**BSc proj. - ArchLens in VS-Code**


| Name | E-mail |
|---|---|
| Casper Holten | casho@itu.dk |
| Sebastian Cloos Hylander | sehy@itu.dk |


Activity code: BIBAPRO1PE
Supervisor: Mircea Lungu

# Dynamically Generated Scoped Module Diagrams in the IDE - ArchLens for VS Code

Casper Holten
*BSc. Software Development*
*IT University of Copenhagen*
Email: *casho@itu.dk*

Sebastian Cloos Hylander
*BSc. Software Development*
*IT University of Copenhagen*
Email: *sehy@itu.dk*

***Abstract* - Architectural diagrams in large software projects easily become hard to maintain and understand, causing them to be abandoned. This results in developers lacking understanding of what their project's architecture actually looks like. We introduce ArchLens for VS Code which is an extension that dynamically generates scoped module views for python projects inside a developers IDE, keeping developers up to date with their project's architecture.**

## I. Introduction

The proliferation of software as a tool to optimise and automate tasks has increased the amount and complexity of software written. As a project grows in size, the increasing amount of internal dependencies makes it difficult for developers to be confident in the ways the software's modules interact with each other [1].

Managing a project's inter-module dependencies without appropriate tools, adherence to the architectural specifications, or the ability to visualize the architecture can result in architecture drift and erosion [1]. This can lead to a hard-to-read and tightly coupled codebase, increasing complexity and cost of writing software [2].

Architectural Lens (ArchLens)[1] is a tool to help developers focus on their project's architecture. ArchLens generates module views of Python source code through a command-line interface or as a GitHub Actions workflow. While ArchLens proved effective as a CLI tool and GitHub Actions workflow, a case study revealed that developers want deeper integration with their IDEs to enable easy-to-access and continuous architectural awareness [3].

A survey about developers' use of software architecture documentation [4] emphasised the importance of architectural views like what ArchLens generates. The survey also mentions that traceability and navigation between views and the developer's source code are essential for the developer to fully understand and work with the diagrams [4].

[1]https://github.com/archlens/ArchLens

Furthermore, a study about developers' usage of IDEs has shown that introducing more structured navigation and better tools for refactoring into a developer's IDE can improve overall efficiency when developing [5].

In this paper, we address ArchLens's limitations as a disconnected CLI tool for generating architectural views by integrating it in Visual Studio Code as an extension. We will do this by dynamically generating and rendering views in Visual Studio Code. Furthermore, we will connect the diagrams from ArchLens to the source code by making the views interactive and navigable, giving the developer an integrated IDE tool to explore the architecture of their project.

## II. State of the Art

Visualisation of software is an integral part of software development. Developers have developed a plethora of tools to visualise software, and these range from hand-drawn paper diagrams to automatically generated UML diagrams that can describe even the most complex systems. Software visualisation helps developers maintain software, gain knowledge about the architecture, and reverse engineer code that is foreign to the programmer [6]. Diagrams are also often a part of the software specifications when developing commercial software. Many methods for creating diagrams have been proposed, and here we present a few:

### A. Drawn Diagrams

Drawn diagrams are diagrams that are drawn using pen and paper or using specialised drawing software. Tools to draw such diagrams include Paint and Gimp, but PowerPoint also sees widespread adoption as a diagram creation tool [7]. Purpose-made tools like Draw.io and Visio can also be used. Drawn diagrams often focus on specific parts of a system, as the time-consuming nature of creating diagrams necessitates prioritisation [8].

Drawn diagrams reflect the diagram creator's understanding of the system, which makes them susceptible to errors or incompleteness, as they rely entirely on the creator's knowledge

and perspective [8]. If drawn diagrams are not maintained, they tend to become outdated representations of the system. This gradual loss of accuracy often leads to these drawn diagrams being abandoned entirely [9].

### B. Diagrams as code

Diagrams can be defined using programming languages or Domain Specific Languages (DSLs) developed specifically to define diagrams. PlantUML[2] and Graphviz[3] are two examples of widely used software visualisation tools, and they both define DSLs.

Defining diagrams in code allows for diagram artifacts to be stored alongside source code in version control, ensuring consistent visual representation across multiple versions and views while integrating seamlessly with existing development workflows.

Diagrams as code suffer from many of the same issues as drawn diagrams, since they are still based on the creator's understanding of the system, and they do not solve the issue of keeping diagram artifacts up to date.

