some conciseness was there in all explanations, because of the word count restrictions in the prompts output indicator. They would use the concise rating more leniently. On the other hand, expert 2 was stricter on the imprecise rating. Small imprecisions like getting some details wrong or using wrong or unusual terminology could already earn the explanation an imprecise rating. Expert 2 was more lenient with it, as long as the explanation did not contain completely false information. For blandness, expert 1 would only rate explanations as bland that they perceived as 'boring'. Also, as they stated, the fewer words an explanation had, the more it would increase the likelihood of receiving this rating. Expert 2 would use this rating if the explanations contained information that was not needed to increase its usefulness.

This explained most of the discrepancies. However, the discussion and additional feedback revealed further observations. All explanations discussing SLO violations mistakenly treated them as single points in time where the condition was breached. This is incorrect—a violation occurs only when the condition (SLI) is breached over a specified period (e.g., a certain percentage of the entire simulation time). Expert 2 consistently marked such explanations as imprecise, explaining the 100% imprecise rating for Problem & Solution Explanations. Ideally, explanations should specify the time intervals during which the SLO was violated rather than isolated moments.

A potential method to address this problem could be, to add even more context regarding SLOs to the prompt. For example, we tried adding the following passage to the context part of the prompt: *"An SLO violation occurs once the number of data points where [SLI] is violated exceeds [SLO PERCENTAGE] of all data points."* In the prompts output indicator we also added a concrete request for time intervals instead of points in time. *"Explanation of which SLO violation occurred at which time intervals and to what extent (How many % of time was SLO condition not fulfilled?)"* Using the new prompt for an example, with an SLO that requires the response time of the system to be under 5 seconds for at least 10% of the time, the generated explanation contained the following information regarding SLO violations: *"The SLO violation occurred primarily in time intervals [6.0-11.0], [23.67-26.0], [42.22-46.67], [61.77-63.33], and [90.77-93.61], with response times exceeding 5 units. The violation exceeded 10% of data points."* As we can see, it does output intervals now. However, the LLM seems to still have problems calculating the exact percentage of points where the SLI was violated.

Other remarks we got from the experts on our explanations were that they would sometimes make up assumptions without context. For example, in one explanation it was stated that a upscaling policy that adds a high number of instances would lead to rapid upscaling. Given that this was one of the explanations that was generated without input data in the prompt, the LLM could not have known that. Maybe that scaling policy was created in anticipation of big workload spikes. In that case, the explanation would have been wrong. In the same vein, the explanations often warned that cooldowns could lead to delayed responses, no matter how big or small they are. The tendency for explanations to construct such scenarios that are not present indicates that the LLM may is overeager to provide the user with tips to improve the policy even when there is little need for it. This overeagerness may needs to be restricted inside the prompt by a stricter, more detailed output indicator.

It was also noted that the explanations use of terminology could be inconsistent at times. For example, sometimes it wrote about adjustments, sometimes about scaling. Sometimes it even used some unusual terminology that finds little use in the area of autoscaling. The easiest way to improve on that is to add even more context of the field to the prompt. We already did this for the antipatterns so they would be named correctly. Depending on how highly we value consistent terminology use, we could go even further. Experimenting on how much added context would actually improve the explanation and how much would be excessive would be an interesting exercise for future work.

Lastly, every once in a while explanations would contain tips and suggestions that were trivial and therefore not helpful. For example, one explanation includes the passage *"Fine-tuning the threshold and cooldown can improve reliability."* Phrases like this are obvious, offer us nothing and do not improve the explanations' usefulness at all. Maybe these are filler phrases the LLM adds to hit the character limit, if it does not have much information to offer on the policy.

