Architectural Lens: A Tool for Generating Comprehensible Diagrams

1st Jesper Kronborg Rusbjerg
MSc in Computer Science
IT University of Copenhagen
Copenhagen, Denmark
Course code: KISPECI1SE
jrus@itu.dk

2nd Nikolai Perlt Andersen
MSc in Computer Science
IT University of Copenhagen
Copenhagen, Denmark
Course code: KISPECI1SE
npan@itu.dk

May 30, 2023

Abstract

In this report, we introduce Architectural Lens, an automated diagram-generation tool. The tool's purpose is to tackle the challenges of creating comprehensible architectural documentation and keeping it up-to-date.

Architectural Lens, demonstrated through a Python-based implementation, is designed to be language-agnostic, and capable of being adapted to various programming languages. The tool empowers developers to generate scoped views of the system architecture, enhancing system comprehension and maintainability. In addition to scoped views, a key feature of Architectural Lens is its "difference view". This feature creates a visual comparison between the local state of the codebase and a specified version, typically the main/master branch in a remote git repository, added or deleted packages and dependencies are colour-coded for clarity. This clear visualization of changes supports developers in maintaining an accurate understanding of the system architecture and what changes.

To assess the tool's effectiveness, workshops were conducted with professional developers working in complex software environments, enabling them to apply the tool to their own real-world projects. Subsequently, a questionnaire was administered to collect feedback from the participants. The feedback was analyzed using thematic analysis to evaluate the tool's impact on creating comprehensible documentation and ensuring its up-to-dateness.

The feedback received from the participants emphasized the effectiveness of Architectural Lens in generating accurate and comprehensible architectural documentation. It not only facilitated the creation of comprehensible and up-to-date diagrams but also offered additional benefits such as promoting a clear understanding of specific system components, facilitating onboarding, improving team communication and enhancing system maintainability. These findings highlight the significant impact of Architectural Lens on various aspects of software development.

This conclusion establishes that Architectural Lens effectively addresses the challenges associated with creating and maintaining comprehensible architectural diagrams for complex software systems. The research outcomes provide valuable insights into enhancing architectural documentation practices in software development and lay a solid foundation for further exploration and refinement in this domain.

Contents

1 Introduction

Software architectural documentation is an important part of software development, as it offers a high-level understanding of a system's structure and design. Effective architectural documentation can facilitate communication among developers, thereby promoting code maintainability and helping prevent the decline of a system's architecture quality. [1]

Despite its importance, creating effective architectural documentation is a complex task, often fraught with common errors and oversights. Inadequate documentation is widespread in software development [2], often resulting from time constraints, lack of prioritization, or insufficient tooling. [3]

One of the most prevalent is the issue of outdated documentation. As the system evolves, architectural documentation can quickly become obsolete if it's not regularly maintained [2].

Another common issue in software documentation is the generation of overly complicated architectural diagrams. When diagrams are too detailed or contain unnecessary information, they can obstruct clear understanding and can hinder effective communication among developers [2].

Keeping architecture documentation up-to-date in complex systems can be a major challenge. [2] Changes to one component can have ripple effects across the system, requiring updates to multiple documentation artefacts. Additionally, the documentation process itself may be slow and cumbersome, making it difficult to keep up with changes in the system. [4]

To investigate the problems of outdated documentation and overly complicated diagrams, the study focuses on the following research question: **"How can the utilization of an automated diagram-generation tool help the creation of software architectural documentation, such that the diagrams are comprehensible and remain up to date, and what are the potential other benefits of using such a tool?".** By **automated diagram-generation tool**, the report refers to a tool that can accurately generate architectural diagrams directly from a system's source code. Additionally, we consider a **complex system** as one with more than 25 interconnected modules. It is important to note that there is no universally accepted standard for complexity, as it can vary depending on multiple factors. Additionally, by **comprehensible diagrams**, we refer to diagrams that can be understood by a developer.

In the realm of automated diagram-generation tools, facing the complexities of modern systems has been a recurring challenge, frequently leading to static, convoluted, and difficult-to-interpret views. To tackle these limitations, we introduce our contribution: Architectural Lens[1], a Python-based automated diagram-generation tool.

Architectural Lens derives its name from its ability to provide developers with scoped and customized views of their systems. Developers have the freedom to determine the level of detail for each view, allowing them to create scoped views of the domain model. This flexibility enables adjustments to be made to simplify complex views, ensuring better comprehension and alignment with their specific requirements. Thereby, by leveraging Architectural Lens, developers can visualize and review specific sub-parts of their systems. Lastly, Architectural Lens has a feature which highlights differences in the source code between two branches.

The diagrams that Architectural Lens generates are rendered in UML, a standard graphical language for illustrating software systems [5]. These UML diagrams are crafted using PlantUML, a tool that facilitates the generation of diagrams from simple textual descriptions. Our focus in this study is on module diagrams, it is important to note that the principles and concepts discussed may be transferable across different diagram types, such as domain or class diagrams, suggesting the broader applicability of our research.

[1] https://github.com/Perlten/Architectural-Lens

To avoid linking Architectural Lens to a specific language, we will refer to Python packages as "modules" and Python modules as "files" throughout this report. However, it is important to note that we will use Python-specific terminology when describing technical details about Architectural Lens. The use of Python-specific language in this context of describing Architectural Lens ensures precision and clarity when discussing Architectural Lens's implementation details.

To answer the research question, we will first explore the background and related work on software architectural diagrams, their importance, and the limitations of existing tools. We will then describe the tool, Architectural Lens. Next, we will conduct workshops introducing the tool and how to use it to developers and allow them to try it on their projects. Subsequently, participants will complete a questionnaire to provide feedback on the effectiveness of Architectural Lens in addressing the challenge of creating and maintaining comprehensible architectural documentation, as well as any suggestions for improvements or future work.

Based on the collected feedback, we will conduct a thematic analysis to examine the results and discuss their implications. This will be followed by a discussion of the limitations and reliability of the research. Finally, we will present a concise conclusion summarizing our contributions, findings, and future work.

Through our research and development efforts, we strive to offer a viable solution for maintaining up-to-date and comprehensible architectural diagrams for complex systems. By utilizing Architectural Lens, we aim to improve software systems' overall quality, maintainability, and understandability.

2 Background

This section explores the challenges and best practices for software architecture documentation in various contexts, such as continuous software development, software life cycle stages and complex systems. Effective documentation ensures software systems remain adaptable and maintainable. [6] [7]

2.1 Challenges in Software Documentation

Numerous studies have been conducted to identify the challenges that developers face while creating software documentation. For example, Rost et al. (2013) conducted a survey of 147 developers to investigate their experiences with software architectural documentation. The study identified four key findings related to the challenges of creating and maintaining documentation. [2]

1. Architecture documentation is often not up-to-date.
2. Architecture documentation is often provided in a "one-size-fits-all" manner.
3. Architecture documentation is often inconsistent.
4. Architecture documentation does often not provide sufficient navigation support to find the right information easily.

In 2020, a paper titled "Software Documentation: The Practitioners' Perspective" provides further insight into the challenges that developers encounter with software documentation. In addition, the paper also highlights the specific areas where developers find the documentation to be lacking regarding up-to-dateness. According to the paper, the two most significant issues are [3]:

1. Missing documentation for a new feature/component (69%)
2. Outdated/Obsolete references (64%)

The aforementioned paper also highlights the most significant concerns of developers regarding the readability of software documentation, which include [3]

1. Clarity (88%)
2. Conciseness (49%)

2.2 Software Life Cycle and the Continuous Process of Architectural Documentation

Software development is a continuous process that encompasses various stages [8]. In this section, we present the role of software architecture documentation in different stages of the software life cycle and how these insights inspire our proposed solution.

2.2.1 Planning and Analysis

During this stage, documentation clarifies system requirements and design, promotes discussions, and ensures the development team comprehends both business and technical needs. [9] The upfront creation of

documentation and architectural views emphasizes the importance of generating architectural documentation or system views throughout development. Continuously generating documentation enables comparisons with the original plans and facilitates discussions to confirm that the implemented architecture aligns with the initial design and requirements. Despite these precautions, it's important to note that architectural knowledge can be lost as soon as development begins, especially if the architectural documentation and the code start to diverge [9]. This lack of conformance can render the documentation irrelevant and of little use to developers [9].

2.2.2 Development and Testing

During the development and testing stage, documentation plays an important role in understanding the system's architecture, modules, and dependencies, reducing errors and inconsistencies [9].

There are two types of documentation to consider in this context. First, documentation is created prior to system development, such as architectural plans or initial design documents. Such documents are valuable for understanding the initial thoughts and intended structure of the system. While this documentation may not fully reflect the actual system as it evolves over time, it provides valuable insights into the initial vision and intended quality attributes of the system.

Second, documentation created during development that accurately captures the evolving system architecture is important. Up-to-date documentation aids developers in effectively navigating the codebase, understanding module purposes and responsibilities, and ensuring a clear understanding of the system's structure.

2.2.3 Deployment and Maintenance

Architectural documentation plays an important role in aiding troubleshooting, and ongoing maintenance. [10] [11] As software systems constantly evolve and updates are made, developers need a comprehensive understanding of the system to minimize the occurrence of architectural mistakes. By integrating architectural documentation creation into the software development process, developers can ensure that they have the necessary knowledge and insights to make informed changes and mitigate risks.

2.3 Software Architecture Adaptability

Designing adaptable and maintainable software architectures is critical for the long-term success of software systems [12]. This is particularly significant as architectural changes, when not handled appropriately, can lead to significant costs. These costs can stem from the extensive resources required to refactor large portions of the code, the risk of introducing new bugs or inconsistencies, and potential impacts on other interconnected systems.

A significant contributor to the need for refactoring is the accrual of technical debt. A study by Alves et al. identifies technical debt to often be a consequence of shortcomings in code design, architectural design and documentation [13].

A key aspect of adaptable software architecture is designing systems that are loosely coupled and highly cohesive. In the context of software architecture, a loosely coupled system is one where each of its components is independent and has little knowledge of the others. On the other hand, high cohesion refers to how closely the responsibilities of a single module or component are related to each other [14]. Designing systems

with these characteristics facilitates change and improves quality attributes, such as maintainability and testability [15].

2.4 Software Architecture Documentation in Complex Systems

Complex software systems present challenges for software architecture documentation. This section will explore the specific challenges in these systems and their impact on the documentation process.

2.4.1 The Complexity of Software Systems

Software systems are often very complex, with many parts working together and depending on each other. This makes documenting the architecture of these systems difficult because it requires capturing a lot of information about different layers and components. Additionally, even when the documentation is done correctly, the large static view can be hard to understand, making it difficult to learn about the system. [2]

2.4.2 Keeping Documentation Up-to-Date

Maintaining current and accurate architecture documentation for complex systems is a significant challenge. Whenever a component of the system changes, it can lead to effects that ripple through the entire system, which necessitates updates to several pieces of documentation. [4] Moreover, the process of updating documentation can often be time-consuming and labour-intensive, making it hard to keep pace with ongoing changes in the system. This raises the importance of having mechanisms that ensure the regular updating of architectural diagrams to mirror the evolving state of the system.

2.4.3 Documenting Interactions and Dependencies

Documenting interactions and dependencies between components in complex systems is critical to understanding the system's overall architecture. [16] However, this can be challenging, as interactions and dependencies may be difficult to capture clearly and concisely. It may be hard to read if a view gets populated with too many arrows and boxes [16].

2.5 Agile Practices and Architectural Documentation

Agile development methodologies have gained significant traction in software development due to their adaptability and responsiveness to change. [17] While these methodologies prioritize working software over comprehensive documentation, it's important to understand how architectural documentation fits into these methodologies.

2.5.1 The Agile Perspective on Documentation

Agile methodologies encourage producing working software and responding to changes over generating comprehensive documentation. [18] However, Sommerville emphasizes the importance of maintaining architectural documentation, even in small-scale systems and agile teams [19]. Despite the agility and adaptability of the agile development methodology, there can be a tendency to overlook or let the architectural documentation become outdated. The value of this documentation diminishes over time unless it is regularly updated

to reflect the evolving system. Documentation that fails to stay in sync with the changes in the system risks losing its relevance [9].

2.5.2 Balancing Agility and Documentation

Balancing agile development with effective architectural documentation can be challenging. In an environment where changes are continuous, keeping documentation up-to-date can be cumbersome. However, outdated or unclear documentation can create misunderstandings and difficulties in maintaining and evolving the system's architecture, especially as teams change over time.

3 Related work

In designing Architectural Lens, our objective was to draw upon established knowledge and best practices in the field of software architecture documentation and visualization. To accomplish this, we conducted an extensive search for relevant literature, examining the findings to uncover insights, recommendations, and potential challenges that could inform our approach. In this section, we discuss the academic research and existing tools we reviewed and elaborate on our criteria for selecting and evaluating the tools we encountered.

3.1 Existing Tools

In the field of software architecture documentation, a variety of approaches and tools have been developed to assist in generating and maintaining architectural diagrams. These tools utilize different techniques and methodologies, ranging from manual drawing approaches to automated solutions. In this section, we explore different categories of tools that have been employed to create architectural diagrams, including manual drawing approaches, diagrams as code, and automated tools. By exploring these different approaches, we can gain insights into the strengths and limitations of each method and provide a comprehensive overview of the existing landscape of software architecture documentation tools.

