1. Software Architecture Design Portfolio

Art Ó Liathain

October 2025

2. Table of Contents

Contents

1. Software Architecture Design Portfolio 1
2. Table of Contents 1
3. Merlin++ 4
   3.1. Current State 4
   3.2. Proposed Architecture 4
   3.3. Reasoning 5
   3.4. Conclusion 5
   3.5. Diagrams 5
      3.5.1. Tech Stack 5
      3.5.2. Domain Model 6
      3.5.3. Utility Tree 7
      3.5.4. Use Case Diagram 7
      3.5.5. 4 + 1 Diagram 8
      3.5.6. Logical View 8
      3.5.7. Physical View 8
      3.5.8. Development View 8
      3.5.9. Process View 9
      3.5.10. Codescene 9
      3.5.11. DV8 11
      3.5.12. SonarQube 12
      3.5.13. Understand 13
      3.5.14. C4 14
   3.6. Fluctuating Finite Element Analysis (FFEA) Case Study 15
   3.7. Tech Stack 15
   3.8. Use Case Diagram 16
   3.9. Utility Tree 17
   3.10. SysML Diagram 18
   3.11. Codesense Diagram 19
   3.12. 4+1 Diagram 20
      3.12.1. Logical View 20
      3.12.2. Process View 20
      3.12.3. Developmental View 20
      3.12.4. Physical View 20
   3.13. SonarQube Diagram 21
   3.14. DV8 22
   3.15. Understand 23
   3.16. C4 Diagram 24
   3.17. Dependency Graph 24
   3.18. Class Diagram 24
4. iDavie Case Study 25
   4.1. Tech Stack 25

4.2. Domain Model 25
4.3. Utility Tree 26
4.4. Use case diagram 26
4.5. 4 + 1 Diagram 27
4.5.1. Logical View 27
4.5.2. Physical view 27
4.5.3. Process View 27
4.5.4. Developmental View 28
4.6. Codescene 28
4.7. Sonarqube 30
4.8. Understand Code Analysis 32
4.9. Class Diagrams 32
4.10. C4 Diagram 33
5. ACTS Case Study 34
5.1. Tech Stack 34
5.2. Utility Tree 35
5.3. Use Case Diagram 35
5.4. 4+1 Diagram 36
5.4.1. Logical View 36
5.4.2. Process view 36
5.4.3. Physcial View 36
5.4.4. Development View 36
5.5. Codescene 37
5.6. Understand 38
5.7. Sonarqube 39
5.8. C4 41
Bibliography 42

List of Figures

Figure 1 MicroKernel Architecture Diagram 4
Figure 2 Merlin Tech Stack 5
Figure 3 Merlin Utility Tree 7
Figure 4 Merlin Use Case diagram 7
Figure 5 Merlin Codescene coupling graph 9
Figure 6 Merlin Codescene maintainability circle 10
Figure 7 Merlin Dv8 Dev tool analysis 11
Figure 8 Merlin Dv8 high level analysis 11
Figure 9 Merlin Technical debt cost 12
Figure 10 Merlin Duplicated code analysis 12
Figure 11 Merlin Function complexity understand 13
Figure 12 Merlin Coding langauge breakdown 13
Figure 13 Merlin Understand code dependency graph 13
Figure 14 Merlin C4 Diagram 14
Figure 15 Tech Stack 14
Figure 16 Tech Stack 15
Figure 17 Use Case Diagrams 16
Figure 18 Utility Tree 17
Figure 19 SysML Diagram 18
Figure 20 Codesense Architecture Diagram 19

Figure 21 SonarQube Complexity Diagram 21
Figure 22 Understand Function Analysis 21
Figure 23 Understand Function Analysis 22
Figure 24 Understand Function Analysis 22
Figure 25 Understand Function Analysis 23
Figure 26 Understand Function Analysis 23
Figure 27 C4 Diagram 24
Figure 28 Dependency Graph 24
Figure 29 Class Diagram 24
Figure 30 Use case state machine 26
Figure 31 Logical View 27
Figure 32 Codescene coupling diagram 40% 28
Figure 33 Codescene Maintenance Diagram 29
Figure 34 Codescene Hotspot overview 29
Figure 35 Sonarqube cyclomatic complexity 30
Figure 36 Sonarqube maintainability 30
Figure 37 Sonarqube reliability graph 31
Figure 38 Duplications Graph 31
Figure 39 32
Figure 40 32
Figure 41 32
Figure 42 C4 Diagram 33
Figure 43 Use Case Diagram 35
Figure 44 Dependency Coupling Graph 37
Figure 45 Technical Debt Codescene 38
Figure 46 Understand High Level Info 38
Figure 47 Understand complexity ratings 38
Figure 48 Class Diagram Inheritance 39
Figure 49 Class Diagram flat 39
Figure 50 39
Figure 51 40
Figure 52 40
Figure 53 41
Figure 54 Context Diagram 41
Figure 55 Component Diagram 42
Figure 56 Container Diagram 42

3. Merlin++

3.1. Current State

The current codebase is an artifact that seeks to achieve Listing 1 (High speed particle simulation). The code has an accidental architecture as an ad-hoc feature development approach meant that the only concrete factor was the tech stack(Figure 2). Figure 6 is a clear example of the outcome of this, a jumbled mess of code is mashed together.

While successful in achieving functionality the code architecture makes change difficult and has slowed the potential development of features due to the complexity of adding new features. This architecture is something that will not last in the long term and a purpose picked architecture is needed to support the l:wqong running development and maintenance of the project in future.

3.2. Proposed Architecture

To address these issues, looking at the Merlin documentation highlights the two key architectural goals the project had in development; to do the job in the simplest form and to produce a set of loosely coupled components which are easily maintainable ([1]). These goals along with the core purpose of the system being simulation leads me to believe that a microkernel architecture is the best architecture for this project. A microkernel architecture is a system built around a single purpose, in this case simulation while allowing a plugin system to introduce new functionality without changing the core code [2].

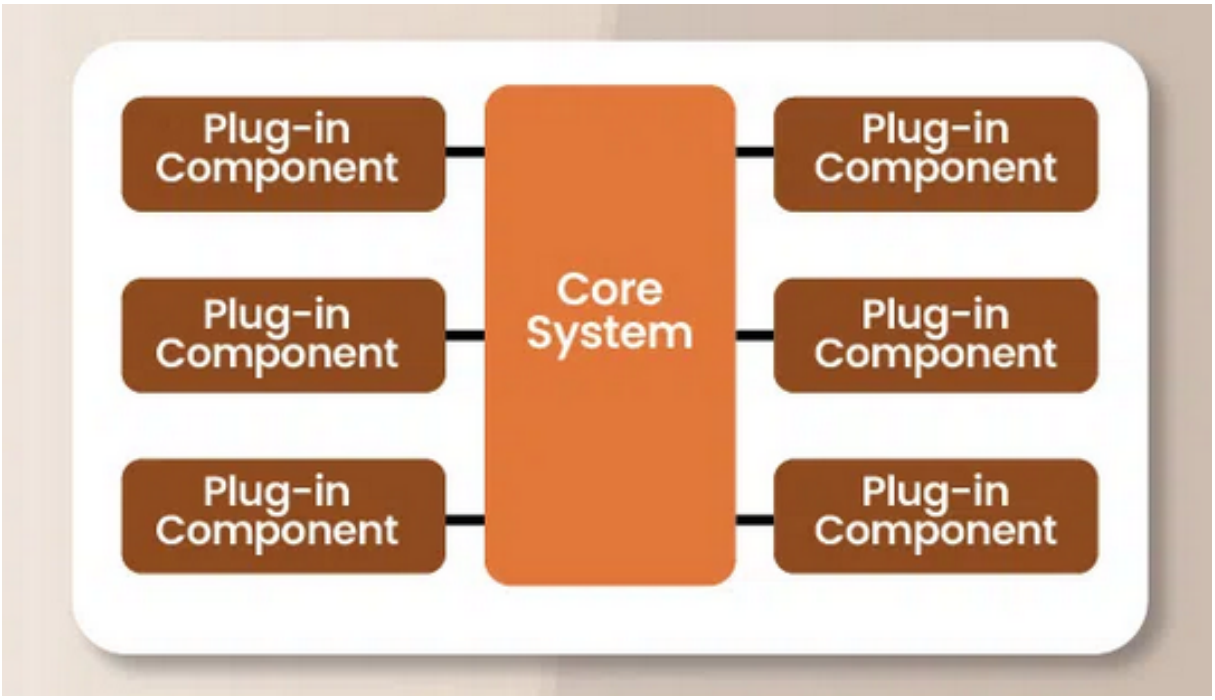


Figure 1: MicroKernel Architecture Diagram

The advantages this brings are numerous:

- The core code being largely static encourages **testability** and **robustness** as a set of comprehensive tests to be developed and maintained.
- Removing the plugins from the core developer responsibilities improves **maintainability** as only the core code must be maintained.
- The plugin system allows for development **flexibility**, allowing new features without changing the core code. Opening the code to extensibility without the new features encroaching on the core simulation.

3.3. Reasoning

This re-architecture is feasible due to the big shutdown happening to Merlin. As just fixing the code is a monumental task, seen here Figure 9. To take full advantage of the shutdown I propose to rewrite the codebase in Rust.

Adopting a microkernel architecture would provide a stable testable core, while rust would optimise the performance and robustness of the simulation, integral to high precision simulations. These changes would improve the long term maintainability of the code, as well as preventing unsafe code from entering the codebase.

Additionally, Rust inherently supports the functional programming paradigm, which I believe is well placed in the plugin-based architecture. If plugins are enforced to be functional, they will act as stateless modules interacting through interfaces, reducing the risk of architectural drift as a plugin with state could become a dependency (CITE HERE). Merlin can then maintain clear separation between the stable simulation kernel and user-developed extensions, ensuring long-term scalability and maintainability while allowing feature flexibility.

3.4. Conclusion

This architecture proposal aligns with the ideal goals Merlin had laid out (CITE HERE). Allow new developers to merlin focus on new features to develop over jumping around spaghetti code trying to understand issues. Shifting the focus from maintenance and compiling being the standard to a robust performant system being at the core of Merlin.

3.5. Diagrams

3.5.1. Tech Stack

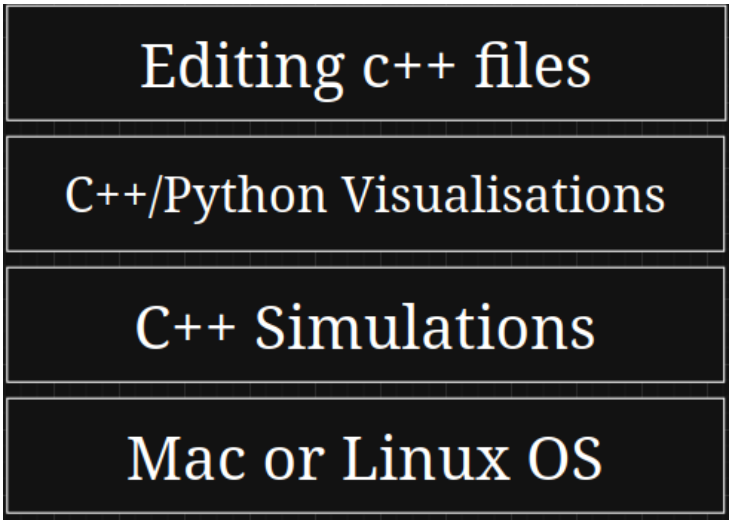
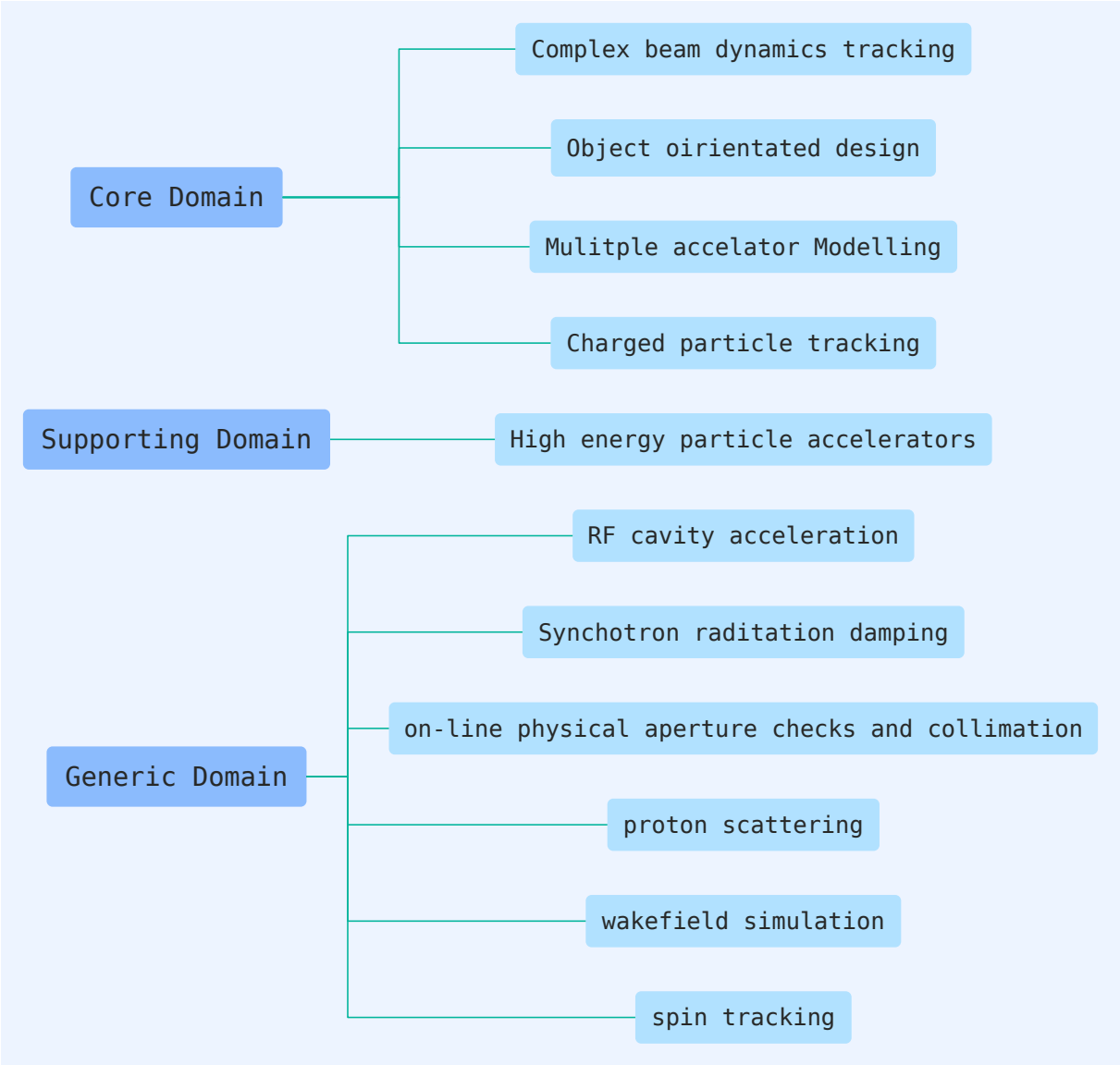