### C. Diagrams derived from source code

It is possible to generate diagrams from source code with tools like UMLDoclet[4] for Java and py2puml[5] for Python. Many IDEs also include tools to generate UML class diagrams and UML module diagrams.

These diagrams reflect the actual state of the system, and they are easier to maintain since they can be automatically generated within seconds. They can also be incorporated into CI-CD pipelines, allowing programmers to use diagrams in the same way as automated testing.

Despite being maintainable and convenient to use, the generated diagrams are often visually complex and tend to show irrelevant information, hindering the user's ability to comprehend them, especially as a program grows large.

### D. View-scoped, source-code derived diagrams

Recent research proposes a way to automatically generate diagrams that are scoped to views defined in code. An example of this is ArchLens. It generates diagrams based on views defined in a JSON configuration file [3]. ArchLens generates user-specified views, and the views only contain elements declared in the configuration file.

[2]https://plantuml.com/
[3]https://graphviz.org/
[4]https://github.com/talsma-ict/umldoclet
[5]https://github.com/lucsorel/py2puml

Another novel approach that combines automatic diagram generation and diagrams defined in Python is Codoc [10], a tool that allows a developer to specify architectural views using Python. Codoc generates diagrams with similar visual information as ArchLens, but the diagrams can only be viewed on the now-defunct project website. This raises concerns about privacy and access to the internet, and the stability of web-based solutions [10].

These two approaches combine the high maintainability of automatically generated diagrams with the relevance and conciseness of drawn diagrams, thus resulting in easy-to-maintain and relevant diagrams [3].

Despite the wide selection of techniques for generating diagrams, the diagrams themselves seem to be difficult to relate to the source code that they represent [4]. Developers have expressed interest in having diagrams that have traceability to source code [4], and this is possible by integrating diagram generation directly with the IDE.

While current diagrams may provide a clear view of what the software looks like, they do not reveal why a system looks a certain way. In this paper, we investigate the possibility of integrating these diagrams as an interactive part of a developer's IDE to allow the user to inspect the architecture of a project.

## III. ArchLens for VS Code

We introduce 'ArchLens for VS Code', an extension for Visual Studio Code that enables users to use ArchLens directly in their IDE. The extension allows a user to visualise the architecture of their program with code-defined and focused architectural views, see how module relations evolve, and explore module and file relations, without ever leaving their editor.

The extension is installed through Visual Studio Code's built-in marketplace[6], or a file[7] available on the project's GitHub[8].

The extension is written using a combination of plain JavaScript, TypeScript, HTML, and CSS, and it utilizes Visual Studio Code's rich API for building extensions. Cytoscape.js[9], a rich graph visualisation library, renders the diagrams.

The extension uses ArchLens to generate the model of the underlying architectural views. To make ArchLens interoperable with the extension, we have modified it in two ways:

[6]https://marketplace.visualstudio.com/items?itemName=ArchLens.archlens-for-vscode
[7]https://github.com/archlens/ArchLens-VsCode-Extension/releases/tag/v0.2.1
[8]https://github.com/archlens/ArchLens-VsCode-Extension
[9]https://js.cytoscape.org/

1) Added support for tracking imports at the file level, enabling ArchLens to identify relationships between files, and not only modules.
2) Implemented JSON export functionality for architectural views.

### A. Configuring ArchLens for VS Code

The extension relies on an `archlens.json` configuration file to define views, GitHub URL, and branch, save location of diagrams, among other settings. The configuration is modified using JSON, and as seen in Listing 1, views can be added to the *views* property. The user can decide which modules to include in the view by adding them to the views' *packages* property, and modules can be ignored by adding them to the *ignorePackages* property.

```
{
  "$schema": "https :// raw. githubusercontent .com/archlens
       /ArchLens/master/src/config .schema.json",
  "name": "ArchLens",
  "rootFolder": "src",
  "github": {
    "url": "https :// github .com/archlens/ArchLens",
    "branch": "master"
  },
  "saveLocation": "./diagrams/",
  "views": {
    "completeView": {
      "packages": [
        {"path": "*", "depth": 3}
      ],
      "ignorePackages": []
    }
  }
}
```

Listing 1. Example `archlens.json`-file with the "Complete View" view. This view shows all modules up to three levels deep in the project structure.