3.2 Manual drawing approach

A manual approach to architectural documentation involves creating it without relying on specific tools or automation. This method allows developers to express their architectural vision in as much detail as desired, enabling a high degree of customization and flexibility. Developers have the freedom to choose the level of granularity and the specific elements they want to include in the documentation. However, there are drawbacks to this approach, manual documentation is prone to becoming outdated as the system evolves over time, requiring significant effort to keep it synchronized with the actual codebase. One tool that supports manual documentation creation is draw.io, which provides a versatile diagramming platform for creating detailed architectural diagrams and visual representations [20].

3.3 Diagrams as code

This approach involves using textual languages, such as programming languages or text-based DSLs, to create diagrams. This approach offers benefits like easier version control, scripting possibilities, and integration into the development pipeline, enabling automation of documentation processes [21]. Various tools, including Structurizr DSL [22], Diagrams [23], PlantUML [24] Mermaid [25] and Graphviz [26], provide the flexibility for developers to create detailed and precise views of the architecture, including only the aspects they deem important. However, as discussed earlier, one challenge with these tools is the difficulty of keeping the documentation up to date.

3.4 Automated Tools

On the other hand, automated tools have taken various approaches to address the challenge of keeping the system in sync with its documentation. These tools offer features such as automatic documentation generation, reminders for documentation updates, and mechanisms to ensure that the documentation accurately reflects the current state of the system.

3.4.1 Swimm

Swimm aims to bridge the gap between documentation and source code by tightly coupling documentation with the codebase itself. [27] It provides an integrated solution that automatically detects changes in the codebase and prompts users to update the corresponding documentation sections, assisting the developer keep the documentation up to date.

3.4.2 Pyreverse

Pyreverse, a tool that is part of pylint [28], offers the advantage of automatically generating architectural diagrams by analyzing the source code. This automation simplifies the process of aligning the documentation with the actual source code. Additionally, Pyreverse provides functionality through its CLI to ignore certain packages, allowing developers to filter certain elements from a system view. However, these automatically generated diagrams tend to be large and complex as they represent the entire system, making them challenging to comprehend. Tools similar to Pyreverse include Doxygen [29] and Umbrello [30].

3.4.3 Archjava

Archjava takes a different approach. Rather than creating documentation based on the source code, they facilitate the creation of source code based on the documentation. This approach ensures that the documentation remains up to date because the code is generated directly from the architectural specifications. Furthermore, tools like archjava prioritize maintaining the architectural design specified by the original design, which is not guaranteed by the other solutions. This integration between the documentation and the code ensures not only accuracy but also adherence to the intended architectural design [31]. Another tool that is also able to convert documentation to source code is Umbrello [30].

3.5 Diagrams as Code and Automated Tools

Both "Diagrams as Code" and "Automated Tools" have their merits. Diagrams as code allow developers to define diagrams using familiar text-oriented tooling, offering control and automation possibilities. However, it requires developers to define the entire system in code, which can be cumbersome and time-consuming to maintain. On the other hand, automated tools offer various approaches to keep the documentation in sync with the codebase. However, it is worth noting that automated tools like Pyreverse may produce complex and extensive diagrams, which can pose challenges regarding comprehension, complexity and readability.

Potentially, integrating the strengths of both approaches could steer towards a more viable solution. By leveraging diagrams as code, which allows for flexible and customizable representations, we can create scoped and precise views. By incorporating automation similar to that of automated tools, we can enhance the efficiency and effectiveness of generating and maintaining comprehensible architectural diagrams. This hybrid approach combines the benefits of both methods, empowering developers to define tailored views while automating the generation process. Merging the principles of **diagrams as code** and **automated tools** lays the foundation for Architectural Lens, a tool which is designed to implement an approach for producing comprehensible architectural diagrams using a combination of the principles.

4 Method

To address the research question, "How can the utilization of an automated diagram-generation tool help the creation of software architectural documentation, such that the diagrams are comprehensible and remain up to date, and what are the potential other benefits of using such a tool?" we applied the following methodological approach.

4.1 Literature Review, Search Strategy and Criteria

This subsection outlines the methodology used in this paper to conduct a literature review and identify relevant background works and related tools. It provides an overview of the search strategies employed and highlights the process undertaken to gather relevant information in the field.

4.1.1 Background

To identify relevant literature for the background works of this study, we employed a search strategy that utilized Google Scholar and focused on keywords such as "Challenges of software architectural documentation", "Current issues with software architectural documentation", "Software Life Cycle and documentation", "Agile software documentation" and "The affect of architectural documentation on software quality". We began by screening the search results, evaluating their relevance to our research objectives, and prioritizing papers with significant academic citations. We then delved into the references of the selected papers to uncover additional material that may be pertinent to our study. We aimed to conduct a comprehensive search while remaining mindful of our project's time and resource constraints.

4.1.2 Tools

To explore related tools in the field, we conducted searches on Google using relevant keywords such as "automated architecture documentation tools", "diagram as code tools", "automated architecture documentation tools in python", and "architectural documentation tools". The search results included websites like Stack Overflow, where developers shared their comments and recommendations on tools they found useful which lead to recommended tool being further explored. We also found dedicated websites directly showcasing various tools for architectural documentation. We carefully examined the feature sets of these tools and, whenever possible, even tested them ourselves to gain a better understanding of their contributions to architectural documentation creation. After identifying a tool, we conducted additional research to investigate if there were any relevant academic studies or publications specifically addressing that tool using Google Scholar. If such studies were found, we leveraged them to gather further insights into the capabilities and potential benefits of the tool.

4.2 Tool Development and Iterative Feedback

We developed a software tool designed for Python projects, although the requirements it adheres to could be implemented in any programming language. This emphasizes the tool's inherent flexibility and adaptability across different programming contexts. The tool is designed with customization in mind, allowing developers to adjust diagrams according to their specifications, thereby facilitating the creation of detailed sub-views that provide a more granular perspective on the domain model.

The validation and reliability of our software tool were ensured through the development of a tool. We solicited iterative feedback from our supervisor, who utilized the system on a Python project. This feedback provided essential insights which were then carefully analyzed and discussed within our team. Based on these discussions, we implemented necessary changes to enhance the system, addressing bug fixes, usability improvements, and the comprehensibility of the generated documentation. Upon reaching a stage where we had a minimum viable product, we initiated the recruitment of developers for our workshop and study.

4.3 Evaluation of Architectural Lens

To effectively assess the potential of Architectural Lens in continuous software development, we have defined a process that outlines how the tool should be used within the context of this study.

Drawing inspiration from the iterative software development approach, which involves cyclic repetition and refinement of development activities, this process guides participants to write code, generate diagrams based on their code, review and refine the diagrams as necessary. This iterative process, referred to as "the process" throughout the questionnaire and subsequent discussions, ensures that each participant has a consistent experience when working with Architectural Lens. By establishing this approach, we aim to maintain uniformity and enable meaningful comparisons across participants in evaluating the impact of Architectural Lens.

4.4 Workshop

We conducted workshops with developers experienced in Python to collect diverse feedback on Architectural Lens. As the tool presented in this paper was developed in Python, we recruited workshop participants based on their programming language experience, specifically with daily Python development. Additionally, we valued Involvement in various projects, and adherence to agile practices such as continuous use of feature branches and pull requests. This selection criterion was important to ensure that they could effectively incorporate the intended use of Architectural Lens into their existing workflows and provide valuable insights and feedback.

The workshops were structured as follows:

1. **Introduction**: The workshop began with a discussion of the purpose and objectives of Architectural Lens and highlighting current issues in architectural documentation. We introduced our solution, aiming to create comprehensible automatic documentation that addresses these challenges.

2. **Presentation**: We demonstrated the Architectural Lens tool and the intended process of iteratively applying Architectural Lens to generate customized views as part of the continuous documentation process on a demo project.

3. **Case Study: Zeeguu**: We chose Zeeguu-API as a case study for applying Architectural Lens. Zeeguu-API is an open-source API designed to track and model a learner's progress in a foreign language, with the goal of recommending paths to accelerate vocabulary acquisition. The application of Architectural Lens to this project illustrates the entire workflow, demonstrating how it can be integrated into the development process to create comprehensible architectural documentation.

4. **Hands-on Activity**: Participants were guided through installing Architectural Lens and applying it to their projects, with assistance provided as needed.

5. **One-week Testing Period**: Participants were given one week to use Architectural Lens in their projects, during which we maintained regular communication to offer support and address any questions or issues.

6. **Reflection and Feedback**: After the testing period, participants completed a questionnaire about their experiences with generating documentation iteratively using Architectural Lens. Their detailed feedback provided insights on usability, effectiveness, and areas for improvement or future research directions.

This structured approach to the workshops allowed us to gather comprehensive and valuable feedback, which was subsequently analyzed using a thematic analysis approach to evaluate the effectiveness of Architectural Lens.

4.5 Analysis

Upon completing the workshop, we used the thematic analysis method to identify common themes and patterns in the data, which informed our conclusions regarding Architectural Lens. Additionally, we identified the potential benefits and drawbacks of this combined approach.

Thematic analysis is a qualitative research method used to identify patterns or themes within unstructured data, such as questionnaire responses. It involves identifying, analyzing, and interpreting patterns or themes in the data that emerge from the participants' responses. [32]

To conduct thematic analysis, we read through the responses multiple times to gain a deep understanding of the content. We then proceeded by labelling sections of the responses with descriptive or interpretive labels that captured the meaning of the text. The labelled questionnaires can be found in the Appendix section.

Afterwards, we reviewed the labels to identify recurring patterns, themes, or categories that emerge across the data. We compared these labels, which allowed us to see how different themes relate to one another and the overarching research question. Finally, we interpreted the themes and their relationships to each other to draw conclusions and implications for the research.

5 The automated tool

5.1 Architectural Lens

Architectural Lens is designed to facilitate the process of generating and maintaining architectural documentation for complex software systems. It aims to provide developers with comprehensible visual representations of a system's architecture, support the identification of potential architectural issues, and track changes between different branches of a project. The tool's main objectives include the following:

1. Automating the generation of architectural documentation and minimizing the risk of outdated or incomprehensible documentation.
2. Facilitating system understanding by providing focused, comprehensible, and developer-defined views of the system's architecture, addressing the challenge of large and complex static views.
3. Assisting in the identification of architectural issues, such as violations of design patterns, thus enabling developers to become aware of and rectify these issues.

Additionally, the tool aims to assist in understanding a system's architecture by automatically generating developer-customized visual representations of the relationships between modules and dependencies based on the developer's specifications for what to include or exclude. The tool achieves this by parsing the source code, creating dependency graphs, and generating two types of customized architectural diagrams in PlantUML, Figure 1 represents the first type: Render View.

Figure 1: Flow of Render View.

The two types of diagrams able to be created are the following:

1. Render views: These PlantUML diagrams display the system's architecture using boxes and arrows to represent modules and their relationships.
2. Difference views: Based on the Render views, these PlantUML diagrams showcase architectural changes between the working branch and a specified branch. Added dependencies or modules are marked in green, and removed ones are marked in red, making it easy to visualize changes in the architecture.

PlantUML is an open-source tool that allows developers to create diagrams using a simple and intuitive textual notation. It supports various types of diagrams, including class diagrams, module diagrams, sequence diagrams, and activity diagrams [24]. With PlantUML, developers can describe the structure and relationships of software systems in a textual format, which is then automatically rendered into visual diagrams.

5.2 Tool Architecture & How the Tool Works

This section provides a detailed explanation of the tool's inner workings, focusing on the process of analyzing the code, generating visualizations, and creating the difference views. The core of the tool lies in the three main algorithms: one that analyzes each source code file in the project using an Abstract Syntax Tree

(AST) to construct a graph of the system, a second that generates a view from the graph by converting it into PlantUML syntax, and the third that builds on top of the second to create a difference view. These algorithms are the pillars and the foundation of the tool's functionality.

5.2.1 Parsing the Source Code with an Abstract Syntax Tree (AST)

The first algorithm analyzes each source code file in the project using an Abstract Syntax Tree (AST). The AST represents the syntactic structure of the code, allowing the algorithm to identify and extract relevant information such as classes, modules, and their dependencies. This information is then used to construct a graph representing these elements' relationships.

5.2.2 Generating Render View: Converting the Graph into PlantUML and Filtering Modules

The second algorithm takes the dependency graph generated by the first algorithm and converts it into PlantUML syntax. This conversion process translates the nodes and edges of the graph into corresponding PlantUML syntax, such as packages and dependency arrows, thereby creating a render view of the project's architecture. The algorithm considers the developer's specifications for what should be included in the view during this process. These specifications are defined in a JSON configuration file, which the developer can use to customize the architectural views.

5.2.3 Creating Difference Views: Extending the Render View

The third algorithm is responsible for generating difference views. This process involves utilizing the first algorithm to create an additional dependency graph based on the specified Git repository and branch from the JSON configuration file. This new dependency graph will be compared to the one created for the render view, to find and highlight the differences.

The algorithm then compares the two dependency graphs: the one generated from the Git repository's specified branch, and the other generated from the developer's local working branch. It identifies the differences between the two graphs, highlighting added dependencies and modules in green and removed dependencies and modules in red. The third algorithm generates a new PlantUML representation that incorporates these differences, extending the render view to create a difference view. The process is presented in Figure 2.

