

# **1. Block 4.2: CS6514: Software Architecture Design Portfolio**

Fluctuating Finite Element Analysis (FFEA) Case Study

Art Ó Liathain

September 2025

## 2. Table of Contents

### Contents

1. Block 4.2: CS6514: Software Architecture Design Portfolio .....	1
2. Table of Contents .....	2
3. Tech Stack .....	3
4. Use Case Diagram .....	5
5. Utility Tree .....	6
6. SysML Diagram .....	7
7. Codesense Diagram .....	8
8. SonarQube Diagram .....	9
9. DV8 .....	10
10. Understand .....	11
11. C4 Diagram .....	12
12. Dependency Graph .....	12
13. Class Diagram .....	13
14. References .....	13

### List of Figures

Figure 1 Tech Stack .....	3
Figure 2 Use Case Diagrams .....	5
Figure 3 Utility Tree .....	6
Figure 4 SysML Diagram .....	7
Figure 5 Codesense Architecture Diagram .....	8
Figure 6 SonarQube Complexity Diagram .....	9
Figure 7 Understand Function Analysis .....	9
Figure 8 Understand Function Analysis .....	10
Figure 9 Understand Function Analysis .....	10
Figure 10 Understand Function Analysis .....	11
Figure 11 Understand Function Analysis .....	11
Figure 12 C4 Diagram .....	12
Figure 13 Dependency Graph .....	12
Figure 14 Class Diagram .....	13

### 3. Tech Stack

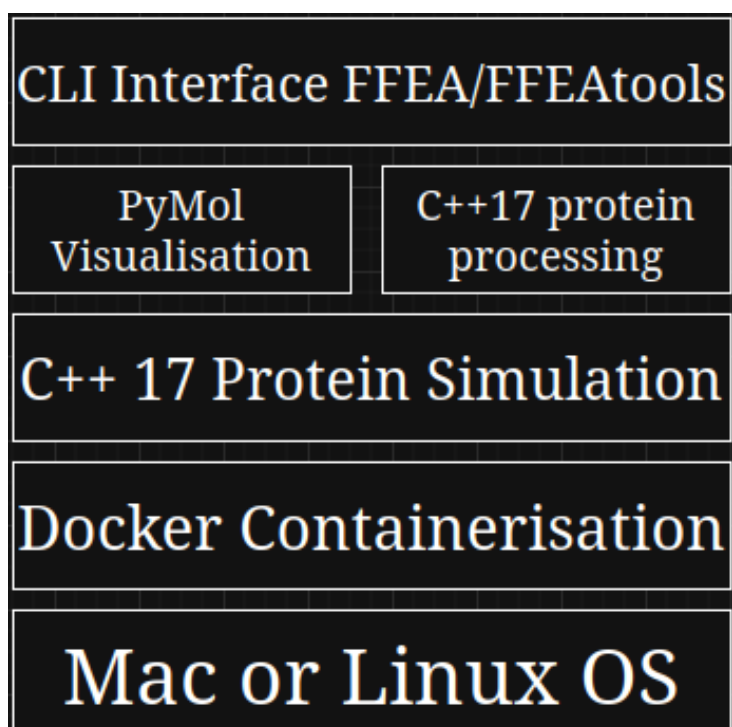
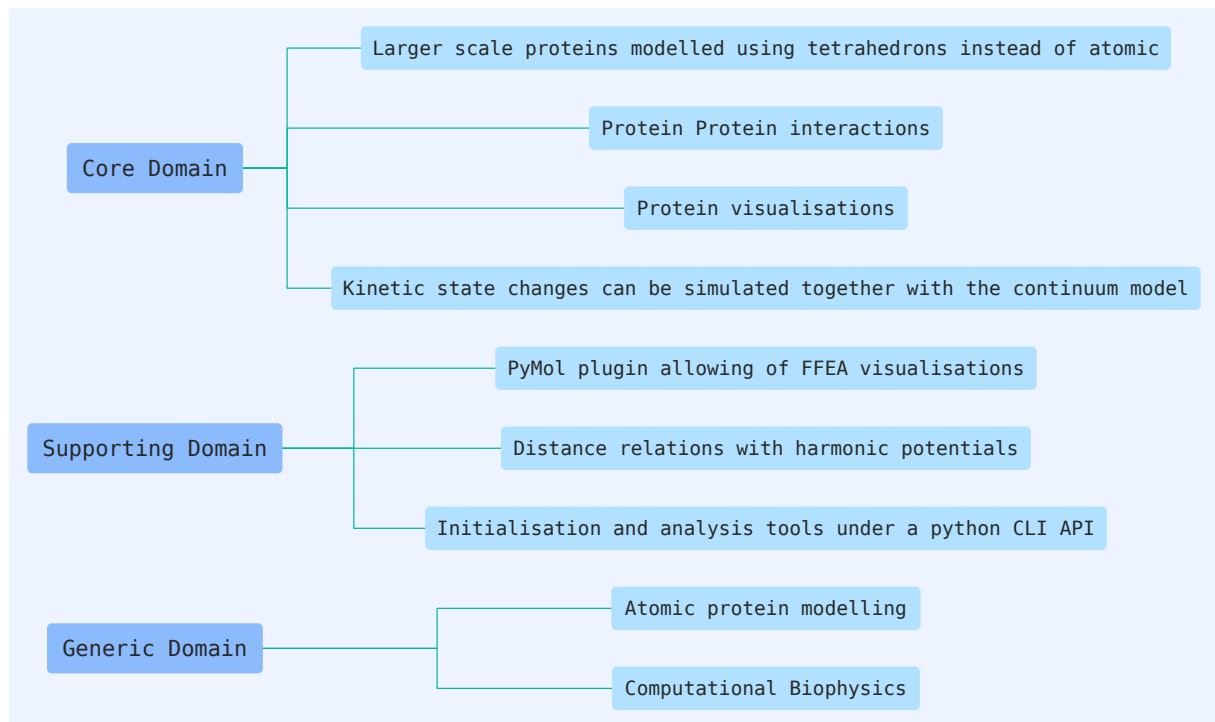