### B. Using ArchLens for VS Code

The extension adds two commands to VS Code's Command Palette:

1) *ArchLens: Setup ArchLens* - checks that everything is set up correctly and otherwise helps the user get ready to use the extension by checking the following:
   - The user has chosen a virtual environment as their Python interpreter.
   - ArchLens is installed in the chosen virtual environment.
   - The user has an `archlens.json`-file, such that they can configure ArchLens.
2) *ArchLens: Open Graph* - opens the extension interface in a new tab, which starts the main functionality of the extension.

### C. Extension user interface

The extension user interface opens as a new tab. As seen in Figure 1, the extension user interface consists of a header, an area to render the selected view, and an explorer menu item 'Dependencies' to display module relation data. The header displays the available views. Users can switch between different views by clicking on the corresponding view button in the header. The header also contains a checkbox that toggles between view modes.

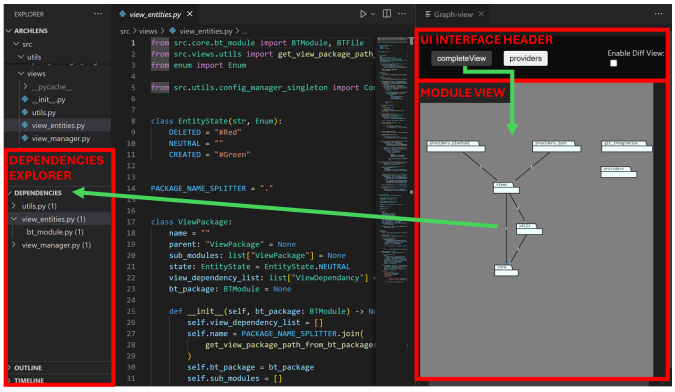


Fig. 1. ArchLens for VS Code being used to visualise the original ArchLens.

Module views can contain a lot of modules, and as seen in Figure 2, the render area of the extension has been given plenty of room to render views.

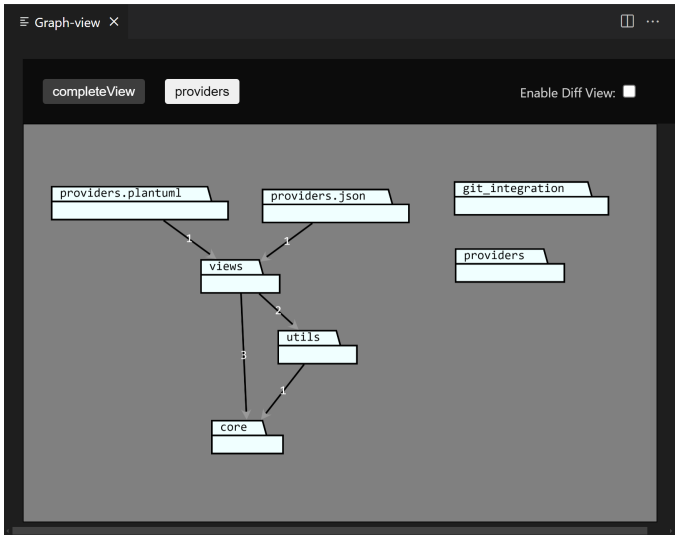


Fig. 2. The 'Module view' shows the overall architecture of the system within a given architectural view.

### D. Module views

The module views contain visual information about the modules included in the view and the relations between them. The labels between modules indicate how often the source module imports the target module.

The extension provides two different types of module views. The regular view shows the different modules and the imports between them. The Difference View (diff view) compares the project on Git's remote to the local state of the project and highlights the differences between the two.

- New edges or edges with added imports are coloured green. The edge label displays how many imports are new compared to the remote and the current number of imports.
- Edges that have been removed or edges with fewer imports than the remote branch are coloured red. The edge label displays the current number of imports and the number of imports removed compared to the remote.
- Entirely new modules are coloured green.
- Removed modules are coloured red.

The diff view gives insight into how the changes a developer has made impact the architecture of a project.