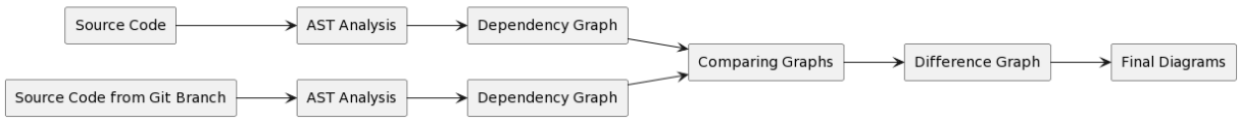

Figure 2: Flow of Difference view.

5.2.4 Rendering Diagrams and Difference Views Using a PlantUML Server

After converting the dependency graph, render view, and difference views into PlantUML syntax using the three algorithms, the tool uploads the generated PlantUML code to our self-hosted PlantUML server. The server then processes the PlantUML code and returns the final diagrams and difference views in the desired output format, such as PNG. These visualizations can be saved to a specified location and used better to understand the project's architecture and changes between branches.

The three algorithms at the heart of the tool provide an efficient and automated way to generate architectural visualizations that facilitate system understanding and help identify potential architectural mistakes.

5.3 Architectural-Lens: Implementation details & Developers perspective

The name Architectural-Lens signifies the tool's ability to provide focused, customizable views of the software architecture, much like a lens can offer a tailored perspective. Python was selected for several reasons: the developers' familiarity with the language and the availability of Python projects for testing, including a project supervised by our supervisor. This decision enabled a smooth testing process and facilitated the development of a tool that works in conjunction with the process.

The Python implementation supports Python 3.9, 3.10, and 3.11. In developing the tool, we leveraged the Astroid library for parsing the source code with an Abstract Syntax Tree (AST). We also developed a custom Graph class for representing dependency graphs and implemented the algorithms for generating render views and difference views in pure Python.

The README and source code for our project can be found here: https://github.com/Perlten/Architectural-Lens.

5.3.1 Language Agnostic

To ensure language agnosticism, Architectural Lens does not rely on any Python-specific features that could limit its compatibility. This design choice allows for the implementation of Architectural Lens in various programming languages, making it language agnostic. Although Architectural Lens is primarily described in the context of its Python implementation in section 5.2, it is important to note that the tool is not limited to Python alone.

5.3.2 Installation

The Python implementation of our tool, Architectural-Lens, has been packaged to enhance accessibility.

Architectural-Lens can be found on the Python Package Index (PyPI) at https://pypi.org/project/Architectural-Lens/, available for installation via the pip package manager. Execute the following command to install:

```
pip install Architectural-Lens
```

5.3.3 Configuration and Usage

The tool utilizes a JSON configuration file to define various user inputs, including the project name, root folder, GitHub repository, branch, save location, and views. To generate the JSON file, users can run the command `archlens init`, which creates a default configuration file named `archlens.json` in the project's root directory. This file serves as a blueprint for customizing the views of the system.

To provide a comprehensive understanding of the JSON file's structure and elements, we will now explain the contents of the entire file. However, in subsequent showings involving the JSON file, we will focus only on the most important aspects to avoid redundancy and repetition.

```
1  {
2      "schema":
           "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
3      "name": "project_name",
4      "rootFolder": "rootfolder_of_project",
5      "github": {
6          "url": "link_to_github_repo",
7          "branch": "branch_to_compare_with"
8      },
9      "saveLocation": "./diagrams/",
10     "views": {
11         "completeView": {
12             "packages": [],
13             "ignorePackages": []
14         }
15     }
16 }
```

- Line 2: Specifies the schema for the JSON configuration file.
- Line 3: Defines the project's name.
- Line 4: Indicates the source folder containing the root package (typically a folder named src).
- Lines 5-8: Provide details about the project's GitHub repository, including the URL and the primary branch's name.
- Line 9: Determines the location for storing generated diagrams.
- Lines 10-16: Outline the views for the architectural diagrams. In this instance, a single view called "completeView" displays the entire system.
- Line 12: Lists the packages to include in the view. When left empty, it will display the whole system view.
- Line 13: Identifies packages to exclude from the diagram.

Upon creating the views, a developer is able to create them using the command line interface, the following are all the available commands:

1. `archlens init`: This primary command generates the initial JSON configuration file.
2. `archlens render`: Produces render views of the architecture based on the JSON configuration.
3. `archlens render-diff`: Creates difference views between two architectural diagrams, facilitating comparison between versions or commits.
4. `archlens create-action`: Establishes repository actions to automate the generation and display of difference views in pull requests.

This subsection has presented various user inputs that are provided to Architectural Lens through a JSON configuration file. As mentioned in the previous subsection 5.3.1 (Language Agnostic), Architectural Lens is designed to be transferable to other programming languages. While the specific method of providing these configurations may vary across different programming languages, the key focus when implementing Architectural Lens in a new language is to maintain its core functionality.

6 Applying the Tool: Demo Project

In this section and the following, we will demonstrate the use of the Architectural Lens tool in two different scenarios. First, we will apply it to a simple demo project, which serves as an illustrative case to thoroughly explain and highlight the tool's core functionalities in the most understandable manner. Once the functionality has been comprehensively introduced and understood, we will proceed to showcase the tool on a real-world open-source system in the following section.

6.1 Setting Up & Entire Domain View

In this example, we will generate a comprehensive domain view of the system using the base configuration file, archlens.json. The JSON file is presented below:

```
1 {
2     "schema":
          "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
3     "name": "demo\_project",
4     "rootFolder": "tp_src",
5     "github": {
6         "url": "https://github.com/JesperRusbjerg/test_project",
7         "branch": "main"
8     },
9     "saveLocation": "./diagrams/",
10    "views": {
11        "completeView": {
12            "packages": [],
13            "ignorePackages": []
14        }
15    }
16 }
```

- Line 3: Defines the project's name.
- Line 4: Path to the root folder of the project, tp_src.
- Lines 5-8: Github link to the project.
- Line 9: Saving diagrams in "./diagrams".
- Lines 10-16: Outline the views for the architectural diagrams. In this instance, a single view called "completeView" displays the entire system.

By executing `archlens render`, Architectural Lens generates the entire view of the demo project based on the provided archlens.json file, which contains the "completeView" view in Figure 3.

The diagram displays various packages and their dependencies, symbolized by arrows linking them. These arrows represent the import relationships between packages. Besides the arrows, the dependency count appears, displaying the number of imports between the connected packages.

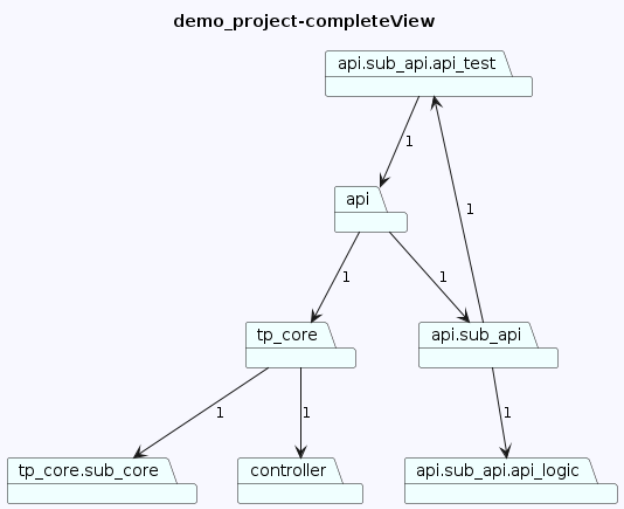

Figure 3: Demo project completeView.

6.2 Package Filtering, Depth View, Ignoring Packages

In complex software systems, the entire view is difficult to understand, [16] developers may prefer to create more focused views. In this example, we will generate an "apiView" that displays only the "api" module and its sub-system. This includes the "api" module and all the packages beneath it. To achieve this, we update the archlens.json file accordingly:

```
{
    "schema":
        "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
    "name": "demo_project",
    "rootFolder": "tp_src",
    "github": {
        "url": "https://github.com/JesperRusbjerg/test_project",
        "branch": "main"
    },
    "saveLocation": "./diagrams/",
    "views": {
        "apiView": {
            "packages": [
                "api"
            ],
            "ignorePackages": [
                "*test*"
            ]
        }
    }
}
```

- Lines 10-15: Defines the "apiView" which focuses only on the "api" and its sub-modules.
- Line 12: Specifies the packages to be included in the "apiView" (in this case, just the "api" package).
- Line 13: Specifies the packages not to be included in the "apiView" diagram (ignores all packages with "test" in their name).

The resulting diagram is shown in Figure 4.

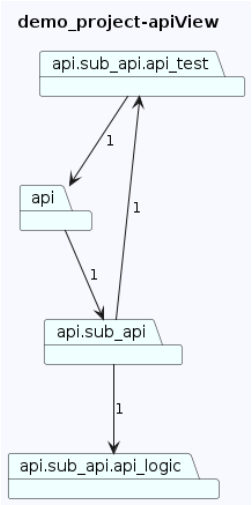

Figure 4: Demo project api filtered view.

Architectural-Lens can simplify views by using a depth specification. By specifying a depth value alongside the "packagePath," the diagram will only display the designated number of sub-package layers. Data from filtered-out sub-packages are aggregated to the nearest parent in the view, allowing the diagram to show only the requested components and their direct dependencies while still retaining relevant information through aggregation. The "packagePath" field specifies the desired package's location and must be used alongside the "depth" field in the same object. By setting the "depth" field to 1, for example, the diagram will display only one layer of sub-packages within the api package, providing a focused view of the architecture. In the JSON configuration file shown below, we use the depth specification to create a focused view of the "api" package with a depth of 1:

```
{
    "schema":
        "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
    "name": "demo_project",
    "rootFolder": "tp_src",
    "github": {
        "url": "https://github.com/JesperRusbjerg/test_project",
        "branch": "main"
    },
    "saveLocation": "./diagrams/",
    "views": {
        "apiView": {
            "packages": [
                {
                    "packagePath": "api",
                    "depth": 1
                }
            ],
            "ignorePackages": [
                "*test*"
            ]
        }
    }
}
```

- Lines 10-22: Defines the "apiView" with depth specification.
- Lines 12-18: Specifies an array of packages to be included in the "apiView." In this case, there is only one package, "api," with an additional depth property.
- Lines 13-17: Specifies an object containing the "packagePath" and "depth" properties for the "api" package. The "depth" property is set to 1, meaning only one layer of sub-packages within the "api" package will be shown in the diagram.

The resulting view with a depth of 1 is shown in Figure 5.

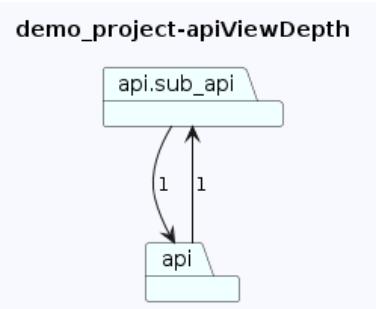

Figure 5: Demo project api view with depth 1.

Using the depth specification in the "apiView" (Lines 13-17), Architectural-Lens generates a diagram with only the "api" package and its direct sub-packages (Figure 5). The depth value of 1 ensures that only one layer of sub-packages is displayed in the view, resulting in a more focused and simplified diagram. To understand why api.sub_api now depends on api, we must look back to Figure 4 and observe that the sub-module api.sub_api.api_test relies on api. However, due to the depth constraint enforced on the view, the api.sub_api.api_test sub-module has been removed. As a consequence, its dependency on api has been aggregated upwards to its nearest parent, which is api.sub_api. Thus, the updated dependency indicates that api.sub_api now depends on api. This simplification of the view does not result in any dependencies being omitted; rather, it consolidates the dependencies through aggregation in a higher-level module.

6.3 Difference View

To create a difference view for the demo project, there is no need to modify the JSON configuration file. Instead, you run the `archlens render-diff` command rather than the `archlens render` command.

Suppose you have made changes to the demo project's codebase within the scope of a view and want to visualize those changes. Executing the `archlens render-diff` command, in conjunction with specifying a GitHub repository and branch within the config file for comparison, will generate the previously specified views. Any added package or dependency will be highlighted in green, and any deleted package or dependency will be shown in red. If the dependency count between two packages changes, the number on the arrows will adjust accordingly. An arrow will turn red only if the dependency count decreases to zero, while a new green arrow will appear if the count rises from zero to one or above.

Now, consider an example where you have added a new package called *controller_new*, which *tp_core* depends on, and you have removed the old *controller* package, resulting in the deletion of the package and all its dependencies.

Showcasing the difference by comparing the render view in Figure 7 and the render diff view in Figure 6, where the latter compares the changes to the master branch of the demo project.

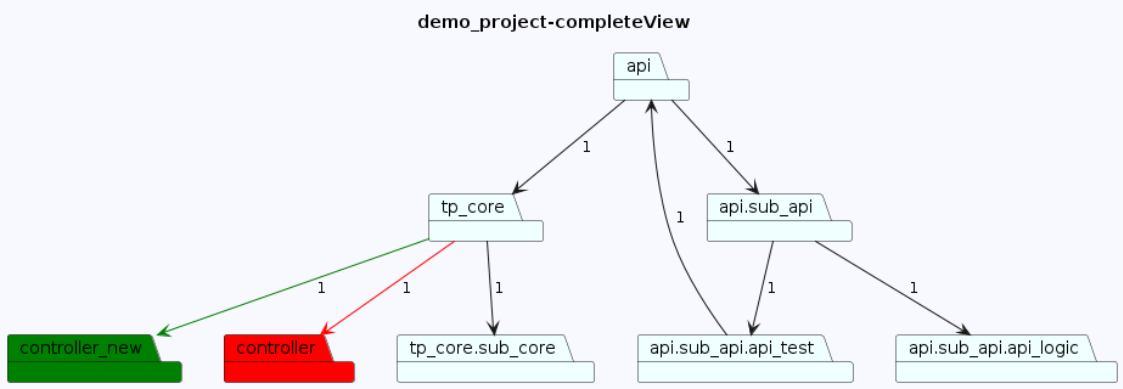

Figure 6: Difference view of the modified demo project.

