1. Software Architecture Design Portfolio

Art Ó Liathain

October 2025

2. Table of Contents

Contents

List of Figures

3. Merlin++

3.1. Current State

The current codebase is an artifact that seeks to achieve High speed particle simulation Listing 1. In development an accidental architecture Figure 14 formed as ad-hoc feature development approach meant that the only concrete architectural factor was the tech stack(Figure 2). Figure 6 is a clear example of the outcome of this, a jumbled mess of code 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. The key issues that must be resolved are:

- The blurring of lines of one off features verses core development goals
- Adding proper tests which run in a consistent reproducible manner
- Improving the flexibility of the code to allow new developers to contribute new features

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].

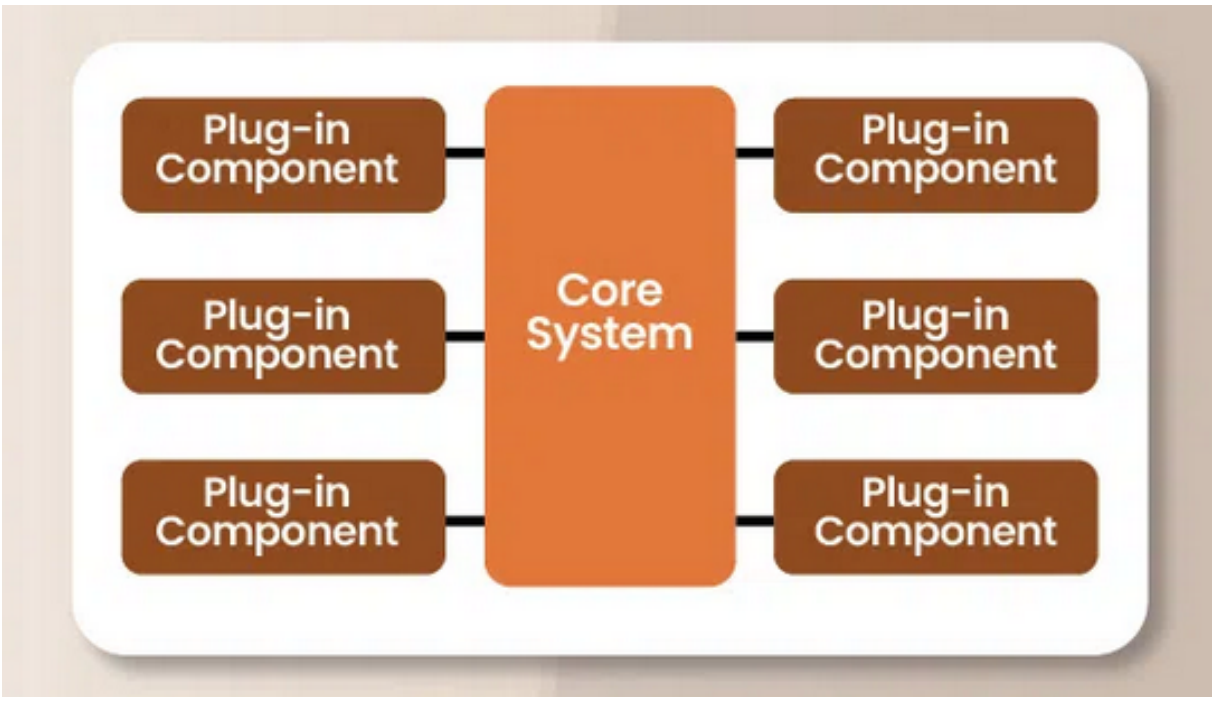


Figure 1: MicroKernel Architecture Diagram

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 (highlighted here Figure 3). 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 [3] as a plugin with state could become a dependency. 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 [1]. Allowing merlin developers to 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 easily extensible system being at the core of CERN high speed particle tracking simulations.

3.5. Diagrams

3.5.1. Tech Stack

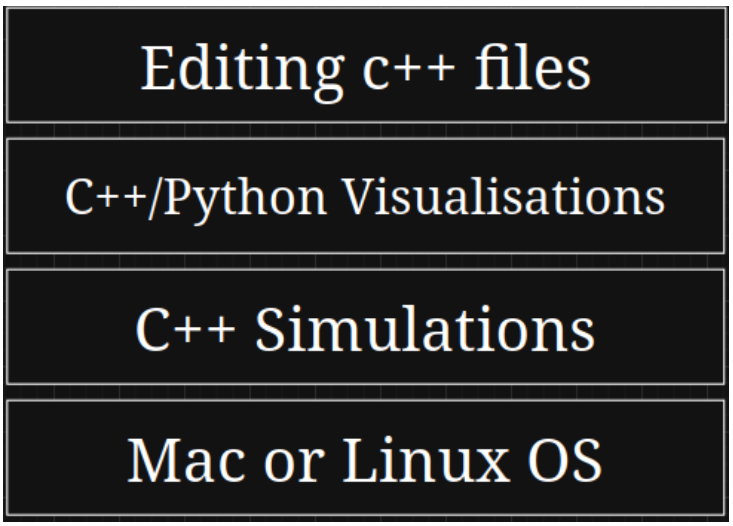
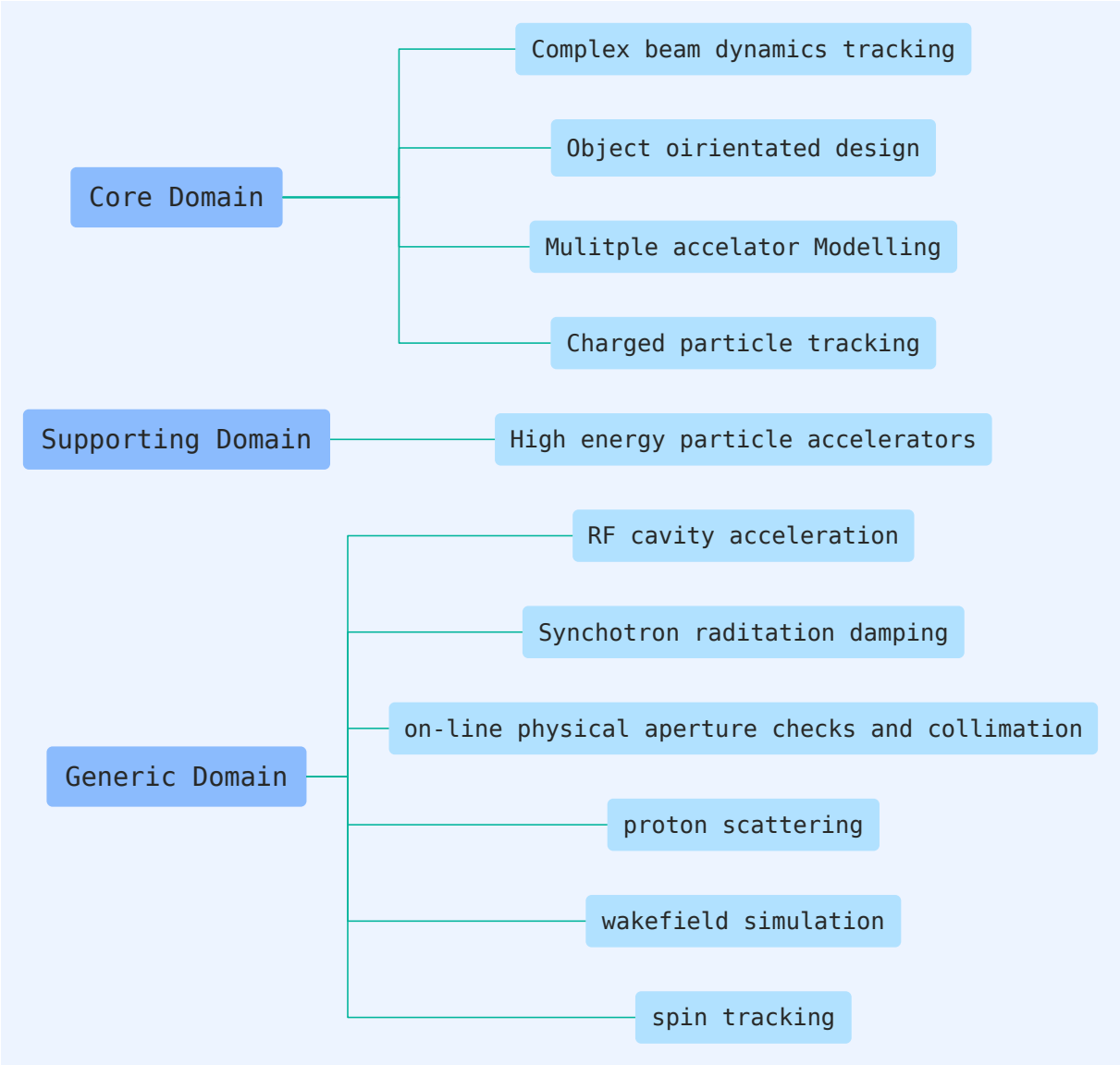


Figure 2: Merlin Tech Stack

3.5.2. Domain Model



Listing 1: Merlin Domain Model

3.5.3. Utility Tree

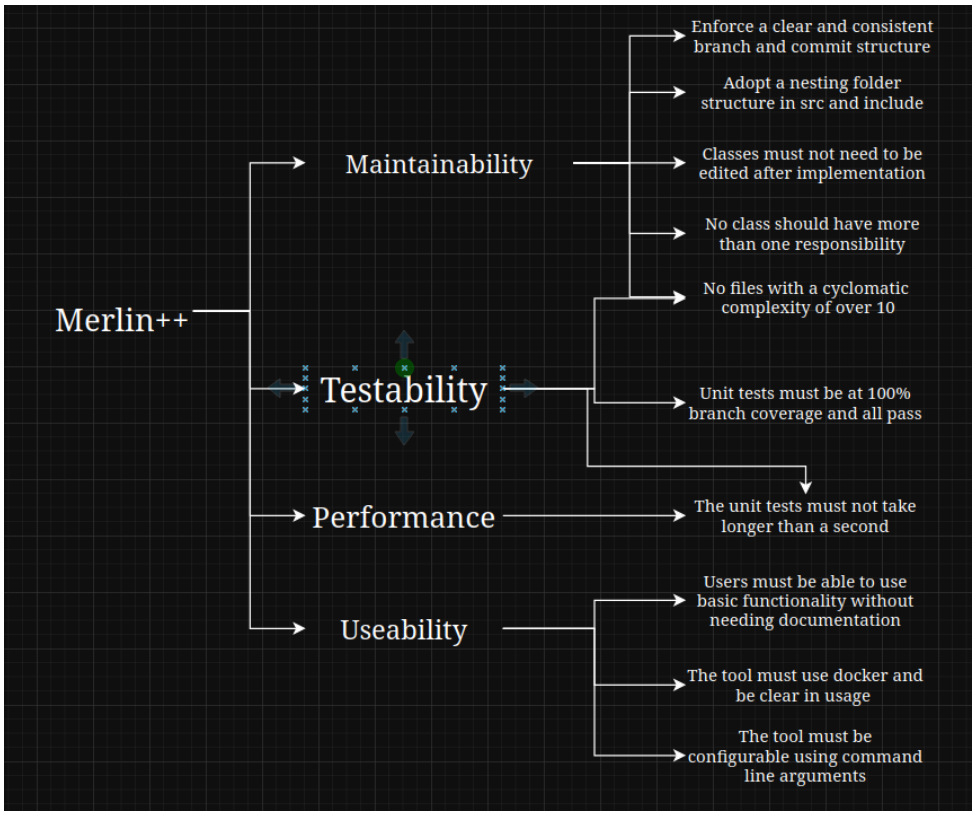


Figure 3: Merlin Utility Tree

3.5.4. Use Case Diagram

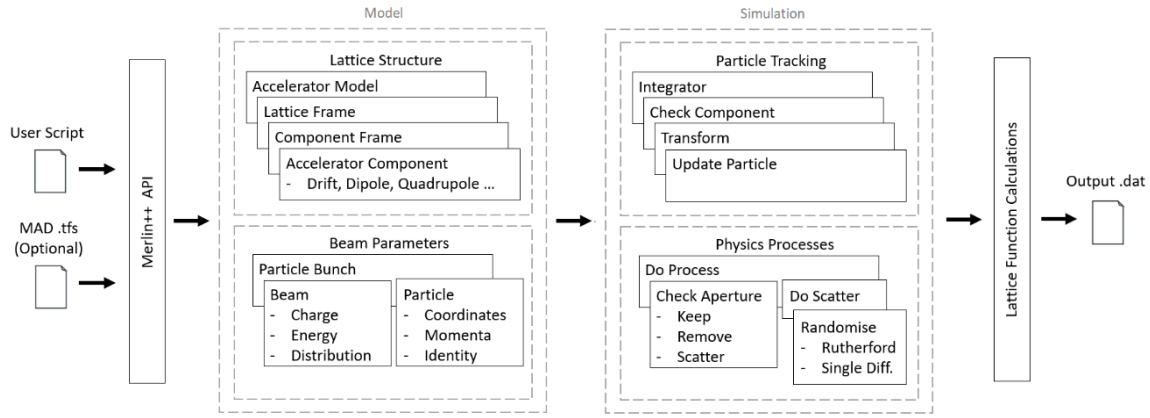
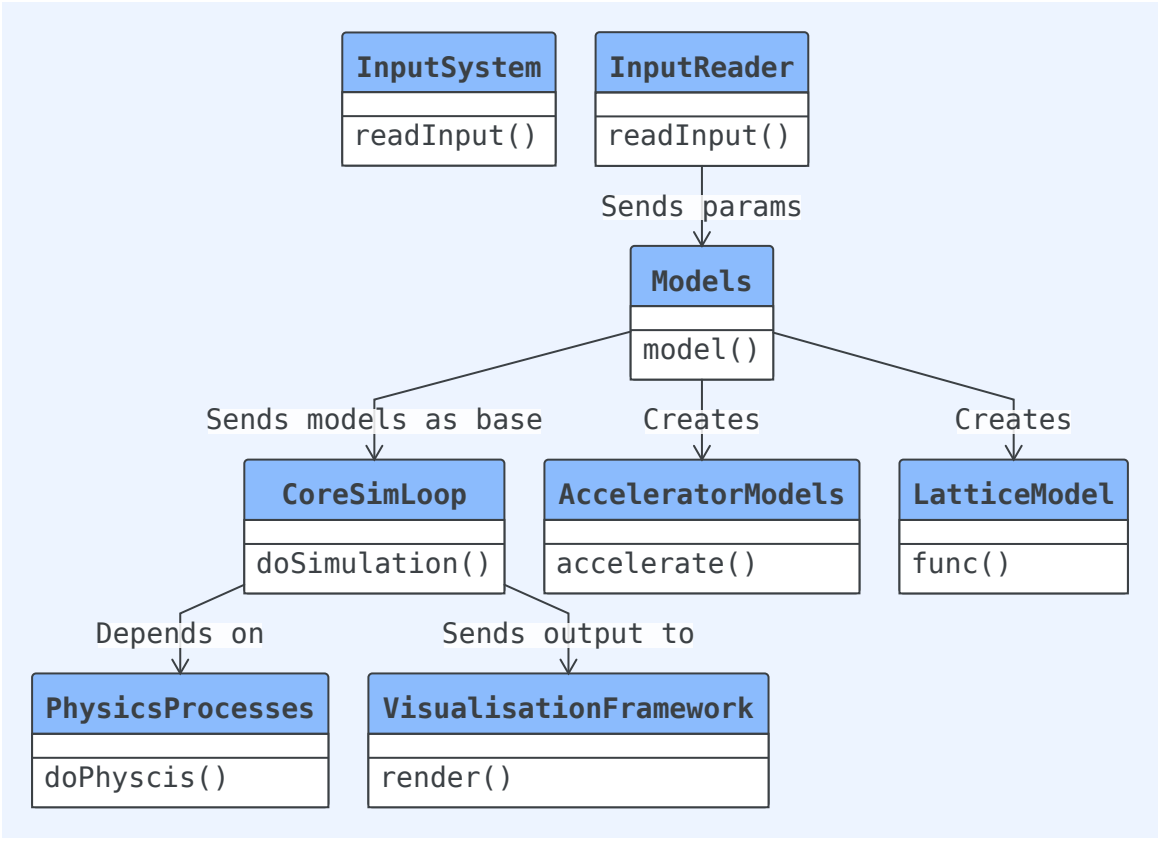