Figure 1: Tech Stack

The tech stack is primarily focused on C++, the processing and the simulations are all carried out in C++, this is a natural choice as performance is a tenant of the program. The visualisations build on the PyMol library with a plugin. This leads to a natural decoupling where visualisation is separated from processing. This is a suitable tech stack for the project allow for a modular approach and following the qualities required for the project.



Listing 1: Domain Model

## 4. Use Case Diagram

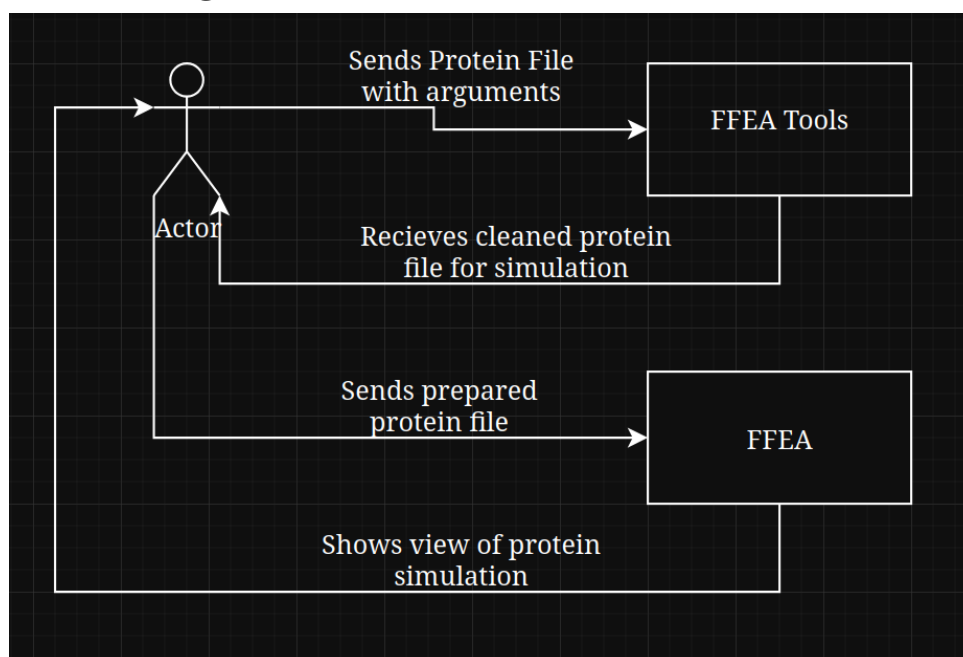


Figure 2: Use Case Diagrams

The primary use case for FFEA is to process protein files to then simulate under user desired conditions. The interface is rudimentary in terms of use as the complexity lies in both the simulation and parameters which means that a simple process still has layers of complexity. Looking at the use case there could be an argument to bundle the tooling as the whole purpose of ffeatools is to prepare files for ffea meaning a single unified process that processes then displays based on cli arguments would streamline the process.