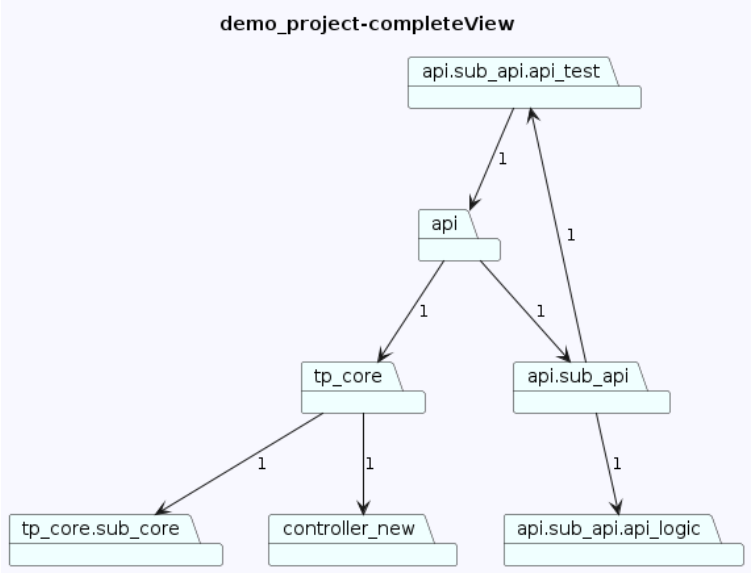

Figure 7: Non-difference view of the modified demo project.

6.4 GitHub Action for the Demo Project

When creating architectural documentation based on changes made to the demo project, a developer may want to create a pull request showcasing those changes. Presenting the entire system view to a reviewer could make it challenging to spot the differences and identify any mistakes. As specified in the configuration and usage subsection 5.3.3, the `archlens create-action` command is designed to generate a GitHub action for the repository where the code is stored. This action showcases difference views based on the archlens.json file, making it easy to spot the differences for the reviewer.

In our implementation, we use GitHub and GitHub actions. When a developer runs the `archlens create-action` command, files are added to the project's .github folder (the folder is created if it doesn't exist). Inside this folder, a GitHub action is placed, which follows these steps:

1. Renders each view as a "Difference view."
2. Uploads the images to our PlantUML server.
3. Attaches the returned images as comments in the pull request, making it easy for the reviewer to understand the documentation and what has changed.

Figure 8 displays the view of the action having run on Github, which is based on the previous example in Figure 6.

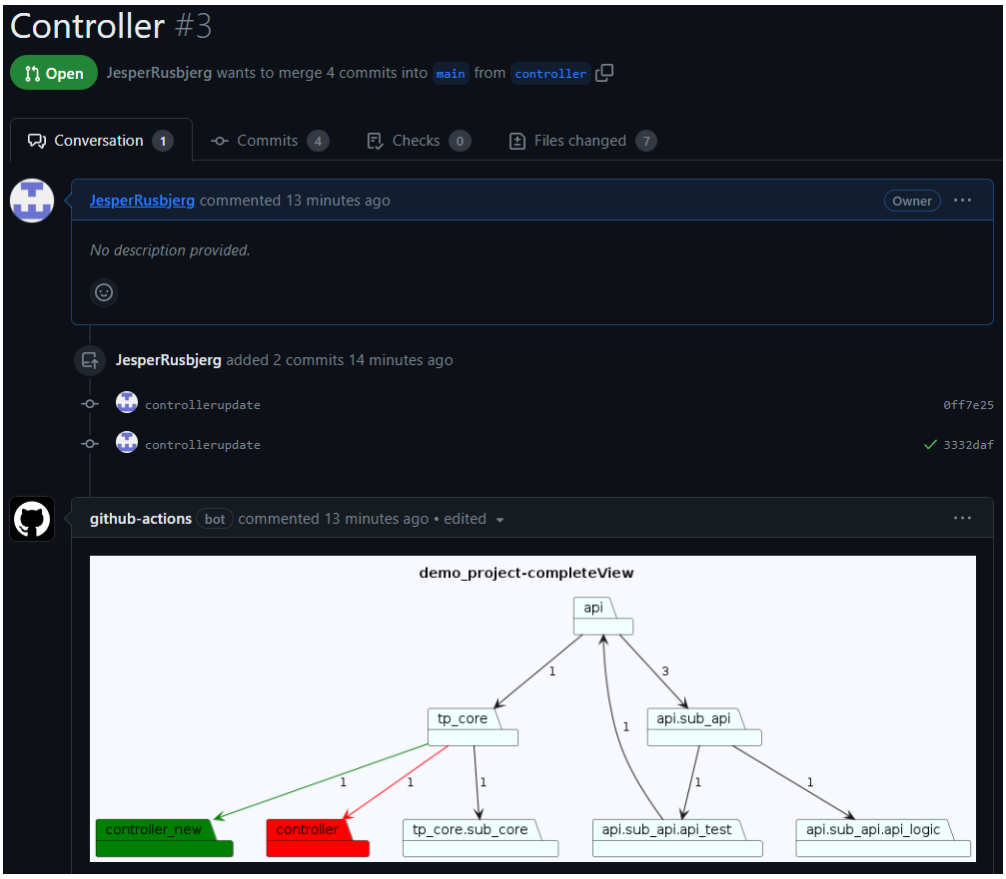

Figure 8: GitHub Action with Difference View.

This allows the reviewer to see the difference view for each pull request created, as long as the archlens.json file and the create-action configuration are present in the project.

7 Case Study: Applying the tool on Zeeguu

After showcasing the Architectural-Lens tool using a basic demo project to illustrate its capabilities, we now focus on a real-world project. This will highlight the tool's capabilities when applied in a more complex environment.

In the following subsections, we will examine the Zeeguu API, a Python project with 36 modules. The Zeeguu API is an open source API that enables tracking and modelling the progress of a learner in a foreign language, aiming to recommend paths for accelerating vocabulary acquisition. The Zeeguu API repository is available on GitHub: https://github.com/zeeguu/api.

The base configuration file for archlens.json is shown below, with the key elements explained:

```
1 {
2     "schema":
          "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
3     "name": "zeeguu",
4     "rootFolder": "zeeguu",
5     "github": {
6         "url": "https://github.com/zeeguu/api",
7         "branch": "master"
8     },
9     "saveLocation": "./diagrams/",
10    "views": {
11        "completeView": {
12            "packages": [],
13            "ignorePackages": []
14        }
15    }
16 }
```

The primary differences in the configuration file compared to the demo project are the project's name, root folder, and GitHub repository URL (Lines 3-8). The other parts of the configuration file remain the same.

In Figure 9, we present a miniature version of the entire view of the Zeeguu API project. Due to its complexity, the full-sized view cannot be accommodated within this report. However, a link is provided for those interested in exploring the complete view in more detail.

As illustrated, even with only 36 modules, the Zeeguu API's domain model is difficult to comprehend and will become increasingly challenging as the system grows. In the next subsections, we will create more digestible views to help explain the system.

7.1 Scoping the domain: Core and API view of Zeeguu

The Zeeguu API is built upon two main sections: the API and core. The API would typically depend on the core, and the core provides the essential business logic for the application. To gain a deeper understanding of the system's architecture, we can start by breaking it into two separate views: core and API.

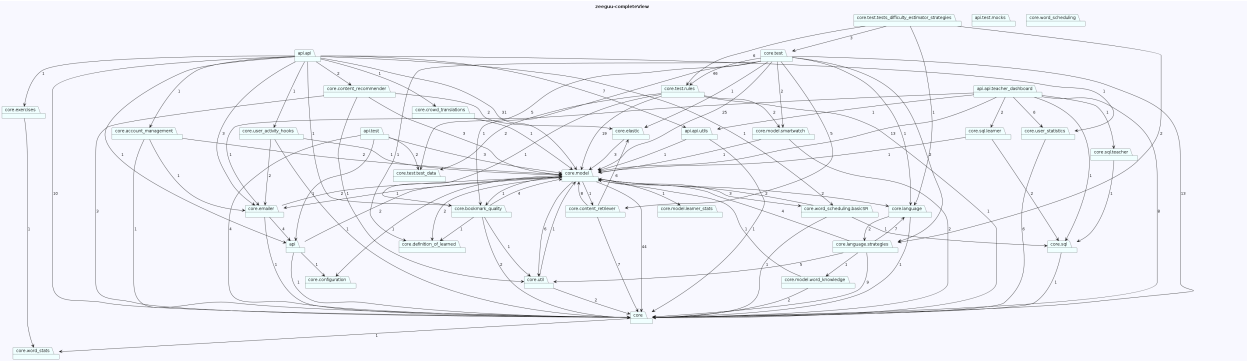

Figure 9: Zeeguu entire-view. Full size image is available here:
https://github.com/JesperRusbjerg/code/blob/master/zeeguu-completeView.png

We present the two views: Core view (Figure 10) and API view (Figure 11).

The core view config:

```
{
    "schema":
        "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
    "name": "zeeguu",
    "rootFolder": "zeeguu",
    "github": {
        "url": "https://github.com/zeeguu/api",
        "branch": "master"
    },
    "saveLocation": "./diagrams/",
    "views": {
        "coreView": {
            "packages": [
                {
                    "depth": 1,
                    "packagePath": "core"
                }
            ],
            "ignorePackages": [
                "*test*"
            ]
        }
    }
}
```

In this JSON example, we create a focused subview called "coreView" instead of generating a view encompassing the entire system. The configuration file assumes the existence of a package named zeeguu.core in the project, as indicated by the packagePath field in the JSON. If this path does not exist, the render will fail. The "packagePath" field specifies the path to the package of interest and must be accompanied by a

"depth" field in the same object. The "depth" field is set to 1, meaning that only one layer of sub-packages within the core package will be shown in the diagram. It is the "depth" field that controls the depth of the sub-packages displayed in the resulting view.

The "ignorePackages" field filters out any package with the word "test" in its name, further refining the view and making it easier to focus on the essential components of the project's architecture.

The coreView configuration generates the architectural diagram seen in Figure 10.

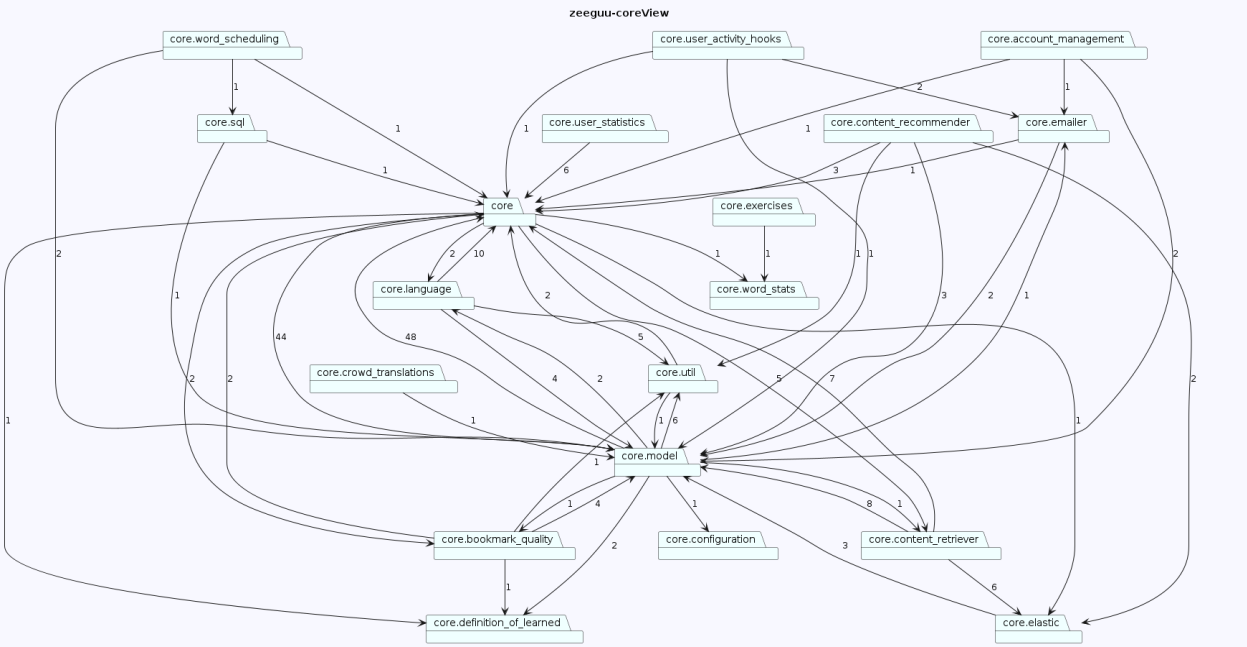

Figure 10: Zeeguu CoreView.

It is important to note that you can add multiple "allowed package paths/objects" and multiple ignorePackages if you wish to create even more specified views.