Figure 4: Merlin Use Case diagram

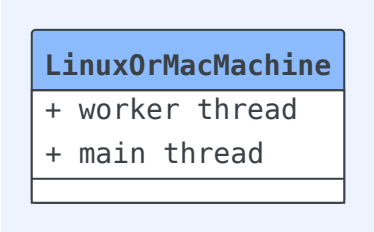
3.5.5. 4 + 1 Diagram

3.5.6. Logical View



Listing 2: Merlin Logical View

3.5.7. Physical View

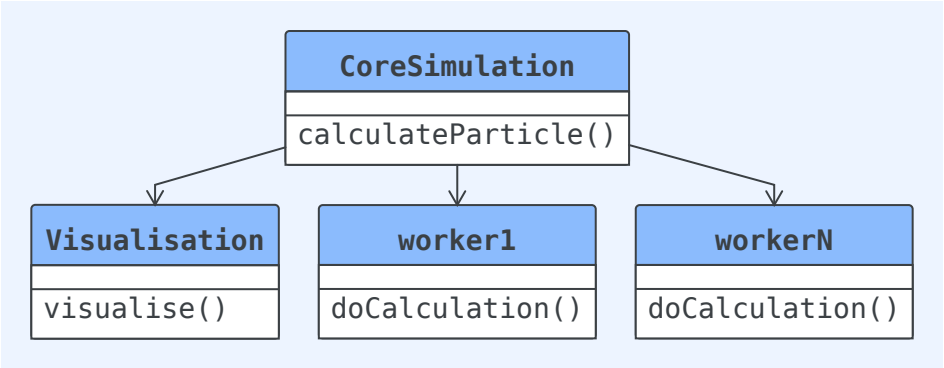


Listing 3: Merlin Physical View

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

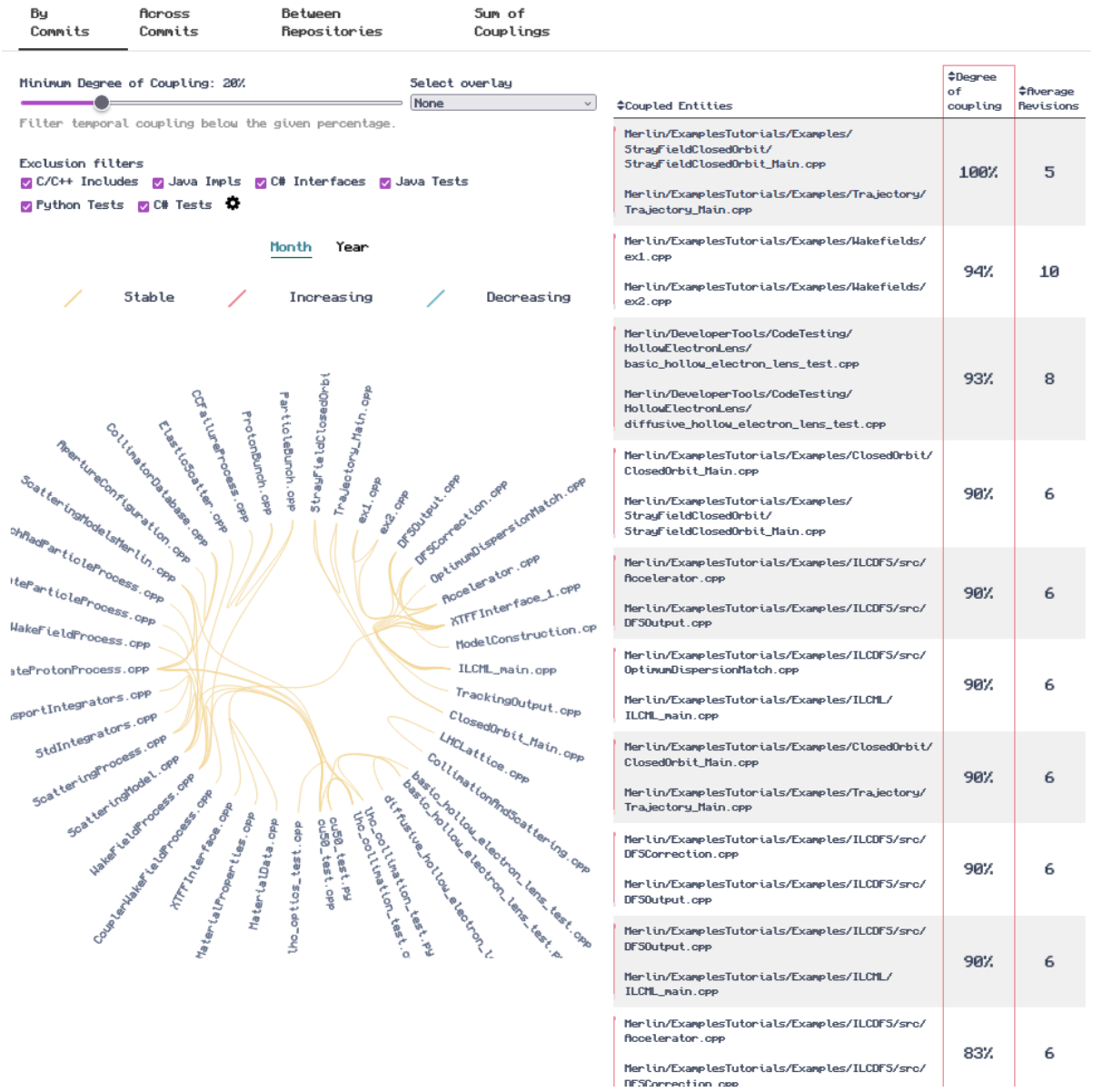


Figure 5: Merlin Codescene coupling graph

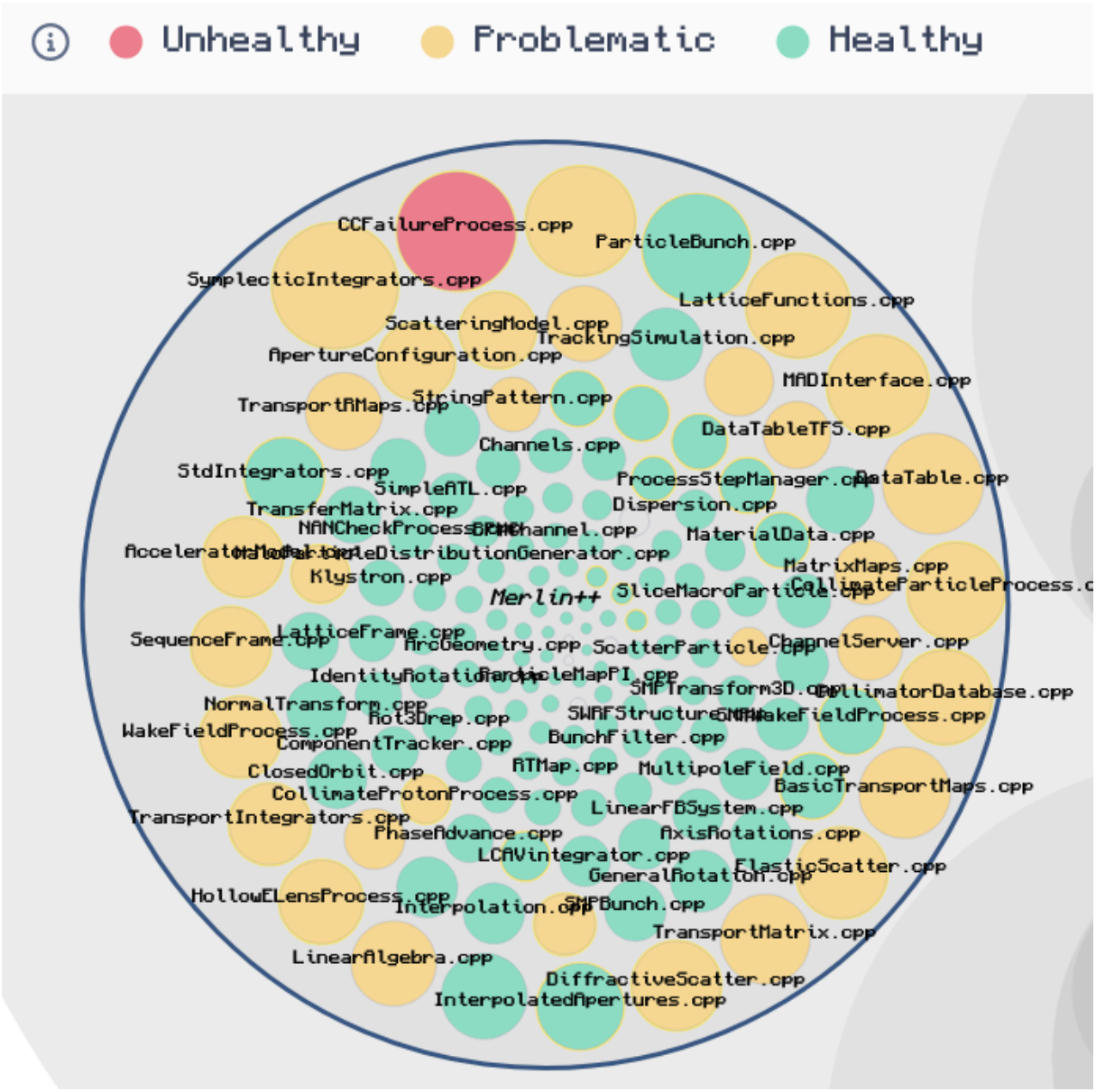


Figure 6: Merlin Codescene maintainability circle

3.5.11. DV8

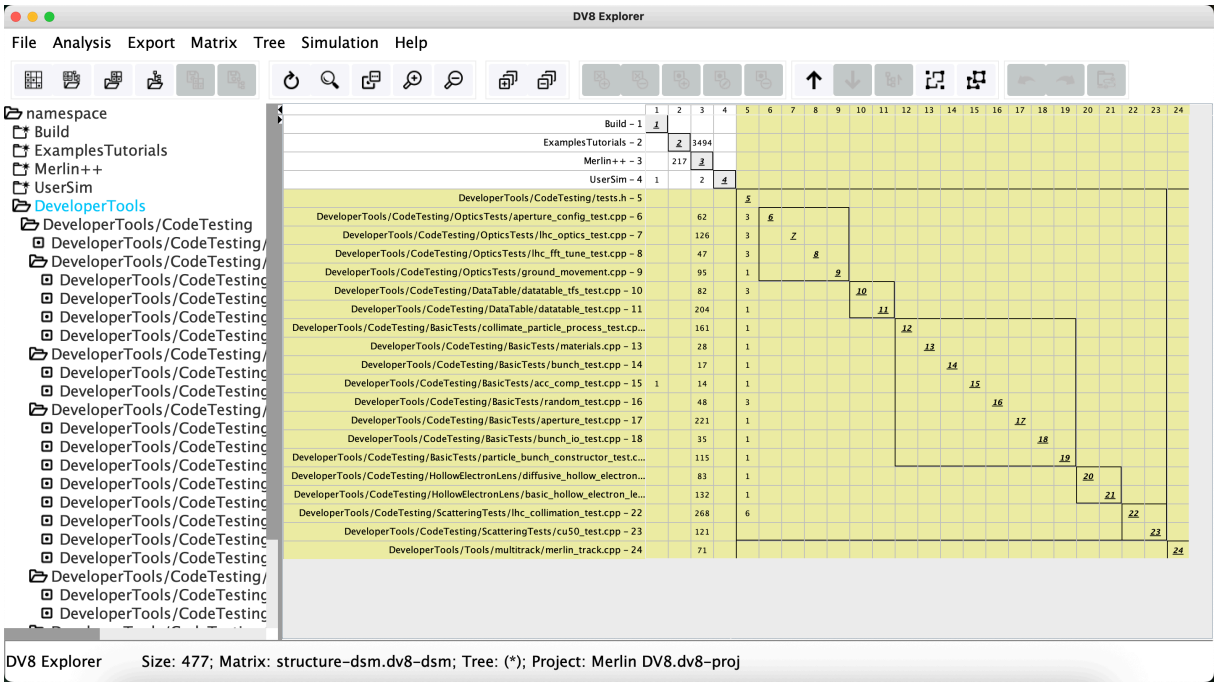


Figure 7: Merlin Dv8 Dev tool analysis

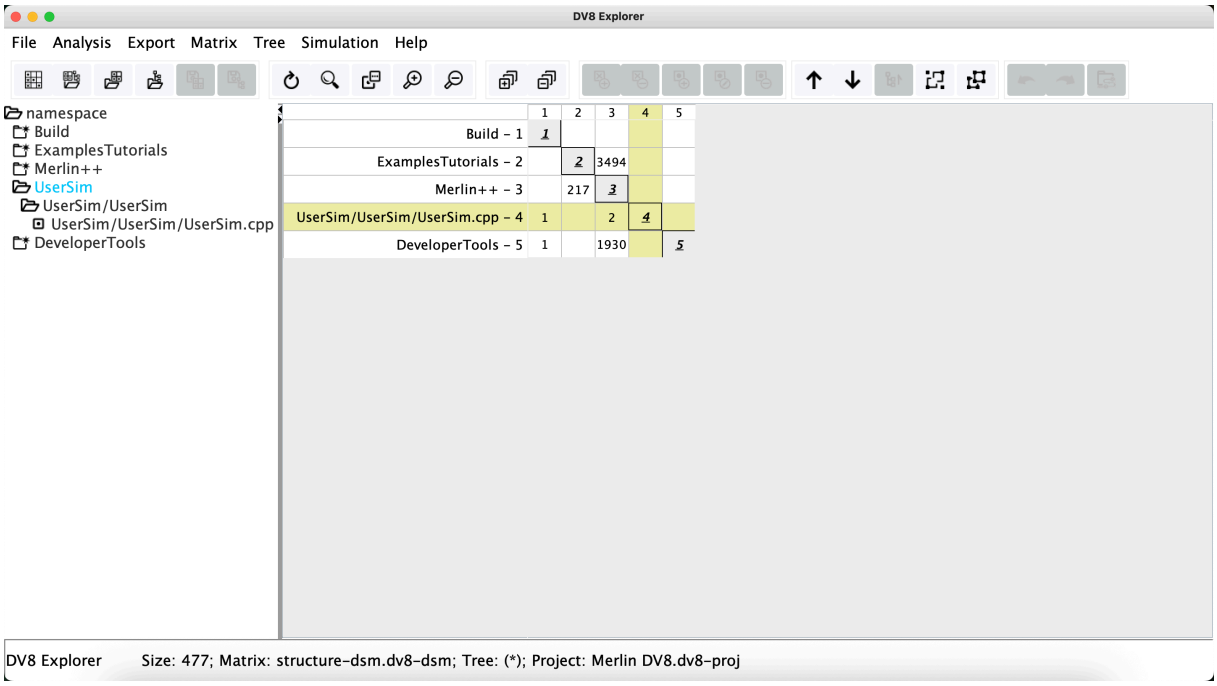


Figure 8: Merlin Dv8 high level analysis

3.5.12. SonarQube

Figure 9: Merlin Technical debt cost

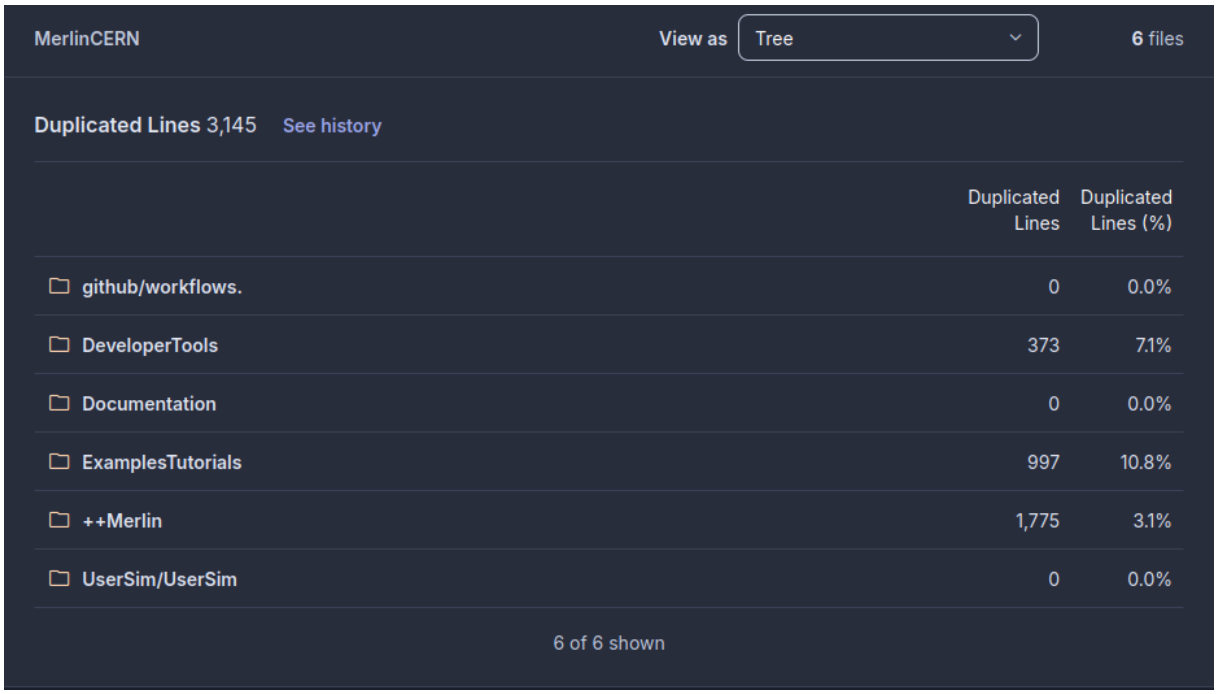


Figure 10: Merlin Duplicated code analysis

3.5.13. Understand

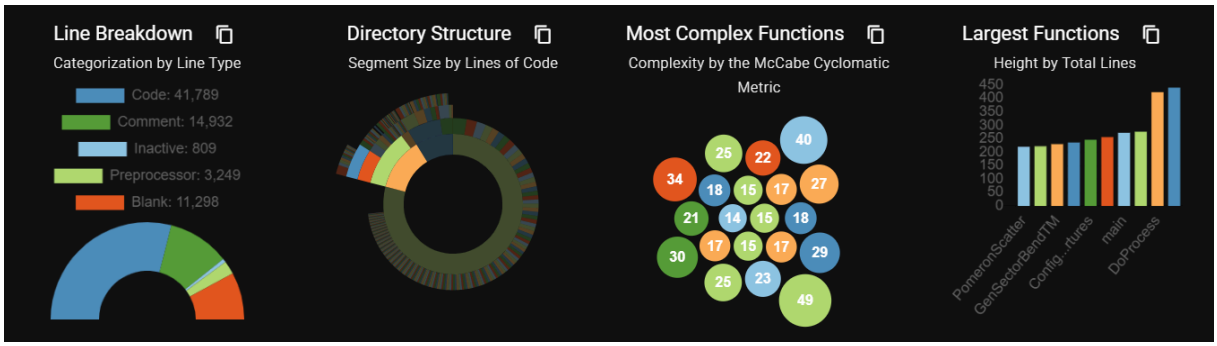


Figure 11: Merlin Function complexity understand

Figure 12: Merlin Coding langauge breakdown

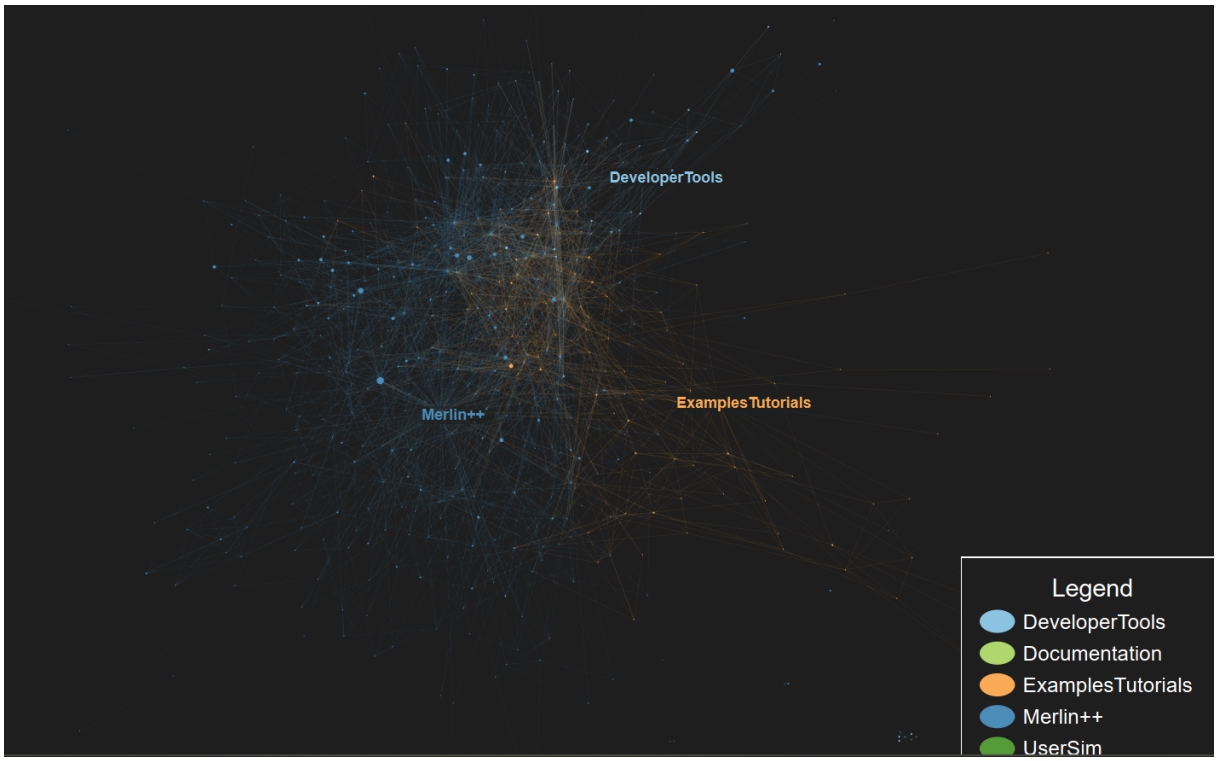


Figure 13: Merlin Understand code dependency graph

3.5.14. C4

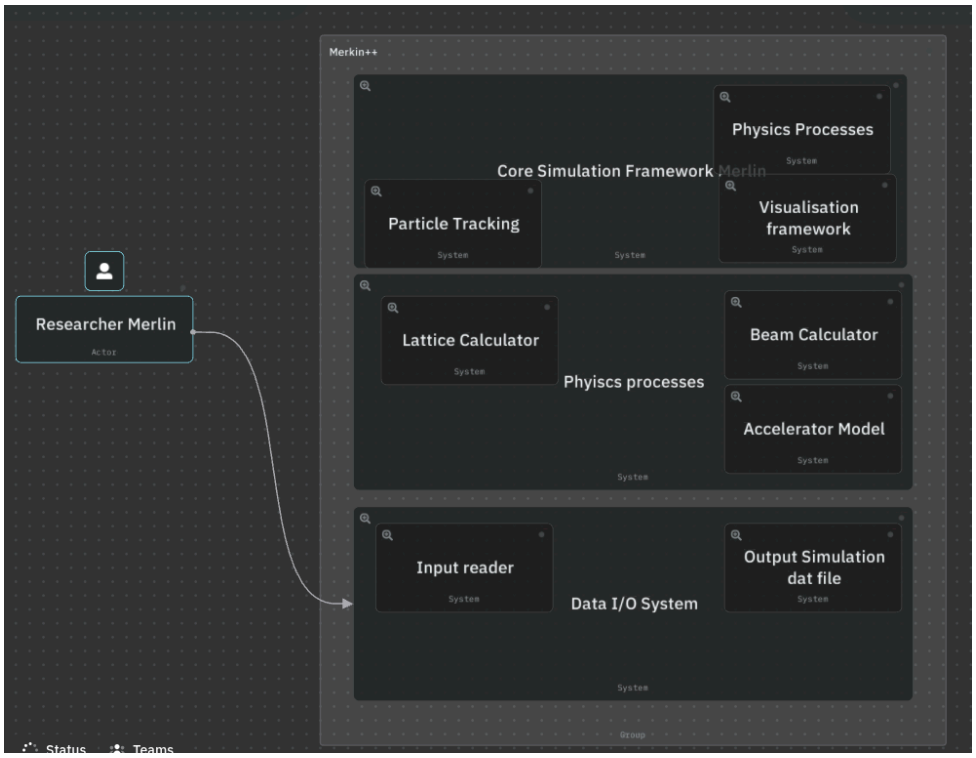


Figure 14: Merlin C4 Diagram

4. Fluctuating Finite Element Analysis (FFEA) Case Study

4.1. Current state

FFEA is a tool primarily focused on simulating larger scale proteins using tetrahedrons (Listing 5) as well as preparing protein files for simulation(FFEA Tools). The tool works as intended but due to the haphazard development a God class based architecture has developed as seen in Figure 28. Overloaded classes blatantly breaking single responsibility has lead to classes with cyclomatic complexity in the hundreds Figure 20.

Complexity at this scale severely reduces the capabilities of new developers contributing to the project as the god class must be understood at a base level. Reducing the number of new experiments that can be carried out.

4.2. Proposed Architecture

The current state of the project needs extensibility and maintainability as core parts of the new architecture as described Listing 6. The architecture that best fits the use case is a microkernel architecture [2].