Figure 2: Merlin Tech Stack

3.5.2. Domain Model



Listing 1: Merlin Domain Model

3.5.3. Utility Tree

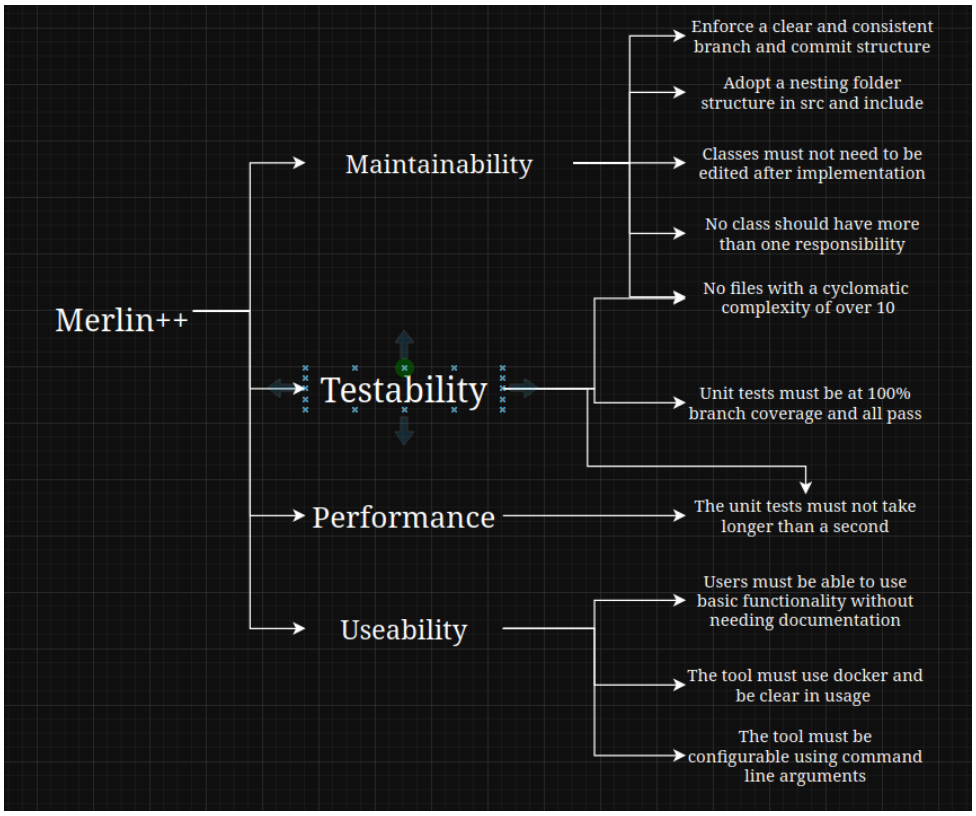


Figure 3: Merlin Utility Tree

3.5.4. Use Case Diagram

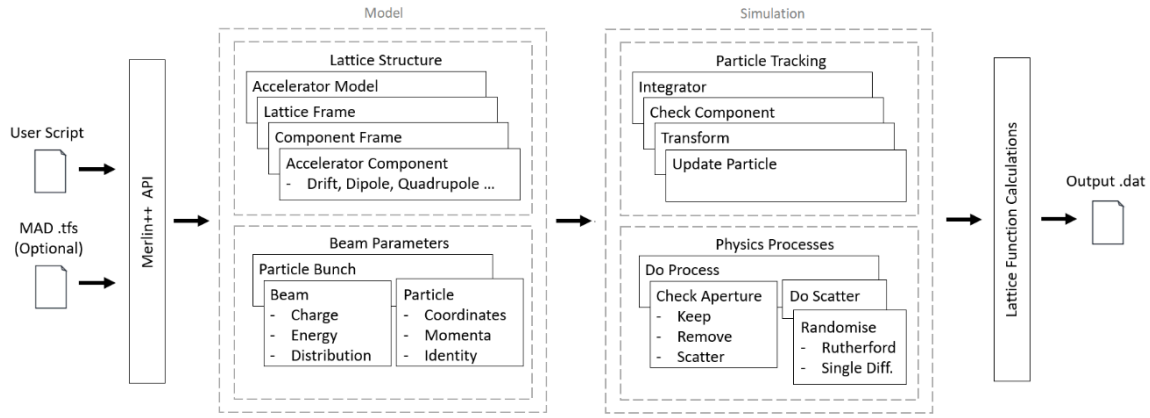
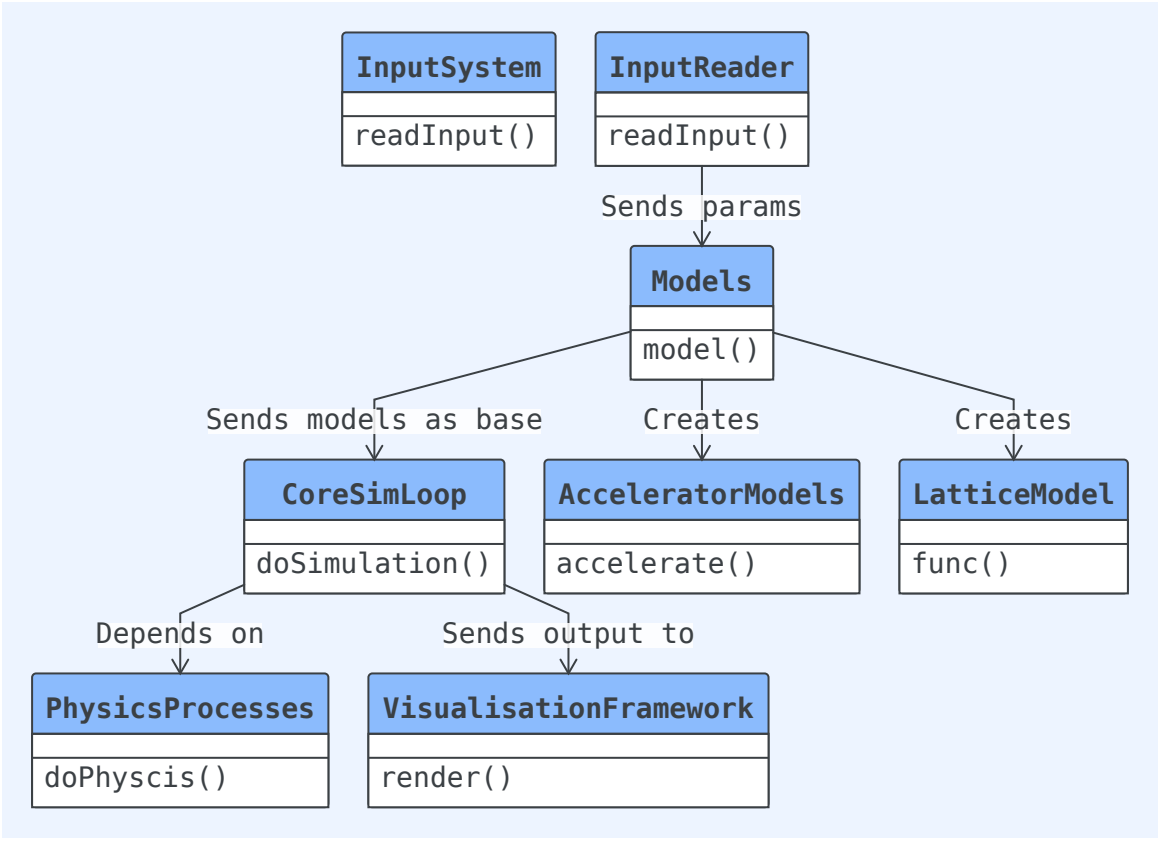


Figure 4: Merlin Use Case diagram

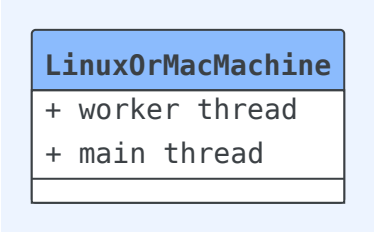
3.5.5. 4 + 1 Diagram

3.5.6. Logical View



Listing 2: Merlin Logical View

3.5.7. Physical View

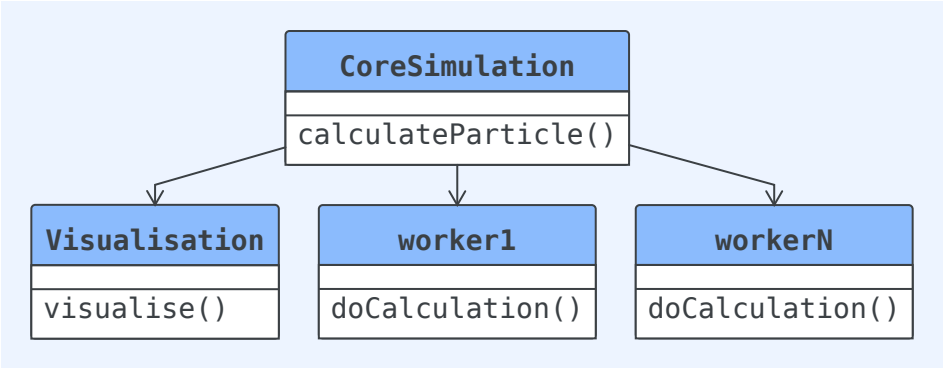


Listing 3: Merlin Physical View

3.5.8. Development View

- The structure of the code is flat there is no file or folder organisation
- There are larger multiclass components such as Particle tracking, Physics processes and lattice calcualtions but these are only grouped in code not in file structure
- There are no git rules for commits

3.5.9. Process View



Listing 4: Merlin Process view

Each node represents a thread in the process view where the Core simulation can spin up worker threads for calculations in a HPC environment