Putting the discussion and feedback aside, we will now discuss the raw numbers we got as the result of the evaluation, compiled in table 6.2. Overall, the explanations had high accuracy numbers, exceeding 80% for both experts. This tells us that the LLM was able to generate explanations that do include factual information, at least partially, for the most part, which confirms Hypothesis 1. However, they would also still contain numerous imprecisions. Expert 1 rated almost 20% of explanations as imprecise, Expert 2 over 50% even. This can be explained by how the LLM had problems keeping the terminology consistent, and also often got details wrong. Including more context to the prompt, or even training an LLM specifically to solve this task could lower these numbers.

Relatively low numbers in conciseness for expert 1, and relatively high numbers in blandness for expert 2 imply that explanations often contain extraneous information that does not contribute to their usefulness. Adding a stricter output template to the output indicator of the prompt, or lowering the temperature of the model could help to reduce the amount of this kind of 'useless' information. Hypothesis 3 can still be seen as mostly confirmed, given the pretty high number of conciseness ratings of expert 2 and their comments on how most of the explanations adhere to the limits given in the output indicator. However, the problem of extraneous information should not be ignored.

Usefulness ratings being around 60% for both experts implies a mediocre performance of the explanations on this criteria, which would contradict hypothesis 2. However, if we break down the explanations to their individual kinds of explanations, we can see a different picture for some kinds. Problem & Solution Explanations and Target Group & SLO Explanations have a much more favorable usefulness rating. The former got a respectable rating of 77,78% from both experts; the latter got 72,72% from expert 1 and a good rating of 81,81% from expert 2. Trigger Behavior Explanations got more middling usefulness ratings, but still at least over 50%. The explanation type that dragged down the overall usefulness rating of all the explanations was the Scaling Policy Explanation. With only 42,86% for expert one, and a withering 14,29% rating for expert 2, these explanations were largely seen as not that useful by our experts. These explanations only describe what a scaling policy does, without analyzing any simulation data. Especially for experts, this information can just be extracted from looking at the SPD-model, so rating them as not that helpful is understandable. If non-experts that are not familiar with the visual language of an SPD-model would find these explanations just as useless is something we could not explore with our evaluation method. This may be an interesting question to investigate in future work.

Overall, we can conclude that our approach shows promise in some quality aspects. It would be interesting to see how much fine-tuning the prompts could improve the quality criteria. Future work could explore ways to make the explanations more useful, concise and precise. Maybe they could even design more kinds of explanations, that would be even more useful to help architects with the creation of SPD-models. It will also be interesting to observe how much LLMs themselves can improve still. Maybe in time these numbers can improve by just generating the explanations with the same prompts, but a different, more sophisticated model that does not exist yet.

6.4 Threats to Validity

The most obvious threat to external validity with our kind of evaluation is our sampling bias. Because we wanted a more educated assessment of our explanations, we picked two experts in the field, with lots of experience working with SPD-models to rate the explanations. This means, however, that we can not know how, for example, a newcomer to the field would receive the explanations. Investigating how the explanations could help someone who only has a passing knowledge of SPD-models, or none at all, to grasps the concepts of autoscaling and understand the models would be an interesting task for future work.

There is also the threat to internal validity of maturation. Because the experts had a lot of explanations to rate, a single experiment run could take quite a large amount of time. According to Microsoft Forms, the experts took 254 minutes on average to fill out the form. This does, of course, not take time away from the keyboard into account, but we can still assume that the rating has been a time-consuming task. We can not exclude the possibility that the participants showed symptoms of fatigue during the long experiment run, which may have led them to, for example, miss small imprecisions in later explanations.

7 Conclusion

This chapter concludes our thesis. First, we summarize the results we have achieved. After that, we describe the benefits of our approach and who can benefit from it. We then discuss some limitations of our approach, particularly those of the provided prototype. Next, we reflect on the lessons learned during the process of writing this thesis. Finally, we conclude with an outlook on the future: how our approach can be further improved and in which direction our topic can be developed.

7.1 Summary

In this thesis, we proposed several types of explanations that could help users gain a better understanding of their SPD model and identify potential antipatterns when anomalies appear in the simulation data. We also introduced prompt templates designed to guide LLMs in generating these explanations as accurately, concisely, and usefully as possible while minimizing imprecisions.