A microkernel architecture is a system built around a single purpose, in this case simulation while allowing a plug-in system to introduce new functionality without changing the core code [2].

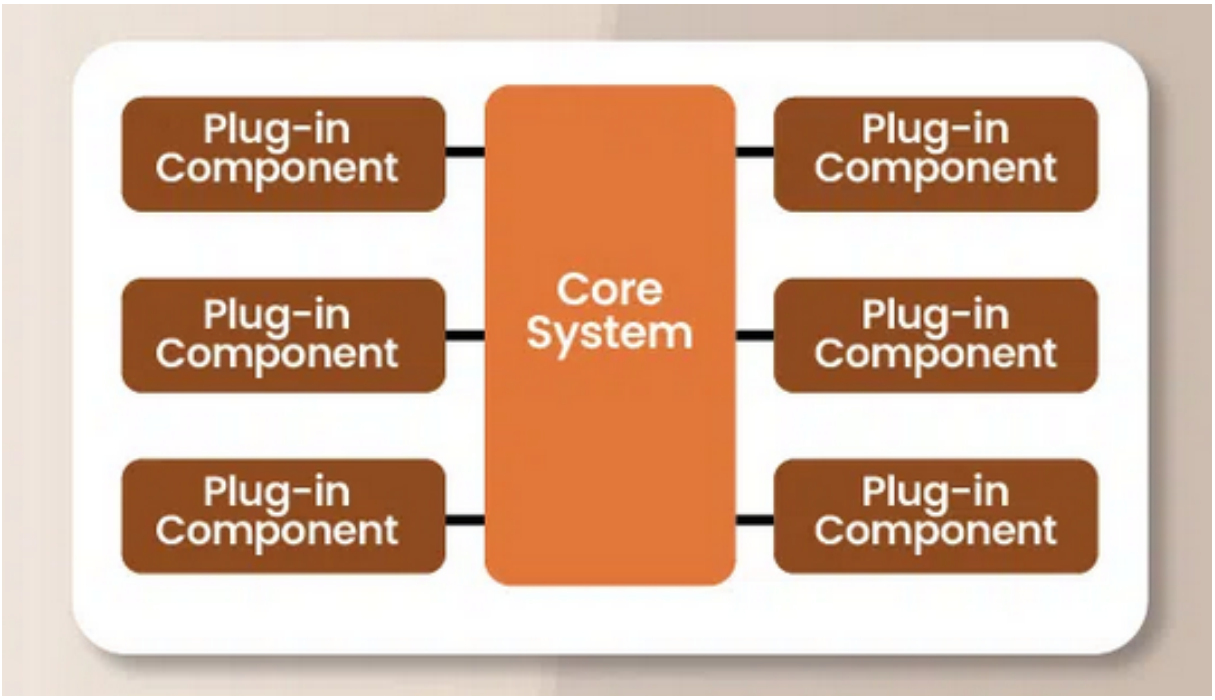


Figure 15: MicroKernel Architecture Diagram

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.
- As the kernel will be the focus of development **performance** can be improved more easily.
- Removing the plugins from the core developer responsibilities improves **maintainability** as only the core code must be maintained.
- The plugin system allows for development **extensibility**, allowing new features without changing the core code.

4.3. Reasoning

The accidental architecture of god classes in FFEA lends itself well inferring what should be made the microkernel. The god class (Seen in Figure 26) in FFEA is a solid starting point to the refactor, tackling the biggest problem (Figure 19) and converting it to a solid foundation to the new architecture. This approach addresses the root of the maintainability issue rather than layering more complexity on top.

By adopting a microkernel architecture, FFEA can evolve into a modular and sustainable framework where new experiments and simulation types can be added without disrupting the existing system. Plug-ins allow for contributors who are focused on testing a new experiment instead of long term code quality can implement features without architecture erosion [3]. This shift towards microkernel prioritizes maintainability performance within the core codebase and extensibility for new scientists, the key qualities identified that FFEA needs.

4.4. Conclusion

There is a lot of work to do with FFEA, the codebase is in a continuous loop of; no development, funding for new experiment, implement experiment and repeat. This vicious cycle is why the code is in the current state as there are no incentives for clean code. The proposed solution fits well to enable extensibility by many researchers while still preserving the core code from deteriorating between funding cycles.

4.5. Diagrams

4.5.1. Tech Stack

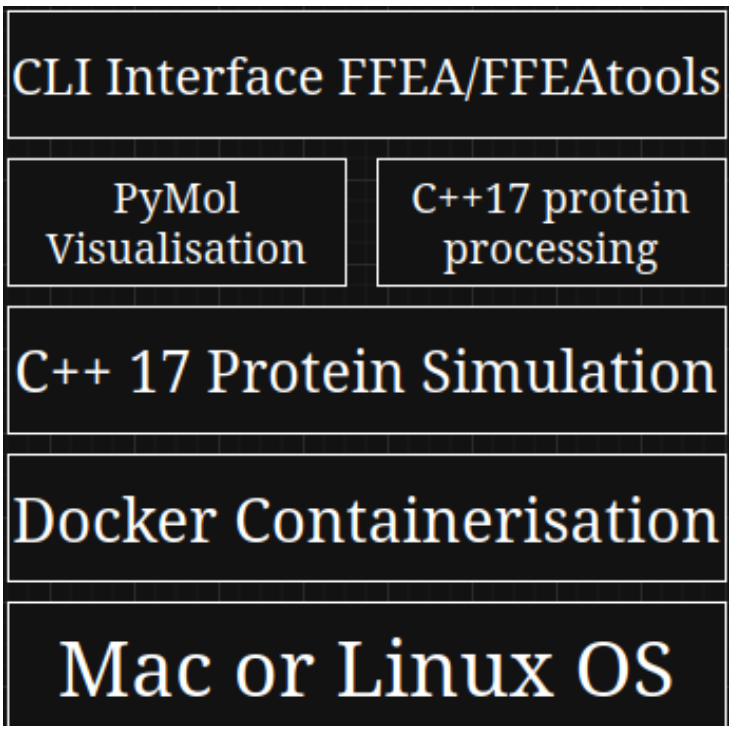
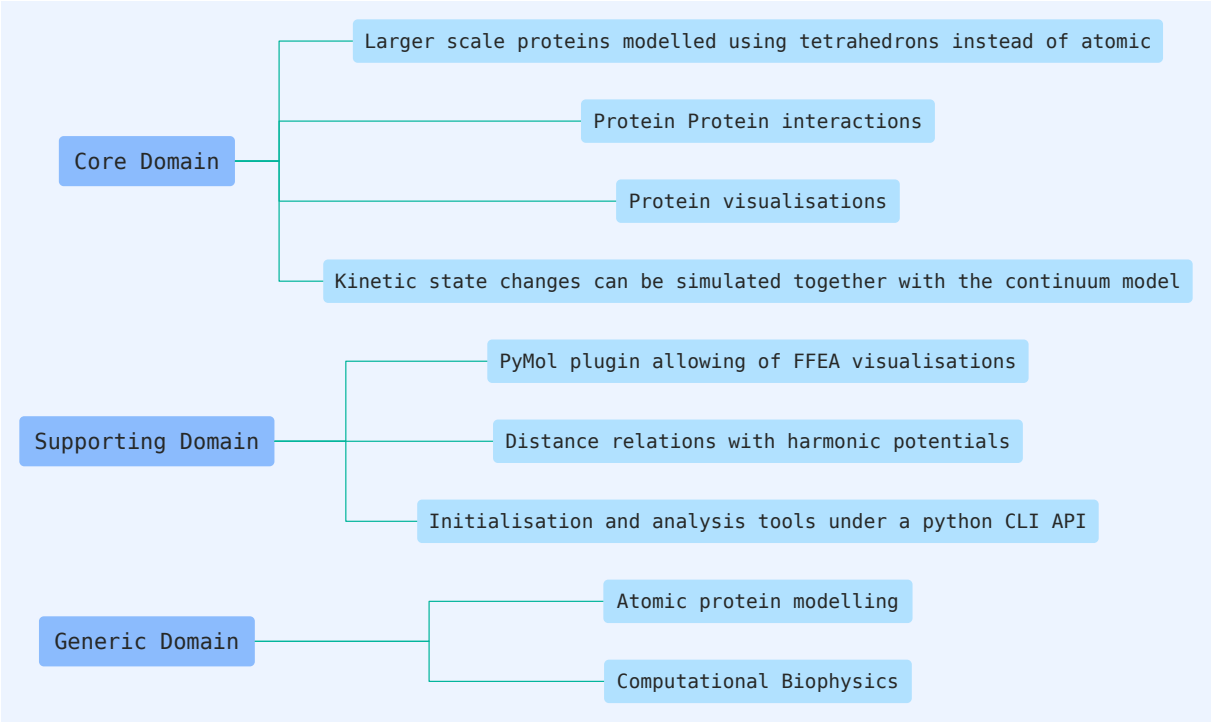