3.5.10. Codescene

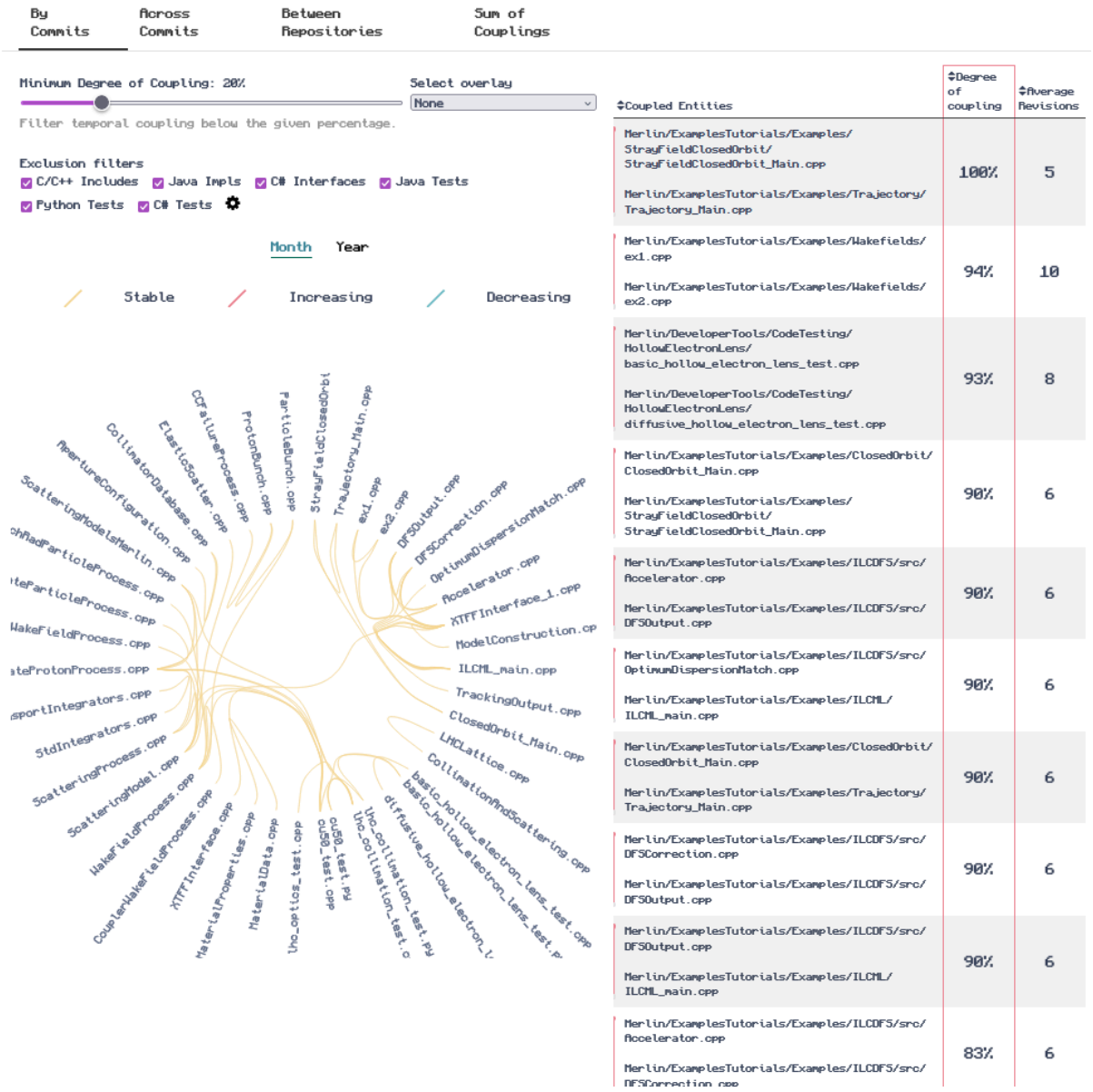


Figure 5: Merlin Codescene coupling graph

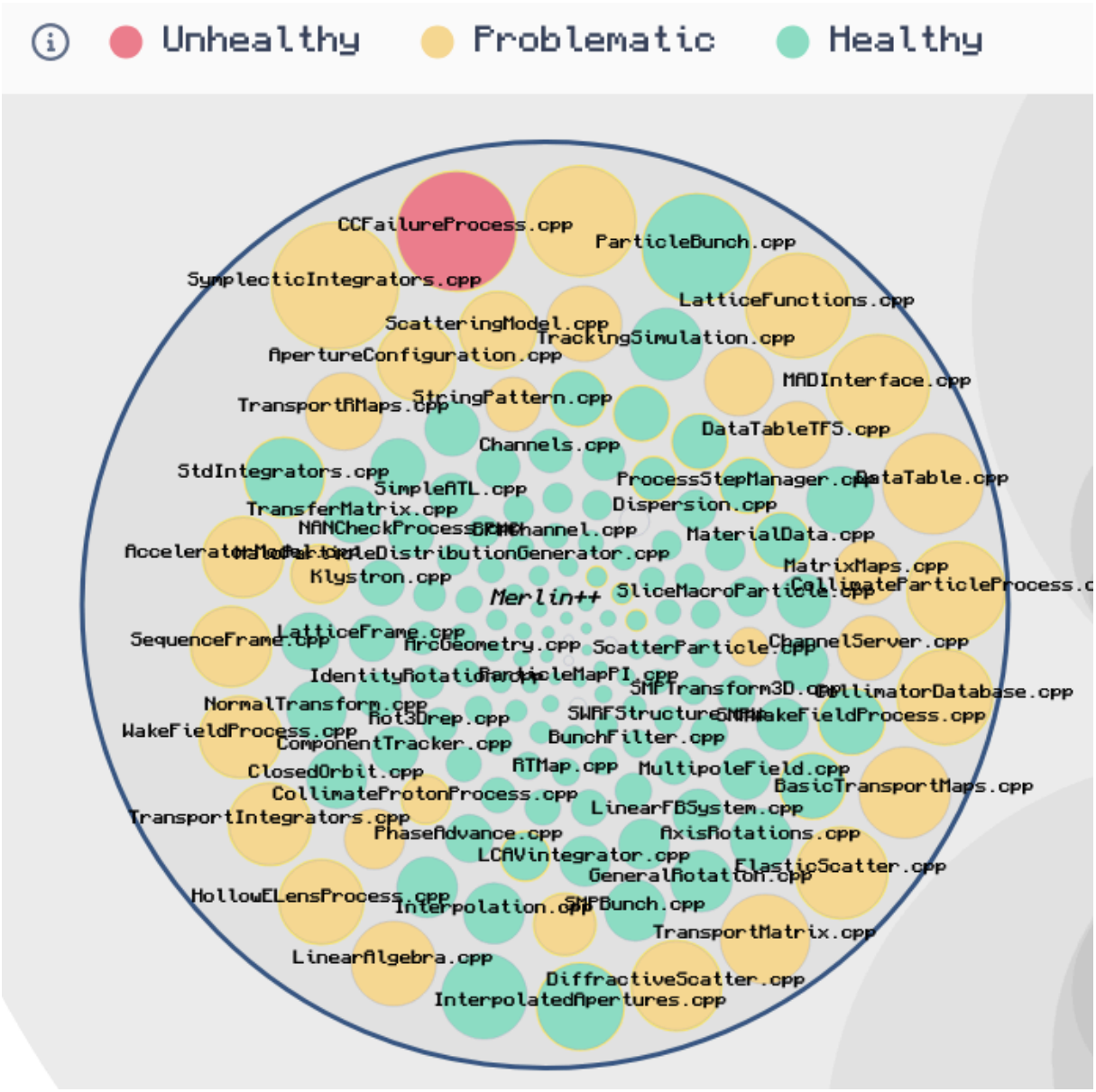


Figure 6: Merlin Codescene maintainability circle

3.5.11. DV8

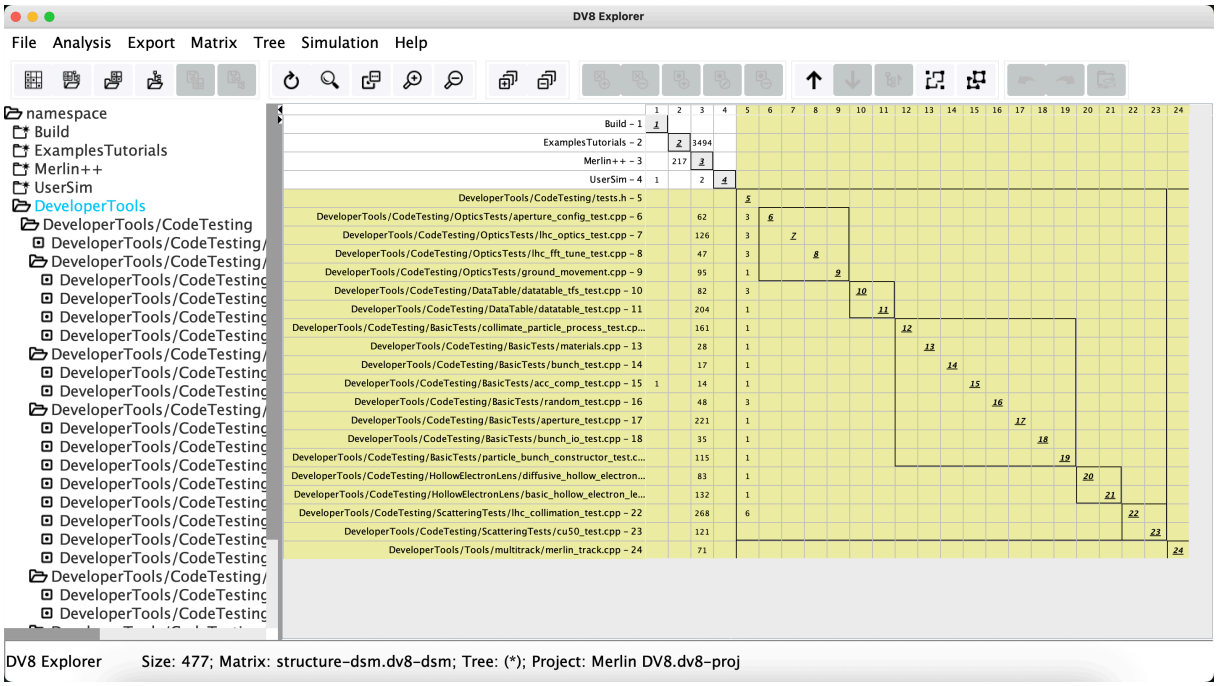


Figure 7: Merlin Dv8 Dev tool analysis

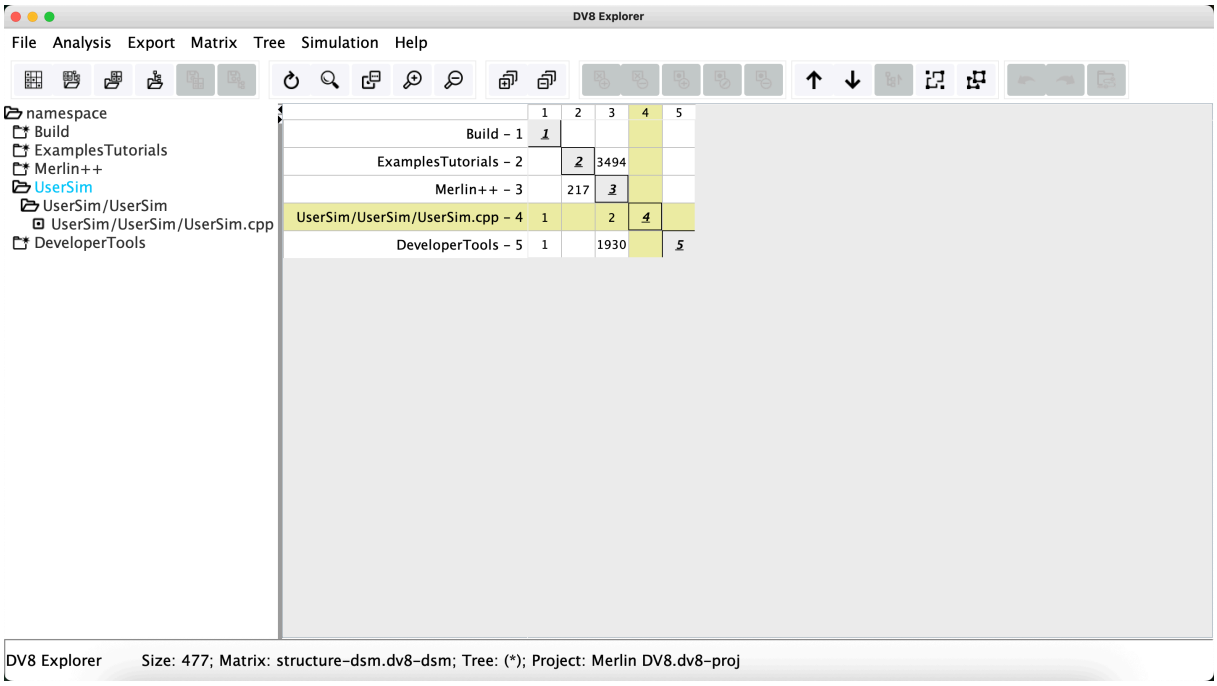


Figure 8: Merlin Dv8 high level analysis

3.5.12. SonarQube

Figure 9: Merlin Technical debt cost

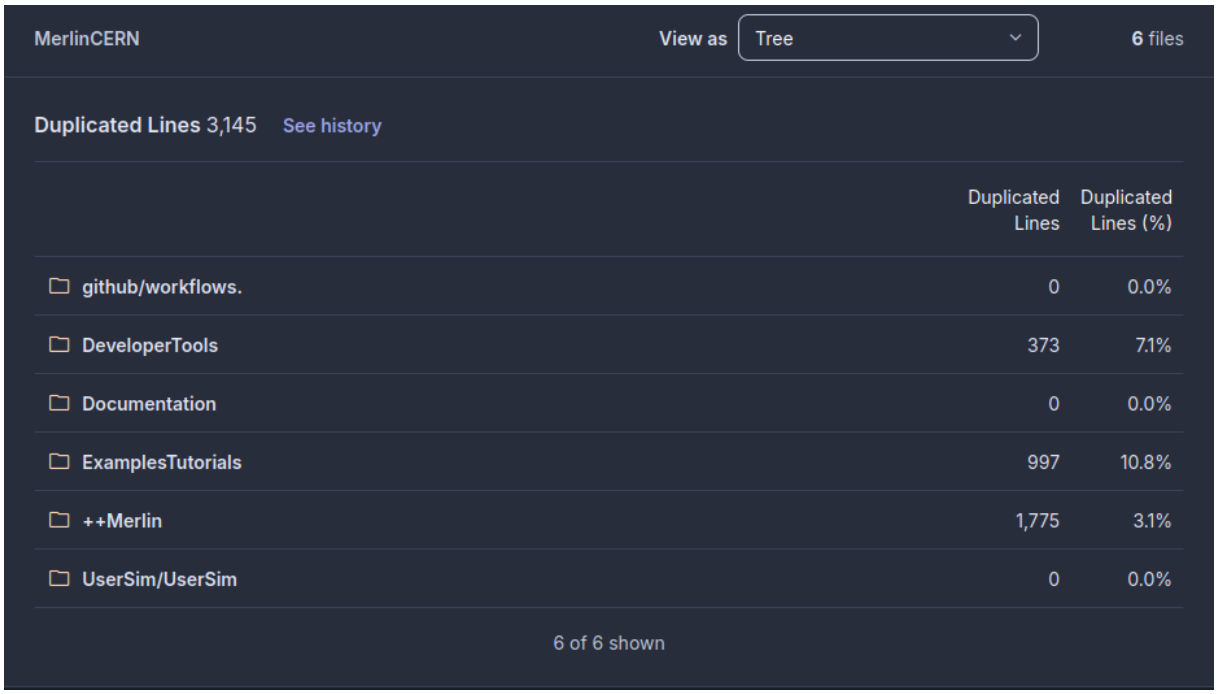


Figure 10: Merlin Duplicated code analysis

3.5.13. Understand

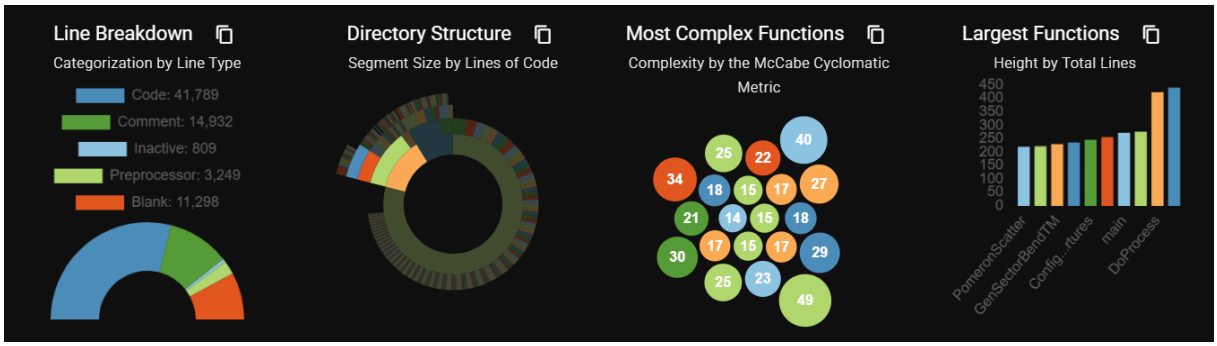


Figure 11: Merlin Function complexity understand

Figure 12: Merlin Coding langauge breakdown

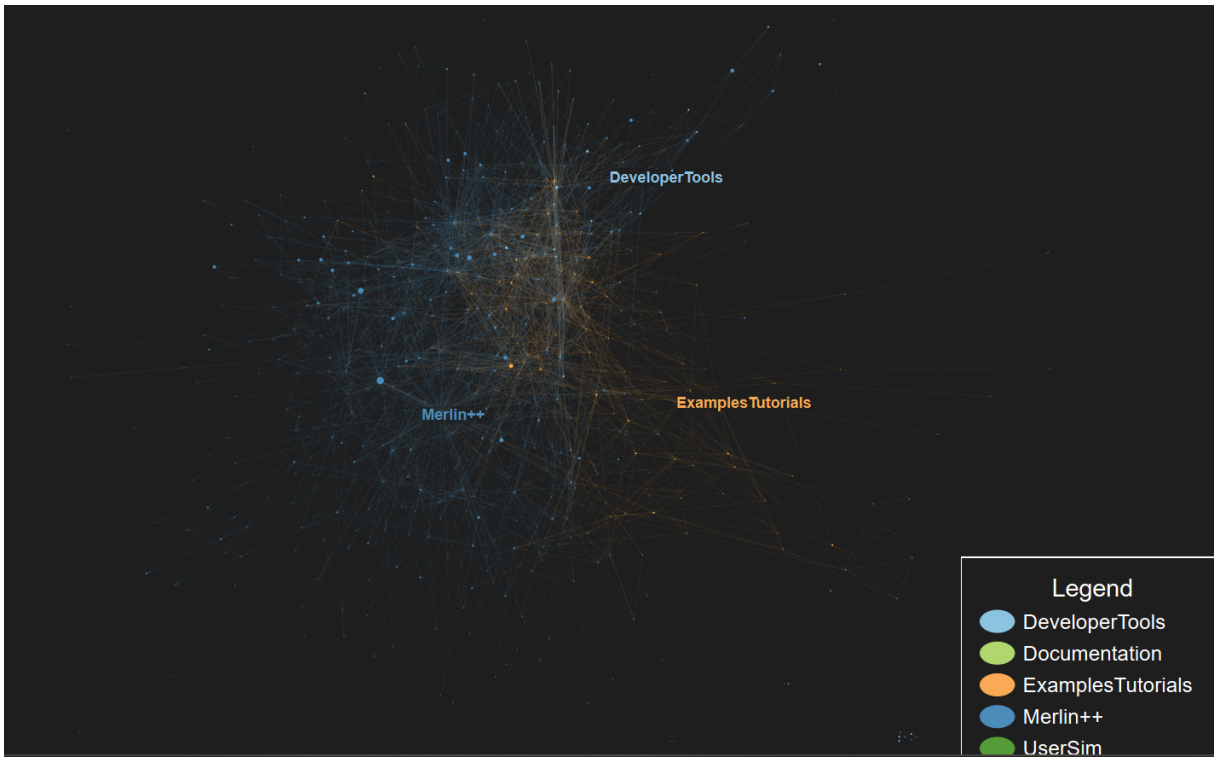


Figure 13: Merlin Understand code dependency graph

3.5.14. C4

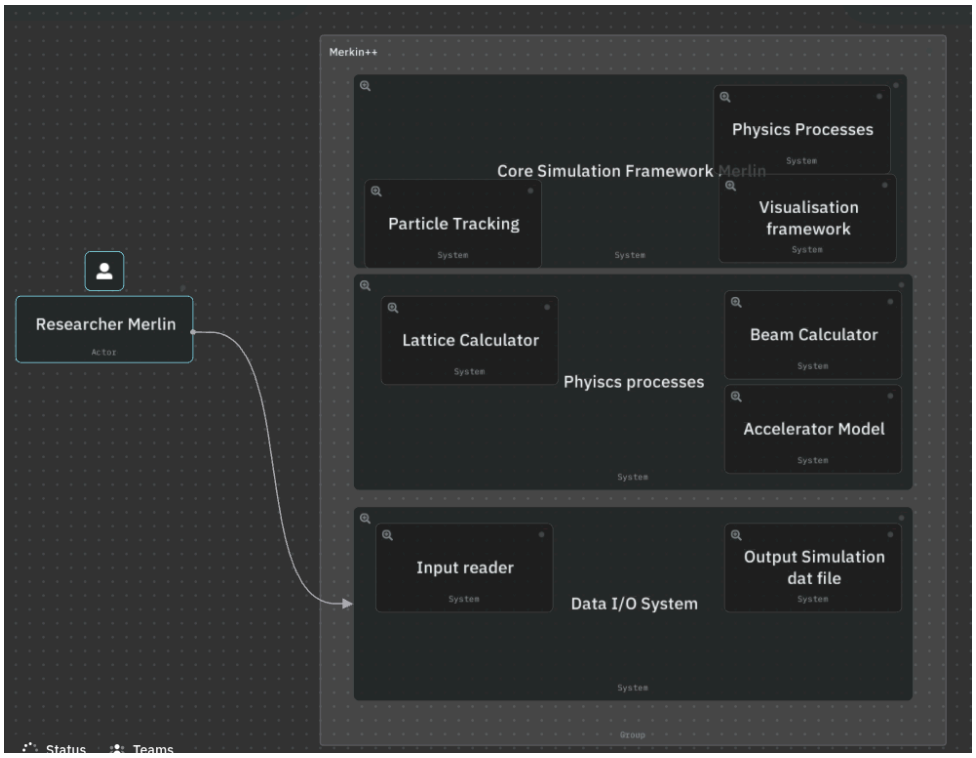


Figure 14: Merlin C4 Diagram

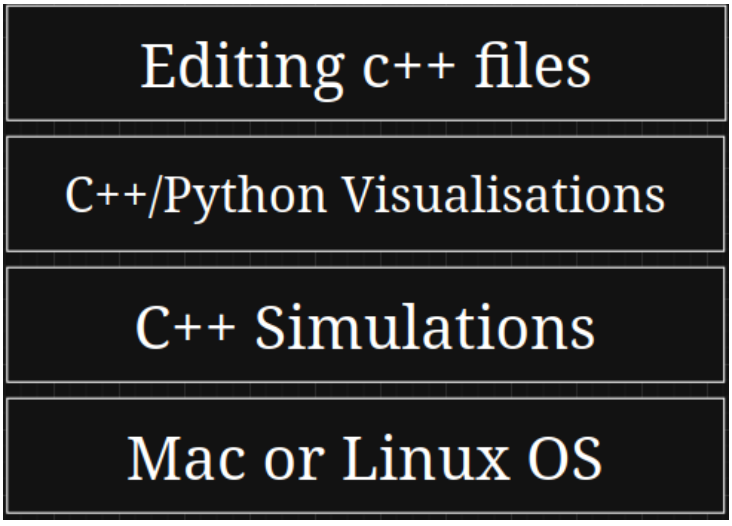


Figure 15: Tech Stack

3.6. Fluctuating Finite Element Analysis (FFEA) Case Study

3.7. Tech Stack

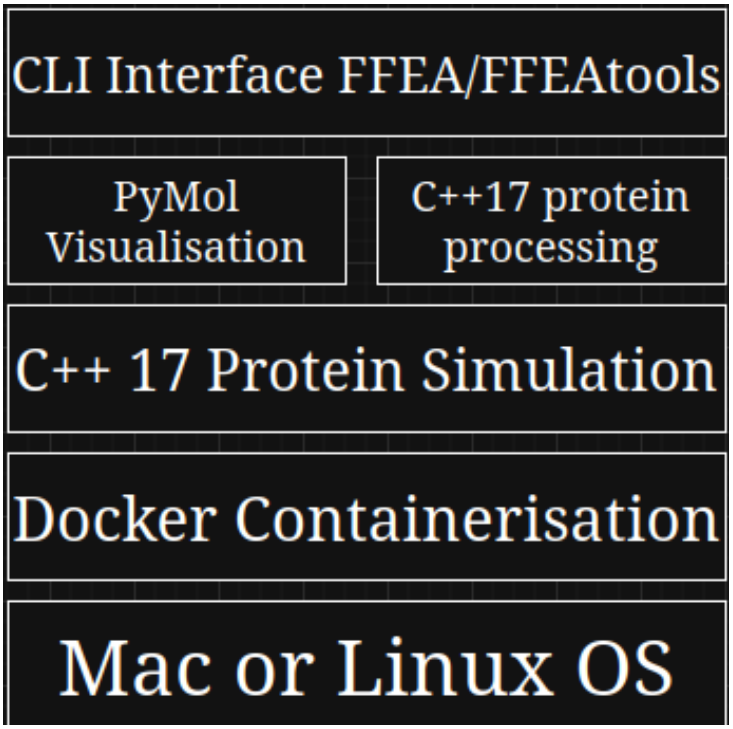
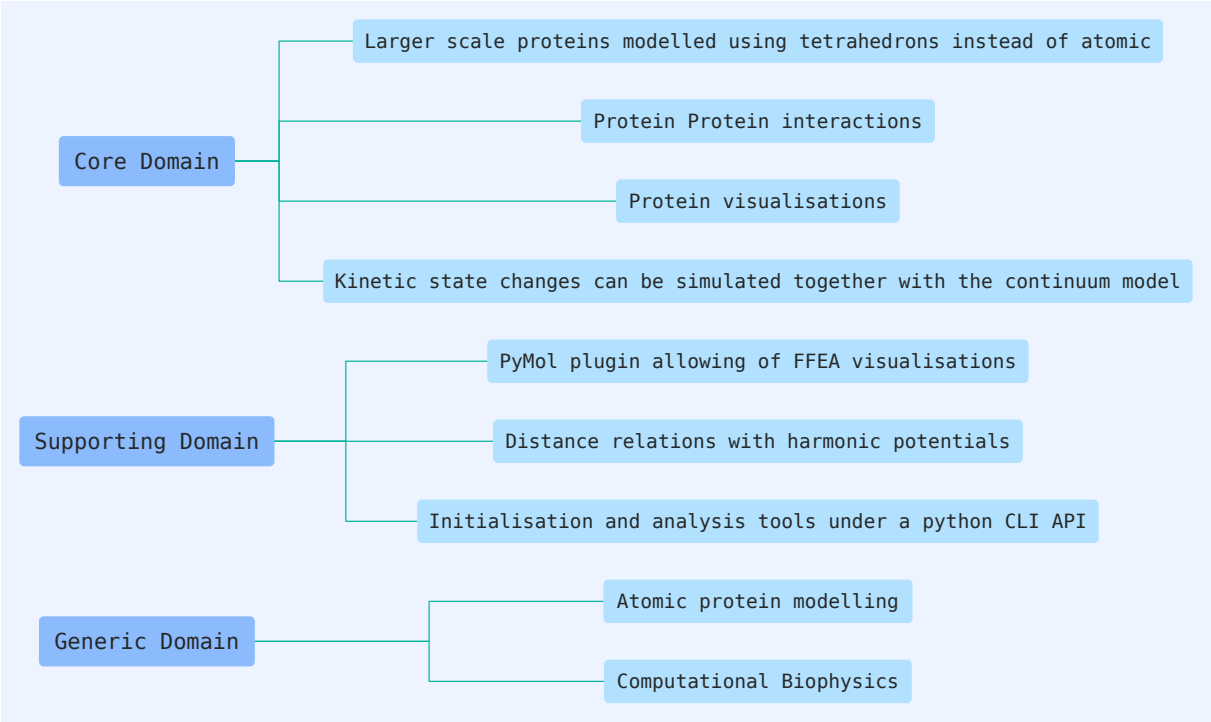


Figure 16: Tech Stack

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 follwing the qualities required for the project.