## 5. Utility Tree

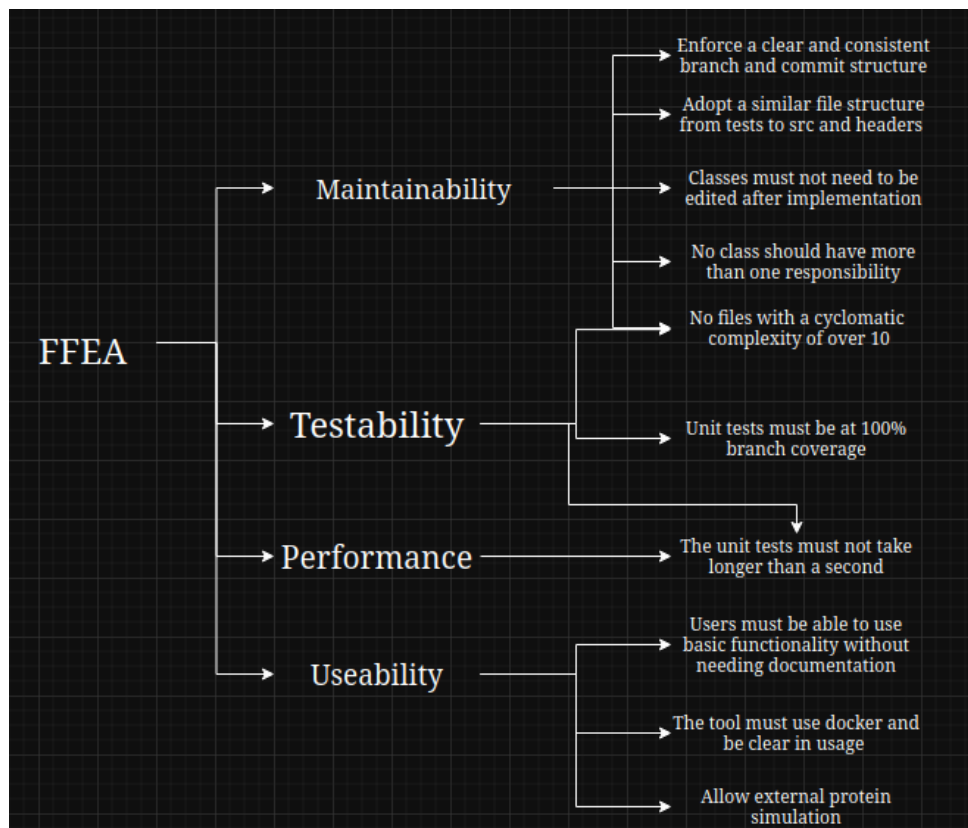


Figure 3: Utility Tree

The utility tree is quite idealistic of aspirational long term goals of the project. There is a heavy focus on maintainability as that must be a tenant of the software for the longevity of the project and research. The maintainability goals have been selected as the foundational work needed in terms of refactoring to move the project to a useable state before any new features are added.

Testability is primarily focused on having a robust test suite to test against to allow simpler refactoring. The goal of 100% test coverage is lofty but a necessary requirement to allow for refactoring with confidence.

Performance is a tenant that is key to the project through the functional requirements but in terms ofilities it is much harder to focus non fucntional requirements through that lens as the testable performance is not easily described.

Useability is targeted towards allowing this tool to be run anywhere regardless of OS as Windows is not supported now but docker can resolve this. The tool must allow for users to interact with it without needing the wiki to see the help commands and simple processing as well to allow for easier and more vaired user testing and feedback.