Figure 16: FFEA Tech Stack



Listing 5: FFEA Domain Model

4.5.2. Use Case Diagram

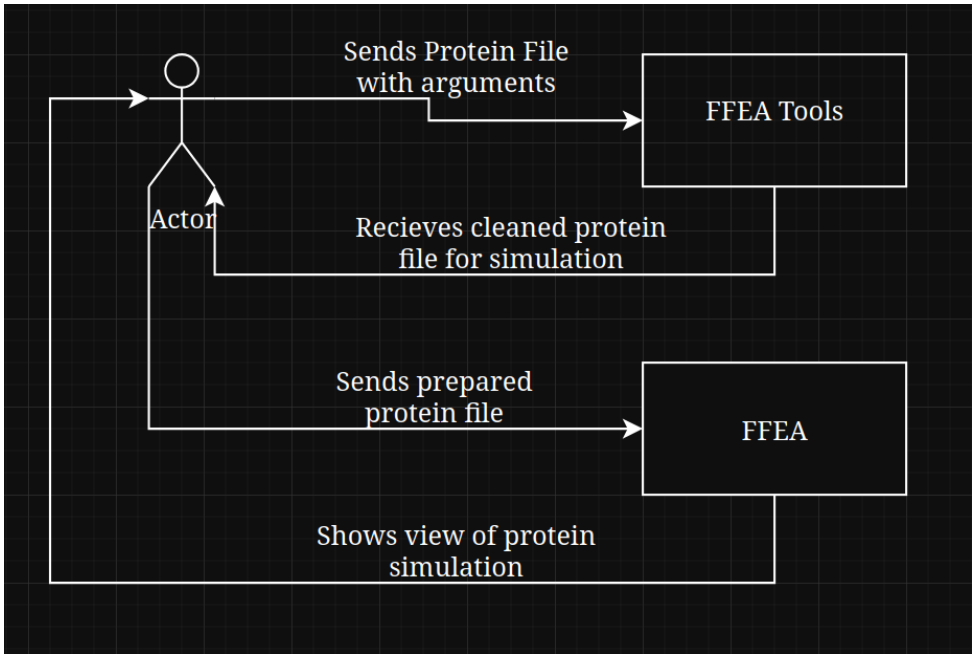
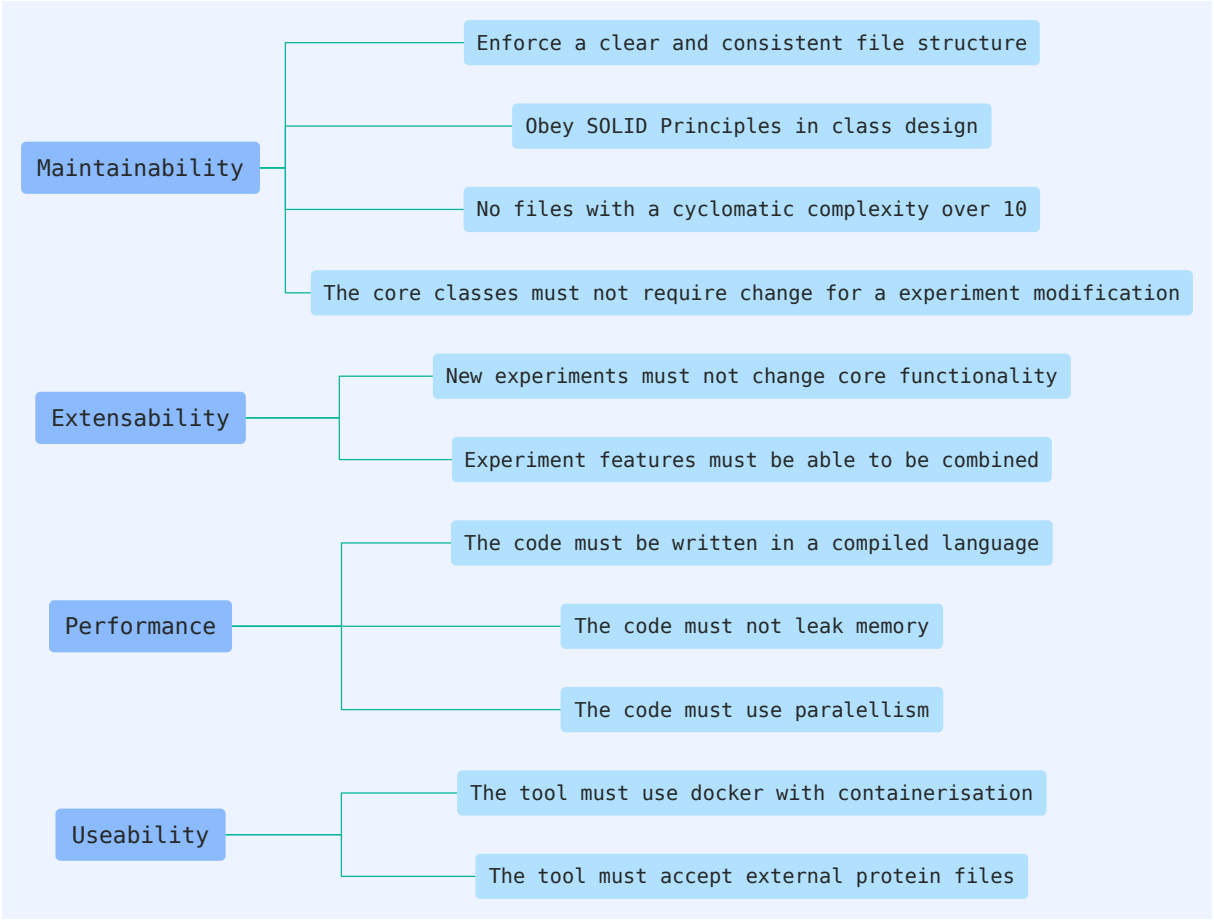


Figure 17: FFEA Use Case Diagrams

4.5.3. Utility Tree



Listing 6: Utility Tree

4.5.4. SysML Diagram

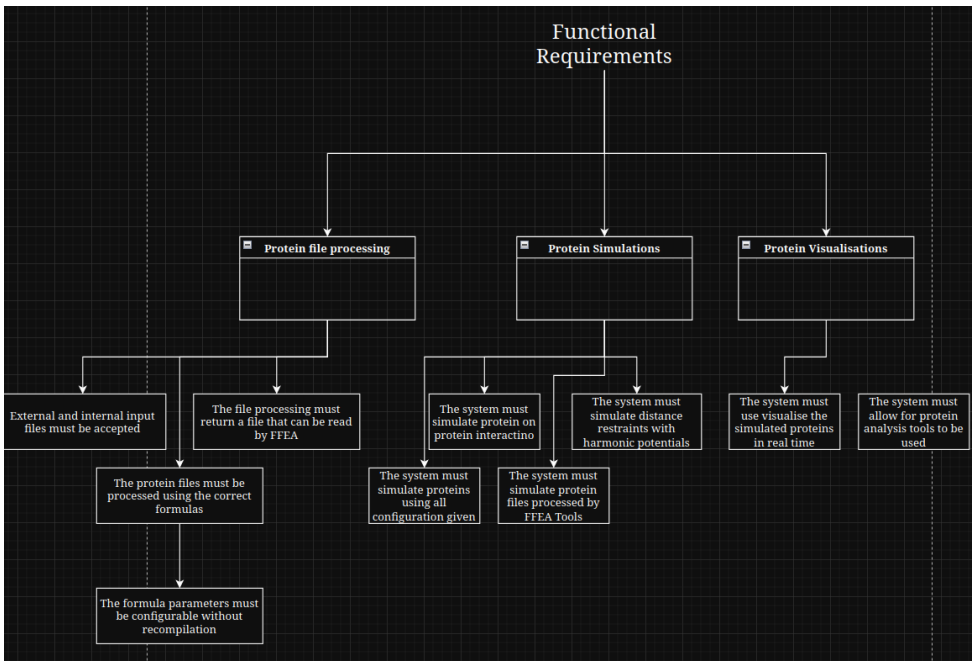


Figure 18: SysML Diagram

4.5.5. Codesense Diagram

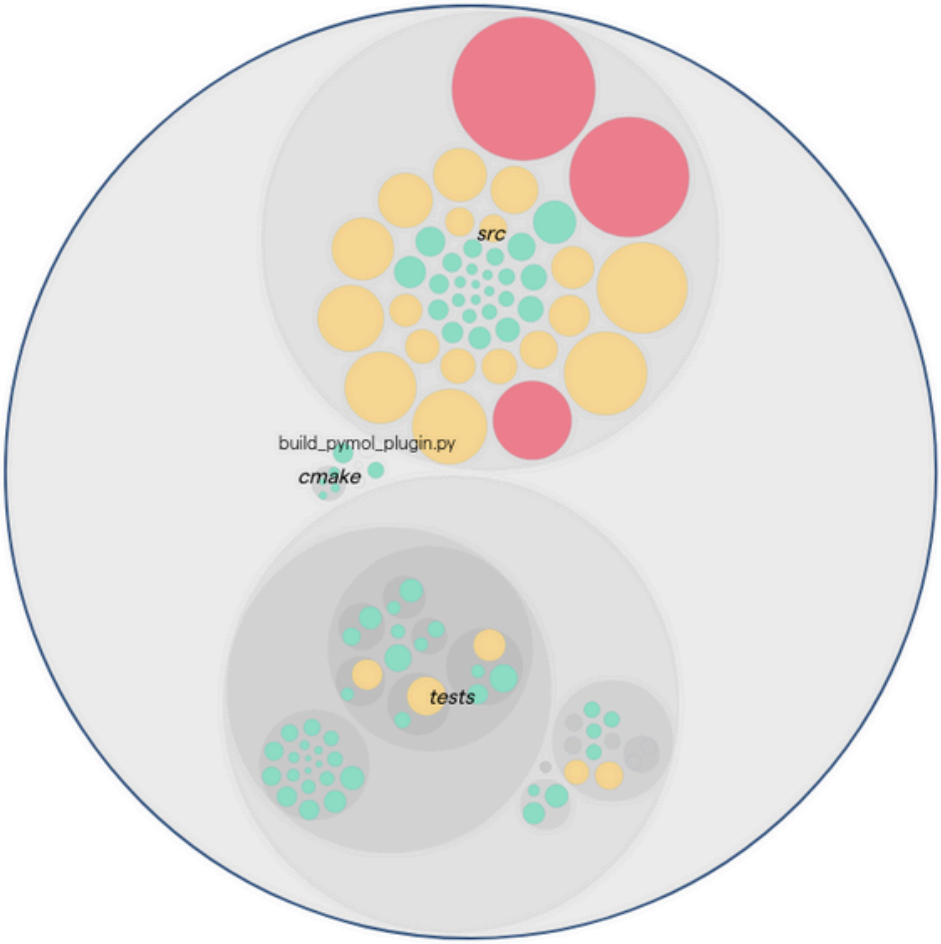
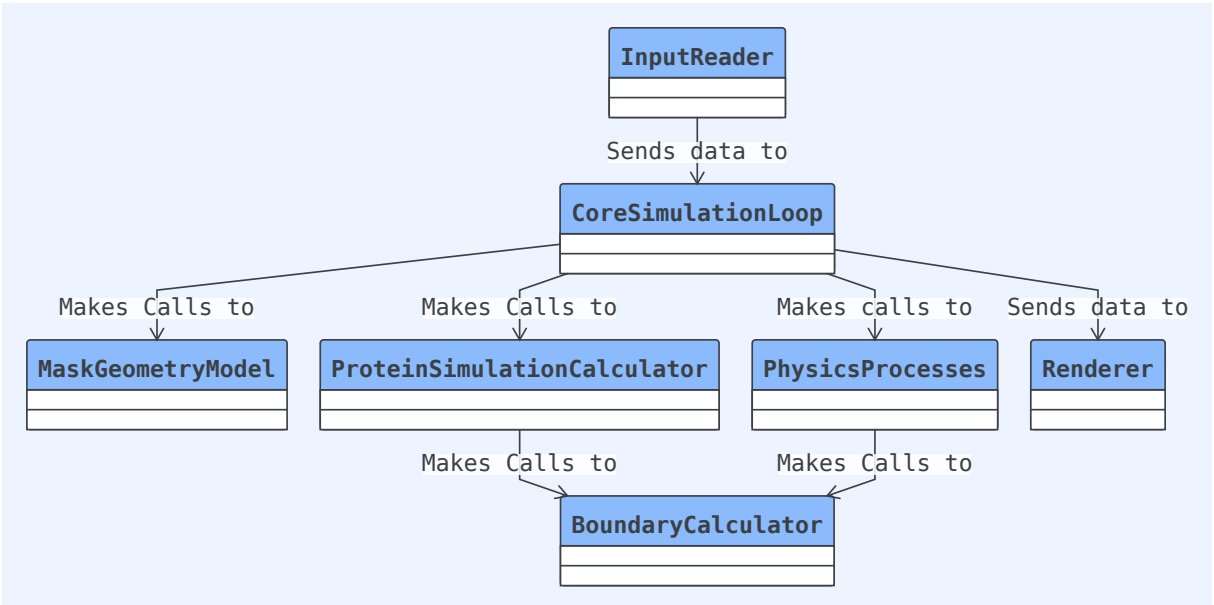


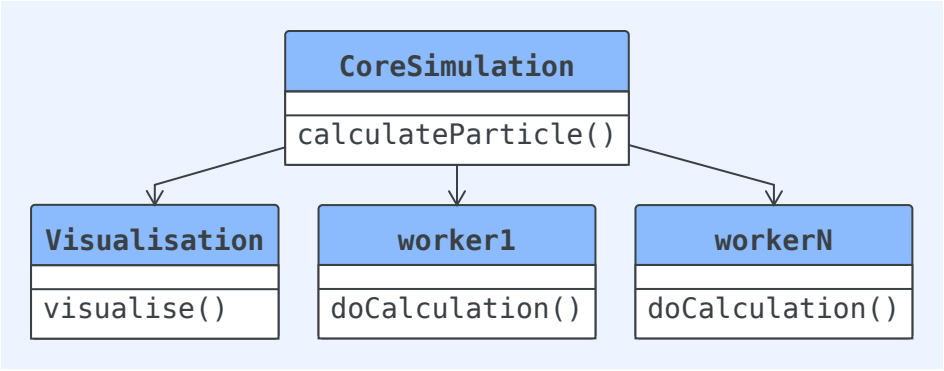
Figure 19: Codesense Architecture Diagram