Listing 5: Domain Model

3.8. Use Case Diagram

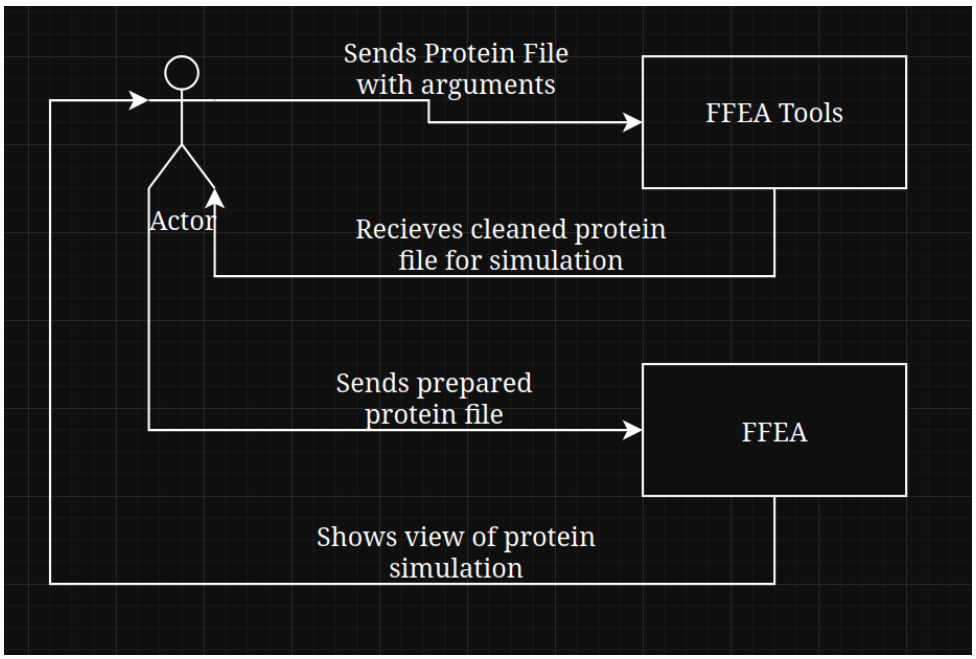


Figure 17: Use Case Diagrams

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.

3.9. Utility Tree

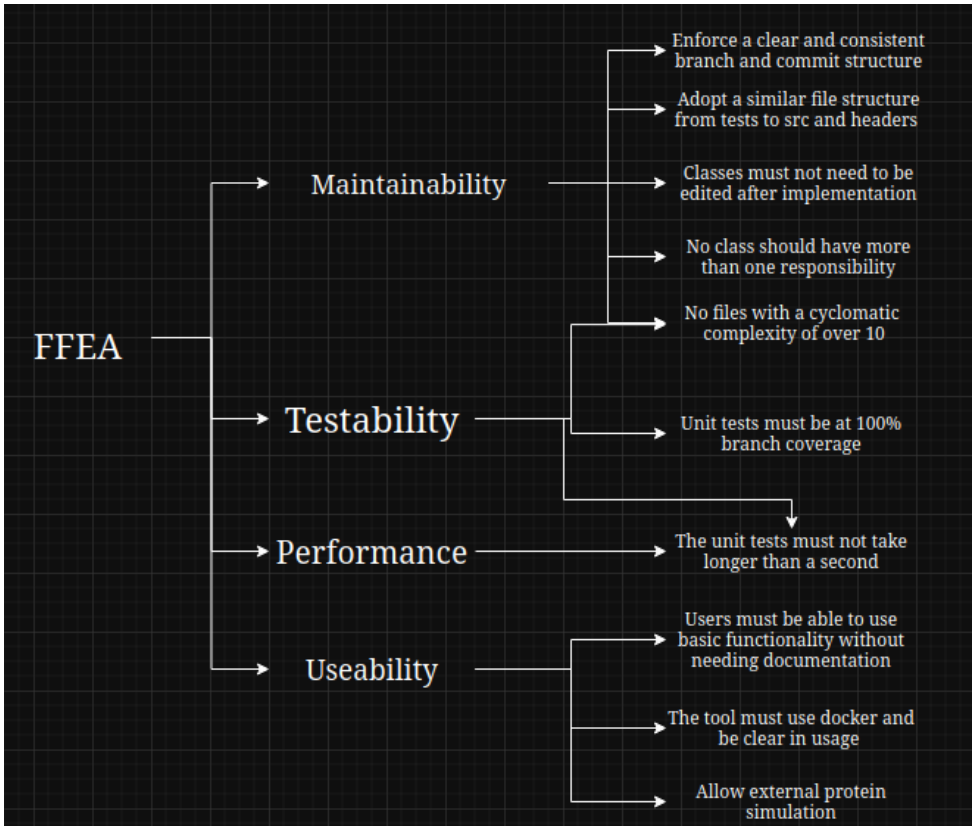


Figure 18: Utility Tree

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 of ilities 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 resilve 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.

3.10. SysML Diagram

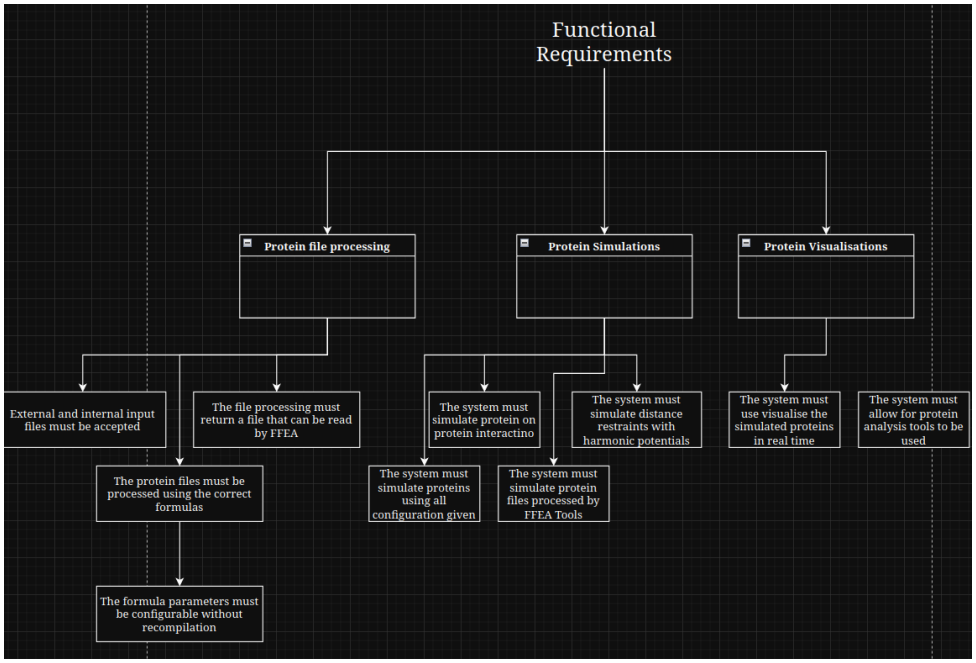


Figure 19: SysML Diagram

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 visualing. This SysML structure allows for room for growth as more requirements are discovered.

3.11. Codesense Diagram

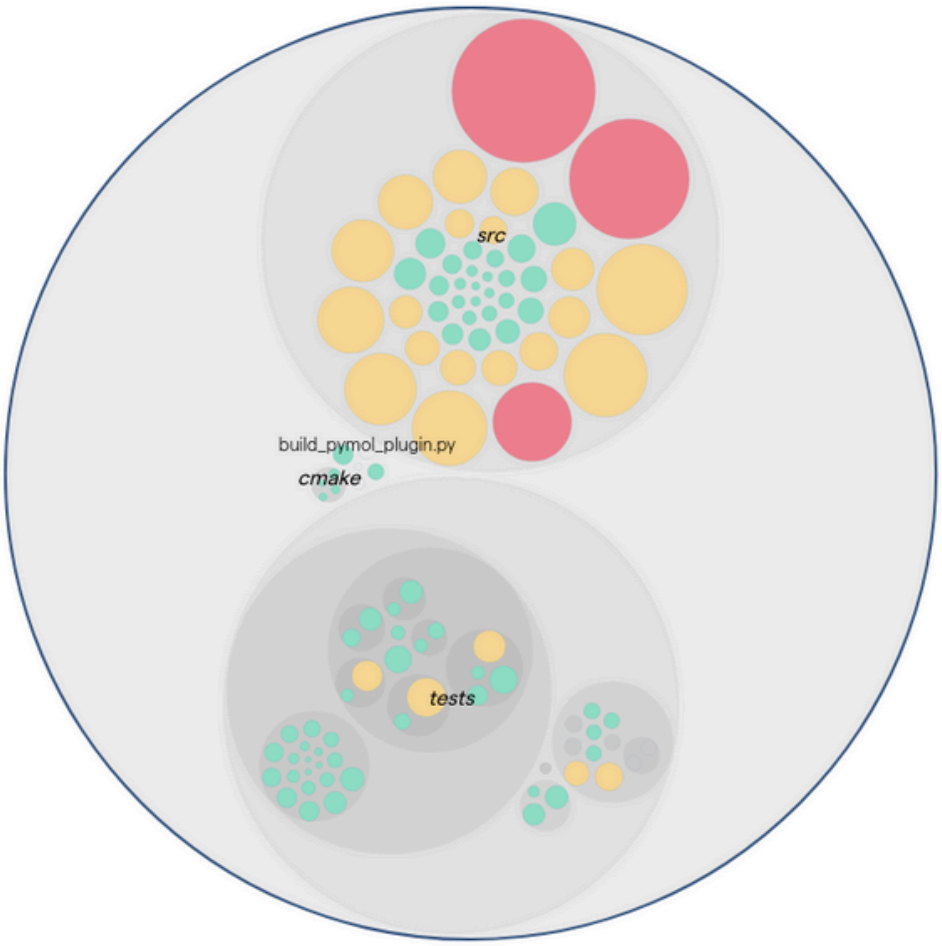


Figure 20: Codesense Architecture Diagram

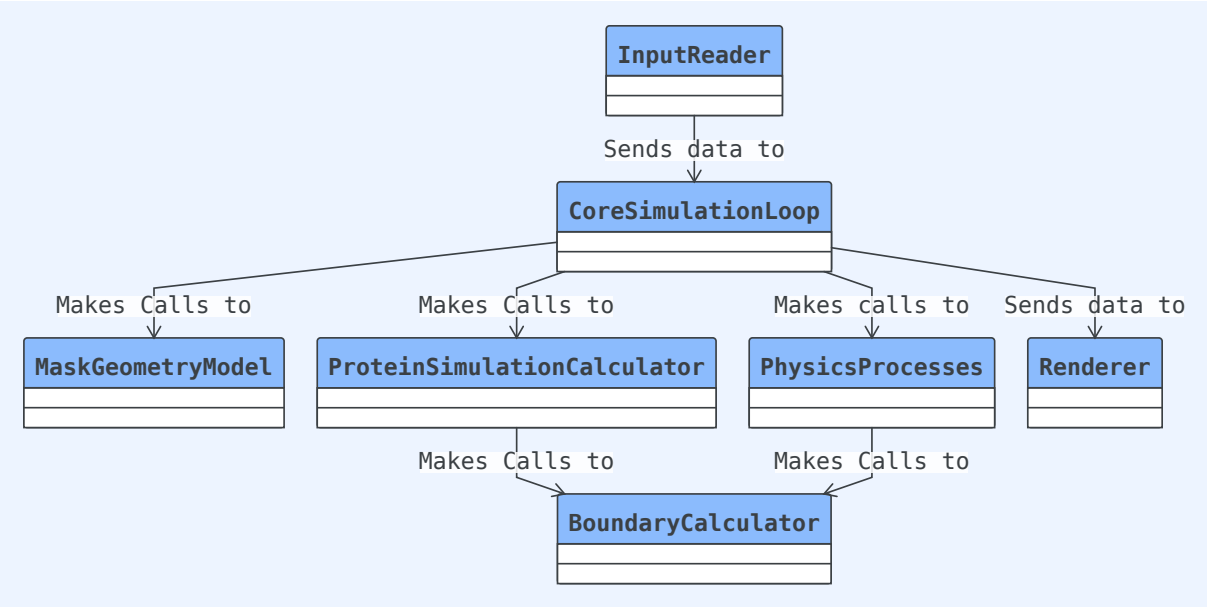
This digram highlights three key things about the architecture decisions taken for this project. The first feature being the flat structure taken for src, this means that all files are the same level and there is no logical grouping for any of the src files which makes grouping much more difficult. The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

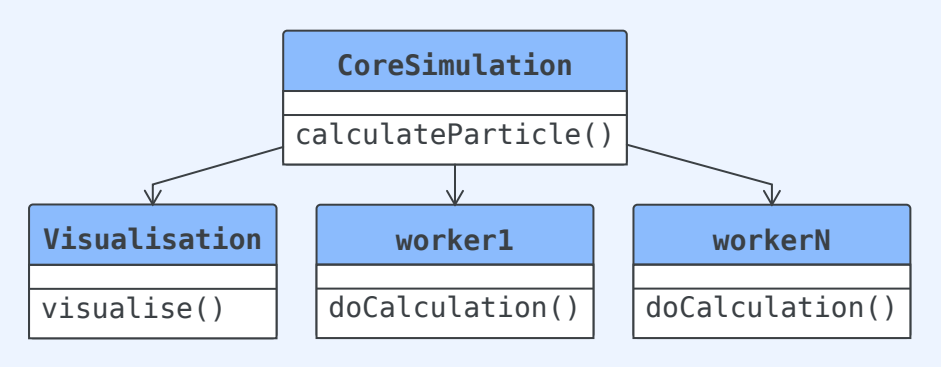
Lastly looking at the colours of the larger files three files are exceptionally difficult to modify and maintain, again a major hit to maintainability

3.12. 4+1 Diagram

3.12.1. Logical View



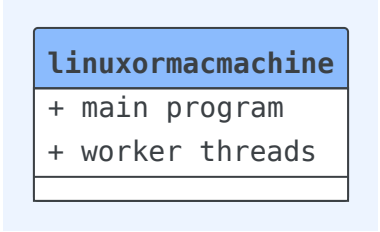
3.12.2. Process View



3.12.3. Developmental View

- The file structure is flat
- There is a high level of redundant code
- There is a god class called world
- The tests dont pass
- There are two main simulation sections being rods and blobs
- There is a logical separation between rendering and simulation due to pyton vs c++ being used