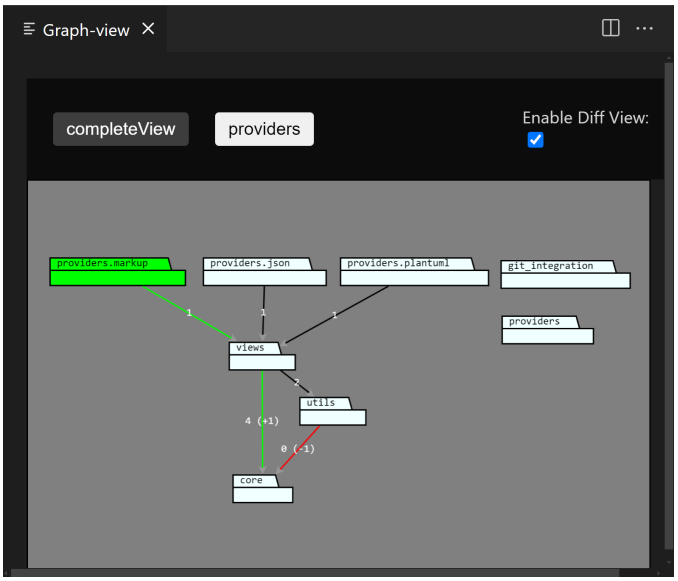


Fig. 3. The diff view compares the local project to the remote GitHub repository and highlights changed dependencies.

The extension monitors changes made in the project, and when a change has been made to a file, the module views are re-rendered. Generating the module views can take a while, so a toast will pop up to notify the user that new views are being generated.

### E. Exploring dependencies

All edges in the view are interactive. When clicked by a user, an edge will indicate that it is selected as seen in Figure 4, and a menu item 'Dependencies' will appear in VS Code's main explorer.

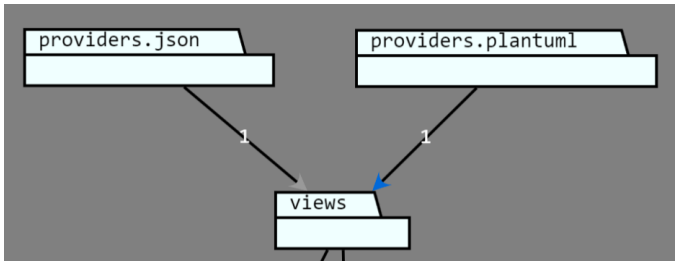


Fig. 4. An unselected edge and a selected edge. The selected edge is indicated by the blue arrow.

The Dependencies explorer shows all file-level relations in the selected edge. The top-level view shows files that import from the target module. The number in the filename, as seen in Figure 5, accounts for how many times the source file imports from the target module.

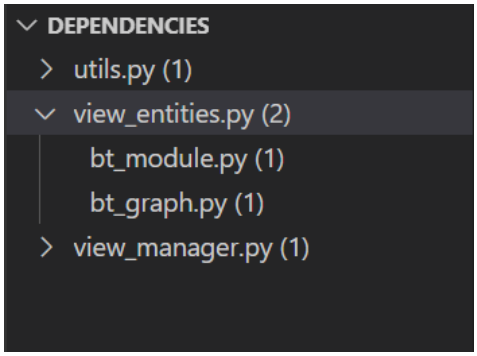


Fig. 5. The Dependencies explorer shows file relations between modules. Top-level displays files importing from the target module, with numbers indicating import counts. Expanded view shows the files imported from the source file, with numbers showing the import count from the target file.

Expanding a file in the top-level view reveals the files that the source file (top-level file) imports. It also shows how many times the target file is imported from the source file. Clicking a file in the Dependencies explorer item will open the file in the editor.

## IV. Case Study: Identifying and Resolving Architectural Issues in Zeeguu

We applied ArchLens for VS Code to Zeeguu[10], a real-world Python project. Zeeguu already uses ArchLens as a CLI tool

[10]https://github.com/zeeguu

and as a GitHub Actions workflow. This case study documents how we used ArchLens for VS Code to discover and resolve architectural issues in Zeeguu.

*1) Initial Exploration:* We loaded Zeeguu in VS Code, and observed that Zeeguu already uses ArchLens through other mediums, as we found four views defined in the project's `archlens.json`. We opened the extension and began exploring the views.