4.5.6. 4+1 Diagram

4.5.7. Logical View



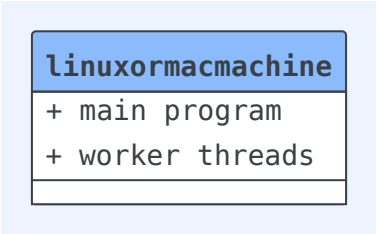
4.5.8. Process View



4.5.9. 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

4.5.10. Physical View



4.5.11. SonarQube Diagram

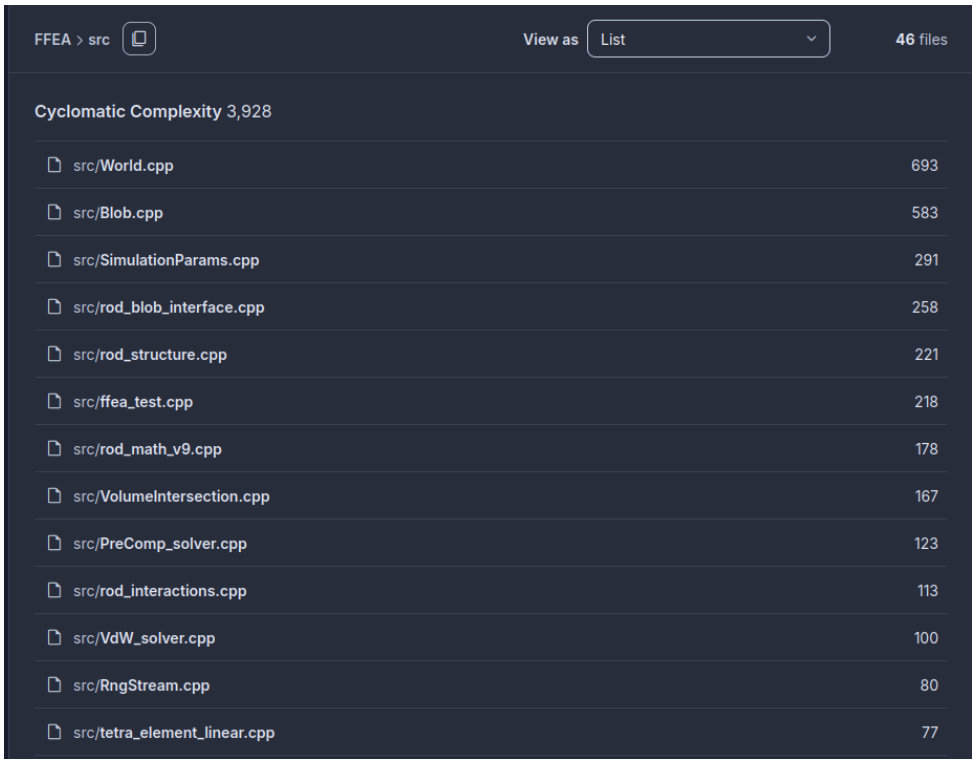


Figure 20: SonarQube Complexity Diagram

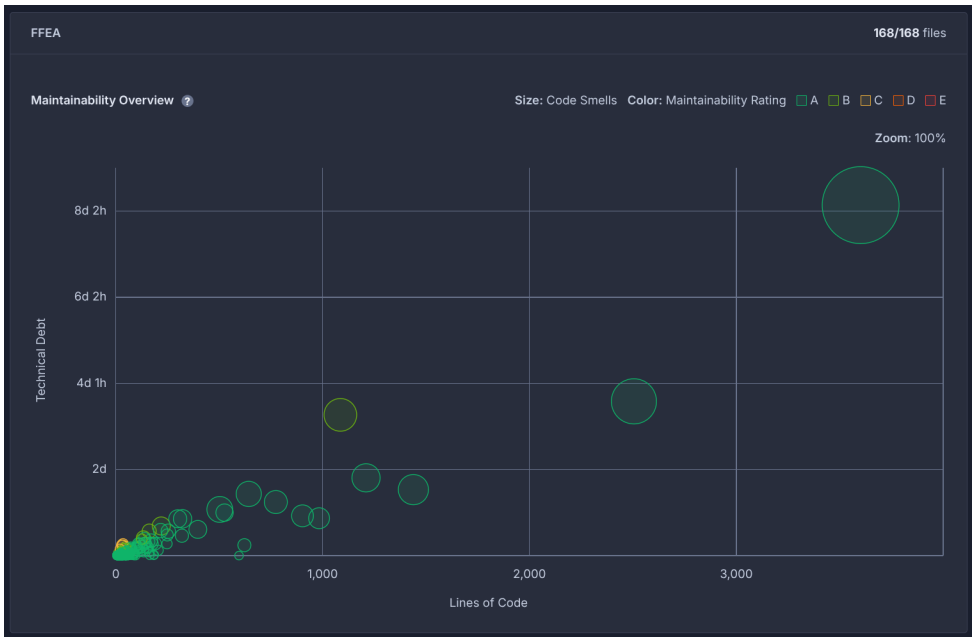


Figure 21: Understand Function Analysis

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

4.5.12. DV8

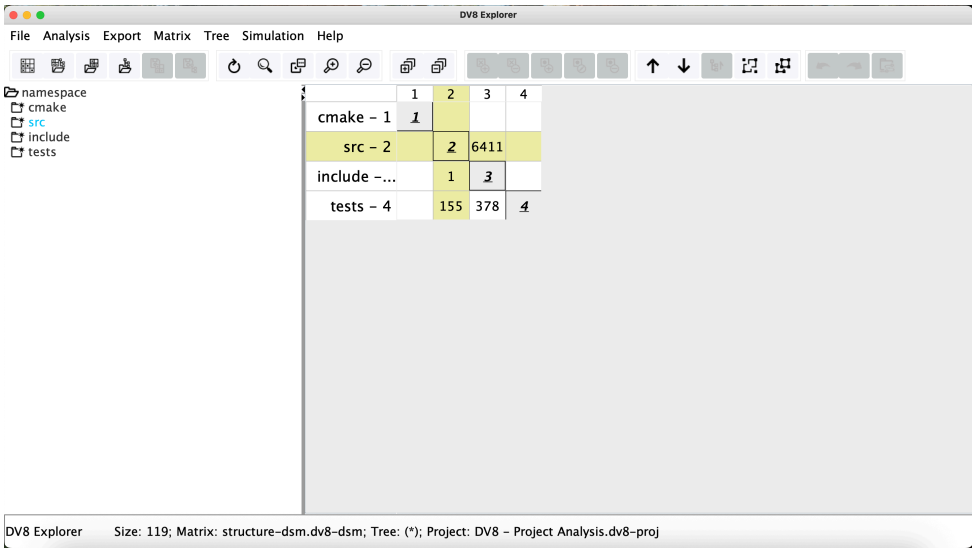


Figure 22: Understand Function Analysis

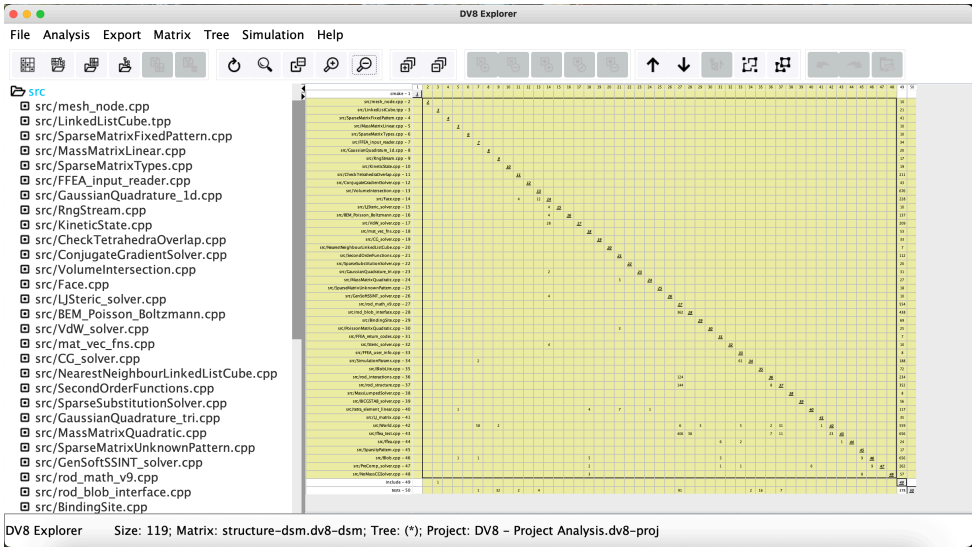


Figure 23: Understand Function Analysis

4.5.13. Understand

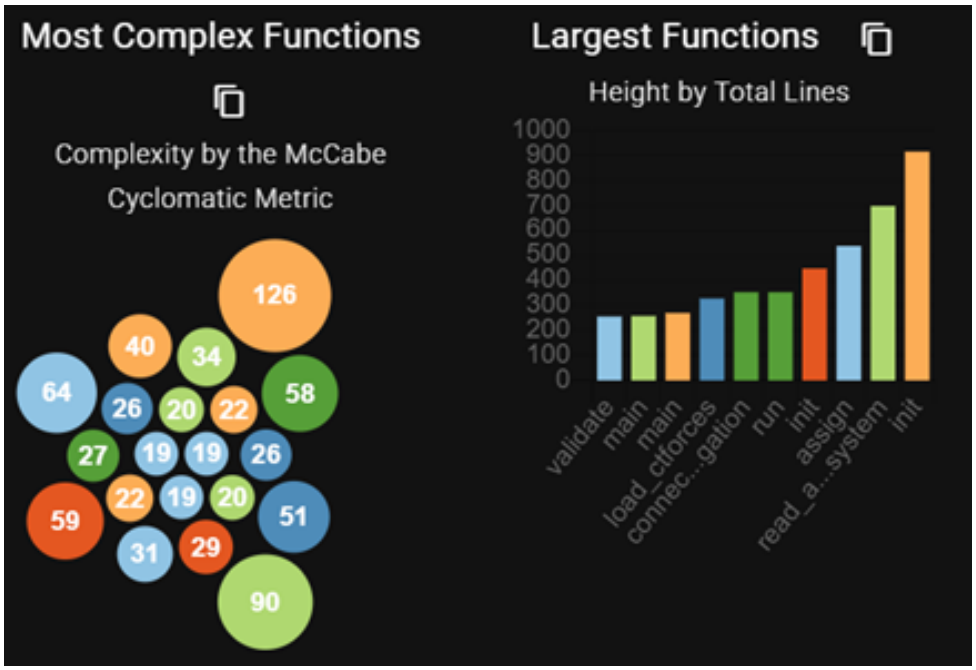


Figure 24: Understand Function Analysis

Figure 25: Understand Function Analysis

4.5.14. C4 Diagram

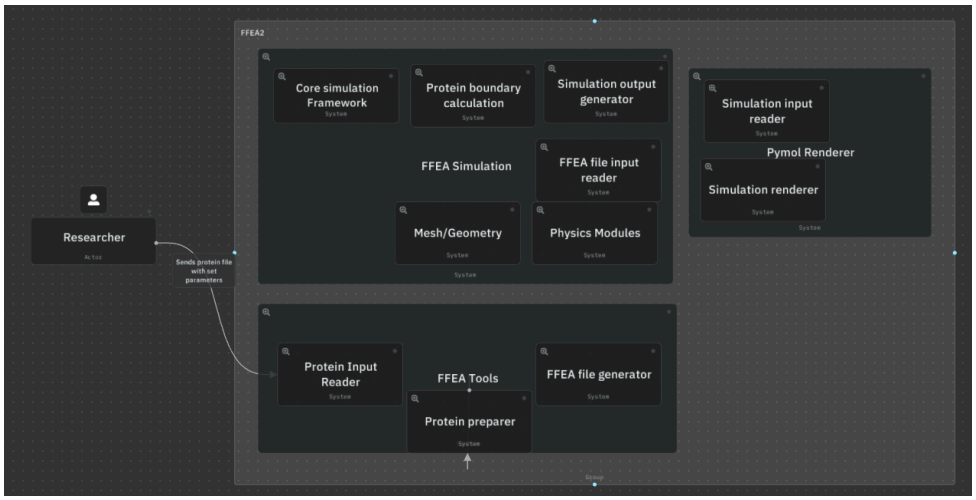


Figure 26: C4 Diagram

4.5.15. Dependency Graph

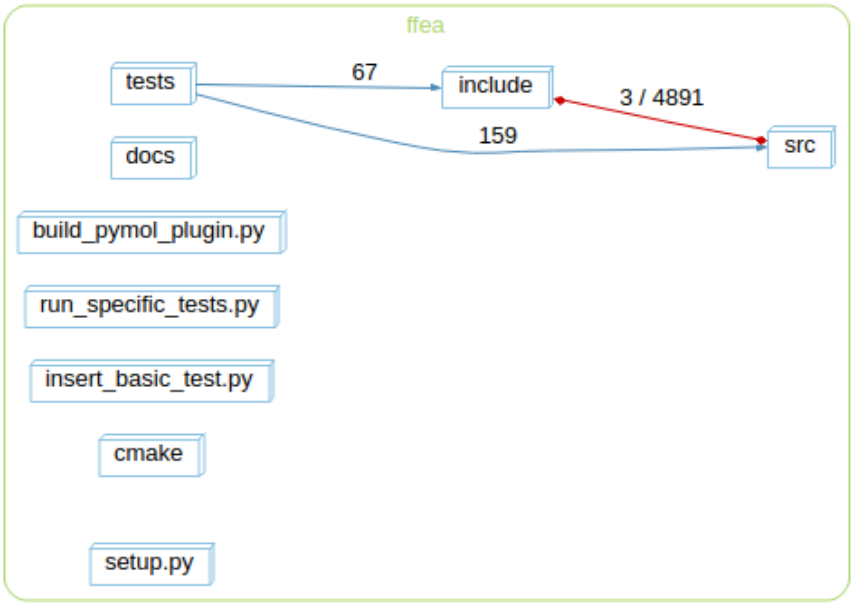


Figure 27: Dependency Graph

4.5.16. Class Diagram

Figure 28: FFEA Class Diagram

5. iDavie Case Study

5.1. Current State

IDavie is primary focused on creating 3d renderings of various inputs that can be explored in VR with data masking Listing 8. A key feature of this being the interactivity between the users and the environment which was created ad-hoc. The current architecture has been reconstructed by the current developers Figure 31. As seen in the diagram, the architecture is disorganised with a disproportionate amount of emphasis put on the DataManager. This accidental architecture came to be due to ever flowing requirements and fixes being applied one on top of the other to the codebase.

Currently the code is in such a state that adding a feature such as particles over the current astronomy rendering requires a fork and a fundamental restructuring to complete. A contributing factor is the high coupling of evolutionary dependencies [3] Figure 32. Maintainability of the codebase is nonexistent and for pull requests, manual testing by the means of a google form must be completed. Showing at a foundational level that standard practice is not being followed allowing additional architectural deterioration.

5.2. Proposed Architecture

The key features IDavie needs are **useability**, **maintainability** and **extensibility** seen here Listing 9. The architecture that best fits the use case is a microkernel architecture [2]. A microkernel architecture is a system built around a single purpose, in this case the astrophysics simulation, while creating a plug-in system to introduce new functionality without changing the core code [2].

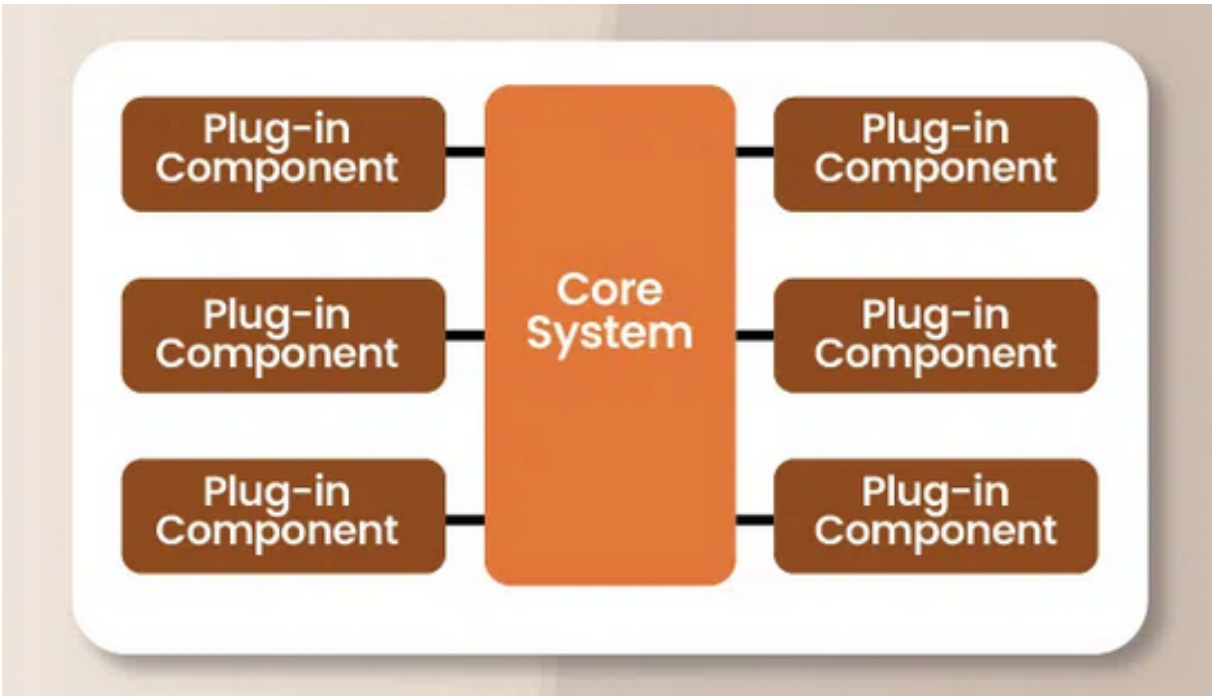


Figure 29: MicroKernel Architecture Diagram

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.
- As the kernel will be the focus of development **performance** can be improved more easily.
- Removing the plugins from the core developer responsibilities improves **maintainability** as only the core code must be maintained.