3.12.4. Physical View



3.13. SonarQube Diagram

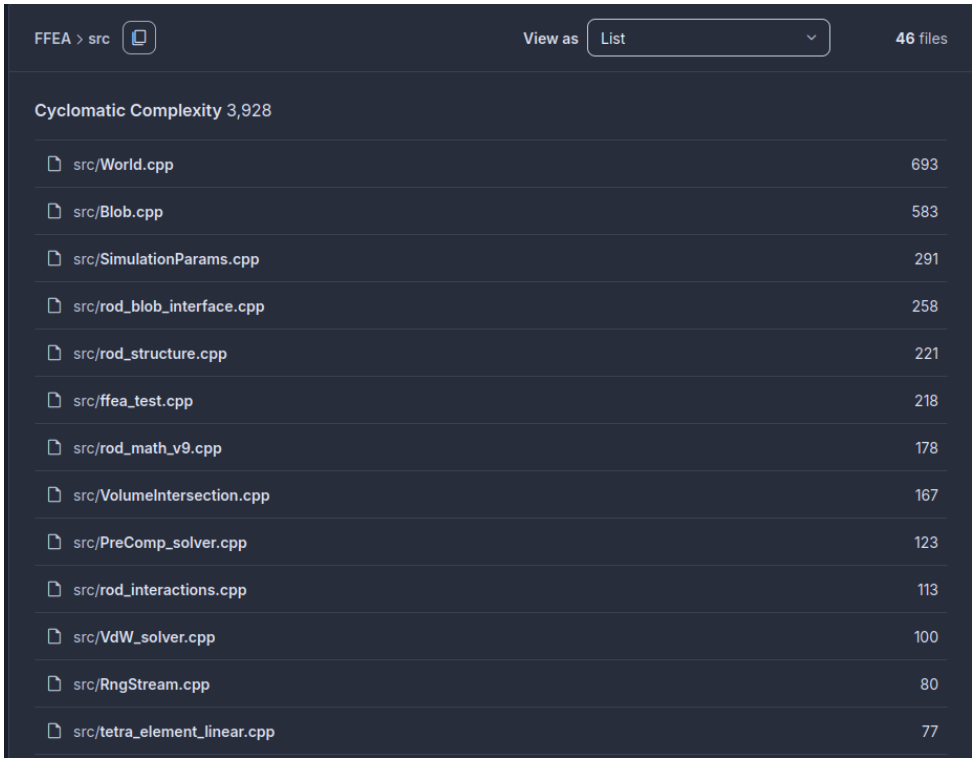


Figure 21: SonarQube Complexity Diagram

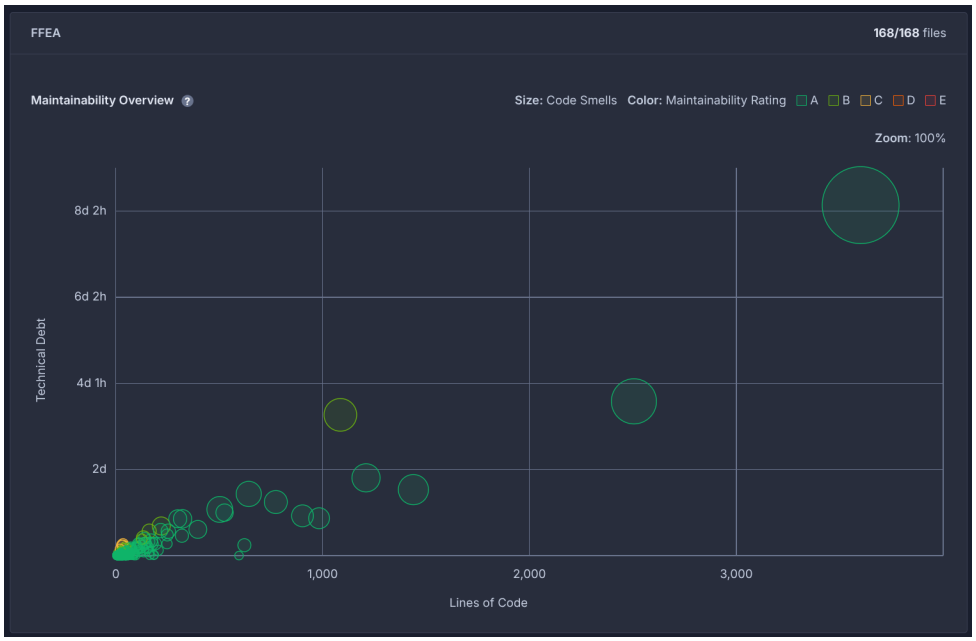


Figure 22: Understand Function Analysis

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

3.14. DV8

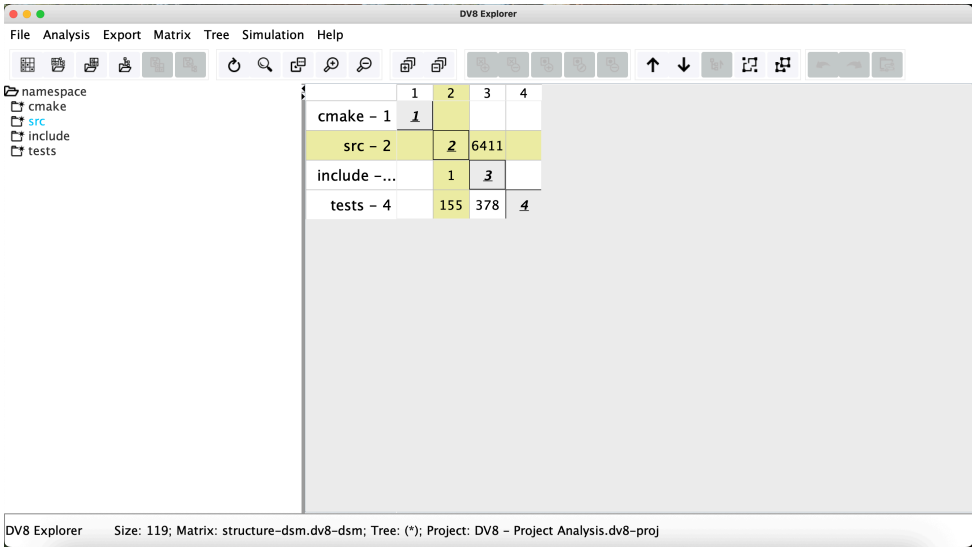


Figure 23: Understand Function Analysis

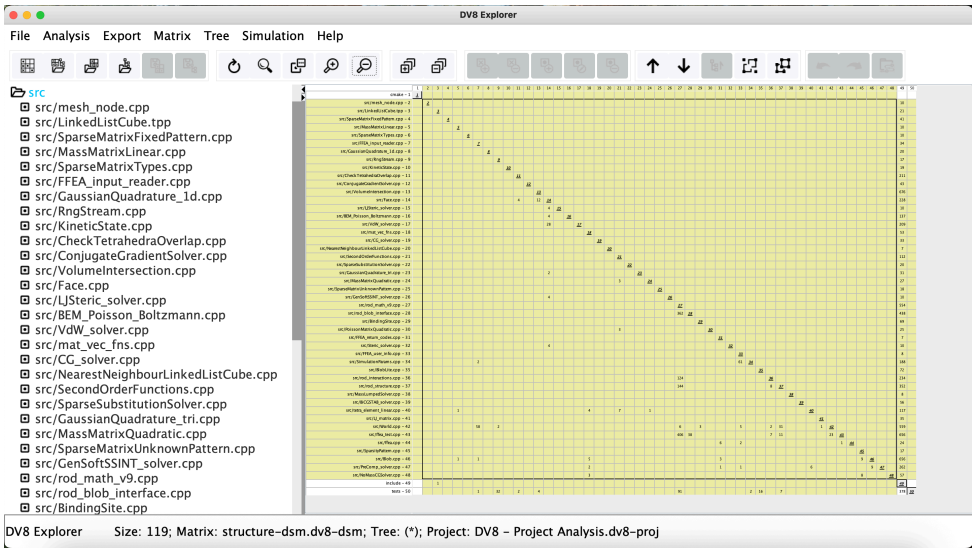


Figure 24: Understand Function Analysis

3.15. Understand

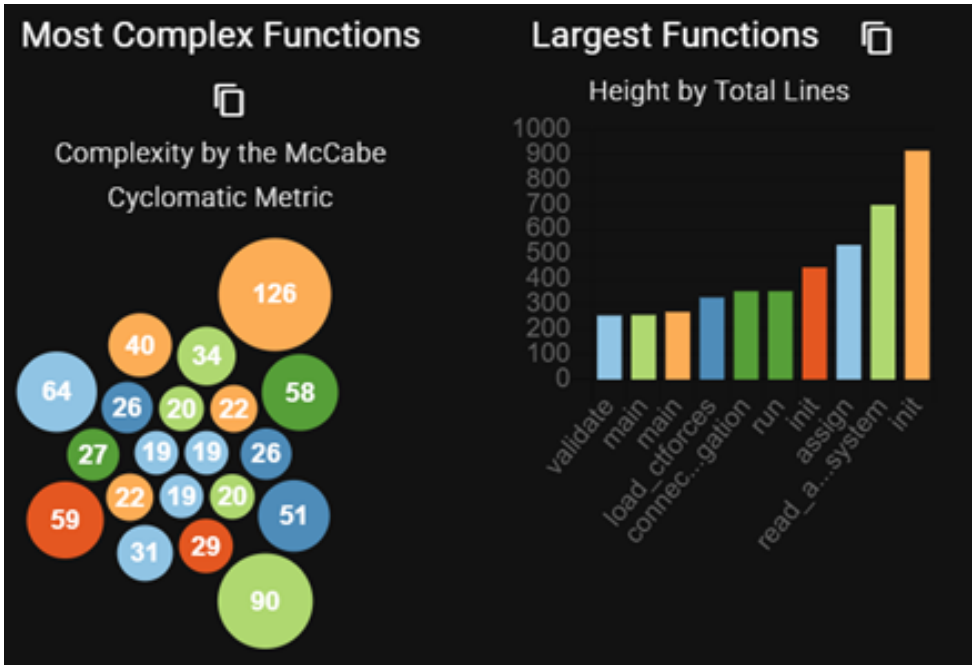


Figure 25: Understand Function Analysis

Figure 26: Understand Function Analysis

3.16. C4 Diagram

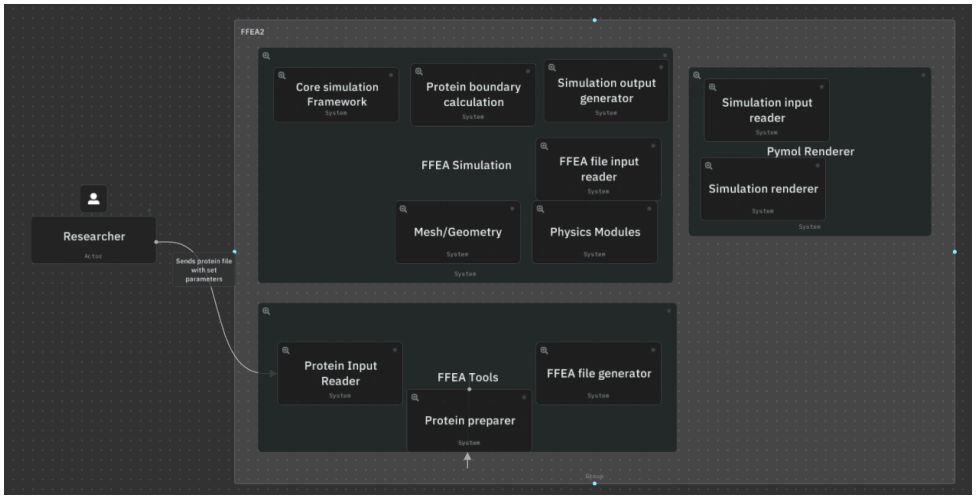


Figure 27: C4 Diagram

3.17. Dependency Graph

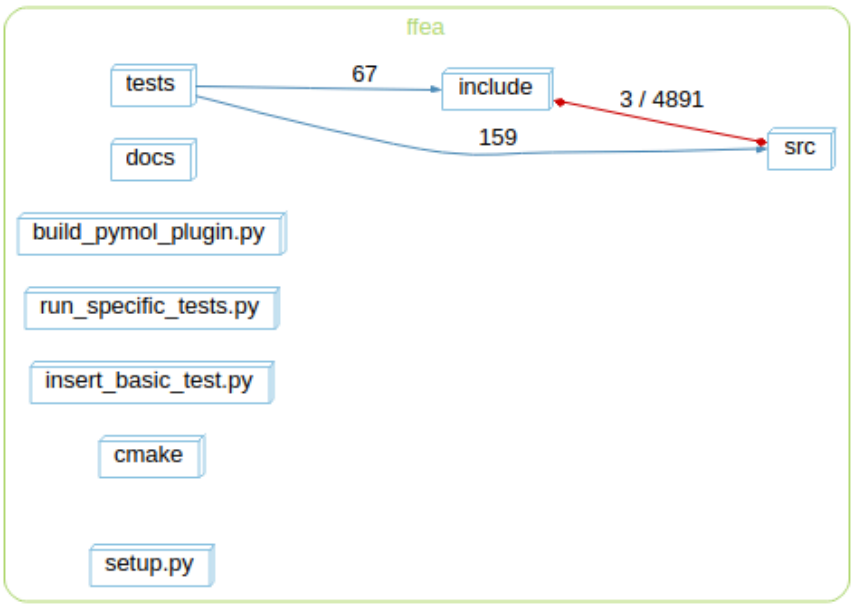


Figure 28: Dependency Graph

3.18. Class Diagram

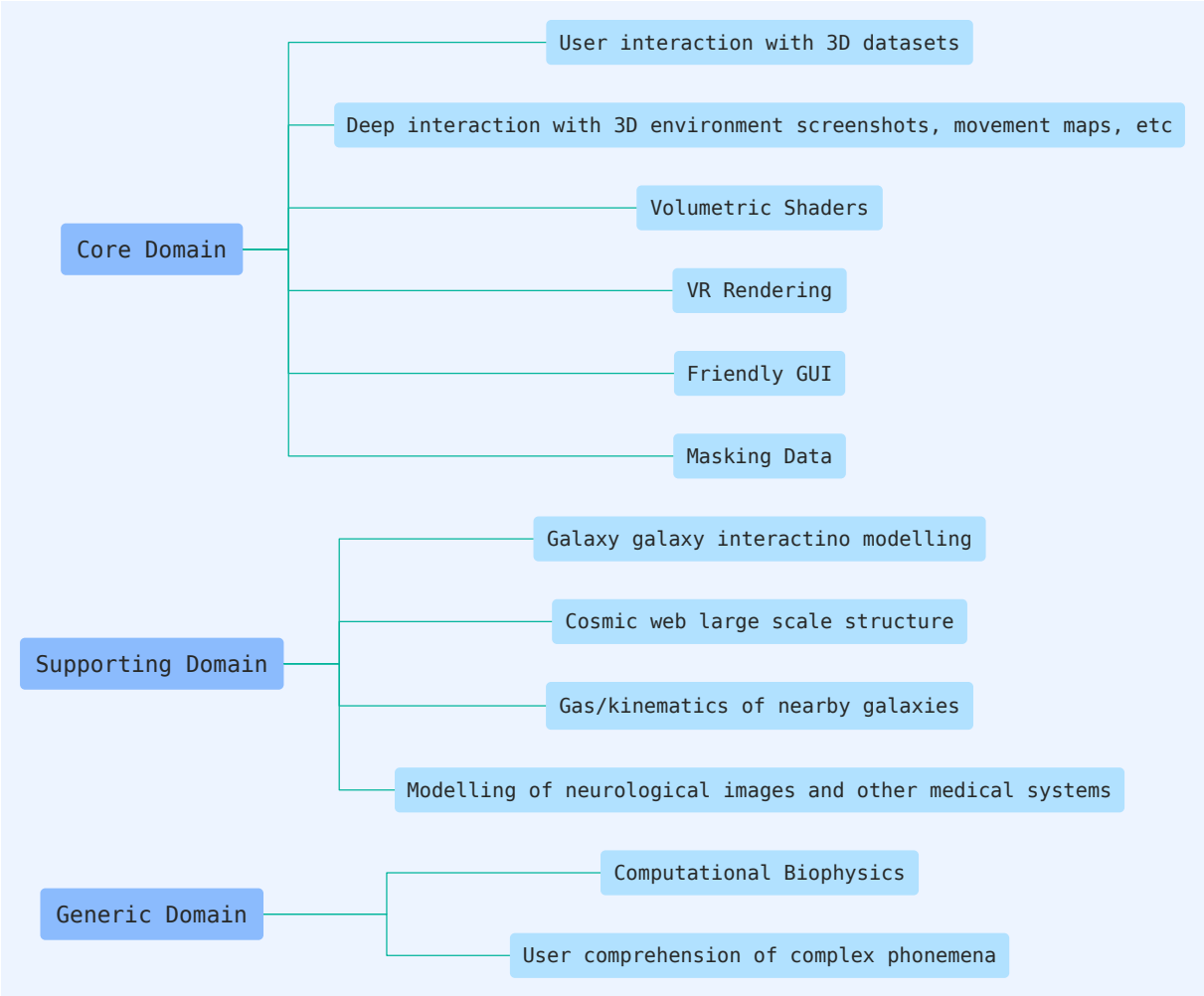
Figure 29: Class Diagram

4. iDavie Case Study

4.1. Tech Stack

- Renderer : Unity with C#
- Interaction Framework : Steam VR
- Simulation Framework : C++ with eigen and boost
- Tests : Ctest with python add ons

4.2. Domain Model



Listing 7: Domain Model

4.3. Utility Tree

- Maintainability
  - Use CI/CD to automatically run test instead of a google form
  - Structure the codebase into a distributed monolith
  - Adhere to SOLID principles
  - Comment the why not the how
  - An average cyclomatic complexity of 15
  - Test branch coverage must be over 90%
- Performance
  - Ensure smmoth usage on minimum specs
  - Unit testing must not take longer than a second
- Useability
  - Create a docker image to allow for repeatable deployments
  - Update the docuementation to allow for a user guide on how to run the code