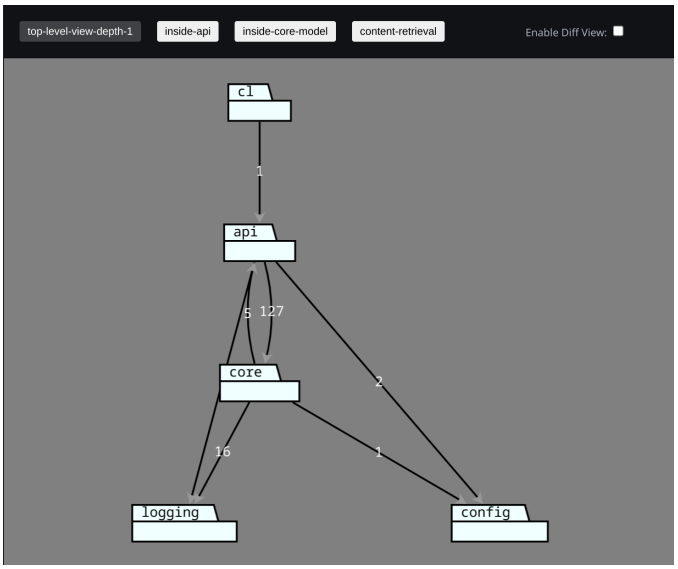


Fig. 6. The `top-level-view-depth-1` view showing Zeeguu's general system architecture. Note the circular dependency between *core* and *api*.

Figure 6 shows the `top-level-view-depth-1` view, which contains all modules at a folder depth of one, visualising Zeeguu's primary system architecture. The circular dependency between *core* and *api*, with *core* importing from *api* 5 times, and *api* importing from *core* 127 times, could indicate an architectural issue.

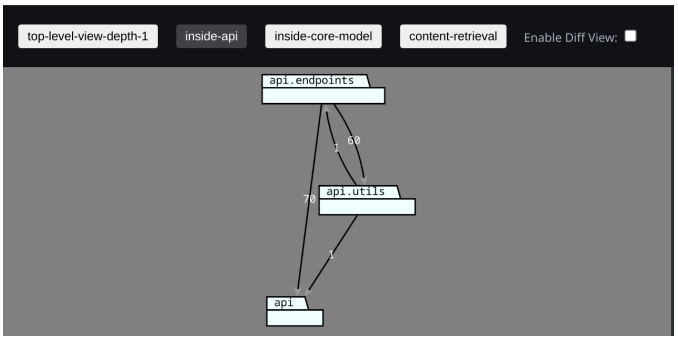


Fig. 7. The `inside-api` view showing the *api* module's architecture.

Exploring the `inside-api` view reveals more potential issues. In Figure 7, we can see a circular dependency between *api.endpoints* and *api.utils*. We believe the *api.utils* module should be decoupled from the rest of the module, and the singular reference from *api.utils* to *api.endpoints* could be the result of architectural erosion.

*2) Exploring problematic dependencies:* To understand where and why these dependencies occur in the relation between *core* and *api*, we used the extension's Dependencies explorer. As seen in Figure 8, clicking the edge from *core* to *api* in the `top-level-view-depth-1` opened the Dependencies explorer and revealed the specific files imported to *core* from *api*.

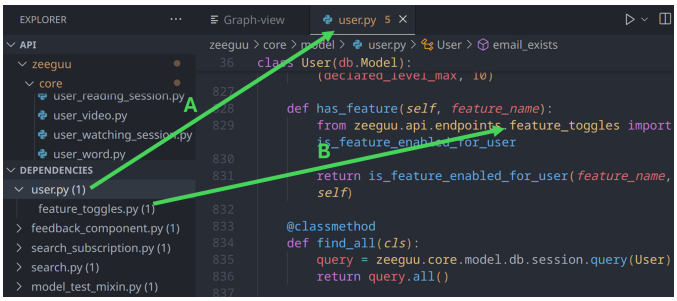


Fig. 8. The Dependencies explorer showing which files are import to *core* from *api*. A) The clicked top-level file is opened in a new tab. B) The file in the expanded drop-down menu is the file imported from *api*. It can be clicked, and the file's source code opens in a new tab.

In the Dependencies explorer, we could immediately navigate to the source code of the files importing from *api* in *core* by clicking the top-level files as seen in Figure 8(A). Here we could observe that some imports were happening in the file scope, and some occurred in functions.

We expanded the top-level files and clicked the files in the expanded drop-down menus as seen in Figure 8(B), such that we could read the source code of the files from *api* and understand why they were imported.

*3) Correcting architectural issues:* After exploring all five dependencies, we observed that these were indeed unintentional.