In addition to core, let's create a view of the "API" side of the Zeeguu API. We've added "api" as an allowed path, meaning that we only allow packages that begin with the name zeeguu.api.

the API View config:

```
1  {
2      "schema":
           "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
3      "name": "zeeguu",
4      "rootFolder": "zeeguu",
5      "github": {
6          "url": "https://github.com/zeeguu/api",
7          "branch": "master"
8      },
9      "saveLocation": "./diagrams/",
10     "views": {
11         "apiView": {
12             "packages": [
13                 "api"
14             ],
15             "ignorePackages": [
16                 "*test*"
17             ]
18         }
19     }
20 }
```

In this scenario, we do not specify a depth, meaning that we will see the entire subsystem of the "api" package (as shown on line 13). Again, we are not interested in seeing any package with the word "test" in its name.

The "API" side of the Zeeguu API can be seen in Figure (11)

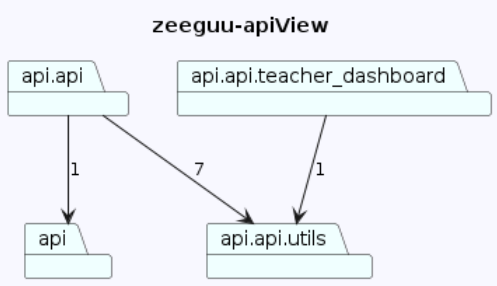

Figure 11: Zeeguu API View.

Now, by simply dividing the domain into the two main sections of the application, it is arguably easier to gain an overview of the system. However, we can scope it even further if necessary to create more focused and comprehensible views of the architecture, providing additional clarity and understanding of the system's components and dependencies.

7.2 Focusing on Specific Domain Components

Suppose you are a developer working on two closely connected modules: `core.model.word_knowledge` and `core.language`. These modules are situated 4-5 layers deep within the system, making them difficult to spot in an automated view of the entire system. To better understand the relationships and dependencies within this specific domain, you can create a focused architectural view for these modules by following the steps we demonstrated in the demo project section. Start by creating a JSON configuration file with the necessary information, as shown below:

```
{
    "schema":
        "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
    "name": "zeeguu",
    "rootFolder": "zeeguu",
    "github": {
        "url": "https://github.com/zeeguu/api",
        "branch": "master"
    },
    "saveLocation": "./diagrams/",
    "views": {
        "languageAndWordView": {
            "packages": [
                "core.language",
                "core.model.word_knowledge"
            ],
            "ignorePackages": []
        }
    }
}
```

This configuration will generate a view of these two modules, their subsystems, and their relationships/dependencies. As with the demo project, you can achieve this by running the `archlens render` function, resulting in the image presented in Figure 12.

Although this information is present in the view of the entire system, it would be challenging to locate and comprehend the clutter and numerous arrows present in the complete view.

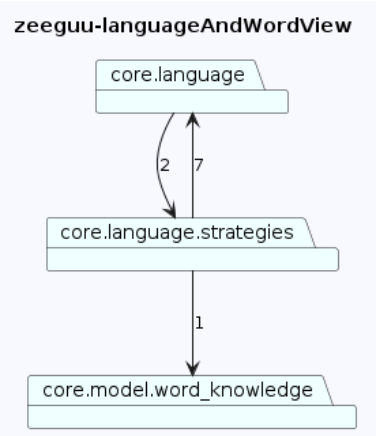

Figure 12: Zeeguu Language and word knowledge view.

7.3 Identifying Architectural Errors

In complex software systems like Zeeguu, spotting architectural errors such as unwanted bi-directional dependencies can be challenging due to the cluttered nature of the overall view. In Zeeguu's case, there is an unexpected bi-directional dependency between its two main components, namely, the core and the API. To identify this issue, one can create a focused view of the two top levels of the system with a depth of 1. This view will display the two modules with their entire sub-systems aggregated within their respective boxes, revealing the bi-directional dependency between them, as illustrated in the following JSON example and Figure 13.

```
{
    "schema":
        "https://raw.githubusercontent.com/Perlten/Architectural-Lens/master/config.schema.json",
    "name": "zeeguu",
    "rootFolder": "zeeguu",
    "github": {
        "url": "https://github.com/zeeguu/api",
        "branch": "master"
    },
    "saveLocation": "./diagrams/",
    "views": {
        "topLevel": {
            "packages": [
                {
                    "packagePath": "",
                    "depth": 1
                }
            ],
            "ignorePackages": []
        }
    }
}
```

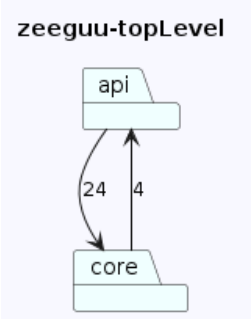

Figure 13: Zeeguu top level view.

The aggregation of sub-systems plays an important role in summarizing dependencies and clearly demon-

strating the relationships between modules at a higher level, making it easier to spot architectural issues.

When dealing with larger views of a system, Figure 13 demonstrates the possibility of detecting the bi-directional dependency between the api and core module, which would have been very hard to spot in the entire system view of Zeeguu (Figure 9). Upon discovering such an issue, developers need to investigate further to locate the precise source of the problem.

These examples showcase the versatility of Architectural Lens in detecting architectural mistakes. Users can create a wide range of custom views to suit their specific needs, such as inspecting a three-layered architecture by aggregating all modules except the top-level ones to examine the adherence to the layered structure.

Summary

Throughout this section, we have demonstrated how the Architectural Lens tool can be utilized to generate various architectural views of a software system. By defining multiple views within a single JSON configuration file, developers can create diagrams that cater to their specific needs, enhancing the understanding and communication of the system's architecture. The integration with GitHub Actions allows for automatically generating difference views during the code review process, making the assessment of architectural changes more efficient.

8 Workshop

As delineated in the methods section, we conducted a workshop with Python-experienced developers to evaluate the Architectural Lens tool and our proposed continuous process for generating customizable architectural diagrams. This workshop provided hands-on experience with the tool and process and facilitated the collection of valuable insights for its effectiveness and potential improvements and future research.

8.1 Questionnaire

To gather feedback from participants regarding their experience with Architectural Lens and the workshop, we have developed a questionnaire specifically designed to capture their opinions and insights on using Architectural Lens. It is important to note that the wording of the questionnaire primarily focuses on the process of working with the tool, as described in the Method section of the report. By following this process, participants use Architectural Lens in a uniform manner, ensuring that their experiences with the tool are aligned and enabling meaningful comparisons across participants.

The questionnaire includes a combination of open-ended and qualitative questions, allowing participants to provide detailed and measurable feedback. The questions cover various aspects, including their perceptions of the tool's effectiveness, usability, and impact on their architectural documentation practices.

1. **Background and Experience**: Participants were asked to briefly introduce themselves and share their background in software development (Q1), as well as describe the system they are currently working on (Q2).

2. **Integration and Workflow**: Participants were asked about their vision for integrating Architectural Lens into their day-to-day work (Q3) and how the process supported their workflow in maintaining up-to-date architectural documentation (Q7).

3. **Comprehension and Scoping Views**: Questions focused on the impact of scoping views on the overall comprehension of system architecture (Q4) and the improvement of comprehensibility in architectural documentation (Q8).

4. **Continuous Process and Documentation**: Participants were asked about the effects of incorporating diagram generation as part of a continuous process on their system's architectural documentation (Q5).

5. **Showcasing Differences**: The questionnaire addressed participants' thoughts on showcasing differences between the "main" branch and their feature branch when creating a pull request using the tool (Q6).

6. **Improvements and Suggestions**: Participants were encouraged to provide suggestions to improve Architectural Lens. (Q9).

7. **Future Work**: The questionnaire asked about additional features or future research directions participants would like to see explored in Architectural Lens (Q10).

The completed questionnaires were collected at the end of the one-week testing period. The feedback collected from the questionnaire will help evaluate the effectiveness of the Architectural Lens tool and identify areas for improvement or future research.

Here are the questions included in the questionnaire:

1. Could you briefly introduce yourself and share your background in software development?
2. Please describe the system that you are currently working on
3. How do you envision integrating the proposed process into your day-to-day work?
4. Considering our proposed process, how do you think the ability to scope views might impact your overall comprehension of your system architecture?
5. In your opinion, what effect could our proposed process of incorporating diagram generation as part of a continuous process have on your system's architectural documentation?
6. What are your thoughts on showcasing differences between the "main" branch and your feature branch when creating a pull request using our proposed process?
7. How well did our proposed process support your workflow and assist you in maintaining up-to-date architectural documentation?
8. How well did our proposed process support your workflow and assist you in improving the comprehensibility of your architectural documentation?
9. Do you have any suggestions to improve the tool to make the diagrams even more comprehensible or valuable?
10. What additional features or future research directions would you like to see explored in Architectural Lens?

9 Analysis

Following the one-week trial period, we conducted a thematic analysis as outlined in the method sections 4.5 to analyze the results. Below presents the findings from the analysis of the questionnaire.

9.1 Improved understanding and spot architectural errors

All five developers who participated in the questionnaire reported that it aided them in gaining a better understanding of the system they were working on. However, this improved understanding was achieved for different reasons. One developer attributed this to the ability to focus on critical parts of the system by scoping the views, stating that *"being able to easily divide your project into subsystems makes it easier to explain."* Another developer found the colour coding in the difference view to be helpful in comprehending the system, stating that *"the color highlighting made it very easy to see what changes occurred in the system before allowing it to be part of your code-repository."*

During the questionnaire, developers mentioned that the enhanced system comprehension aided in identifying design flaws. Certain developers believed that scoping views to lower levels were the reason, while others attributed it to the integration of GitHub action. One developer stated, *"In addition, as the documentation was generated frequently, it assisted me in spotting architectural errors before committing them to the codebase, which in the long run may help my system quality improve, as it is less complex and has fewer architectural errors."*

One of the developers pointed out that for the tool to achieve its full potential, it is essential that all team members embrace it. This developer noted that "Generating customized views tailored to specific aspects of the system, like showing only the components related to "data storage" or "user management", has made it easier to understand the architecture. *"But again, I think it's important to note that for this process to really take effect, it is important that the entire team at least adopts the idea, much like we have with unit testing".*

9.2 Onboarding

Four out of the five developers who participated in the questionnaire mentioned onboarding as a potential benefit of using the tool. However, this is not necessarily because of the tool itself but due to the general benefits of having precise documentation. However, the automatic generation of the documentation of our tool ensures that new developers always have access to the most up-to-date diagrams, and the scoped views make it easier for them to comprehend the system, as discussed in the previous section.

As one developer pointed out, *"Additionally, when it comes to overall team efficiency and onboarding, I think that visual learning is an incredibly powerful way to quickly understand and get an overview of a new project. Having pre-generated and precise views of our system for onboarding of new employees could help with reducing man hours used on onboarding both for the new employee and the senior developers."* This highlights the potential for our tool to not only benefit current developers but also streamline the onboarding process for new team members.

9.3 Architectural alignment

Three out of the five developers who participated in the questionnaire identified potential benefits in terms of aligning the architectural design within the team. They saw this as a possibility through the continuous

process of pull requests and architecture review in parallel with traditional code review. One developer specifically mentioned that the tool could be used *"as a collaborative tool to make sure other team members align with my design and that my changes don't introduce undesired coupling."* This could potentially lead to a more coherent and consistent system design across the team, reducing the risk of errors and inconsistencies in the codebase.

9.4 Usability

One aspect that could quickly be overlooked was the resources that it takes to set up the tool to part of a systems continuous integration. In this regard we have made tried to make this as simple as possible, so that a developer only need to run two commands to set up a view and add the tool to github PR review. This easy of use is also something some of the developers have remarked upon. One of the developers remarked that *"Because it was so easy to integrate, I was up and running almost immediately and started generating architectural views based on live changes."* another developer also mentioned that it was easy for him to add the tool to his existing workflow.

9.5 Up-to-datenss & scoped views

This report aimed to solve two significant issues: outdated architectural diagrams and diagrams that became too complex to be comprehensible for developers. Feedback from the developers who participated in our study showed that these issues were indeed a concern. One developer mentioned that *"Incorporating the diagram generation into a continuous process has helped ensure that our architectural documentation stays up-to-date and accurately reflects the current state of the system."* Similarly, four out of five participants provided similar responses regarding up-to-dateness. All five developers noted the benefits of using scoped views to improve their understanding of the system. One participant explained that segmenting the diagram into views helped them focus on the relevant parts of the system, rather than getting lost in an overwhelming, all-encompassing diagram. The ability to create custom, scoped views was cited as a key factor in enhancing comprehension.

9.6 Human error

In addition to keeping the documentation up to date and improving the team's understanding of the system, the tool can also help avoid human errors in the documentation. As one of the developers pointed out, *"With it happening automatic, this ensures you have an up to date diagram, and developers don't have to question whether diagram is incorrect either from human errors or just not recently updated."*

Keeping diagrams updated manually can be a tedious and error-prone task, as developers may forget to update them after making changes or make mistakes in the process. By automating the process of generating diagrams, the tool can help ensure that the documentation is always up to date and accurate, without requiring additional effort from the developers.

9.7 Improvements

The developers also came with lots of suggestions as to what could improve the tool.

1. Export the internal dependency graph as some ingestible format (json, txt) so it can be used for further code analysis made by the user itself.

2. Make the views interactive by making extensions to popular IDE and code editors such as vscode, and allow users to navigate the systems architecture by clicking on the different modules, and automatically save the discovered views to the JSON file.