- The plugin system allows for development **extensibility**, allowing new features without changing the core code.

5.3. Reasoning

IDavie at its core it a 3D volumetric rendering tool in VR that allows user interaction seen in this use case diagram Figure 30 [4]. To continue expanding into new fields such as particle simulation, it needs a structure that prioritises controlled extensibility.

The microkernel architecture would create a core simulation, regulating features such as astrophysics or medical imaging to plug-ins [4]. Therefore changing the identity of IDavie to be a platform to allow all kinds of volumetric rendering, allowing developers to contribute independently without disrupting the core system.

In practice maintainability and usability would be core tenants of the new architecture. The static core would reduce the maintenance load on the team as well as allow for a planned consistent user interface as well as other core features to be developed that would not change based on ad-hoc new features.

5.4. Conclusion

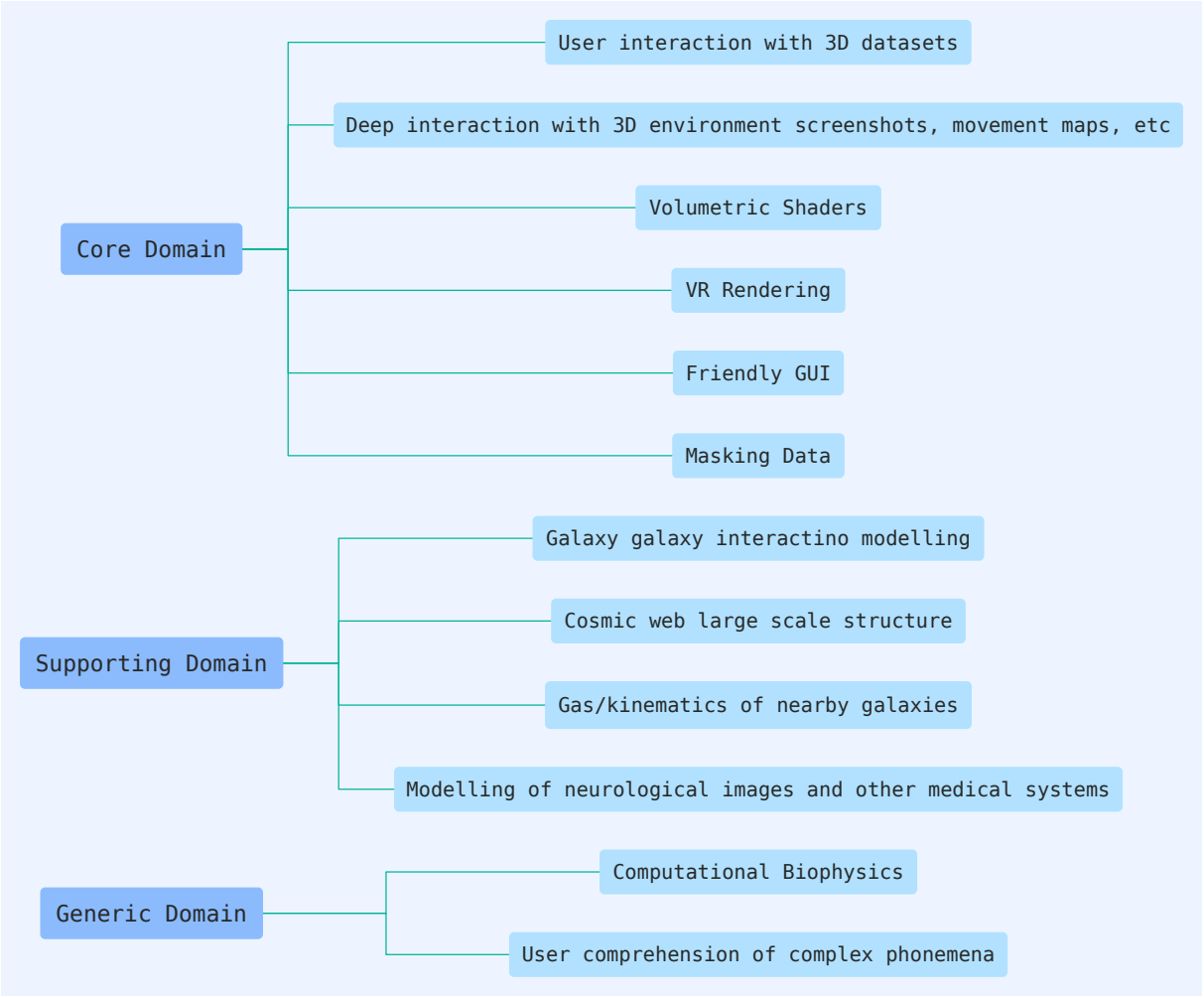
In the long term, transitioning to a microkernel design would change IDavie from a tightly coupled tool into a robust, maintainable, extensible platform. It would enable consistent development on the core features, while encouraging new developers to add functionality through plugins. This shift would not only improve current performance and usability but also ensure IDavie can continue to grow as new rendering technologies and scientific needs emerge.

5.5. Diagrams

5.5.1. Tech Stack

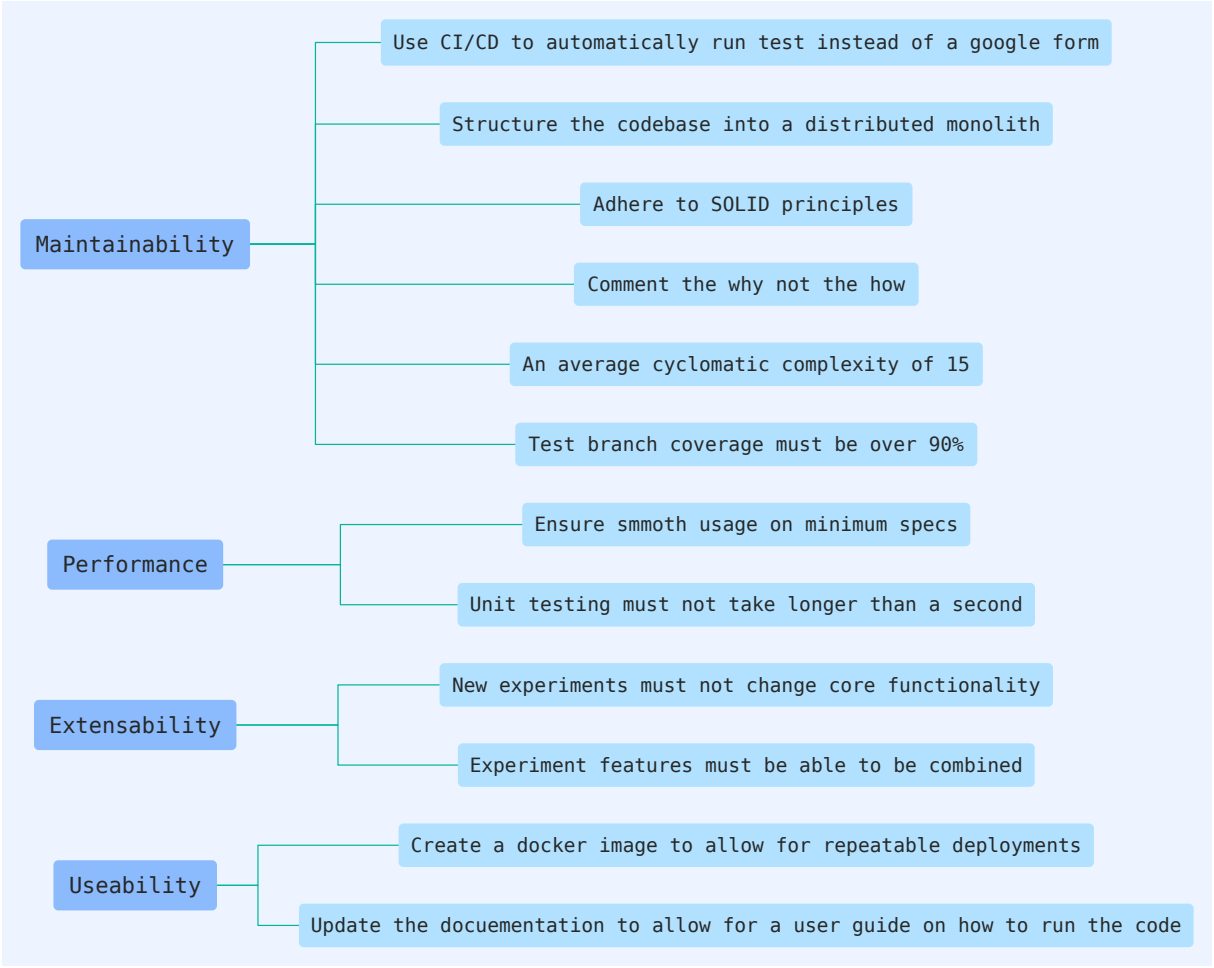
- Renderer : Unity with C#
- Interaction Framework : Steam VR
- Simulation Framework : C++ with eigen and boost
- Tests : Ctest with python add ons

5.5.2. Domain Model



Listing 8: Domain Model

5.5.3. Utility Tree



Listing 9: Utility Tree

5.5.4. Use case diagram

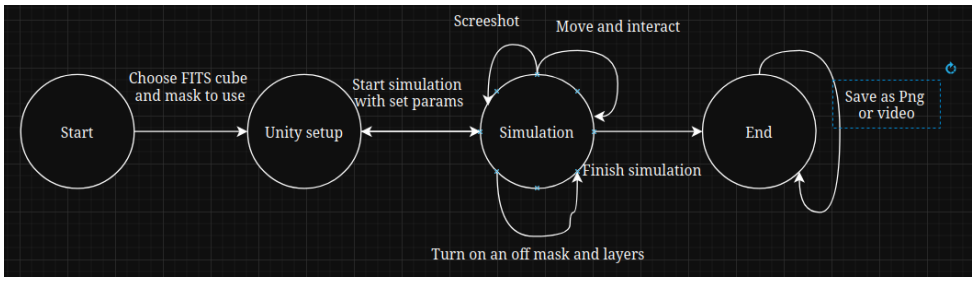


Figure 30: Use case state machine

5.5.5. 4 + 1 Diagram

5.5.6. Logical View

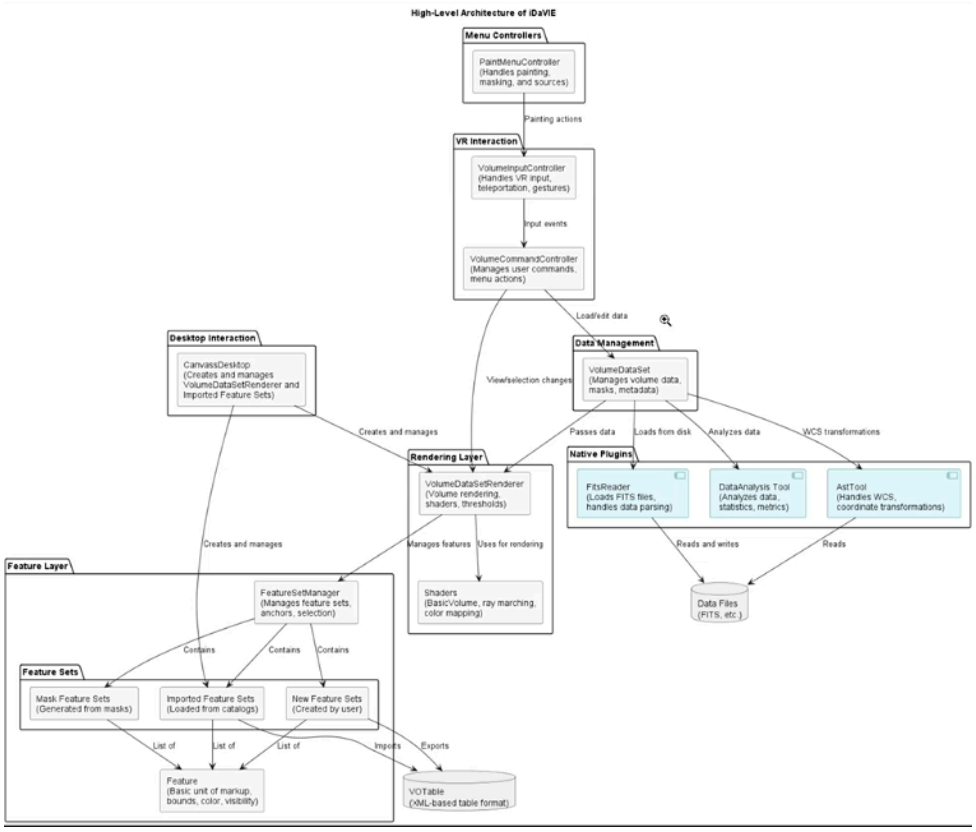
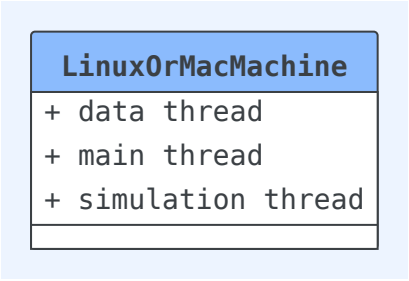
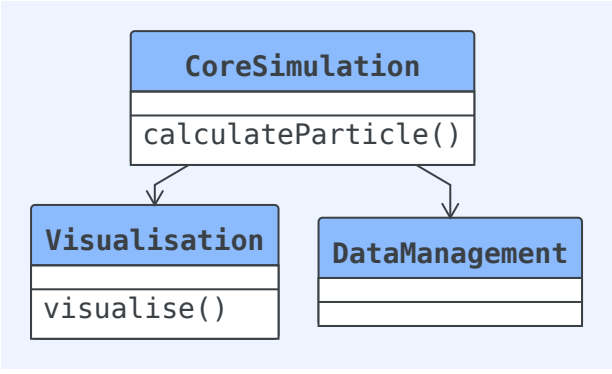