## 6. SysML Diagram

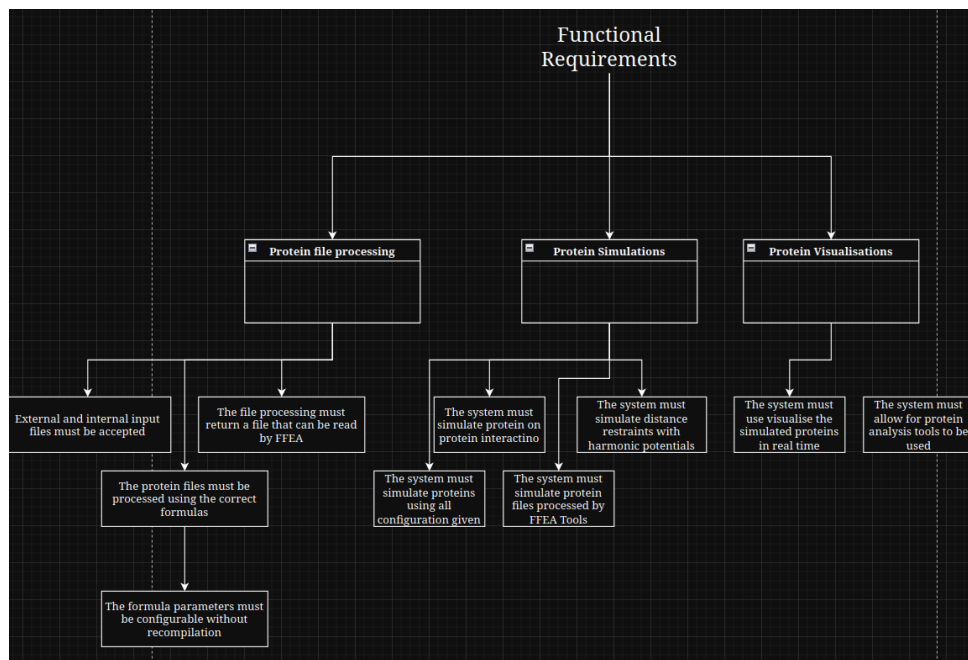


Figure 4: SysML Diagram

The SysML diagram highlights the standout requirements of the tool to allow for accurate protein simulations to be done with respect to user input. There are undoubtedly many more functional requirements but assuming the simulations as primarily a black box the requirements centre around accurate simulations to be done and visualised based on the user configuration allowing for the correct interactions to be observed. The requirements are broken into three large logical sections being the processing, simulating and visualizing. This SysML structure allows for room for growth as more requirements are discovered.