3. Identifying common design mistakes and emphasizing them in the diagrams.

4. Currently, the tool can only analysis within a single system. However, as more and more systems rely on patterns such as microservices, it would be beneficial to be able to analyse across codebases to show dependencies between them.

5. Create another GitHub action that automatically adds the newest diagrams to the repo on every new commit.

10 Discussion

In this section, we compare the insights gathered from our workshop data analysis to the established literature in the related works and background sections. We aim to explore the implications of our findings. We will examine how the tool addresses the challenges of creating and maintaining comprehensible architectural documentation. Through these discussions, we aim to provide a comprehensive understanding of the significance and implications of our research in the context of software architecture documentation.

10.1 Comprehensible and up-to-date diagrams

Participants in our study, as elaborated in analysis section 9.1, reported an improved understanding of the systems they were working on, attributed to features such as scoped views and colour-coded difference views. These features not only enabled developers to focus on the relevant parts of the system but also facilitated swift identification of changes. As a result, developers were better equipped to identify and rectify design flaws, potentially improving system quality and reducing complexity.

Reflecting on this, it becomes apparent that the power of Architectural Lens lies not just in its ability to generate diagrams, but in its capacity to make those diagrams highly relevant and actionable to developers. The tool's provision for scoped and colour-coded difference views can potentially transform how developers engage with system architectures, potentially leading to more thoughtful design decisions and ultimately, more robust and maintainable systems.

Incorporating the insights from analysis section 9.6, we acknowledge that keeping diagrams updated manually can be a tedious and error-prone task, as developers may forget to update them after making changes or make mistakes in the process. As highlighted by a developer in Section 9.6, the automation offered by Architectural Lens assures the continuous accuracy of the diagrams requires minimal effort from developers. This automation eliminates doubts about potential human errors or outdatedness.

Reflecting on this, it becomes apparent how important automation is in maintaining reliable architectural documentation. By eliminating the need for manual updates, we not only save developers' time but also greatly reduce the chances of inconsistency between the codebase and its corresponding diagrams. This could lead to increased productivity and job satisfaction as developers are able to engage more deeply in problem-solving and creative tasks. In addition, accurate and up-to-date diagrams can enhance the understanding of the system architecture among team members, potentially improving collaboration and the overall quality of the software.

However, while automation eliminates the need for manual updates, it's important to note that the quality of the generated diagrams is contingent upon the correct implementation and configuration of the tool. Any issues in these areas could potentially lead to inaccurate diagrams, which could mislead developers and negatively impact their work. Therefore, proper setup and maintenance of the tool are critical for ensuring the accuracy of the diagrams and reaping the full benefits of automation.

10.2 Scoped views

Developers participating in the workshop, as discussed in analysis section 9.1, found scoped views valuable for understanding and explaining the system. By focusing on specific parts of the architecture, developers could eliminate irrelevant details and concentrate on critical aspects, facilitating better comprehension and more efficient design processes.

Furthermore, analysis section 9.3 highlighted potential benefits of scoped views in aligning architectural design within teams. Through continuous pull requests and architecture review alongside code review, scoped views can serve as a collaborative tool for ensuring design alignment and avoiding undesired coupling. This can result in a more coherent and consistent system design, reducing errors and inconsistencies in the codebase. By fostering better architectural alignment within the team, scoped views can contribute to improved system quality and maintainability. Developers can identify and correct design flaws more easily, leading to a more robust and adaptable system.

Lastly, scoped views align closely with the principles of agile and trunk-based development practices, where collaboration is valued over extensive documentation, scoped views offer a solution by creating clear and concise documentation for each part of the system being developed.

10.3 Software Life Cycle

The use of Architectural Lens has the potential to positively impact various stages of the software life cycle, as discussed in the background section 2.2. Our analysis, detailed in sections 9.2 and 9.3, highlighted that developers appreciate the precise and up-to-date diagrams generated by Architectural Lens. By providing accurate views, the tool aids in several phases of the software development process.

During the planning and analysis phase, Architectural Lens, offers a mechanism for maintaining alignment with the original system design. By facilitating the generation of accurate, focused architectural views, it enables continuous comparison of the evolving system with the initial blueprint. Continual comparison with the original design is beneficial as it helps to manage architectural drift, ensuring that design decisions made early in the development process continue to be relevant and effective. This iterative validation ensures the final product accurately reflects the envisioned architecture.

During the development phases, the tool's capability to create scoped views can enhance productivity by enabling developers to concentrate on specific aspects of the architecture. This approach enables a more efficient design and coding process as developers gain a better understanding of the system. Furthermore, automated diagram generation ensures the documentation remains current and accurate, minimizing the risk of misunderstandings or errors that could stem from outdated or inaccurate diagrams.

The benefits of Architectural Lens extend into the maintenance phase as well. As the tool keeps the diagrams in sync with the codebase, developers have access to up-to-date documentation that can aid in understanding the system during maintenance, making it easier to identify and fix issues. [11]

In the post-development phase, Architectural Lens could also play a significant role. Precise and current diagrams could aid in system analysis, evaluation, and potential system refactoring.

10.4 Onboarding

While onboarding wasn't explicitly identified as a challenge in the background and related works sections, developers participating in our study acknowledged the value of Architectural Lens in facilitating a smoother onboarding experience, as detailed in the analysis section 9.2.

Our analysis revealed that developers found it easier to grasp and explain the system's architecture when utilizing our tool. A particularly salient point noted by a developer, highlighted in section 9.2, was that the availability of pre-generated, precise views of the system could significantly reduce the man-hours spent on onboarding - a benefit for both the new employee and the senior developers tasked with their training.

Reflecting on these insights, the use of Architectural Lens could transform the onboarding process by

providing a comprehensive, up-to-date, and easily comprehensible view of the system's architecture. This could fast-track the new developer's understanding of the system, allowing them to contribute more quickly and meaningfully to the project. However, it's important to recognize that the tool's effectiveness in this regard is tied to its correct setup and maintenance.

The insights gathered from developers, coupled with the previously discussed implications, highlight the potential of Architectural Lens to enhance the onboarding experience, overall team productivity, and comprehension of the system.

10.5 Enhancing Quality Attributes: Impact of Architectural Lens

Developers mentioned in the analysis section 9.1 that the enhanced system comprehension aids in identifying design flaws, leading to a more error-free and maintainable system architecture, which in the long run can help improve both the quality and maintainability of the system.

Based on the insights gathered from the developers we interviewed, there is an indication that Architectural Lens contributes to improvement in the system's quality attributes, specifically maintainability. However, further studies and evaluations involving a more extensive and more diverse sample of developers would be valuable in providing stronger evidence and insights regarding the specific contributions of Architectural Lens to system quality attributes.

10.6 Usability and ease of use

An important aspect of any tool is its usability and ease of use, which can influence the likelihood of its adoption by development teams. The analysis section 9.4 highlights that developers appreciate the ease of setup and minimal learning curve associated with Architectural Lens.

By making it easy for developers to try and use our tool, we increase the chances of its adoption, which may lead to better software documentation and improved system quality. These benefits are supported by the developers' experiences and opinions shared in the analysis 9.4, emphasizing the importance of usability and ease of use in the successful implementation of Architectural Lens within the software development process.

10.7 Understanding Comprehensible Diagrams

Initially, our understanding of 'comprehensibility' in the context of software diagrams was quite basic: a developer should be able to understand the information presented in the diagram. The initial design of Architectural Lens focused on this notion, incorporating the concept of scoped views to improve the clarity of the diagrams. As we further developed the tool, we recognized the need to distinguish between different changes in the architecture visually. Hence, we introduced colour highlighting for branch differences, where red indicated deletions, and green represented new additions, further enhancing the comprehensibility of the diagrams.

In addition to these highlighted views, the suggestions and feedback from the developers during the workshop suggest that comprehensibility extends beyond our initial understanding. It is not merely about visual clarity or the ability to grasp the presented information. Instead, comprehensibility includes the relevance and usefulness of the information presented. Features such as detecting potentially harmful architectural patterns, integrating the tool directly into the Integrated Development Environment (IDE), and creating an interactive filtering or selection view for generating the configuration file for the diagrams were suggested by

the developers. These suggestions indicate a more nuanced and complex understanding of comprehensibility, which encompasses not just clarity, but also relevance, usefulness, and ease of access within developers' workflow.

Therefore, while Architectural Lens has made strides in enhancing the comprehensibility of software architecture diagrams through scoped views and colour-coding, our understanding of what constitutes 'comprehensibility' has evolved. The future work in this area should further explore and expand on this understanding of comprehensibility in the context of software architecture diagrams.

Summary

In this discussion, we have delved into how Architectural Lens tackle the challenges identified in the background and related work sections. We have examined the advantages of producing up-to-date and comprehensible diagrams, scoped views, and the tool's ease of use, as well as their impact on onboarding and the software life cycle. Additionally, we have explored improvements in the quality attribute maintainability, and the significance of usability in the tool's adoption.

11 Limitations & Reliability

In this section, we will examine the limitations and reliability of our study, taking into account the methodology employed, the sample size and diversity, and the tool's applicability to other programming languages. By addressing these concerns, we aim to provide a transparent and comprehensive evaluation of our research, acknowledging the potential shortcomings and biases that may have influenced our findings. Furthermore, this analysis will help to contextualize the results and inform future research efforts.

11.1 Sample Size and Diversity

In this research, we exclusively interviewed software developers. The insights of other stakeholders, such as architects, testers, or product owners, might offer additional perspectives on the tool's efficacy and potential enhancements. Subsequent research could involve a more extensive set of stakeholders to obtain a more comprehensive understanding of Architectural Lens impact on the software development process.

Our study encompassed a limited sample size of 5 developers, all originating from Northern Europe and within the age range of 25-50. Although we conducted a thematic analysis and attained saturation, with all responses demonstrating similarities and consistent themes, it is important to acknowledge that larger or more diverse samples might produce different outcomes. To enhance the generalizability of the findings, future research must prioritize including a significantly larger number of participants drawn from a wide range of geographical locations, age groups, and professional backgrounds.

11.2 Methodological Limitations

Our research methodology was solidly grounded in an extensive review of background and related works, which offered valuable insights into the problem at hand. However, in retrospect, there are aspects that could have been approached differently for potentially richer outcomes:

- In the initial phase of the tool's development, engaging more directly with developers could have provided an additional layer of insight into potential solutions. Although our methodology didn't include this step, it could have facilitated a better understanding of their concerns and expectations during the development of Architectural Lens.
- A limitation of our study is that we provided instructions on how to apply Architectural Lens using an iterative process. While this approach allowed us to examine the tool's effectiveness within a controlled setting, it is important to acknowledge that the provided guidance may have influenced participants' usage patterns and potentially restricted their exploration of alternative approaches. By offering specific instructions, we may have inadvertently limited the creative possibilities and diverse ways in which developers could fully leverage the potential of Architectural Lens.
- While we conducted the workshops ourselves to ensure consistency, it is important to acknowledge that this may have introduced some degree of bias in the presentation and explanation of the tool.

These considerations are important to acknowledge as they highlight potential areas where our methodology could have been improved. The lessons learned from these limitations can guide the design and execution of future research in this area.

11.3 Tool Limitations

While our objective was to address the problem generically, our tool focused on Python. As a result, this report solely collects data from Python developers. Developers utilizing other programming languages may offer different insights and opinions regarding the tool's effectiveness. Due to time constraints, we were unable to involve more programming languages in our study. This limitation should be considered when evaluating the extent to which our approach generically addresses the identified problem across various languages and developer communities.

11.4 Diagram Limitations

Currently, our tool generates module diagrams exclusively. It is plausible that combining module and class diagrams or even incorporating a domain-level view could yield clearer and more comprehensive visualizations of the system. This limitation should be considered when evaluating the effectiveness of our approach, as additional diagram types may further enhance the overall understanding of the system's architecture.

12 Conclusion

To investigate the research question, we developed an automated tool implemented in Python. The objective of the tool was to facilitate the creation and maintenance of customizable architectural diagrams for complex software systems, ensuring their comprehensibility and up-to-dateness. Subsequently, we conducted a questionnaire-based survey involving developers to evaluate the tool's effectiveness, gather valuable insights, and analyze the survey data using thematic analysis.

To enhance the comprehensibility of diagrams, our tool provides users with the flexibility to create custom views based on their specific needs. This includes the ability to define views that aggregate multiple layers of the system or focus on individual modules. Users can also filter out modules that are not relevant to the current view, further refining the visual representation of the architecture. Lastly, the tool offers the capability of creating "difference views" which enable visual comparisons between the current state of the codebase and a specified version. This feature allows developers to easily identify and visualize changes in the diagrams, providing valuable insights into the evolving architectural structure of the software system.

To address the challenge of maintaining up-to-date architectural documentation, Architectural Lens utilizes static analysis to analyze the code and generate diagrams directly from the source code. This approach enables developers to effortlessly and efficiently obtain the most recent and accurate version of their documentation by running Architectural Lens as needed. Additionally, Architectural Lens offer seamless integration with a GitHub action, allowing developers to visualize their architecture and changes through difference views during pull request reviews.

From the thematic analysis, it is evident that developers recognize the value of scoping different views, difference views and automating the generation of these views once they are defined. By ensuring that architectural documentation remains up-to-date and comprehensible, our participants also identified several other benefits, besides comprehensibility and up-to-dateness including:

- Supporting onboarding of new team members and facilitating better communication within the development team.
- Enhancing software quality attributes, such as maintainability.
- Easier for developers to spot architectural errors.
- Removes the element of human error.