- `core/model/users.py` imports `feature_toggles.py`. This can be corrected by using dependency inversion and passing the feature toggle checks as function parameters instead of using imported functions from *api*.
- `core/model/feedback_component.py` has an unused import from `abort_handling.py`. This import is removed.
- `core/model/search_subscription.py` imports a `make_error` from *api*, that handles HTTP errors. We refactor the function to throw an exception instead. This exception can be caught, and the appropriate HTTP error can be handled in *api*.
- `core/model/search.py` has an unused import from `abort_handling.py`. This import is removed.

- `core/test/model_test_mixin.py` is tightly coupled to `api`, as it uses it to initialize a test program. This import will require a larger architectural review to extract tests to a separate module or project. This import is left for now.

We tried to refactor the files in question to conform to the intended one-directional relationship from *api* to core, such that *core* was completely decoupled from *api*. This was not accomplished, but four of the five references were removed, and the coupling from *core* to *api* was lowered. The changes can be seen in Figure 9.

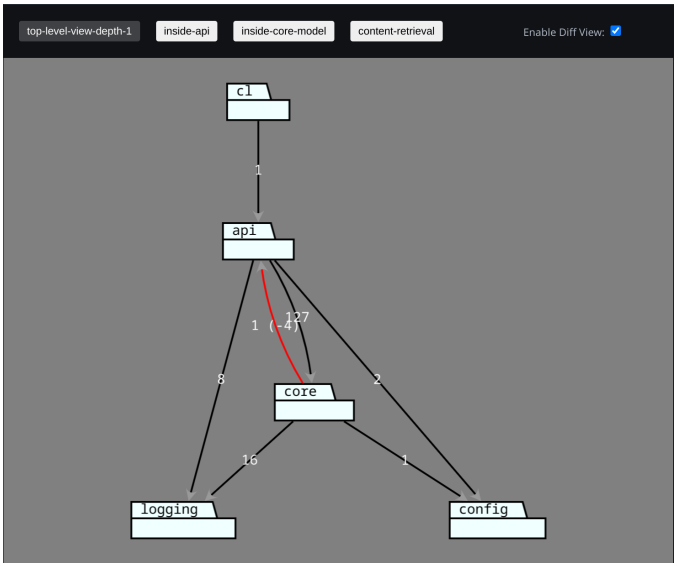


Fig. 9. `inside-api` seen through the diff view. The red edge indicates that four of the five imports from *core* to *api* have been removed.

The complete removal of a circular dependency in `inside-api` can be seen in Figure 10, where a one-directional relation *api.utils* and *api.endpoints* is observed after refactoring the module.

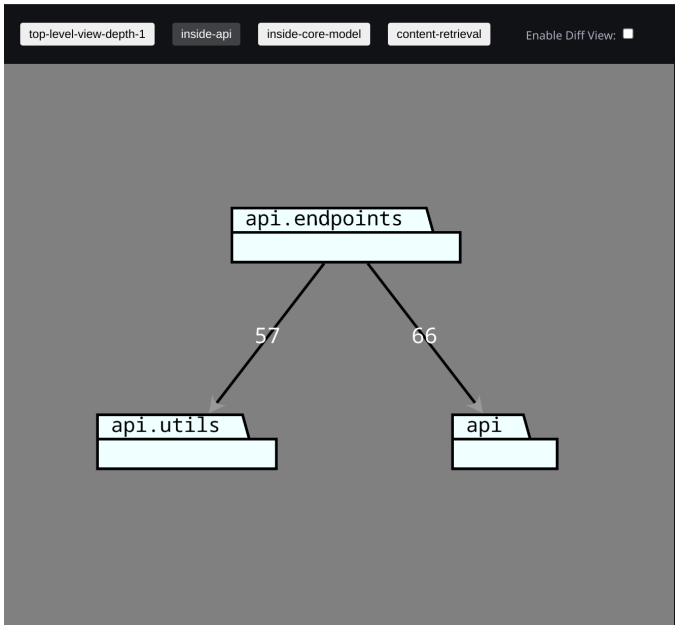


Fig. 10. `inside-api` after removing the circular dependency between *api.utils* and *api.endpoints*.

After refactoring `top-level-view-depth-1` and `inside-api`, we wanted to see how our local changes compare to Zeeguu on Git's remote. We turned on the Diff View, and as seen in Figure 9, the changes only remove the unwanted dependencies, without having an unintended impact on the architecture. No other developer has introduced architecture-eroding dependencies while we were working.