## 7. Codesense Diagram

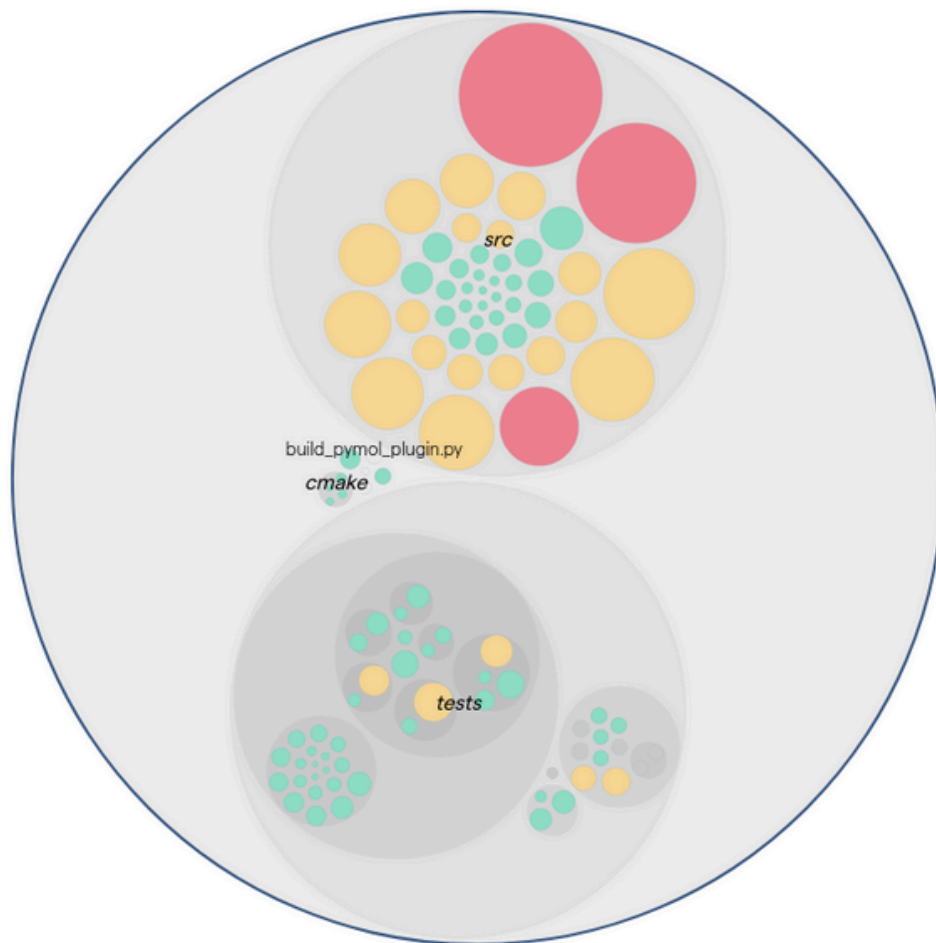


Figure 5: Codesense Architecture Diagram

This diagram highlights three key things about the architecture decisions taken for this project. The first feature being the flat structure taken for `src`, this means that all files are the same level and there is no logical grouping for any of the `src` files which makes grouping much more difficult. This is in spite of the fact that the test files are very well structured.

This leads into the second point of the severe size difference in size between `tests` and `src`, this signifies a significant amount of the code is untested making it much more difficult to add to the codebase with confidence.

Lastly looking at the colours of the larger files three files are exceptionally difficult to modify and maintain, again a major hit to maintainability