Additionally, we developed a prototype that allows users to request explanations directly by clicking on components of their completed SPD model within the graphical SPD editor. These explanations are then displayed directly within the program.

Our evaluation found that while the generated explanations show promise in terms of quality, there is still room for improvement through further prompt fine-tuning.

7.2 Benefits

The concepts introduced in this thesis were developed with architects of self-adaptive systems in mind. We believe they can benefit from our approach when designing SPD-models. The Problem & Solution Explanations will help them identify and address antipatterns at design time, while the suggested solutions will assist in resolving these anomalies. The Trigger Behavior Explanations will provide insights into when and why policies were triggered, helping architects gain a deeper understanding of the simulation data. The Target Group & SLO Explanations will clarify which policies remain necessary to fulfill the SLO and which might introduce problems. These assumptions are supported by the solid usefulness scores these explanations received in expert evaluations. Even the less favorably rated Scaling Policy Explanations may still help newcomers develop a deeper understanding of what the SPD-model represents.

Comparing our concept to the approach proposed by Haas [Haa23], both share a similar goal, despite differences in their use cases. Using LLMs to generate explanations eliminates the need to manually define cases and explanations, providing greater flexibility. Even if unexpected cases arise—ones not anticipated by the designer—the LLM may still analyze the data and generate a relevant explanation. Haas' approach, on the other hand, is more reliable for predefined cases. Since its explanations are strictly predefined, they are free from imprecisions, extraneous content, or hallucinations. This, however, highlights the greatest weakness of our approach. Even with fine-tuning of prompts, as we suggested, or the use of more sophisticated future models, it is unlikely that imperfections can be entirely eliminated. Nevertheless, we believe that due to its flexibility and the reduced workload for experts, our approach offers significant benefits to self-adaptive system architects.

7.3 Limitations

As discussed in Chapter 5, the proposed prototype has limitations regarding prompt input size and the choice of LLM due to its reliance on the LangChain4j Demo API key. These constraints can be overcome by purchasing a dedicated API key for any LLM one wishes to use, as LangChain4j supports most commonly used models.

However, even with a proper API key, restrictions on input size would still apply. While we have argued that, depending on the input structure, it may be possible to include several million simulation data points, this might still be insufficient for very large datasets. In such cases, preprocessing the simulation data would be necessary to manage input size effectively.

Another inherent limitation of using LLMs is their tendency to hallucinate, potentially generating misleading or even incorrect explanations. This issue may never be fully resolved, regardless of how advanced LLM technology becomes. Therefore, it remains crucial to validate the generated explanations by cross-referencing them with the simulation graphs provided by Palladio to ensure accuracy and reliability.

7.4 Lessons Learned

One lesson we learned during the quality evaluation is that even experts in the same field can have vastly different perspectives on a problem. We cannot assume they will reach the same conclusions when rating explanations independently. Differences in how they interpret evaluation criteria and how strictly they apply them can lead to significant discrepancies in their ratings. After our discussion meeting, we realized that conducting the evaluation as a group discussion from the outset might have been a better approach. This format would likely have led to greater consensus on each rating. Additionally, the discussions would have provided immediate comments, feedback, and observations on the explanations, aiding our analysis.

However, this approach would have presented its own challenges. Coordinating all participants at the same time for an extended task would have been difficult. Given the number of explanations to evaluate, along with potentially lengthy discussions for each, the process could have taken several hours—perhaps even an entire evening. Considering the busy schedules of the experts, securing that much time may have been unrealistic, no matter how beneficial it would have been for the thesis.

7.5 Future Work

Since the evaluation phase of this thesis focused solely on assessing the quality of the explanations in isolation—and only through expert review—the next logical step would be to investigate how these explanations are received when incorporated into a prototype, by both experts and newcomers. This would provide a clearer understanding of whether the explanations truly aid users in comprehending SPD models and identifying antipatterns during design time.