Through this case study, we have demonstrated that ArchLens for VS Code can be used to identify and rectify architectural issues. The views allowed us to find circular dependencies that violated the intended system architecture, and we could quickly navigate to relevant source code using the Dependencies explorer. The responsive nature of the views allowed us to verify that the refactoring had the intended changes, and the diff view allowed us to verify that no unintended architectural changes were introduced in the process.

## V. Discussion

### A. Reflections on the case study and ArchLens for VS Code

The case study presented was conducted by the authors themselves, and Zeeguu is a project developed and maintained by our supervisor. While this allowed us to gain access to a project that already uses ArchLens and has useful views configured, it introduces biases and a level of knowledge about the program that unaffiliated developers would not have.

We attempted to maintain objectivity throughout the case study, and focus on concrete observations from using the extension, but the connection between the extension, project

and the authors could influence the results. The case study revealed how ArchLens for VS Code can be used to spot architectural erosion, navigate from diagram to relevant source code, and verify the architectural correctness of a project while refactoring.

In line with what our own case study found, the case study from the original ArchLens Paper also showed that ArchLens as an IDE extension has the potential to add value and insight to a project throughout development [3].

However, future studies should be conducted that include developers who have no affiliations or prior experience with ArchLens for VS Code or Zeeguu, as this would provide a more objective assessment of the extension's utility in a project.

### B. Performance

It takes a long time to generate views. ArchLens parses the entire project as an Abstract Syntax Tree (AST), which can be time-consuming on large projects. This might not be an issue when ArchLens is used as a CLI tool, as the developer occasionally and intentionally runs ArchLens. Since the extension refreshes the views far more often, the developer will spend time waiting for the views to update.

This is especially noticeable when using the diff view, since it clones the remote repository and parses both the local and remote projects as ASTs before comparing them to each other.

On a project like Zeeguu, generating diff views can take upwards of a minute. This will automatically happen every time the view is updated, which is every time the developer saves a change in VS Code. Due to this, using diff views actively while developing is impractical, as the developer would spend most of their time waiting on the diff view to regenerate.

With the current performance issues, the diff view is mostly suited as a tool to verify changes to the architecture before committing changes to the remote.

To fix the performance-related issues, we propose two solutions.

*1) Enhancing the diff view:* By using the local `.git`-file, the time it takes to generate diff views could be improved. The `.git`-file would be used to create a local temporary project to compare against, rather than cloning the remote. This could improve the time it takes to generate diff views, making it more practical to use the diff view while coding.

*2) Use VS Code's code analysis functionality instead of ArchLens:* Another way to improve performance of the extension would be to use VS Code's built in code analysis tools to generate the views instead of ArchLens, as ArchLens was not originally made to be used with a VS Code Extension. Integrating the code analysis as part of the extension would allow direct access to the user's open workspace and the API for the IDE's Language Server Protocols (LSP), allowing faster parsing of the source code.

By moving the code analysis to the IDE, it would be easier to allow other extensions to extend ArchLens for VS Code, since we could expose a language API. This language API could be used by other extensions to implement code analysers for other languages, e.g., JavaScript, C#, or C.

This would remove the dependency on the original Python-based ArchLens and make it easier to support other languages in the extension.

### C. Envisioned features

The extension provides the user with a more seamless integration of the architectural views generated by ArchLens. The user also has direct access to the source code, giving them a tool for exploring and fixing potential problems in the architecture. However, we believe that both the vision of insightful and automatically generated scoped views as well as ArchLens for VS Code can be further improved.

Here we present a few visions for how to increase the usefulness of ArchLens for VS Code.

*1) More diff views:* The diff view is currently set to compare the local project with the latest commit in the remote repository. To improve the value of the diff view, we want the user to be able to configure different diff views. This would allow a developer to define diff views that compare the local state of the project to important commits in the Git history. A way this can be done is by adding a diff view section in `archlens.json`, where a developer can specify different commits to compare to, as seen in listing 2.

```
{
   [... start of archlens.json ...]
  "views": { ... },

  "diffViews": {
    "lastCommit": -1,
    " latestRelease ": "92fdc792"
  }
}
```

Listing 2. Example `archlens.json`-file with the proposed concept of defining different commits for the diff view.

In the example, we have defined two different diff views.

1) *lastCommit* - A view that always compares the program to the last commit in the log. By writing a negative integer, it is interpreted as tracing 1 commit back in the commit history.

2) *latestRelease* - A view which compares to the commit hash with the latest release of the project.