## 8. SonarQube Diagram

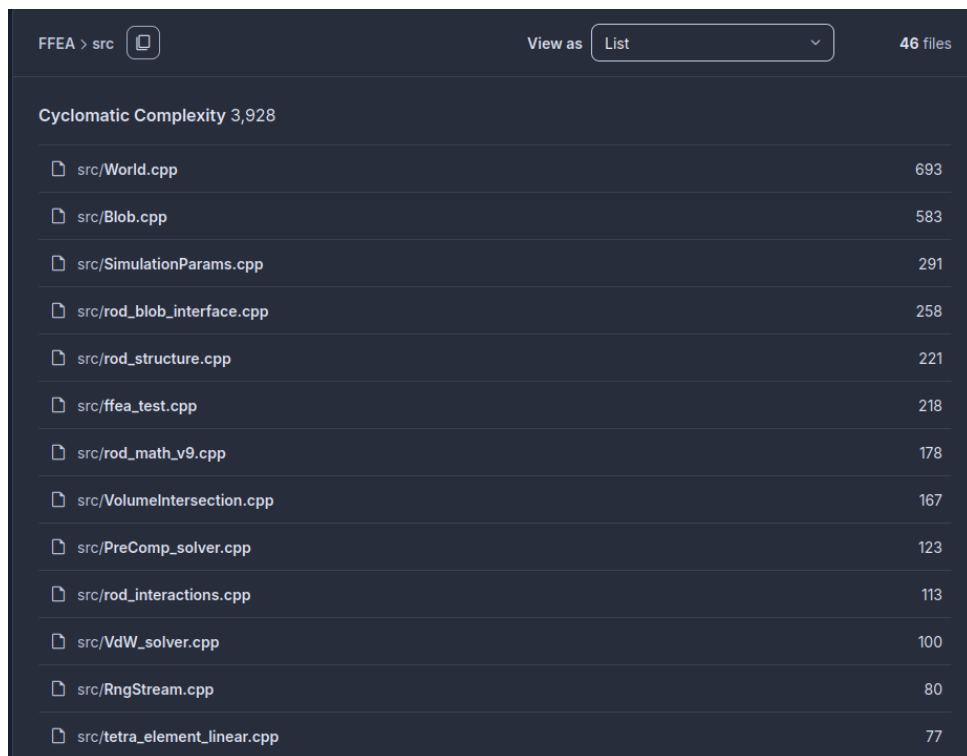


Figure 6: SonarQube Complexity Diagram

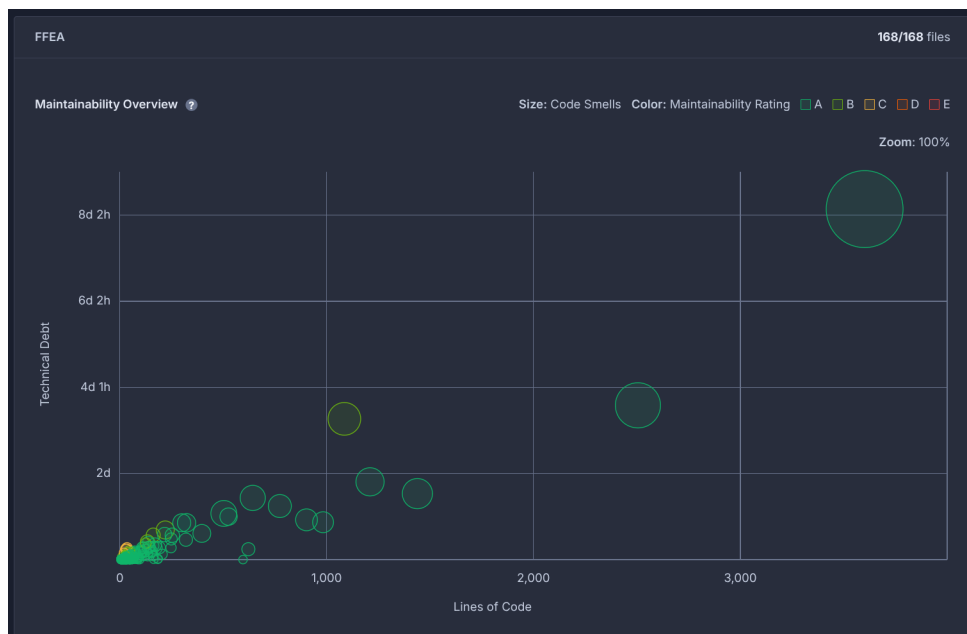


Figure 7: Understand Function Analysis

This is the cyclomatic complexity of the offending files and others from the codesense analysis.