Another important avenue for future research is refining the prompt engineering process to enhance the quality of explanations in a qualitative evaluation, such as the one conducted in this thesis. Some key questions to explore could include: How much additional context is required to minimize inaccuracies? Are there alternative explanation types that could achieve higher usefulness scores? Could enforcing a stricter output structure improve conciseness and reduce blandness? These and other questions could be addressed using our explanation quality evaluation method.

In this thesis, we exclusively used GPT-4o, as it is one of the most widely used state-of-the-art LLMs. However, it would be valuable to examine how other models perform on this task. Could some models achieve better results? What trade-offs might arise? Since Large Language Models continue to evolve rapidly, with new models being introduced frequently, future research should explore how these advancements impact our approach. Would newer models naturally generate higher-quality explanations without significant modifications to our concept, simply due to their improved capabilities? The continuous advancements in LLMs may unlock untapped potential for identifying autoscaling antipatterns at design time using generated explanations.

Bibliography

[AYF+24] A. B. Abacha, W.-w. Yim, Y. Fu, Z. Sun, M. Yetisgen, F. Xia, T. Lin. “Medec: A benchmark for medical error detection and correction in clinical notes”. In: *arXiv preprint arXiv:2412.19260* (2024) (cit. on p. 39).

[BD10] R. Bruns, J. Dunkel. *Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Springer-Verlag, 2010 (cit. on p. 4).

[CMM+23] E. Cambria, L. Malandri, F. Mercorio, M. Mezzanzanica, N. Nobani. “A survey on XAI and natural language explanations”. In: *Information Processing & Management* 60.1 (2023), p. 103111 (cit. on p. 9).

[CR94] V. R. B. G. Caldiera, H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 37).

[DBC21] D. Das, S. Banerjee, S. Chernova. “Explainable ai for robot failures: Generating explanations that improve user assistance in fault recovery”. In: *Proceedings of the 2021 ACM/IEEE international conference on human-robot interaction*. 2021, pp. 351–360 (cit. on p. 10).

[Des] O. Designer. *The Professionel Solution to Deploy Sirius - Obeo Designer*. URL: https://www.obeodesigner.com/en/ (cit. on p. 29).

[DuM93] W. DuMouchel. “Thoughts on graphical and continuous analysis”. In: *Center for Quality of Management Journal* 2.4 (1993), pp. 20–22 (cit. on p. 14).

[Gir23] L. Giray. “Prompt engineering with ChatGPT: a guide for academic writers”. In: *Annals of biomedical engineering* 51.12 (2023), pp. 2629–2633 (cit. on p. 20).

[Goo] Google. *Google Scholar*. URL: https://scholar.google.com/ (cit. on p. 11).

[Haa23] J. Haas. *Integrating Explanation Generation into the Palladio Tool Chain*. MA thesis. University of Stuttgart. May 2023 (cit. on pp. 6, 45).

[Hah23] B. Hahn. *Enriching Graphical Editors for Scaling Policy Definitions with Feedback from Simulations*. BA thesis. University of Stuttgart. June 2023 (cit. on pp. iii, v, 1, 5).

[JLL+24] B. Jury, A. Lorusso, J. Leinonen, P. Denny, A. Luxton-Reilly. “Evaluating llm-generated worked examples in an introductory programming course”. In: *Proceedings of the 26th Australasian Computing Education Conference*. 2024, pp. 77–86 (cit. on p. 10).

[JWNS] C. Jones, J. Wilkes, M. Niall, C. Smith. *Site Reliability Engineering - Service Level Objectives*. URL: https://sre.google/sre-book/service-level-objectives/ (cit. on p. 7).

[KAY24] S. Kang, G. An, S. Yoo. “A quantitative and qualitative evaluation of LLM-based explainable fault localization”. In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 1424–1446 (cit. on pp. 10, 11, 39).