Listing 8: Utility Tree

4.4. Use case diagram

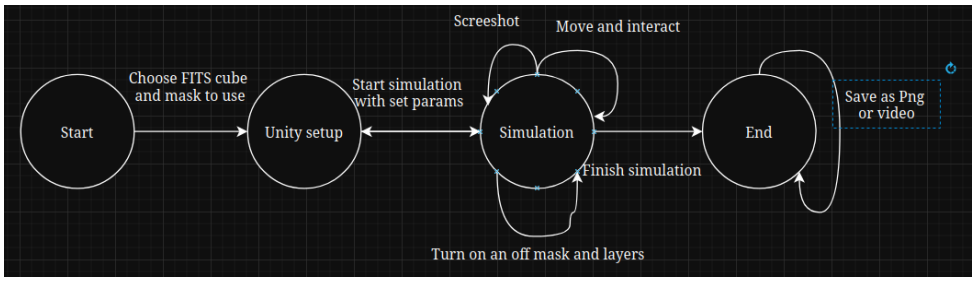


Figure 30: Use case state machine

4.5. 4 + 1 Diagram

4.5.1. Logical View

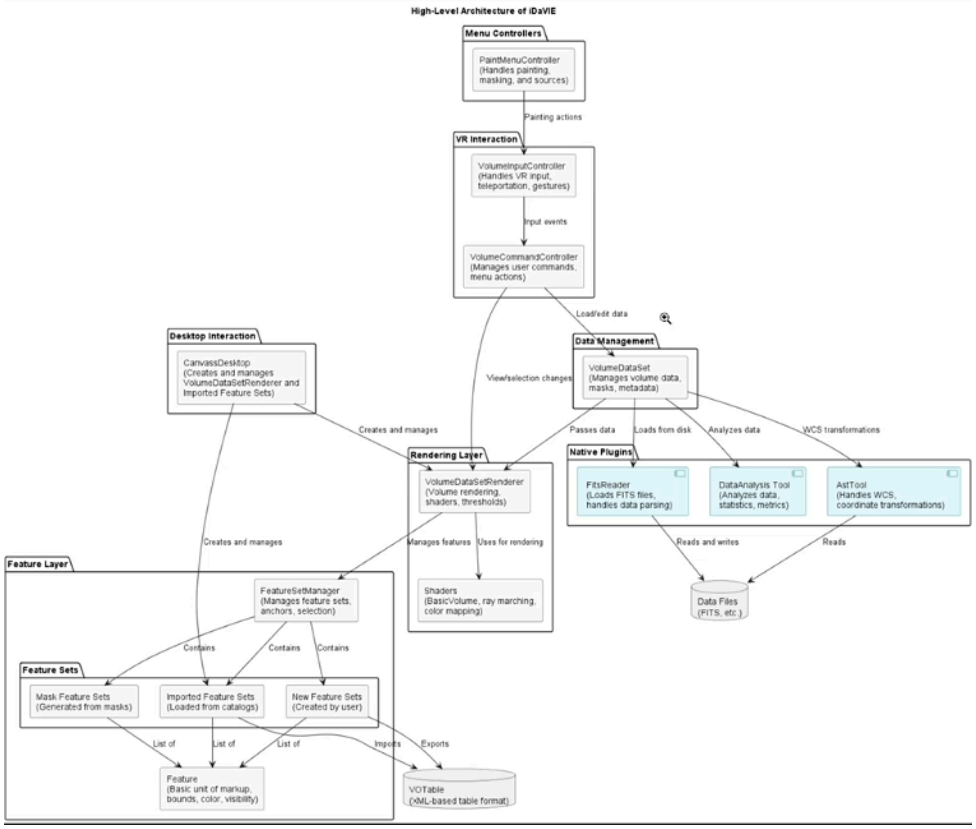
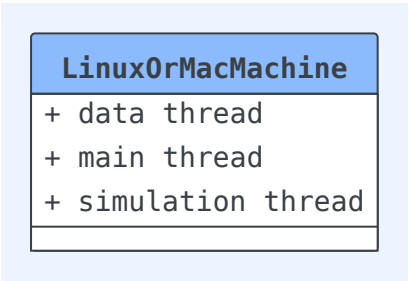
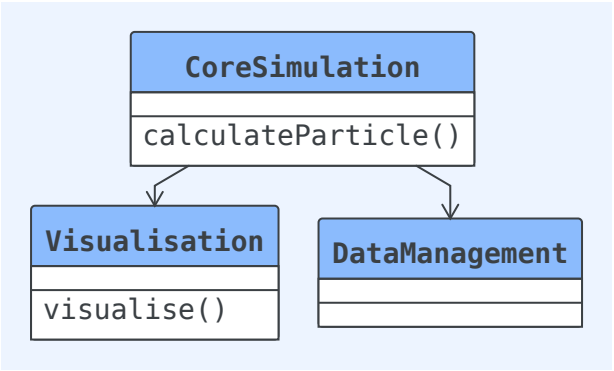


Figure 31: Logical View

4.5.2. Physical view



4.5.3. Process View



4.5.4. Developmental View

- Not extensible, forking is recommmended to add large features
- High change propagation
- Current core is fairly static
- Data management is the largest dependency and a core of the codebase
- There is logcial grouping of files
- No automated tests
- PRs need to go through manual testing via form for changes

4.6. Codescene

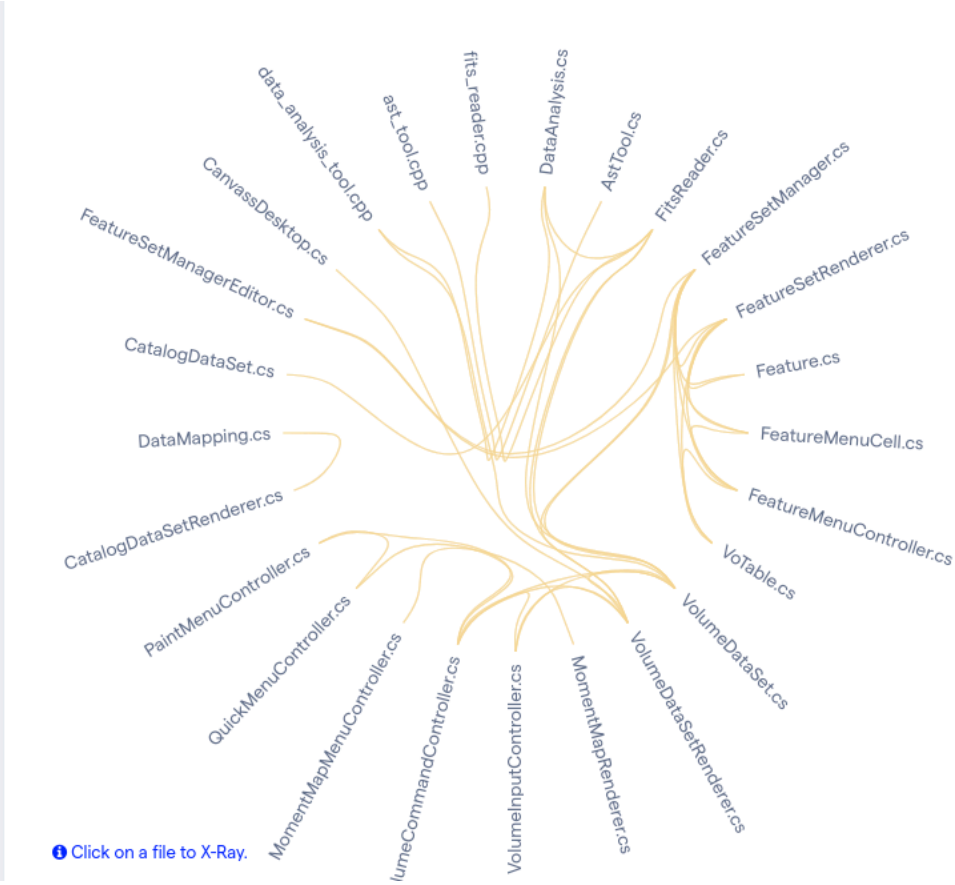


Figure 32: Codescene coupling diagram 40%

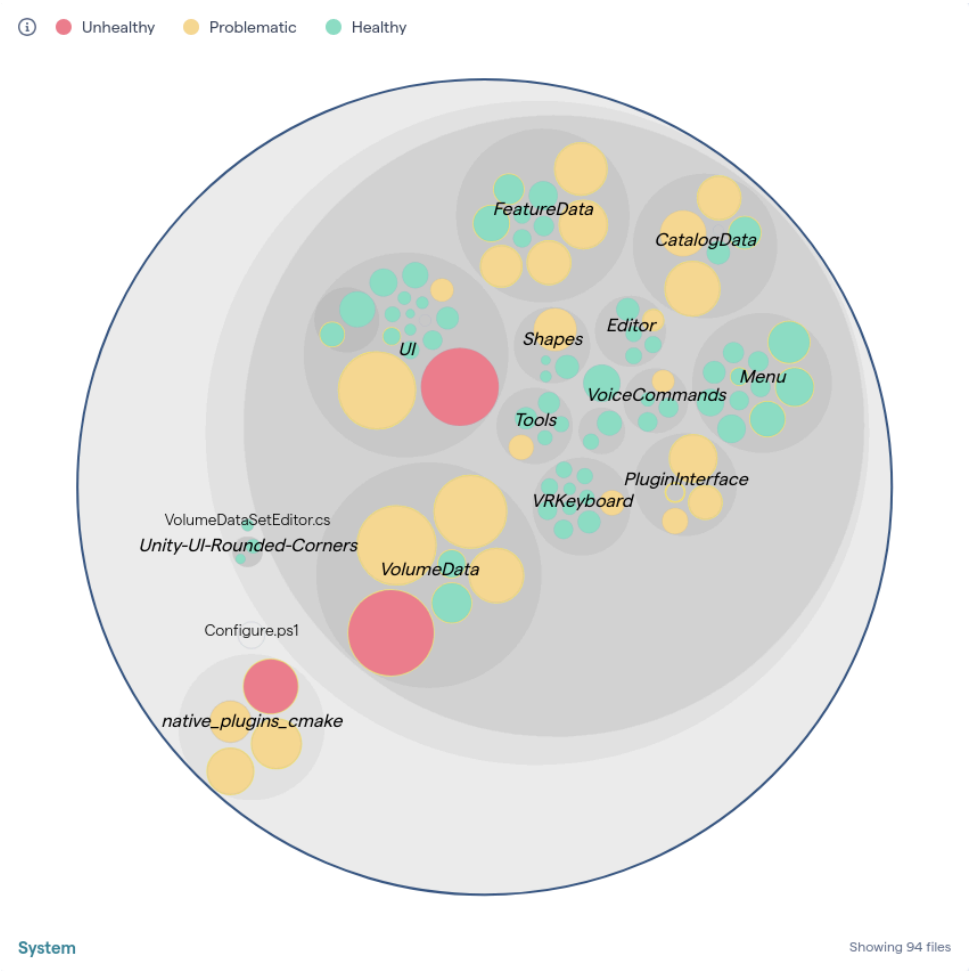


Figure 33: Codescene Maintenance Diagram

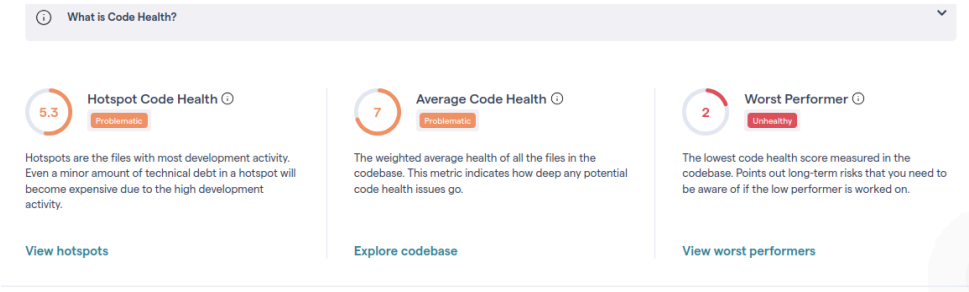


Figure 34: Codescene Hotspot overview

4.7. Sonarqube

Figure 35: Sonarqube cyclomatic complexity

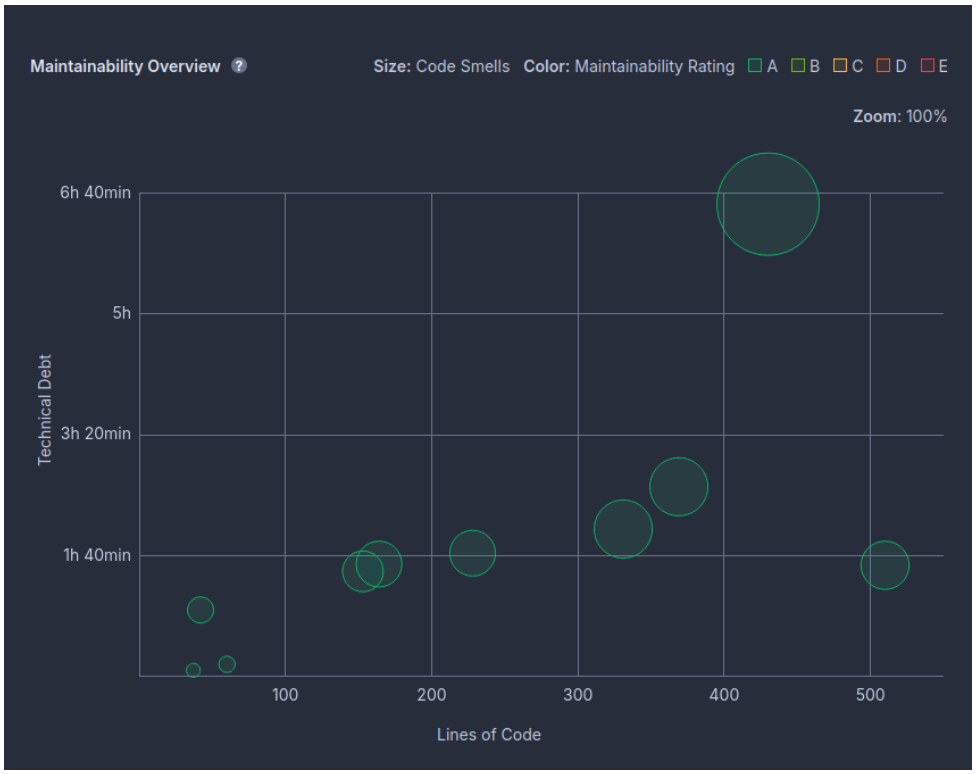


Figure 36: Sonarqube maintainability

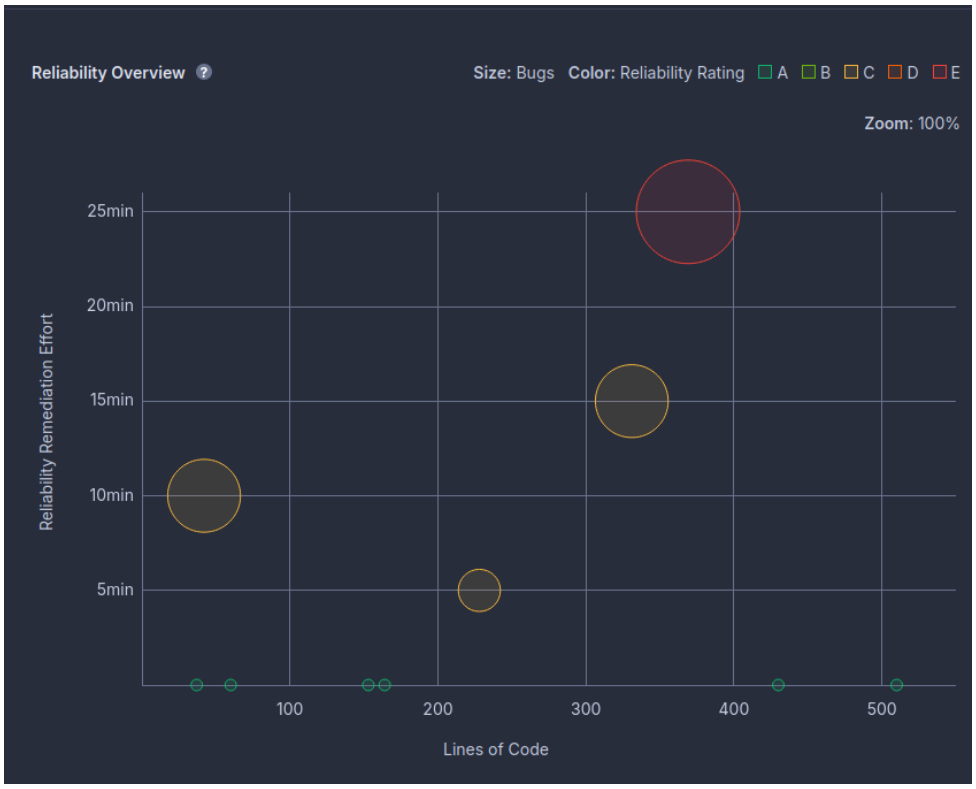


Figure 37: Sonarqube reliability graph

Figure 38: Duplications Graph

4.8. Understand Code Analysis

Figure 39:

4.9. Class Diagrams

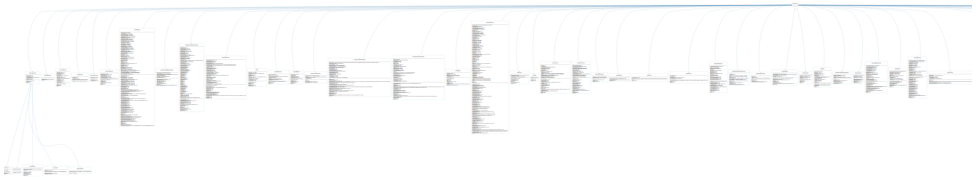


Figure 40:

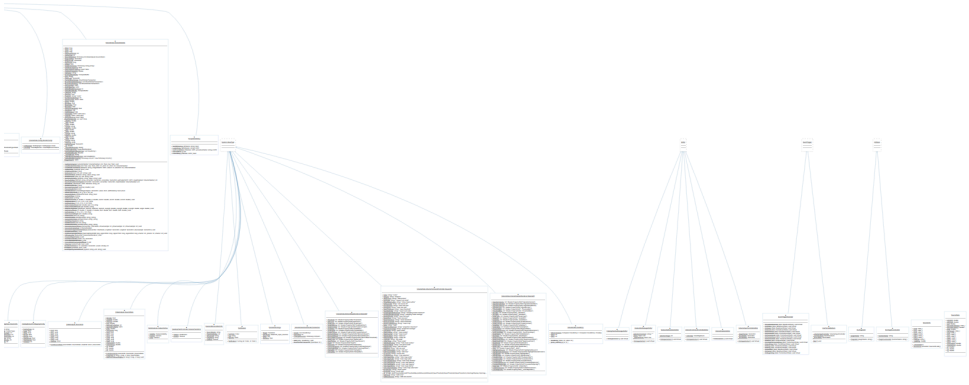


Figure 41:

4.10. C4 Diagram

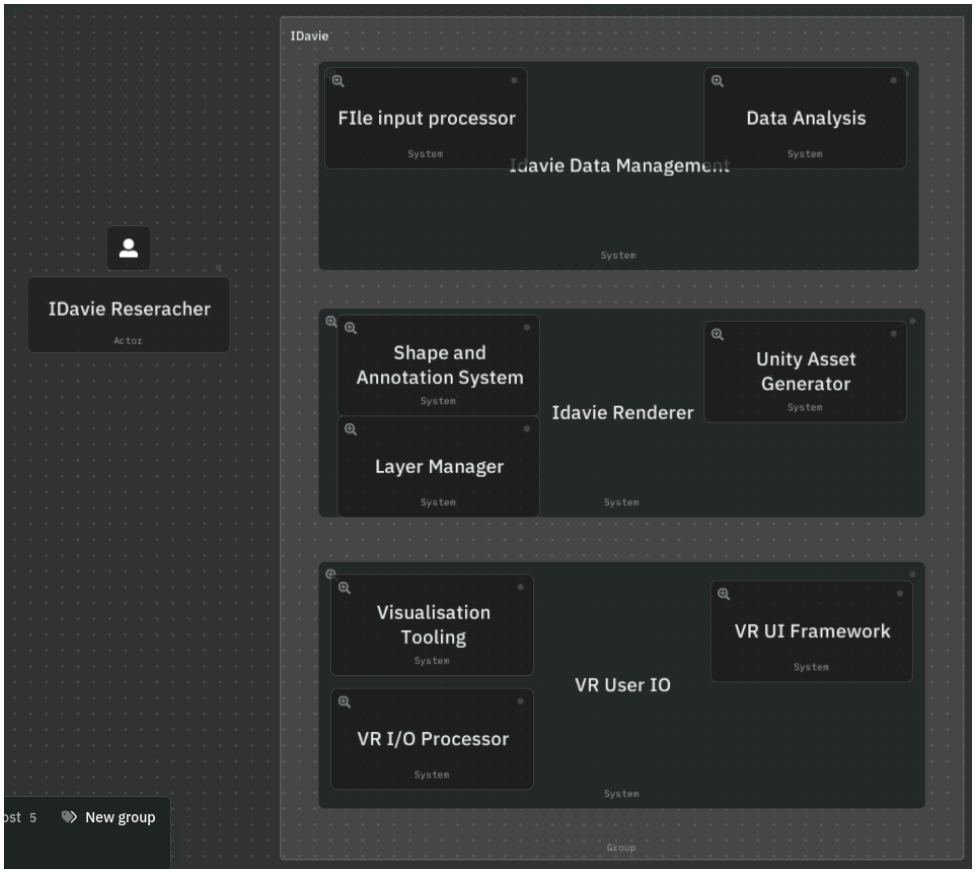


Figure 42: C4 Diagram

5. ACTS Case Study

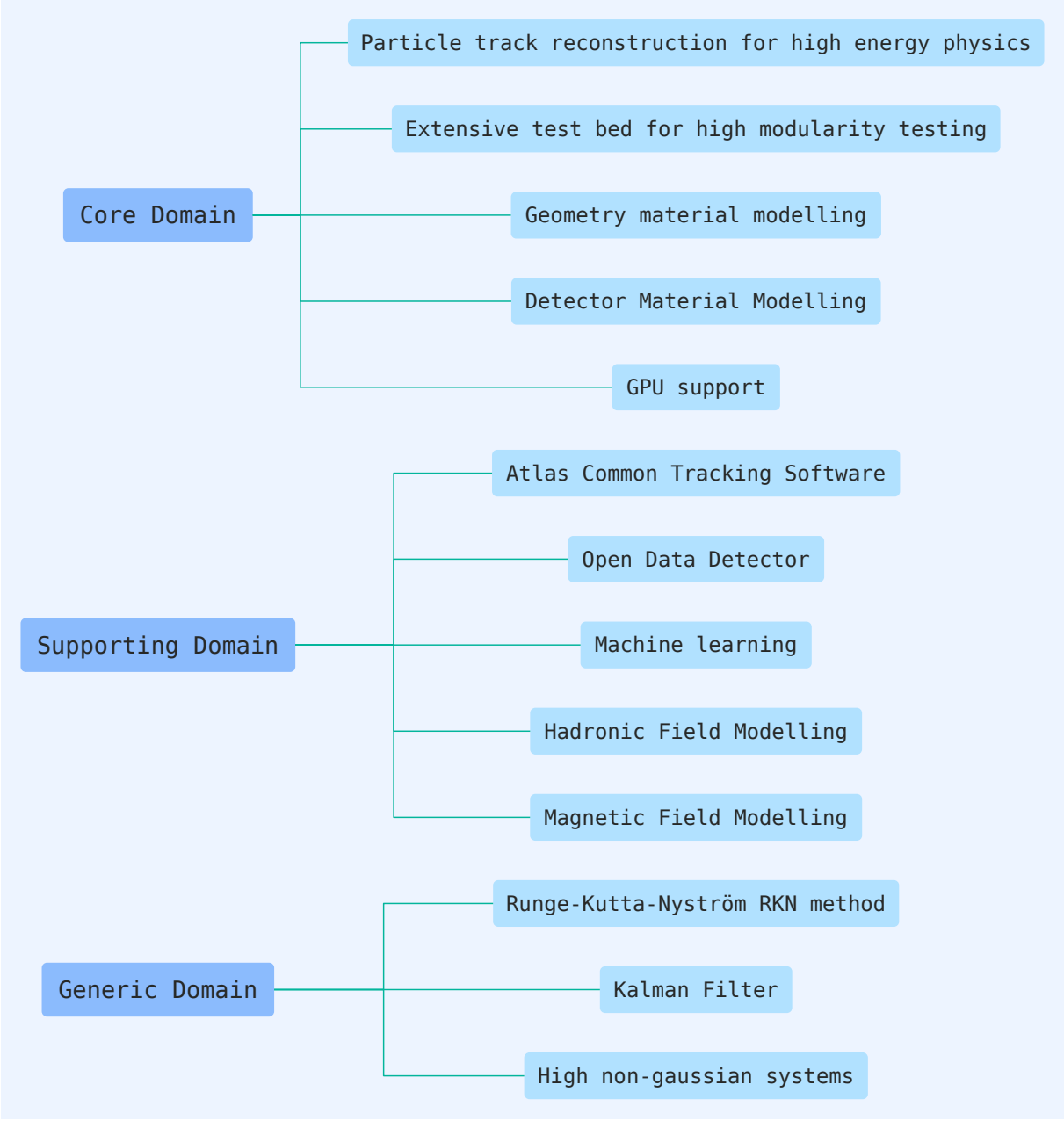
5.1. Tech Stack

Main Language: C++

Heavily used external libraries: Eigen, boost, cmake

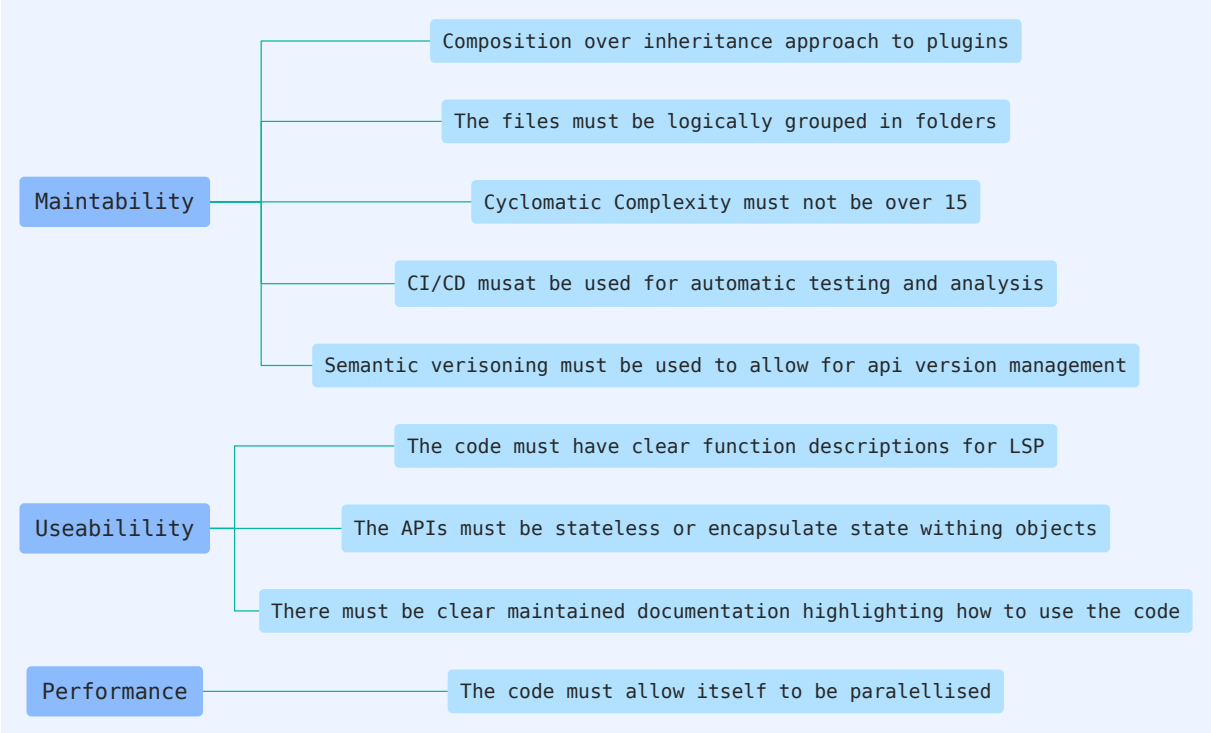
Used for example scripts: Python bindings using pybind11

Containerisation: Docker



Listing 9: Domain Model

5.2. Utility Tree



Listing 10: Utility Tree

5.3. Use Case Diagram

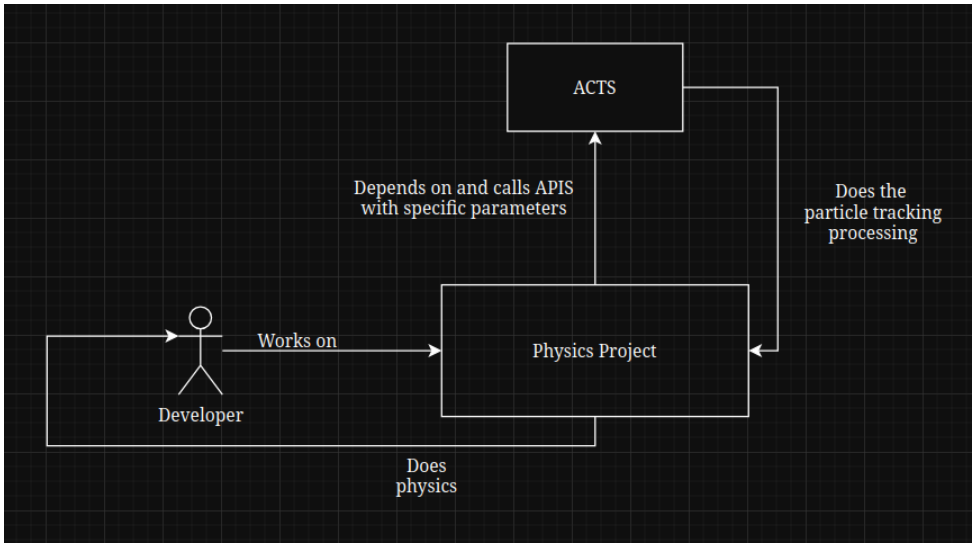
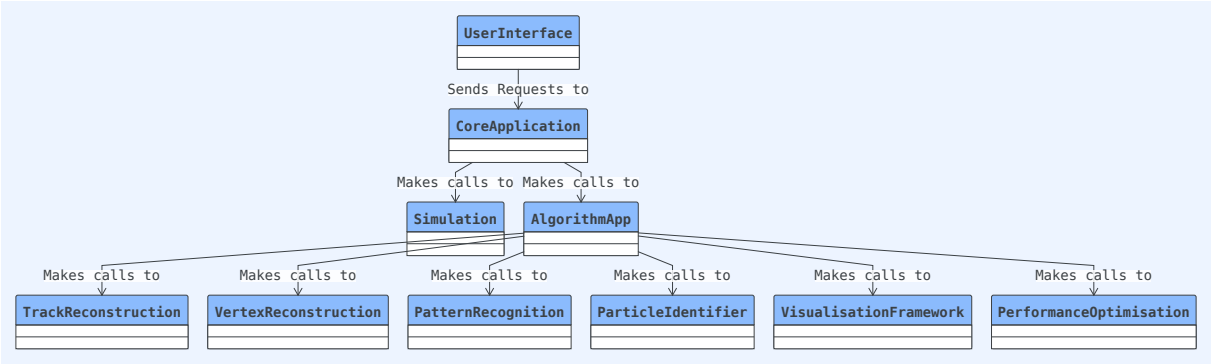


Figure 43: Use Case Diagram

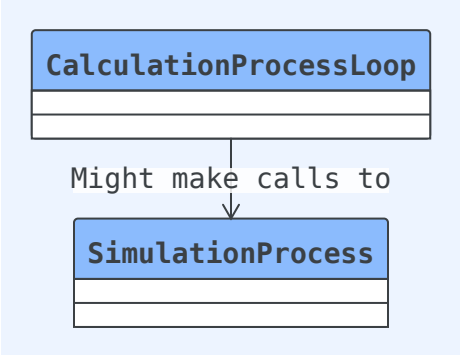
5.4. 4+1 Diagram

5.4.1. Logical View



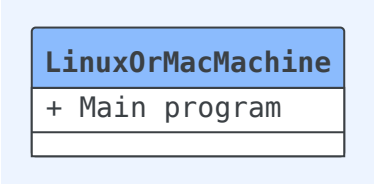
Listing 11:

5.4.2. Process view



Listing 12: Process view

5.4.3. Physcial View



5.4.4. Development View

- There are two layers of folder organistion in the codebase. The first is include/ and src/ follwing that there are logical groupings such as Geometry/ but there are some subfolders with a high file count befitting another layer of depth which is not in place. Geometry/ has over 30 files flat
- There is a consistent commit structure and rule
- Integration tests are loosely handled in