In light of our findings, Architectural Lens has demonstrated its potential to address the challenges identified in the literature and the background section. By offering an automated, customizable, and user-friendly solution for creating and maintaining architectural diagrams, the tool empowers developers to manage the complexity of architectural documentation leading to better system understanding. This increased understanding and control can lead to better decision-making during the development process.

The flexibility of our approach, exemplified by the tool's ability to adapt to other programming language, indicates a potential for further investigation, refinement, and broader application in the field of software architecture documentation.

Overall, the findings of this report demonstrate the effectiveness of Architectural Lens in addressing the challenges of creating comprehensible diagrams and maintaining architectural documentation for complex software systems. By utilizing such a tool, developers can benefit from improved system understanding, quality, and efficiency in software development.

13 Future Work

Based on the findings of our study and the feedback from developers, we have identified several potential avenues for further research and improvement of Architectural Lens. These future work ideas could build upon the foundation laid in this report and further enhance the tool.

From the analysis section G, we gathered several suggestions for improvements provided by the developers, including:

1. Exporting the internal dependency graph in an ingestible format (e.g., JSON, TXT) for further code analysis by the user, potentially enabling more in-depth system understanding and analysis.

2. Making the views interactive by integrating with popular IDEs and code editors, such as Visual Studio Code, allows users to navigate the system's architecture by clicking on different modules and fostering an even more seamless development experience. Furthermore, upon finding the desired view, it can be automatically inserted into the Architectural Lens's configuration. This ensures that the chosen view is generated each time the tool is run, providing a consistently up-to-date and relevant perspective on the system's architecture.

3. Suggesting potential improvements for common design mistakes, which could help developers optimize their system's architecture and reduce the likelihood of issues arising in the future.

4. Expanding the tool's capabilities to analyze dependencies across multiple codebases, particularly relevant for systems that rely on microservices architecture, and enhancing the tool's applicability and usefulness in modern software development.

5. Developing a GitHub action that automatically adds the latest diagrams to the repository on every new commit, streamlining the process of keeping architectural documentation up-to-date.

While some of these improvements, such as creating a new GitHub action, may be relatively straightforward to implement, others, like suggesting improvements for common design mistakes, may require more extensive work but could greatly enhance the tool's value to developers.

In addition to these suggested improvements, we also identified potential benefits that could be investigated in future research:

- Best practice for applying Architectural Lens: This study provided instructions on how to use the tool, further investigation is warranted to observe how developers naturally would incorporate Architectural Lens into their software development workflow without guidance. By observing their usage patterns and strategies, we can gain insights into alternative approaches and uncover effective ways to leverage the tool.

- Reduced architectural mistakes: In order to investigate the impact of Architectural Lens on reducing architectural mistakes, it is important to gather quantifiable data. While our study currently provides qualitative data based on feedback from 5 developers, obtaining quantitative data would provide a more comprehensive understanding of the tool's effectiveness. This investigation could involve examining the frequency and types of architectural mistakes identified during the review process, and comparing projects with and without the integration of Architectural Lens.

- Reduced technical debt: Although challenging to evaluate within the scope of our study, we hypothesize that Architectural Lens could help minimize the accumulation of technical debt over time by encouraging continuous validation and refinement of system architecture. Exploring this hypothesis could involve longitudinal studies that track the technical debt of projects using Architectural Lens, comparing it with similar projects not employing the tool. Quantitative metrics, such as code complexity, cohesion, and coupling, could be used alongside qualitative assessments of code quality and maintainability to measure potential reductions in technical debt.
- Diverse Developer Feedback for Tool Improvement: By adapting the tool to accommodate more programming languages, we could gather a broader range of feedback from a more diverse group of developers. This could provide invaluable insights into potential enhancements for both the tool. Developers from different languages could bring unique perspectives and challenges, enriching our understanding of how the tool can be optimized to better address issues in software architecture documentation.
- Conduct research to better understand the factors that contribute to the comprehensibility of software diagrams, considering not only visual representation but also context, relevance, and usability of the information presented.
- Explore the potential of machine learning and artificial intelligence techniques in automatically detecting adverse architectural patterns and generating actionable insights for developers.

By exploring these possibilities, we can better understand how Architectural Lens can be adapted and enhanced to meet the evolving needs of the software development community.

Bibliography

[1] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International conference on software architecture (ICSA)*. IEEE, 2018, pp. 176–17 609.

[2] D. Rost, M. Naab, C. Lima, and C. von Flach Garcia Chavez, "Software architecture documentation for developers: A survey," in *Software Architecture: 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings 7*. Springer, 2013, pp. 72–88.

[3] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 590–601.

[4] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE software*, vol. 20, no. 6, pp. 35–39, 2003.

[5] B. Selic, *Using UML for Modeling Complex Real-Time Systems*, ser. LNCS. Springer-Verlag, 1998, vol. 1474, pp. 250–260.

[6] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of uml in software maintenance," *IEEE Transactions on software engineering*, vol. 34, no. 3, pp. 407–432, 2008.

[7] A. Abdurazik and J. Offutt, "Using uml collaboration diagrams for static checking and test generation," in *UML 2000—The Unified Modeling Language: Advancing the Standard Third International Conference York, UK, October 2–6, 2000 Proceedings*. Springer, 2001, pp. 383–395.

[8] AWS. [Online]. Available: https://aws.amazon.com/what-is/sdlc/

[9] W. Hasselbring, "Software architecture: Past, present, future," *The Essence of Software Engineering*, pp. 169–184, 2018.

[10] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

[11] E. Tryggeseth, "Report from an experiment: Impact of documentation on maintenance," *Empirical Software Engineering*, vol. 2, no. 2, pp. 201–207, 1997.

[12] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *Journal of Systems and Software*, vol. 82, no. 6, pp. 981–992, 2009.

[13] N. Alves, T. Mendes, M. de Mendonca, R. Spinola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001743

[14] V. Saxena, S. Kumar *et al.*, "Impact of coupling and cohesion in object-oriented technology," *Journal of Software Engineering and Applications*, vol. 5, no. 9, pp. 671–676, 2012.

[15] Microsoft, "Make the application loosely coupled," Nov 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/contact-manager/iteration-4-make-the-application-loosely-coupled-cs

[16] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: views and beyond," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 740–741.

[17] A. Begel and N. Nagappan, "Usage and perceptions of agile software development in an industrial context: An exploratory study," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007).* IEEE, 2007, pp. 255–264.

[18] e. a. Beck, K., "Manifesto for agile software development," Website, February 2001, accessed: May 21, 2023. [Online]. Available: http://agilemanifesto.org/

[19] I. Sommerville, "Software engineering 10th edition - web chapter: Documentation," 2010, accessed: 22. May 2023.

[20] "Drawio," Website, accessed: May 21, 2023. [Online]. Available: https://www.drawio.com/

[21] S. Brown, "Diagrams as code," October 2020. [Online]. Available: https://dev.to/simonbrown/diagrams-as-code-20eo

[22] "Structurizr dsl," Website, accessed: May 21, 2023. [Online]. Available: https://github.com/structurizr/dsl

[23] "Diagram as code," Website, accessed: May 21, 2023. [Online]. Available: https://github.com/mingrammer/diagrams

[24] "Plantuml in a nutshell," Website, accessed: May 22, 2023. [Online]. Available: https://plantuml.com/

[25] "Mermaid," Website, accessed: May 21, 2023. [Online]. Available: https://mermaid.js.org/

[26] "Graphviz," Website, accessed: May 21, 2023. [Online]. Available: https://graphviz.org/

[27] "Swimm," Website, accessed: May 21, 2023. [Online]. Available: https://swimm.io/integrations/

[28] "Pyreverse," Website, accessed: May 21, 2023. [Online]. Available: https://pylint.readthedocs.io/en/latest/pyreverse.html

[29] "Doxygen," Website, accessed: May 27, 2023. [Online]. Available: https://www.doxygen.nl/

[30] "Umbrello," Website, accessed: May 27, 2023. [Online]. Available: https://apps.kde.org/da/umbrello/

[31] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 187–197.

[32] V. Braun and V. Clarke, *Thematic analysis.* American Psychological Association, 2012.

Appendix

Below are the responses from the completed questionnaires. The *italicized* text indicates the labels assigned to the answers during the thematic analysis.

Questionaire from Participant 1

Question 1: Could you briefly introduce yourself and share your background in software development?

I have been working as a software engineer for over 15 years now. Most of these years i spent as a consultant in various roles, being a full stack cloud engineer, backend developer, software coach and architect.

Question 2: Please briefly describe the system that you are currently working on

Currently i work on a system for a large home improvement chain in the Netherlands. Specifically the customer loyalty program. One of the most important things we enable is the loyalty program: a customer can save points when they buy something, and these points can be spent later on for example a discount. Other things include the "my account" stuff: letting customers manage their personal information, receipts, online orders, their paints and so on.

Question 3: How do you envision integrating the proposed process into your day-to-day work?

The company i work for has several teams working on various systems, and to encourage cooporation across all teams there are several guilds responsible for different things. One of the guilds i am part of is the architecture guild, where we aim to have a cohesive architecture and guidelines for all teams to follow. I can see views generated by applying the process and using Architecture Lens being a great aid in gaining insight in the various systems during these architecture guild meetings, while also providing a base for discussion *(alignment with design, improve understanding of system).*

Question 4: Considering our proposed process, how do you think the ability to scope views might impact your overall comprehension of your system architecture?

Some parts of the system are more critical then others, so i can definitely see that creating scoped, more narrow views of areas of the system that are more business critical then others will be very valueable. *(focus in critical parts of system, scoped views)* It is very important to me to reduce my cognitive load, so i can focus on the stuff that really matters to me.

Question 5: In your opinion, what effect could our proposed process of incorporating diagram generation as part of a continuous process have on your system's architectural documentation?

Currently, the architectural documentation is being maintained by hand by someone. All the stereotypical problems are there: this is a person that does not work on each of these systems, so he needs to get the information 3rd hand and trust that it's accurate. *(Lack of human error)* The documentation does not get updated all that often, which further reduces accuracy. Also, it is hard to find: it's hidden on a sharepoint drive somewhere, and you can never be sure you have the correct version. Finally, it is a single image with

all components on it, which makes it impossible to parse at first glance.

I am a big proponent of generating documentation and publishing it as part of the deliverables of a project. Just like you would package up the source code in maybe a deployable docker image, generating and publishing the documentation that goes with that deployable would make a lot of sense to me. This would mitigate a lot of the issues described above. Therefore, Applying to the proposed process and continiously generating views of the system using Architecture Lens which can be stored on github can have a huge impact.

Question 6: What are your thoughts on showcasing differences between the "main" branch and your feature branch when creating a pull request using our proposed process?

Our pull requests are mostly reviewed in the diff view, to see what code got removed and what code got added. Just by looking at code changes it can be challenging to really understand the consequences these changes might have. So, just like how we rely on a CI pipeline to verify the changes did not break the build, having something as part of the pull request that shows the impact on the architecture will help in providing a better code review.*(alignment with design)*

Question 7: How well did our proposed process support your workflow and assist you in maintaining up-to-date architectural documentation?

Very well. Having several diagrams generated as part of the build pipeline feels right at home with generating and publishing the other artifacts. We can even keep track what version of the architecture is in production, and which is on staging to really have a clear picture of what is going on.*(improve understanding of system)*

Question 8: How well did our proposed process support your workflow and assist you in improving the comprehensibility of your architectural documentation?

It does take some experimentation find scoped views that are applicable for the system as a whole, and creating scoped views that really zoom in on the part that is under development. But, dialing in the diagrams also helped me understand what parts i cared and what parts were less relevant, which in turn also increased my understanding of the system as a whole. *(improve understanding of system)*

Question 9: Do you have any suggestions to improve the tool to make the diagrams even more comprehensible or valuable?

Expand the process to also be able to include connected systems. In todays microservice world, it would be awesome to generate documentation that shows how the microservices are connected to one another. Including REST calls, topics, queues, databases, etc.*(improvements)*

Question 10: What additional features or future research directions would you like to see explored in Architectural Lens?

- Make a VSCode and/or PyCharm plugin that can show the diagrams directly in the IDE.
- Make the diagrams interactive, allowing users to click through and zoom the diagrams. Additionally, provide the capability to save scoped views in the configuration.

- Have the tool analyze the architecture and give suggestions on how to improve based on best practices.*(improvements)*

Questionaire from Participant 2

Question 1: Could you briefly introduce yourself and share your background in software development?

I'm a senior developer with over 5 years of experience in software development, mostly working on Python applications. I've been involved in various projects, mainly involving web development and data analysis.

Question 2: Please briefly describe the system that you are currently working on

At the moment, I'm working on a web-based data visualization platform, allowing users to explore and analyze large datasets through interactive visualizations, like generating heatmaps or creating custom graphs to compare data points.

Question 3: How do you envision integrating the proposed process into your day-to-day work?

I attended a workshop on Architectural Lens, and it was pretty straightforward to set up and start using in my own project.*(easy to setup)* Over the past week, I've been integrating it into my daily work, and it's been helpful in automating the generation of architectural diagrams which i and my colleagues understand and keeping them up-to-date.

Question 4: Considering our proposed process, how do you think the ability to scope views might impact your overall comprehension of your system architecture?