[KHB21] F. Klinaku, A. Hakamian, S. Becker. “Architecture-based evaluation of scaling policies for cloud applications”. In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE. 2021, pp. 151–157 (cit. on p. 1).

[KKB21] J. Katic, F. Klinaku, S. Becker. “The Slingshot Simulator: An Extensible Event-Driven PCM Simulator (Poster).” In: *SSP*. 2021 (cit. on p. 4).

[KKSB23] F. Klinaku, J. Katić, S. S. Stieß, S. Becker. “Designing Elasticity Policies for Cloud-Native Applications with Slingshot”. In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE. 2023, pp. 19–23 (cit. on p. 3).

[Lam25] J. Lammert. *Identifying Autoscaling Antipatterns at Design Time using LLM-based Explanation Generation - Master's Thesis Dataset*. Mar. 2025. URL: https://doi.org/10.5281/zenodo.15045589 (cit. on pp. 15, 40).

[LHS+23] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, B. A. Becker. “Using large language models to enhance programming error messages”. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 563–569 (cit. on p. 10).

[LTF+24] S. Lubos, T. N. T. Tran, A. Felfernig, S. Polat Erdeniz, V.-M. Le. “LLM-generated Explanations for Recommender Systems”. In: *Adjunct Proceedings of the 32nd ACM Conference on User Modeling, Adaptation and Personalization*. 2024, pp. 276–285 (cit. on p. 10).

[LTM+24] Y. Liu, S. Tao, W. Meng, J. Wang, W. Ma, Y. Chen, Y. Zhao, H. Yang, Y. Jiang. “Interpretable online log analysis using large language models with prompt strategies”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 2024, pp. 35–46 (cit. on p. 10).

[McH12] M. L. McHugh. “Interrater reliability: the kappa statistic”. In: *Biochemia medica* 22.3 (2012), pp. 276–282 (cit. on p. 40).

[MTM+22] S. MacNeil, A. Tran, D. Mogil, S. Bernstein, E. Ross, Z. Huang. “Generating diverse code explanations using the gpt-3 large language model”. In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*. 2022, pp. 37–39 (cit. on p. 10).

[Nie15] M. A. Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015 (cit. on p. 8).

[NMH+24] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, B. Myers. “Using an llm to help with code understanding”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13 (cit. on p. 10).

[Pal] Palladio. *Modeling and Simulating Software Architectures, The Palladio Approach*. URL: https://www.palladio-simulator.com/ (cit. on p. 4).

[RBB+11] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, M. Kuperberg. “The palladio component model”. In: (2011) (cit. on p. 4).

[RQWK24] E. Reif, C. Qian, J. Wexler, M. Kahng. “Automatic Histograms: Leveraging Language Models for Text Dataset Exploration”. In: *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 2024, pp. 1–9 (cit. on p. 9).

[SEK+23] M. Straesser, S. Eismann, J. von Kistowski, A. Bauer, S. Kounev. “Autoscaler Evaluation and Configuration: A Practitioner’s Guideline”. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 31–41 (cit. on pp. iii, v, 1, 7).

[Sir] Sirius. *The easiest way to get your own Modeling Tool*. URL: `https://eclipse.dev/sirius/` (cit. on p. 4).

[Sli] Slingshot. *Engineering Elasticity Policies for Cloud-native Applications with Slingshot*. URL: `https://www.palladio-simulator.com/Palladio-Documentation-Slingshot/` (cit. on pp. 3, 4).

[Sum22] T. Summerer. *Graphical Editors for Defining Scaling Policies Analysable Using Simulations*. BA thesis. University of Stuttgart. June 2022 (cit. on pp. 1, 5).

[Wol23] S. Wolfram. *What Is ChatGPT Doing:... and Why Does It Work?* Wolfram Media, 2023 (cit. on p. 8).

All links were last followed on March 20, 2025.

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