This would allow us to generate diff views that compare the current state of the project to specific commits in the commit history, rather than the somewhat arbitrary state of the project that is currently on the remote.

*2) Highlight differences:* To help a developer notice changes to their views and architecture, a possible addition to the extension would be to add a UI element to the header that shows which views have had changes to them. The Dependencies explorer should also show which files have had changes to them, and whether or not imports have been added or removed from the files.

*3) The extension as a navigation tool:* The extension could also be improved by adding improved navigation between the views and the source code, as well as between the views themselves. Currently, the views and the source code are connected via the Dependencies explorer, but introducing more features for structured navigation could benefit the developer's ability to navigate from diagram to source code.

In the future, a developer might be able to navigate by:

- Opening a module folder in the file explorer by clicking the corresponding node in the diagram. This allows for visual navigation to the modules in a project, possibly making navigation in large projects like Zeeguu more manageable.
- Linking views to diagram nodes to allow navigation from diagram nodes to related views. This could improve the navigation in the project and between the views themselves, e.g., say the developer could link a module from one view to another view, then clicking the *api* module in the `top-level-view-depth-1` view in the Zeeguu project could open the view `inside-api`, allowing the developer to gain further information about the structure of *api*.

## VI. Conclusion

In this paper, we have introduced ArchLens for VS Code. A VS Code extension that dynamically generates module views for a Python project. A user of the extension can define specific and insightful views of subsystems or interesting parts of their project. These views will automatically be refreshed when changes have been made to the project. Many features from the original ArchLens have made it to the extension, including the diff view. We have also introduced interactive edges and the Dependencies explorer, which allows the developer to navigate from diagram to source code.

A case study was conducted on the project Zeeguu, which showed that ArchLens for VS Code can be used to spot architectural erosion, navigate from diagram to relevant source code, and verify the architectural correctness of a project while refactoring. This helps developers focus on the architecture of a program throughout their development while not changing their workflow substantially.

Lastly, we went through a few possible improvement to the extension, mainly focusing on either improving performance or adding additional features for a developer using ArchLens for VS Code.

The next step is to evaluate the extension on a group of unaffiliated developers to see exactly how they respond to ArchLens for VS Code.

## References


[1] D. Pollet and S. Ducasse, " Software Architecture Reconstruction: A Process-Oriented Taxonomy ," *IEEE Transactions on Software Engineering*, vol. 35, pp. 573–591, July 2009.

[2] E. E. Ogheneovo, "On the relationship between software complexity and maintenance costs," *Journal of Computer and Communications*, vol. 2, no. 14, pp. 1–16, 2014.

[3] J. K. Rusbjerg and N. P. Andersen, "Architectural lens: A tool for generating comprehensible diagrams," Master's thesis, IT University of Copenhagen, may 2023.

[4] D. Rost, M. Naab, C. Lima, and C. von Flach Garcia Chavez, "Software architecture documentation for developers: A survey," *Lecture Notes in Computer Science*, pp. 1–88, 2013.

[5] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. R. Nair, and D. Shepherd, "Chapter 5 - a practical guide to analyzing ide usage data," in *The Art and Science of Analyzing Software Data* (C. Bird, T. Menzies, and T. Zimmermann, eds.), pp. 85–138, Boston: Morgan Kaufmann, 2015.

[6] R. Koschke, "Software visualization in software maintenance, reverse engineering, and reengineering: A research survey," *Journal on Software Maintenance and Evolution*, vol. 15, pp. 87–109, 03 2003.

[7] R. F. Ciriello, A. Richter, and G. Schwabe, "Powerpoint use and misuse in digital innovation," in *ECIS 2015 Completed Research Papers*, p. Paper 32, AIS Electronic Library (AISeL), 2015. University of Zurich.

[8] C. Lange, M. Chaudron, and J. Muskens, "In practice: Uml software architecture and design description," *IEEE Software*, vol. 23, no. 2, pp. 40–46, 2006.

[9] A. M. Fernández-Sáez, M. Genero, and M. R. V. Chaudron, "Empirical studies concerning the maintenance of UML diagrams and their use in the maintenance of code: A systematic mapping study," *Information and Software Technology*, vol. 55, no. 7, pp. 1119–1142, 2013.

[10] C. W. Bang and M. Lungu, "Codoc: Code-driven architectural view specification framework in python," in *2021 Working Conference on Software Visualization (VISSOFT)*, pp. 120–124, 2021.