## 9. DV8

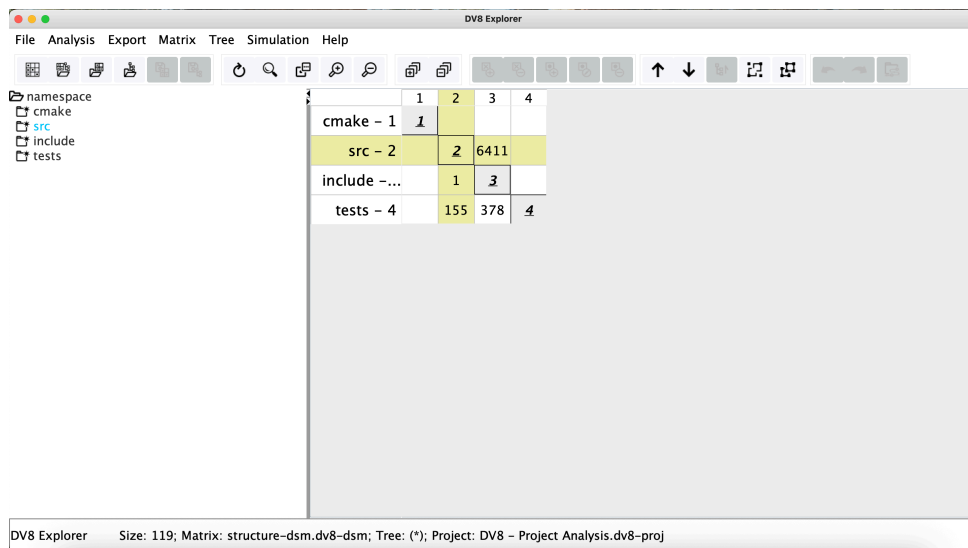


Figure 8: Understand Function Analysis

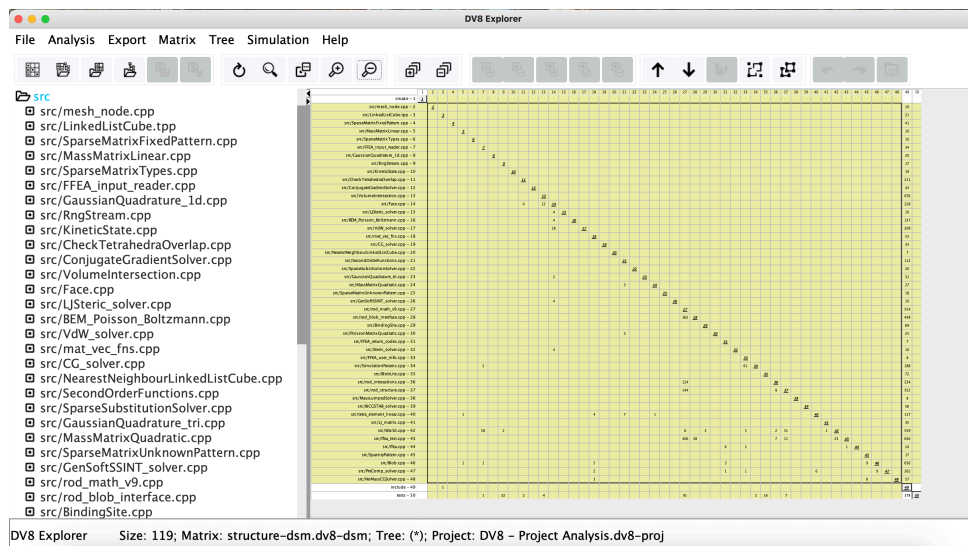


Figure 9: Understand Function Analysis

## 10. Understand

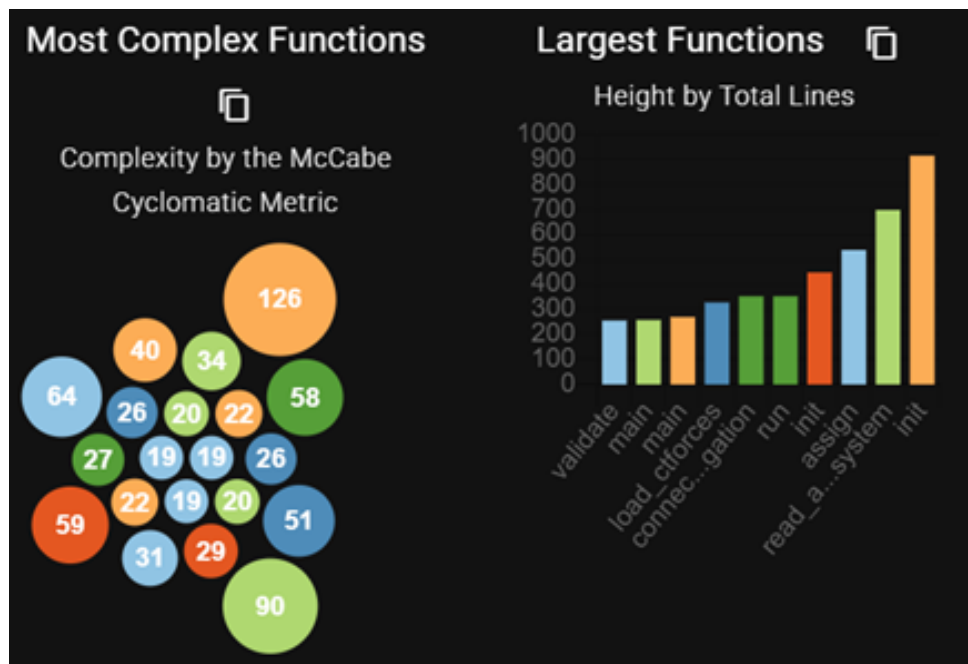


Figure 10: Understand Function Analysis

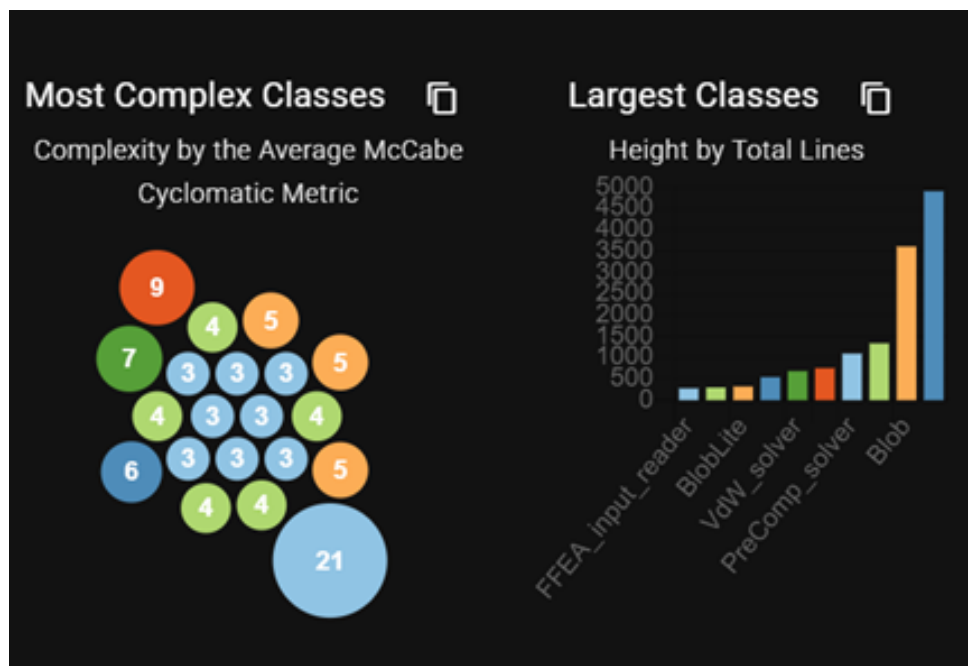