Figure 31: Logical View

5.5.7. Physical view



5.5.8. Process View



5.5.9. 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

5.5.10. Codescene

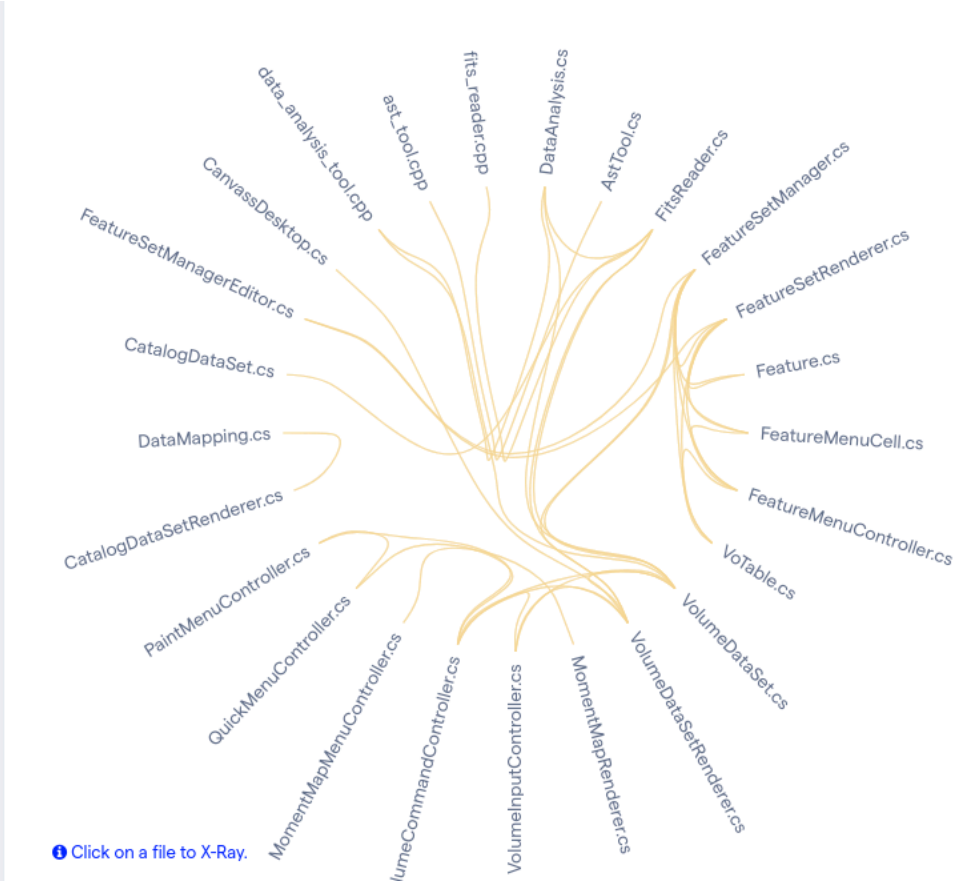


Figure 32: Codescene coupling diagram 40%

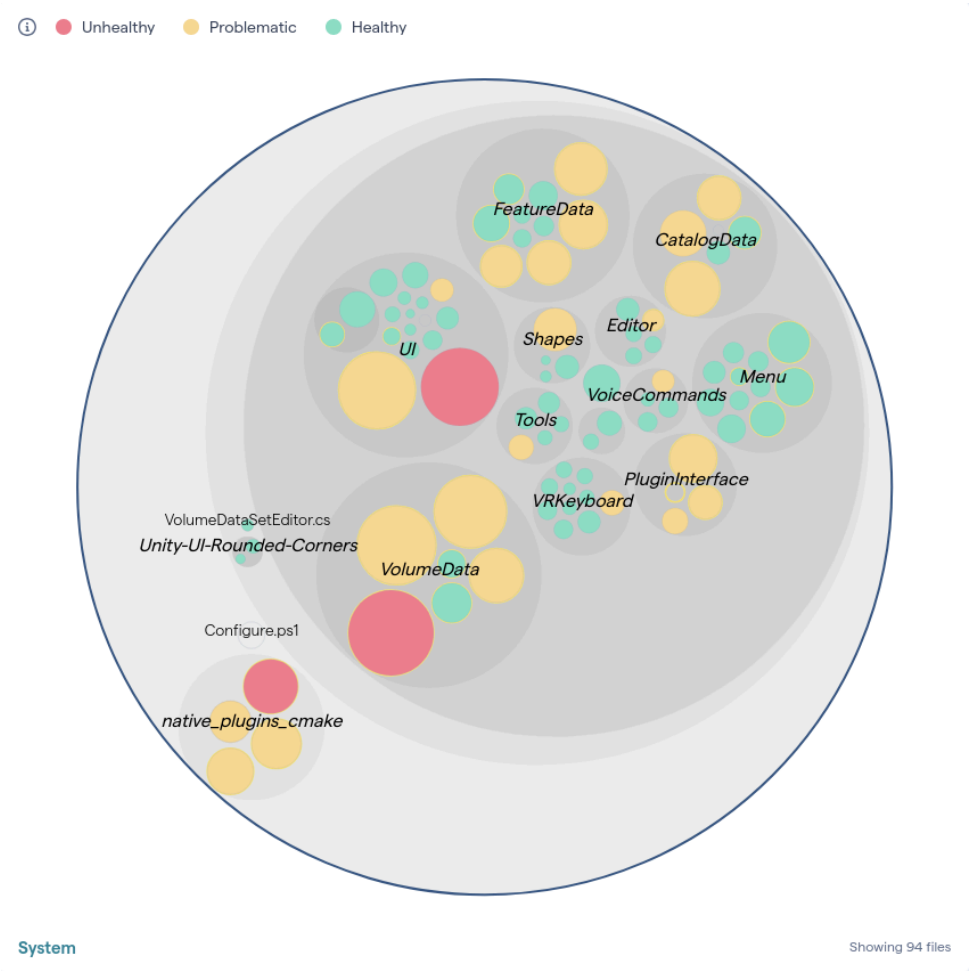


Figure 33: Codescene Maintenance Diagram

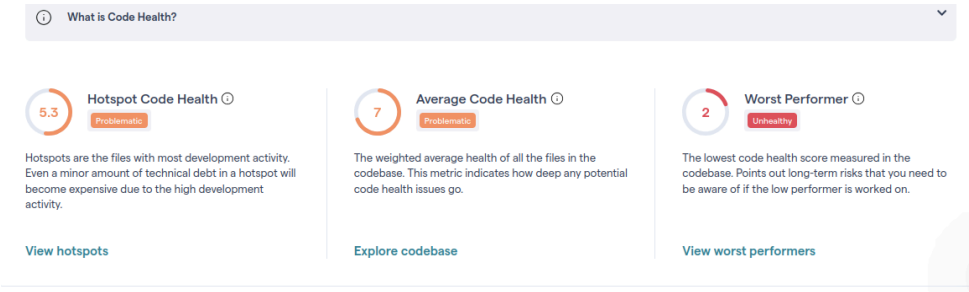


Figure 34: Codescene Hotspot overview

5.5.11. Sonarqube

Figure 35: Sonarqube cyclomatic complexity

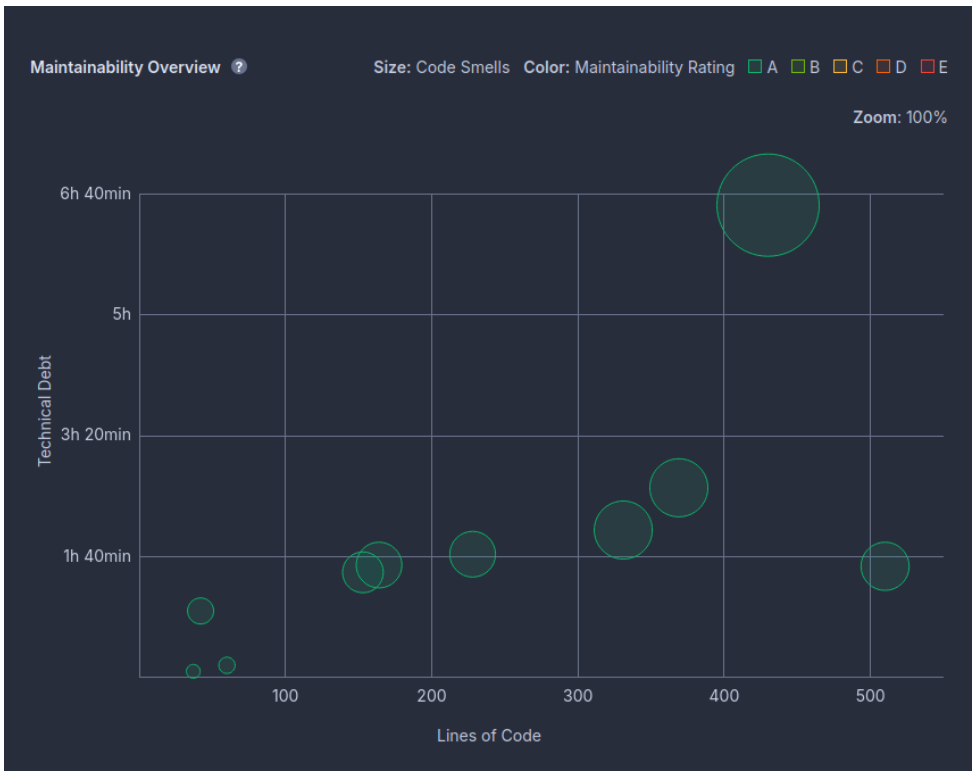


Figure 36: Sonarqube maintainability

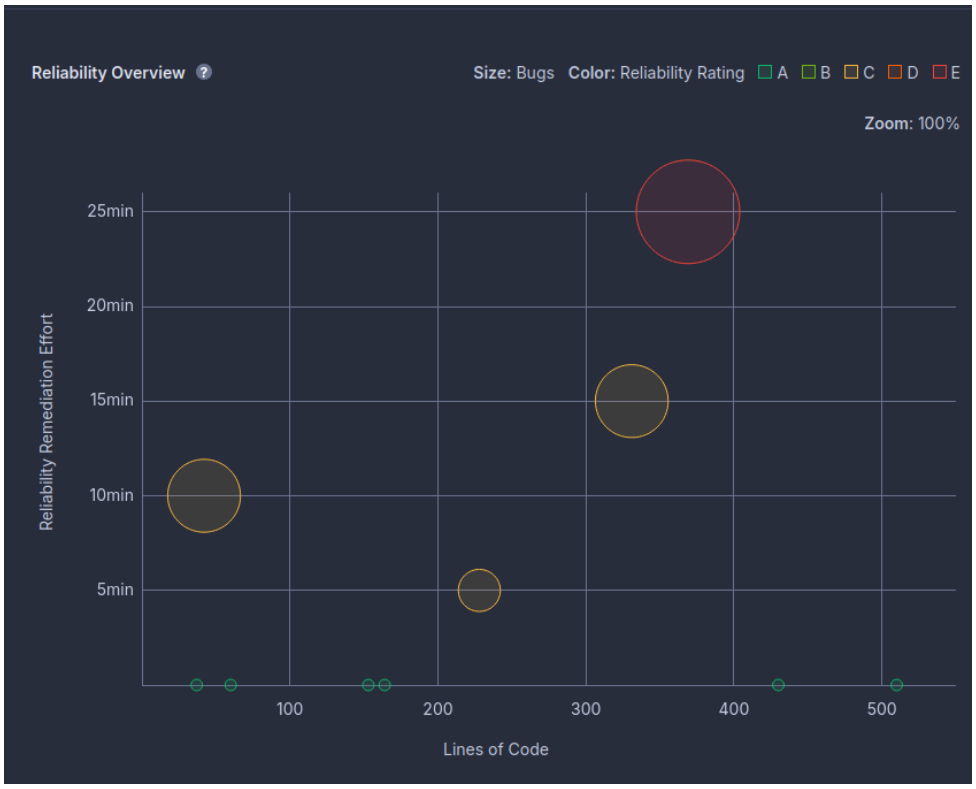


Figure 37: Sonarqube reliability graph

Figure 38: Duplications Graph

5.5.12. Understand Code Analysis

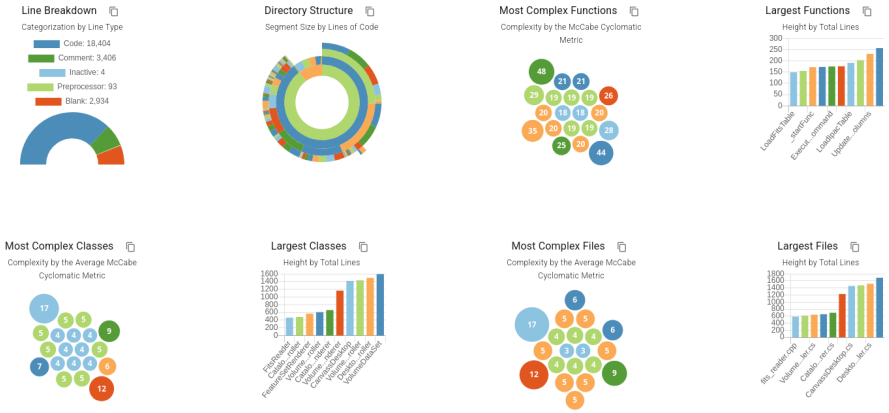


Figure 39: Understand IDavie

5.5.13. Class Diagrams

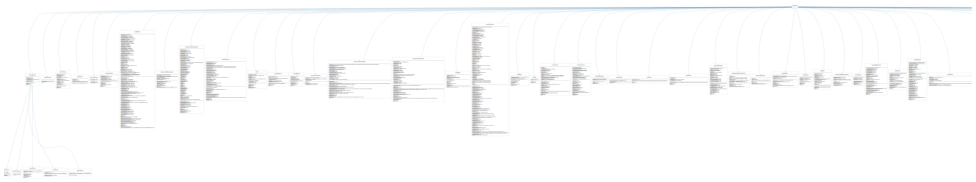


Figure 40: FLat class diagram Idavie

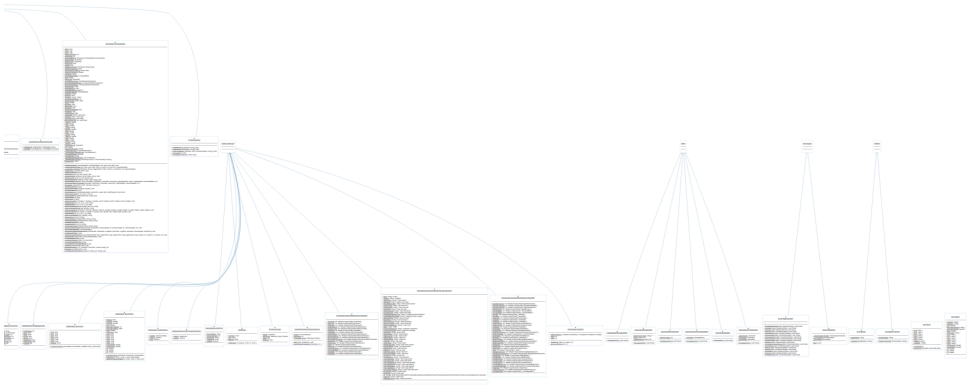


Figure 41: Hierarchical Class diagram Idavie

5.5.14. C4 Diagram

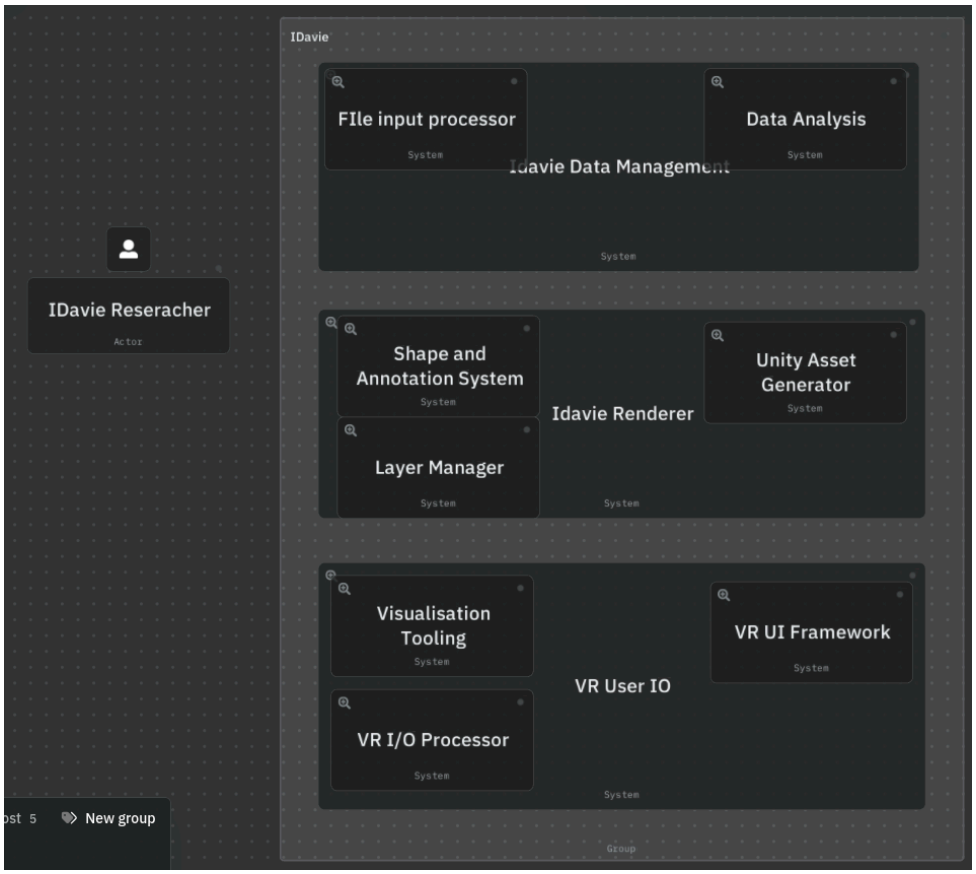


Figure 42: C4 Diagram

6. ACTS Case Study

6.1. Current State

ACTS is a particle tracking software library built on the ATLAS common track as a base. [5] The current goal of the software is to provide high quality particle tracking in a performant modular way, captured in this domain model Listing 10.

The current architecture is closest to a microkernel architecture [2], this fits well for the use case focusing development on static core to the codebase. The primary issues with the codebase are:

- Some plugins are moving to be a core dependency which is mentioned in the documentation, introducing architectural drift [3].
- The current codebase is still dealing with technical debt from building upon the ATLAS codebase. Figure 50
- There are examples which are being called integration tests taking a majority of the codebase Figure 45
- There is extremely high coupling 90% across many components these evolutionary dependencies slow down features severely Figure 44

6.2. Proposed Architecture

To address these issue I believe a modular monolith with purely functional APIs is the best approach. A regular monolithic structure would have one single, indivisible unit whereas a modular monolith takes this approach but breaks the structure down into logical high level groupings which could be represented as containers in a C4 diagram Figure 55.