Although i already knew the system quite well, after using the ability to scope views in the tool while developing as the process states, my understanding of the system architecture has improved.*(scoped views)* It allowed me to focus on specific parts of the system that are most relevant to my work, like the components responsible for data processing or user authentication.*(focus in critical parts of system)*

Question 5: In your opinion, what effect could our proposed process of incorporating diagram generation as part of a continuous process have on your system's architectural documentation?

Incorporating the diagram generation into a continuous process has helped ensure that our architectural documentation stays up-to-date and accurately reflects the current state of the system. *(up-to-dateness)* This has made it easier to troubleshoot issues or onboard new team members.*(onboarding)*

Question 6: What are your thoughts on showcasing differences between the "main" branch and your feature branch when creating a pull request using our proposed process?

After trying the feature that showcases differences between the "main" branch and my feature branch when creating a pull request, I found it useful in identifying potential issues or inconsistencies in the architecture, For instance, I was able to spot a new feature inadvertently introducing a circular dependency, which I addressed before merging the changes.*(spot errors with diff view)* Additionally it was useful just to understand what changes in your system before allowing it to be part of your code-repository, the color highlighting made that very easy to see. *(improve understanding of system)*

Question 7: How well did our proposed process support your workflow and assist you in maintaining up-to-date architectural documentation?

The process and tool has streamlined my workflow and helped me maintain up-to-date architectural documentation. *(up-to-dateness)* I no longer have to spend as much time manually updating diagrams, allowing me to focus on development tasks. The documentation has however not yet been accepted by my bosses yet, so it is only me using it right now, I think if it is to be really useful, my colleagues need to do something similar.

Question 8: How well did our proposed process support your workflow and assist you in improving the comprehensibility of your architectural documentation?

Generating customized views tailored to specific aspects of the system, like showing only the components related to "data storage" or "user management", has made it easier to understand the architecture. *(scoped views)* But again, I think its important to note, that for this process to really take effect, it is important that the entire team atleast adopts the idea, much like we have with unit testing, which I was informed that the process was inspired by. *(team adoption)*

Question 9: Do you have any suggestions to improve the tool to make the diagrams even more comprehensible or valuable?

One suggestion for improvement might be to add more customization options for the diagrams, such as different visual styles or layout options, to further enhance their readability and appeal. *(improvements)* For example, I'd like to have the option to color-code components based on their purpose or layer in the architecture.

Question 10: What additional features or future research directions would you like to see explored in Architectural Lens?

I'd be interested to see Architectural Lens and the proposed process extended to support other programming languages and development environments. It would also be useful to explore ways to integrate the tool more seamlessly with existing development tools and platforms, like incorporating it into popular IDEs or version control systems, but it is nice that it is currently available through pip. *(improvements)*

Questionaire from Participant 3

Question 1: Could you briefly introduce yourself and share your background in software development?

Master's degree in science and engineering, 4 years of experience working as a software developer on Python, Scala, Java projects in banking and game development.

Question 2: Please briefly describe the system that you are currently working on

It is dockerized Python system designed to transfers files over SFTP as either client or server. The users of the system are external banking partners that all have different demands on file formats and encryption standards making configurability and modularity a big focus of the system.

Question 3: How do you envision integrating the proposed process into your day-to-day work?

I see two main modes of operation:

1. As a local tool for prototyping sweeping changes to the system (large refactorings or new modular functionality).*(large refactor)* The tool would be run often on my local machine, as with unit tests.
2. As a collaborative tool to ensure that other team members align with my design and that my changes don't introduce undesired coupling.*(alignment with design)* This would take place in a space such as a pull request.

Question 4: Considering our proposed process, how do you think the ability to scope views might impact your overall comprehension of your system architecture?

I think scoped views is what makes the tool usable for a system of moderate to high complexity.*(scoped views)* Being able to pick apart the system and focus on one specific area at a time helps when onboarding people*(onboarding)* (as that process is naturally gradual) and also when discussing how specific parts of the system are implemented. If you could not scope the views, and only had a top-level view, it would not be as helpful since details would easily be missed.

Question 5: In your opinion, what effect could our proposed process of incorporating diagram generation as part of a continuous process have on your system's architectural documentation?

Documentation tends to have a higher quality when it is often iterated upon. Generating an architectural view of a system with every change vastly increases the odds of the documentation being correct and precise*(up-to-dateness)*. I think it would also lower the barrier for people to start engaging in maintaining proper documentation, which further spreads knowledge of a system through a team of developers and designers.*(easy to setup)*

Question 6: What are your thoughts on showcasing differences between the "main" branch and your feature branch when creating a pull request using our proposed process?

This is the most exciting feature for me as it is, due to the following reasons:

1. Checked in and audited in version control.
2. Automated and integrated into build pipelines.
3. Automatically reflecting the changes of the raised pull request (PR), providing a mutual understanding and solid basis for discussion between reviewers.*(alignment with design)*
4. Automatically updated upon further code changes based on the evolution of the PR, providing a solid trail of changes for new reviewers.*(evolution of design in PR)*

Question 7: How well did our proposed process support your workflow and assist you in maintaining up-to-date architectural documentation?

The proposed process required minimal modification to my workflow and was easy to integrate.*(easy to setup)* The updates to the architectural documentation are naturally iterative and updated often, granting

the documentation a higher degree of confidence. In addition, as the documentation was generated frequently, it assisted me in spotting architectural errors before commiting them to the codebase, which in the long run may help my system quality improve, as it is less complex and has less architectural errors. *(helps spot architectural errors)*

Question 8: How well did our proposed process support your workflow and assist you in improving the comprehensibility of your architectural documentation?

Following the process and applying Architectural Lens helped me gradually gain a more fine-grained understanding of the system and its components. *(improve understanding of system)*

Question 9: Do you have any suggestions to improve the tool to make the diagrams even more comprehensible or valuable?

Searchable diagrams would be of large help as it aids in breaking down top-level diagrams (which are still useful if they can be navigated).
Circular dependencies could probably be highlighted with thicker arrows. *(improvements)* More often than not you do not want them, if the tool can identify them that'd be a plus.

Question 10: What additional features or future research directions would you like to see explored in Architectural Lens?

It'd be nice to see what data types flow between modules most often. This is perhaps quite dependant on the language being analyzed, but seeing if certain types leak very far outside of the module where they are defined might indicate a leaky abstraction. *(improvements)*

Questionaire from Participant 4

Question 1: Could you briefly introduce yourself and share your background in software development?

Advanced Higher Vocational Education Diploma in Software Development and have been working in software development for a little bit more than 7 years as a developer and tech lead across various platforms and languages.

Question 2: Please briefly describe the system that you are currently working on

I work for a large financial institute with tightly coupled Python systems, maintained by several teams serving, which handles all the business needs for accounting and book keeping purposes.

Question 3: How do you envision integrating the proposed process into your day-to-day work?

I can see it being extremely useful in the change process to evaluate impact and scope of proposed changes to the system, which otherwise usually relies on manually updated (and often outdated) documentation or the knowledge of experienced developers. *(alignment with design)* If we could more easily identify our change impact, which is made difficult by having many integrations, we would could ease the burden of senior developers but also in turn more efficiently onboard new developers, speeding up the change process at the

same time and hopefully lowering our risks related to changes. *(helps spot architectural errors)* It could also similarly ease the onboarding of new colleagues by simply serving as an alternative to exploring the system in an illustrative way rather than by code. *(onboarding)*

Question 4: Considering our proposed process, how do you think the ability to scope views might impact your overall comprehension of your system architecture?

The scoped view functionality is key for the process serving as a multi-purpose tool. *(scoped views)* With scoped views it could be used for all steps in the development process. A low level scope can be used by the developer immediately to identify unintended side effects. *(helps spot architectural errors)* Whilst a top level view allows designers, testers and architects to properly evaluate module coupling. *(improve understanding of system)*

Question 5: In your opinion, what effect could our proposed process of incorporating diagram generation as part of a continuous process have on your system's architectural documentation?

In my experience, the only documentation that always stays up to date is the one that is done automatically and when it's integrated into the change process this ensures that our documentation always up to date and factual, automatically. *(up-to-dateness)* In turn I would hope this would further incentivize developers to take part in documentation.

Question 6: What are your thoughts on showcasing differences between the "main" branch and your feature branch when creating a pull request using our proposed process?

This is essential for the tool to be used in the change process and offers a lot of value for stakeholders as it can be used to more accurately measure impact and risk for proposed changes. *(helps spot architectural errors)* This means it could also be used to troubleshoot or better understand divergences in system output or state after a change by comparing previous views to the current one.

Question 7: How well did our proposed process support your workflow and assist you in maintaining up-to-date architectural documentation?

Because it was so easy to integrate I was up and running almost immediately and started generating architectural views based on live changes. *(easy to setup)* As it was done automatically from the codebase it was less prone to mistakes or inaccuracies and in turn also earlier in the development process. *(lack of human error)* Traditionally documentation would be done, by hand, after the change was in place but now we could fully explore architectural differences in a much earlier stage.

Question 8: How well did our proposed process support your workflow and assist you in improving the comprehensibility of your architectural documentation?

The proposed process allowed me to in an illustrative way explore our system in an up-to-date fashion. *(up-to-dateness)*

Question 9: Do you have any suggestions to improve the tool to make the diagrams even more comprehensible or valuable?

Perhaps the views could distinctly showcase different types of dependencies (e.g. circular dependencies, bi-directional dependenices, uni-directional dependencies). *(improvements)*

Question 10: What additional features or future research directions would you like to see explored in Architectural Lens?

If it could also export in some raw data format, rather than views, users could more easily parse it with code for analysis purposes depending on use case, allowing even more flexibility for the tool. *(improvements)* It would also mean that users could, potentially, build their own presentation/view layer for the architectural views which potentially would fit their own use case better.

Questionaire from Participant 5

Question 1: Could you briefly introduce yourself and share your background in software development?

Bachelor in software engineering and Software developer/operations engineer with 4 years of experience working in python backend systems

Question 2: Please briefly describe the system that you are currently working on

Web based monitoring system using precision temperature and humidity sensor hardware to monitor and validate manufactoring processes for food and pharma industries

Question 3: How do you envision integrating the proposed process into your day-to-day work?

Implementing views to current documentation and most valuable is hooking renders up to PRs and seeing the diff views. *(Difference views)* After attending the workshop, it was easy for me to get started using the process and tool.

Question 4: Considering our proposed process, how do you think the ability to scope views might impact your overall comprehension of your system architecture?

I think it'll mostly help me with keeping eye on maintability of the system. To easily spot if any new dependencies on a new PR has arisen and make sure to avoid any unwanted coupling from happening before merging. *(helps spot architectural errors)*

Additionally, when it comes to overall team efficiency and onboarding, I think that visual learning is an incredibly powerful way to quickly understand and get an overview of a new project. Having pre-generated and precise views of our system for onboarding of new employees could help with reducing man hours used on onboarding both for the new employee and the senior developers. *(onboarding)* While a new employee should never be afraid to ask questions to understand a new system, this also takes time off the senior developers having to explain the system, which in itself can be hard to do. Being able to easily divide your project into subsystems makes it easier to explain, and possibly reduce questions to senior devs how the system works. *(improve understanding of system)*

Question 5: In your opinion, what effect could our proposed process of incorporating diagram generation as part of a continuous process have on your system's architectural documentation?

Having this as the source for documentation makes sure its always updated, and no manual work apart from making new views when needed helps reducing time spent on documentation (which most developers prefer not to spend time on).*(up-to-dateness)* We've tried using automated tools in the past, but have not found any option which gave us anything of value (entire system views are hard to understand, only found tools which can do this).*(automation)* I like that you can create specific views in the json file, meaning that i only have to create the json file once, and then i can always generate the updated views when changes are made and save them for others to see.*(scoped views)*

Question 6: What are your thoughts on showcasing differences between the "main" branch and your feature branch when creating a pull request using our proposed process?

As previously mentioned, seems like a very nice way of keeping maintainability by visually being able to spot dependency changes, I tried using it with my colleague to spot errors in our pull requests, it feels like a clear visual way of validating your architecture continiously.*(Difference views)*

Question 7: How well did our proposed process support your workflow and assist you in maintaining up-to-date architectural documentation?

This reduces time spent on updating system diagrams, which was done once in a while. With it happening automatic, this ensures you have an up to date diagram, and developers don't have to question whether diagram is incorrect either from human errors or just not recently updated.*(lack of human error, up-to-dateness)* I have started storing some of the render views in my github, where they are relevant, it would be nice if you could specify where they are saved and have a github action in addition which refreshes them each time a commit is made.*(improvements)*

Question 8: How well did our proposed process support your workflow and assist you in improving the comprehensibility of your architectural documentation?

Being able to segment into views helps with focusing on what part of the system you're working on now, instead of seeing the whole system in itself and get lost in an enormous diagram. *(scoped views)*

Question 9: Do you have any suggestions to improve the tool to make the diagrams even more comprehensible or valuable?

I could see it being very handy to have renders being interactive. So you're able to see the entire system. From there on click and choose views you'd like to see, highlighting the once you've chosen, and grey out/lower opaque of the modules you haven't chosen in the view. Additionally as previously mentioned, being able to give render views a path where they get saved and then have them refreshed on each commit would be nice to keep them up to date, after saving them.*(improvements)*

Question 10: What additional features or future research directions would you like to see explored in Architectural Lens?

This being implemented to other git platforums such as Azure Devops, gitbucket etc. *(improvements)*