5.5. Codescene

Figure 44: Dependency Coupling Graph

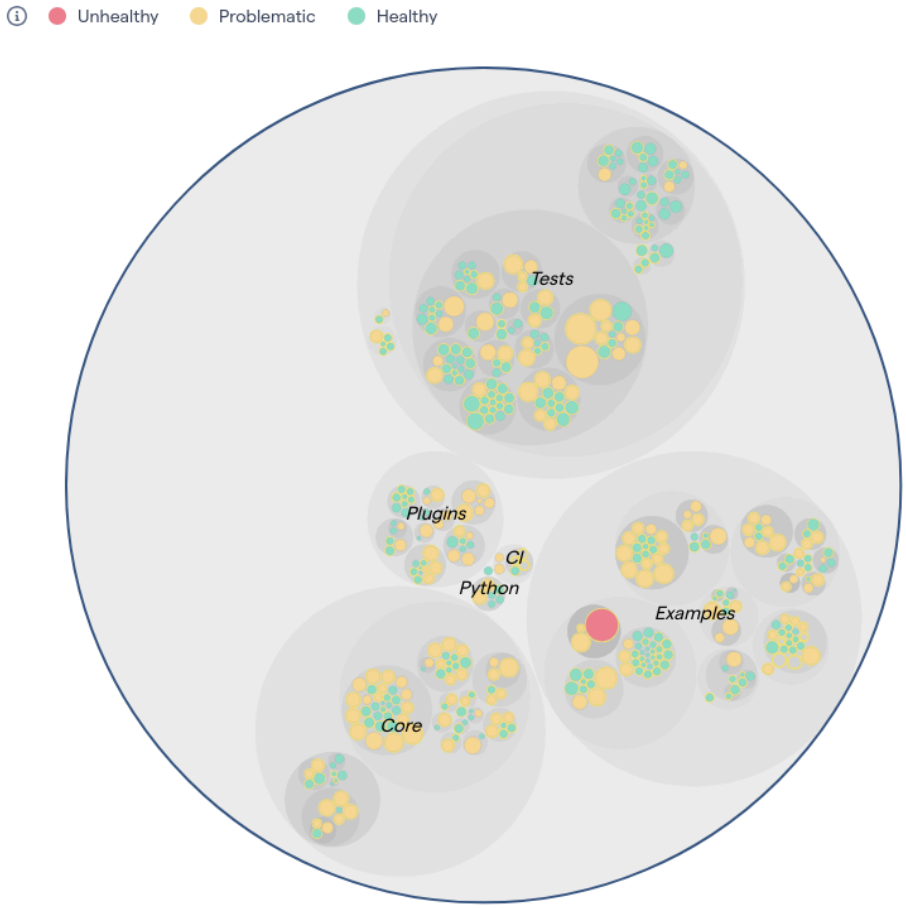


Figure 45: Technical Debt Codescene

5.6. Understand

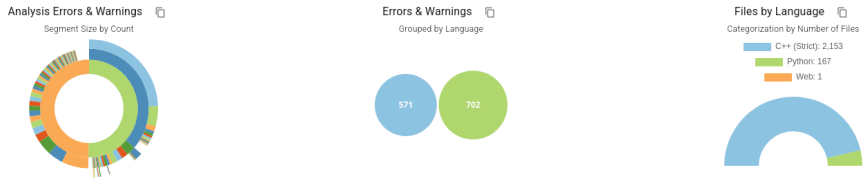


Figure 46: Understand High Level Info

Figure 47: Understand complexity ratings

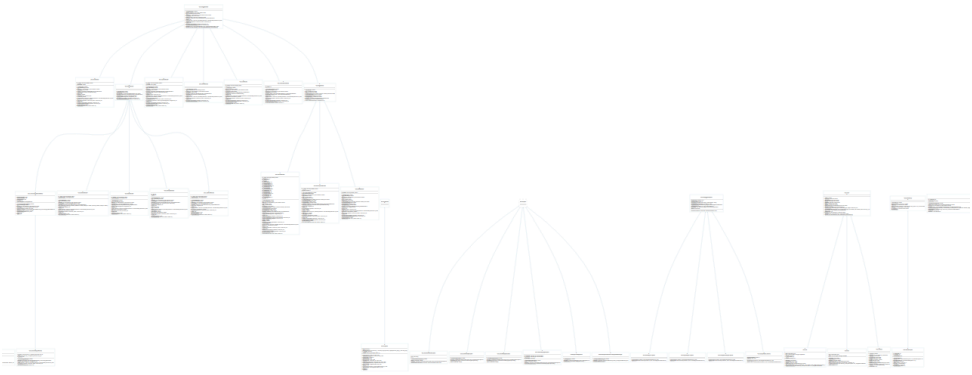


Figure 48: Class Diagram Inheritance

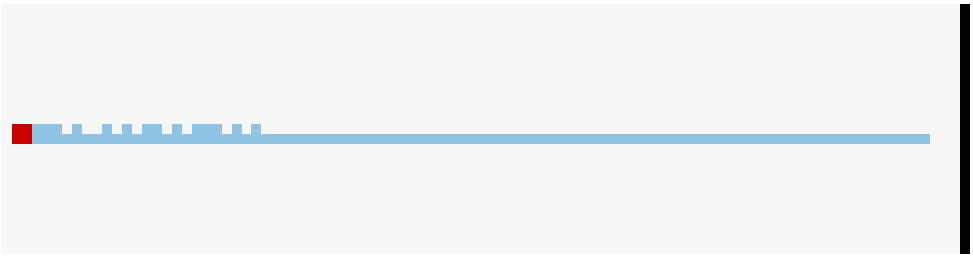


Figure 49: Class Diagram flat

5.7. Sonarqube

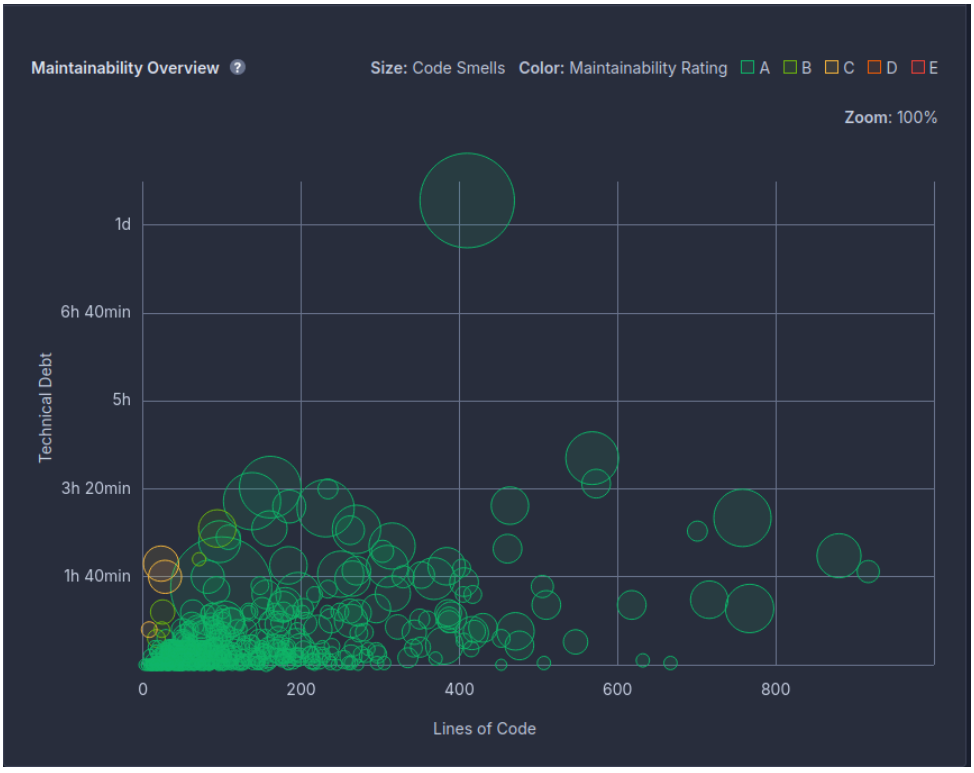


Figure 50:

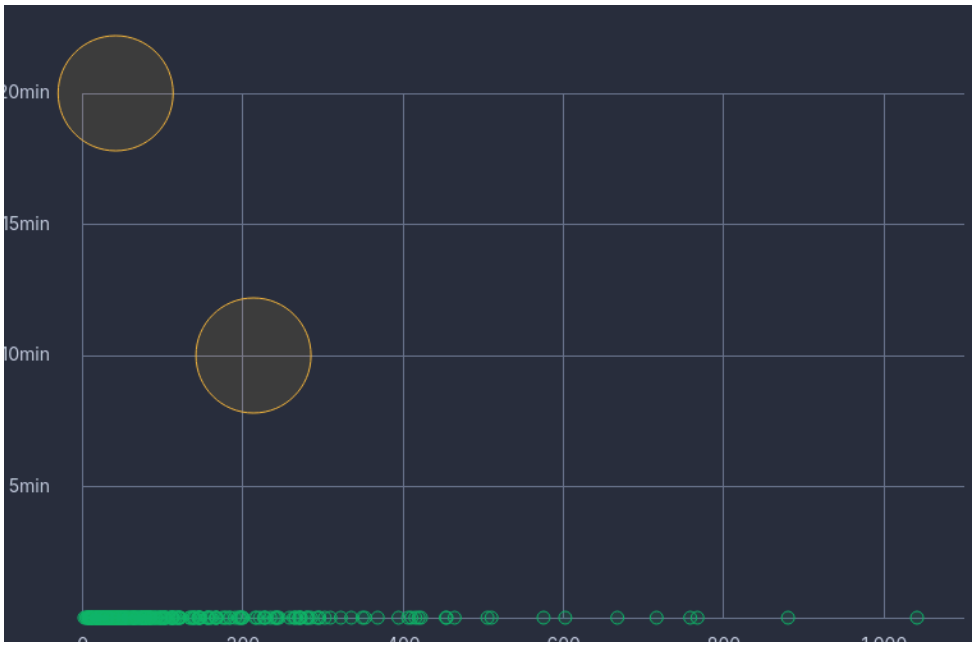


Figure 51:

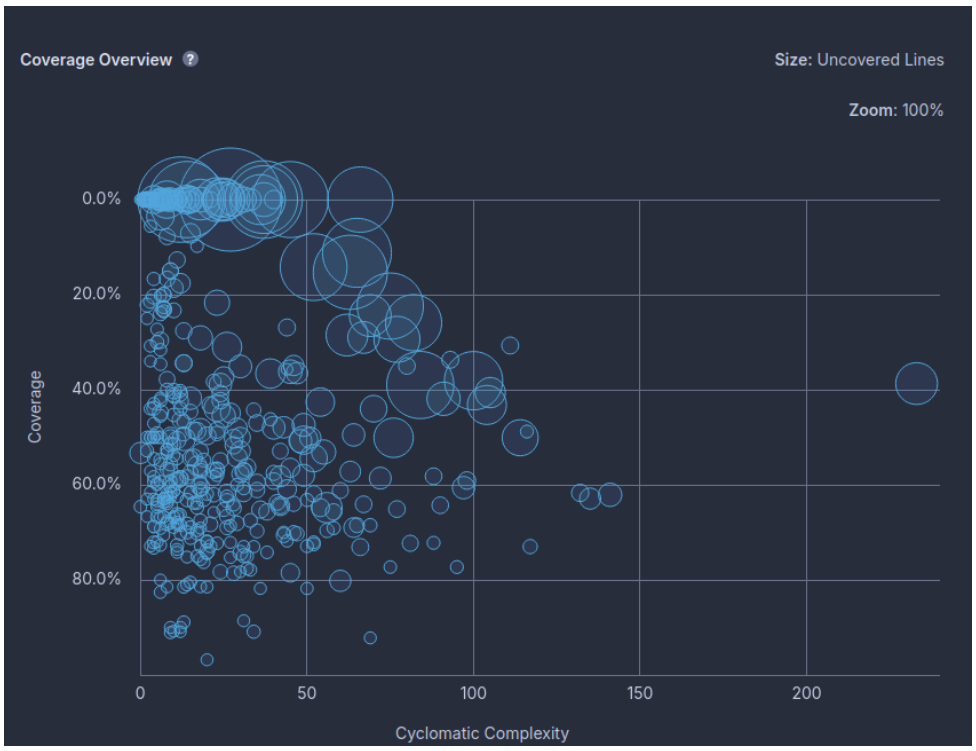


Figure 52:

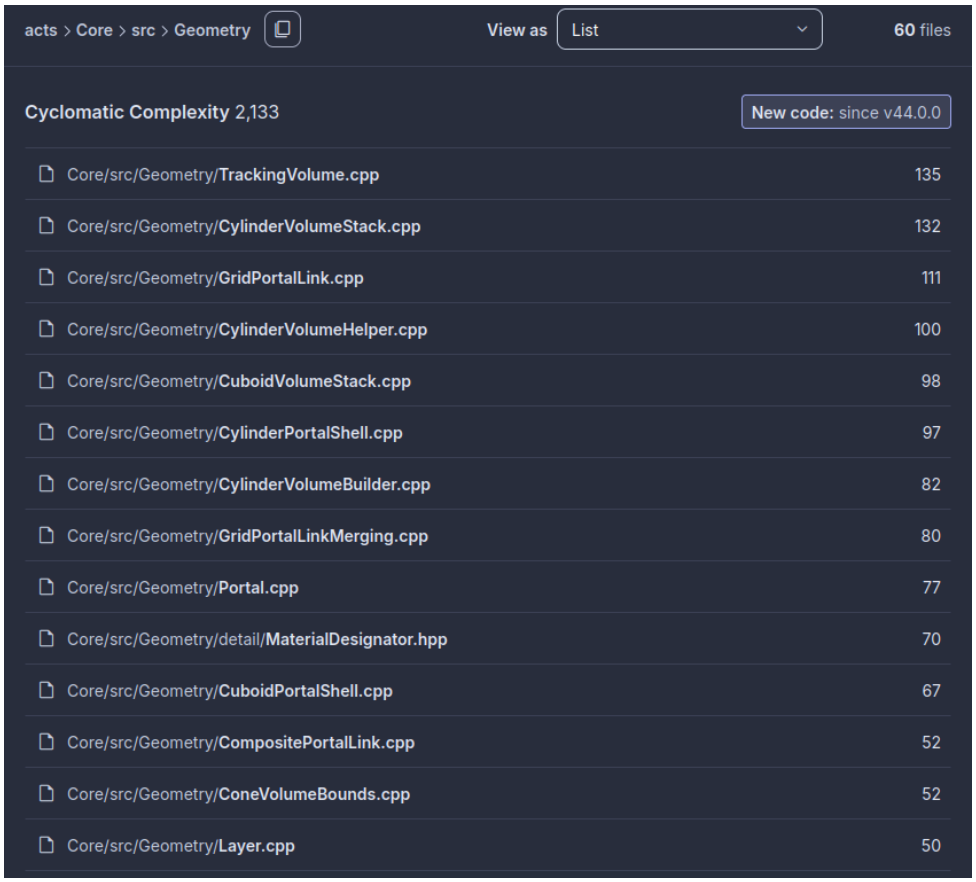


Figure 53:

5.8. C4

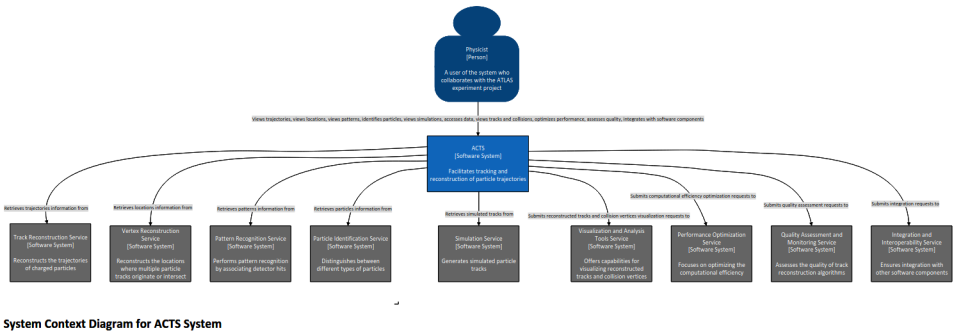


Figure 54: Context Diagram

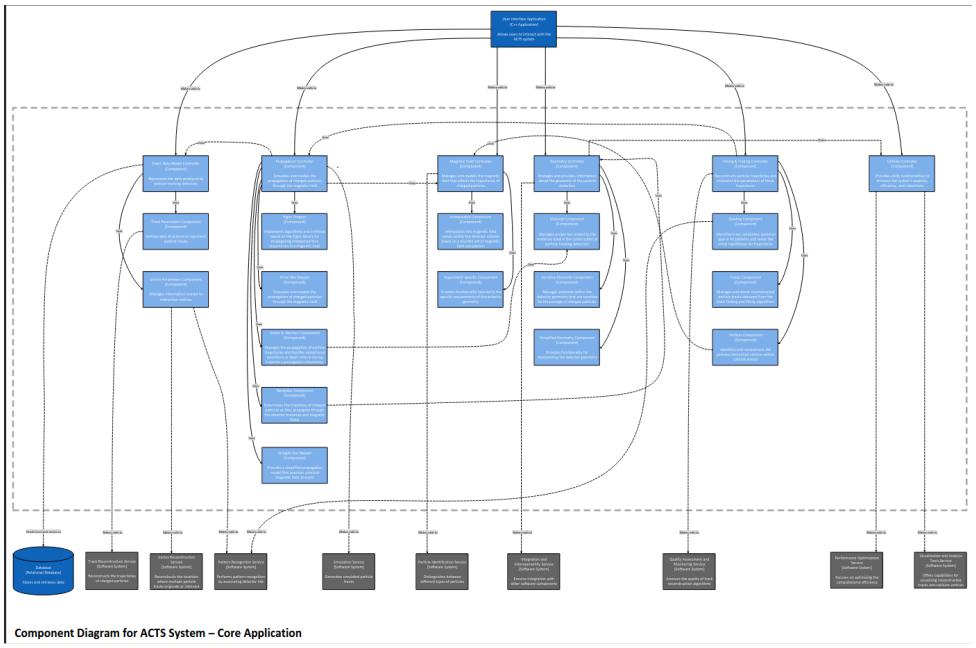


Figure 55: Component Diagram

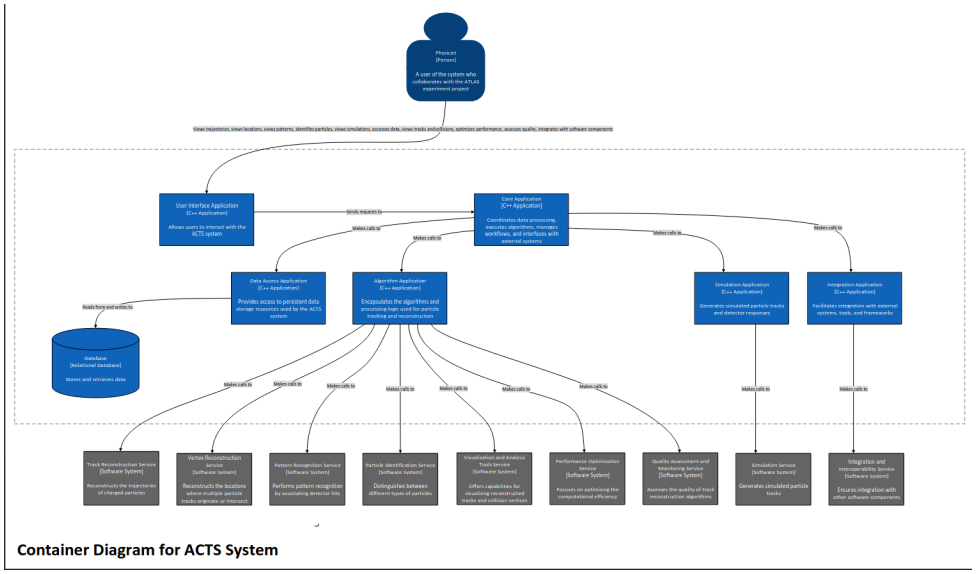


Figure 56: Container Diagram

Bibliography

[1] R. B. Appleby *et al.*, "Merlin++, a flexible and feature-rich accelerator physics and particle tracking library," *Computer Physics Communications*, vol. 271, p. 108204, Feb. 2022, doi: 10.1016/j.cpc.2021.108204.

[2] M. Rana and S. Baul, *A Survey on Microkernel Based Operating Systems and Their Essential Key Components*. 2023. doi: 10.2139/ssrn.4467406.