Figure 11: Understand Function Analysis

## 11. C4 Diagram

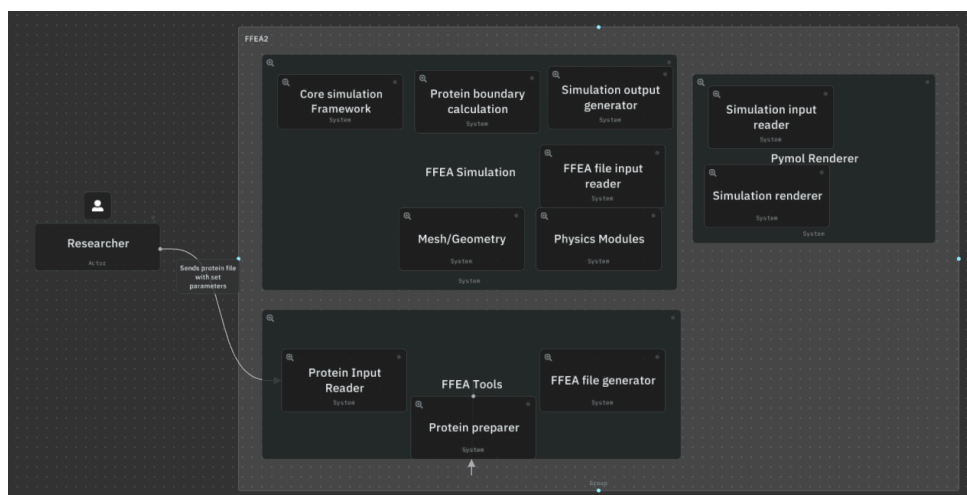


Figure 12: C4 Diagram

## 12. Dependency Graph

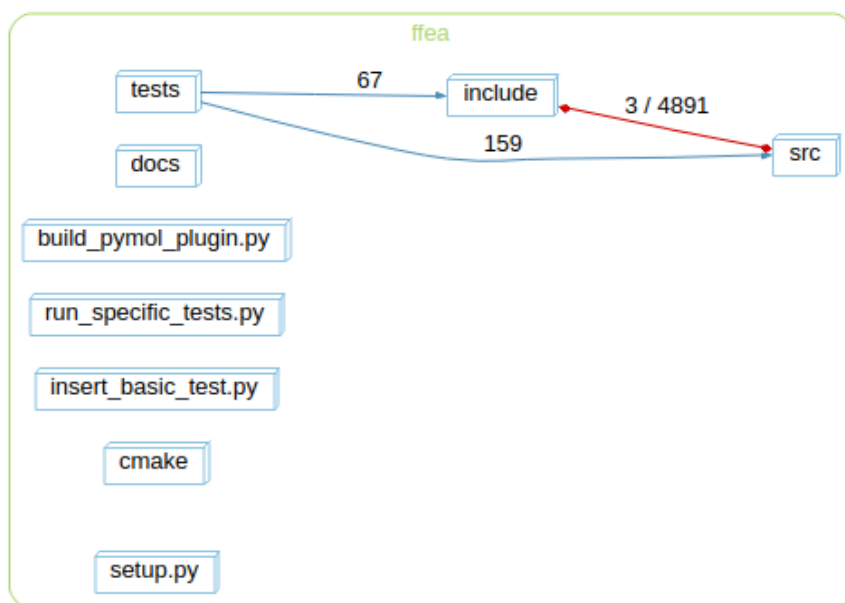


Figure 13: Dependency Graph

## 13. Class Diagram



Figure 14: Class Diagram

## 14. References

- Typst Documentation: <https://typst.app/docs/>