The benefits this brings are:

- Strong grouping of components making any coupling explicit and allowing changes in components without altering others similar to how a layered architecture.
- The functional interfaces would simplify parallelism and improve performance.
- It still allows extension in the form of plugins, as new plugins can add new modularity without changing core behaviour
- A shared runtime ensures the entire codebase is on the same dependencies not allowing version drift over time between plugins

6.3. Reasoning

The principle goal for acts is to model particle tracking, with the key qualities for usage being performance and reliability, while maintainability is at the forefront for developers Listing 11. The current microkernel-style design has worked well for flexibility, but the growing coupling and architectural drift show that the boundaries between the kernel and plugins have blurred over time. A functional modular monolith aligns directly with these priorities.

Grouping related code into cohesive components naturally reduces cross cutting dependencies [6] and encapsulates the code logically. Similar to a layered architecture this separation of concerns would encourage concurrent development without the overhead of distributed complexity. The shared monolithic runtime also helps avoid version drift between plugins/components and ensures consistent performance across all modules, something that's critical for simulation reproducibility.

Adopting purely stateless function calls through the functional paradigm allows for users to expect consistent **deterministic** results regardless of internal state or previous function calls. This characteristic directly parallels the deterministic nature of physical simulations; given identical inputs, the output must always be consistent. Such a design not only supports reproducibility and testability but also aligns the software's logical model with the scientific principles. It also allows for

safe parallel execution and reproducibility across hardware configurations, which are essential for CERN-scale computations.

6.4. Conclusion

Long term this architecture strikes a balance between maintainability, performance and precision. Enabling the library to grow without fragmenting into disparate components or becoming tightly coupled to a repository wide cross cutting dependency. Ensuring ACTS will remain a strong developing library for high speed particle tracking in the long term.

6.5. Diagrams

6.5.1. Tech Stack

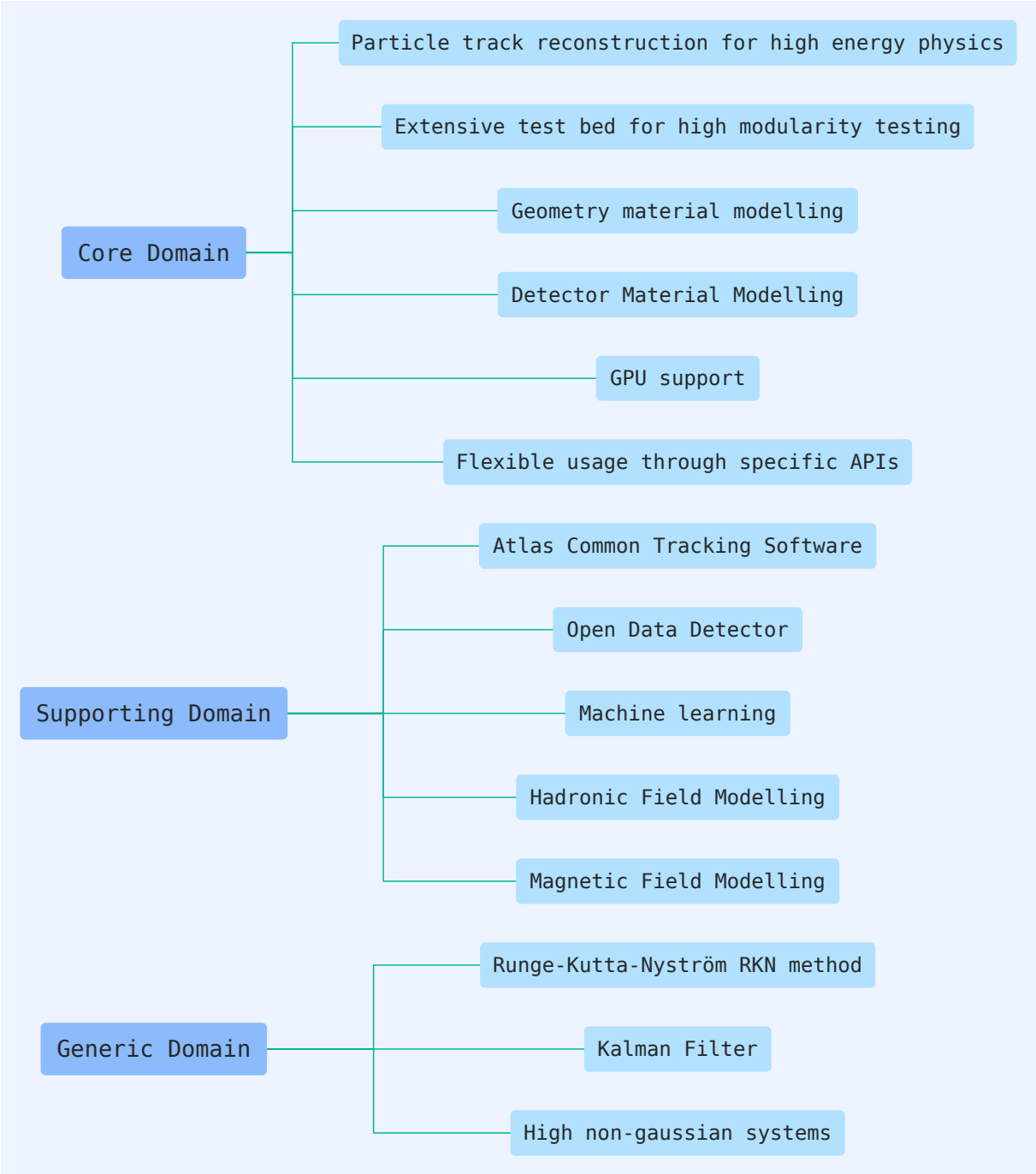
Main Language: C++

Heavily used external libraries: Eigen, boost, cmake

Used for example scripts: Python bindings using pybind11

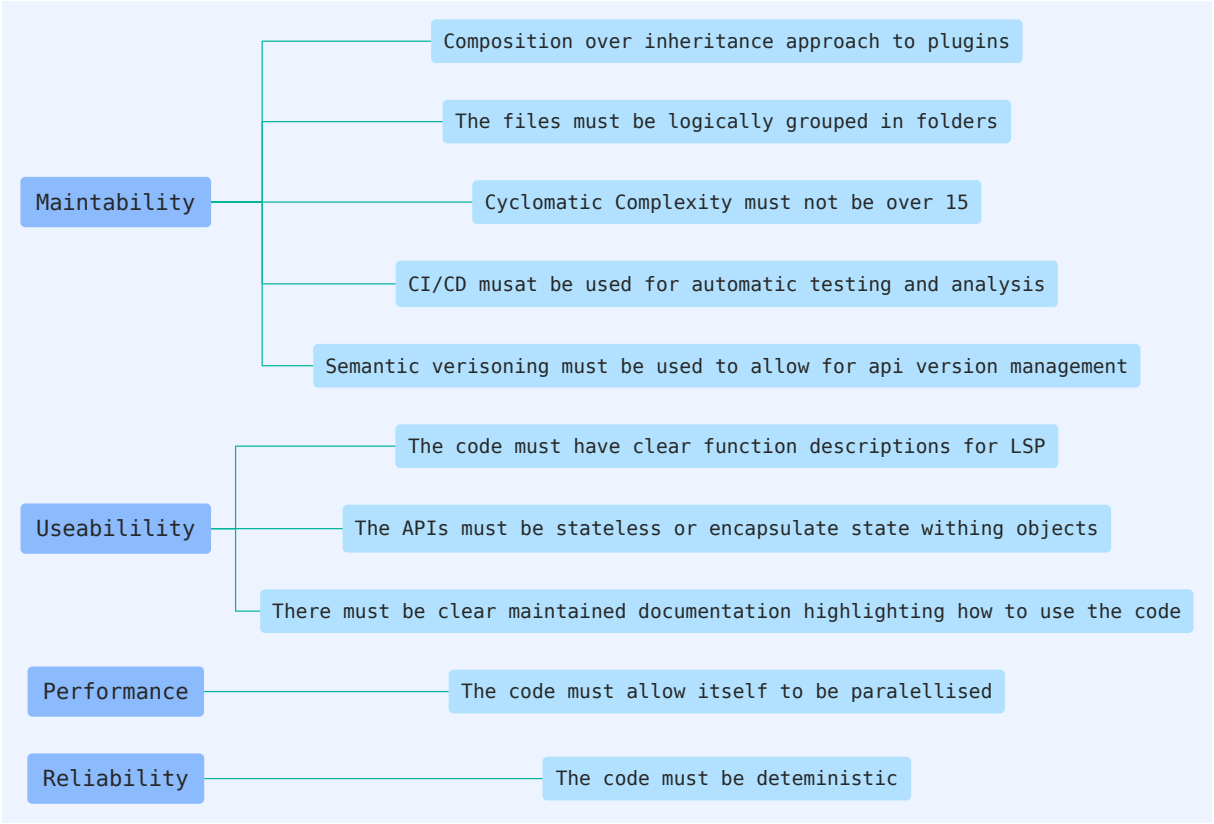
Containerisation: Docker

6.5.2. Domain Model



Listing 10: Domain Model

6.5.3. Utility Tree



Listing 11: Utility Tree

6.5.4. Use Case Diagram

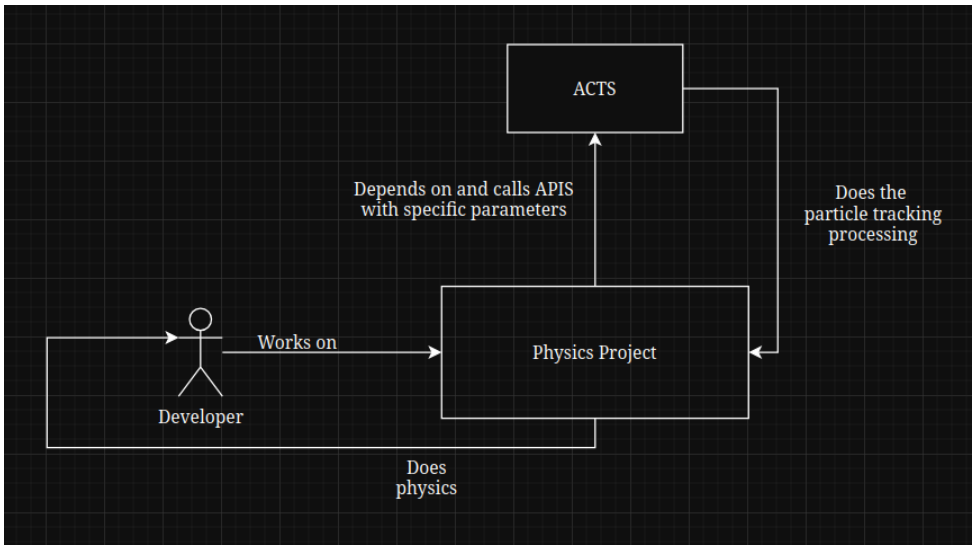
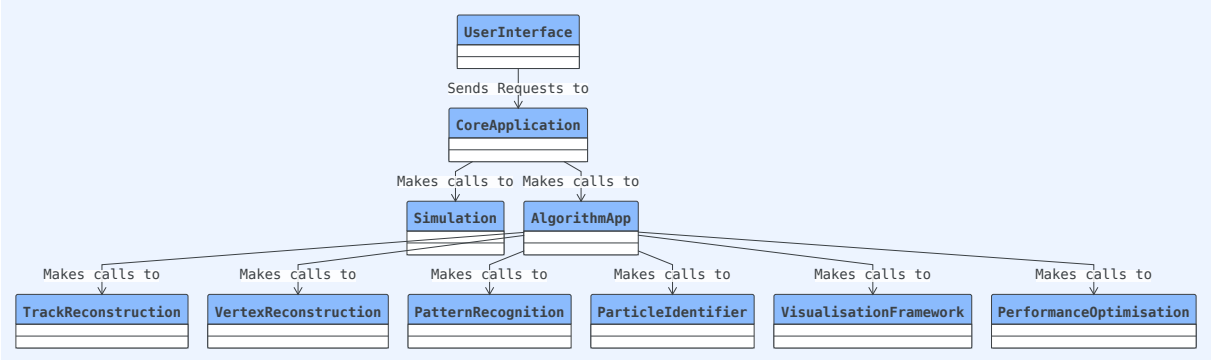


Figure 43: Use Case Diagram

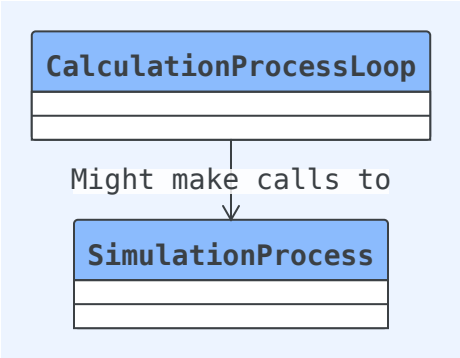
6.5.5. 4+1 Diagram

6.5.6. Logical View



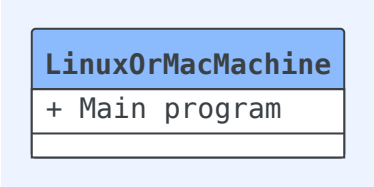
Listing 12:

6.5.7. Process view



Listing 13: Process view

6.5.8. Physcial View



6.5.9. 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

6.5.10. Codescene

Figure 44: Dependency Coupling Graph

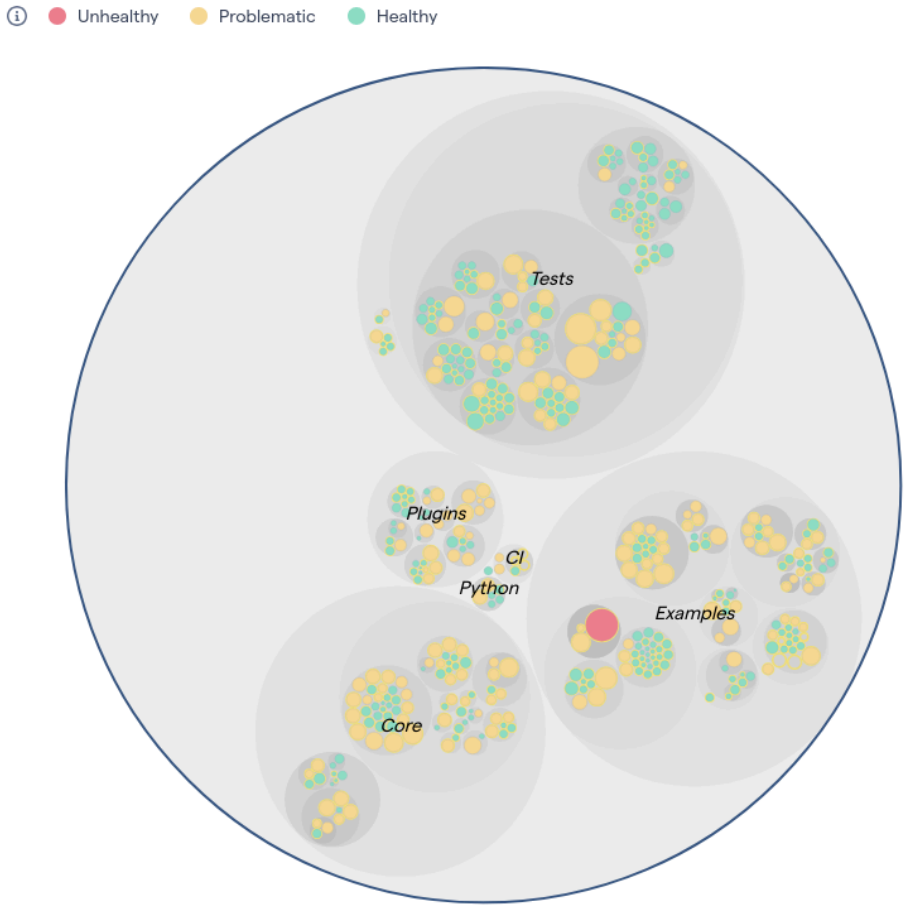


Figure 45: Technical Debt Codescene

6.5.11. Understand

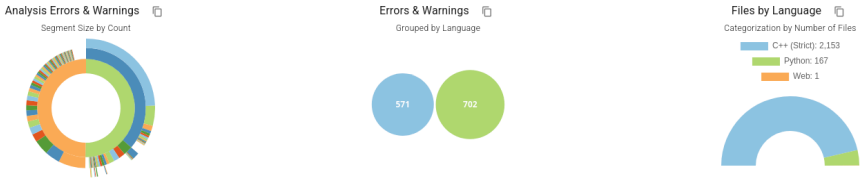


Figure 46: Understand High Level Info

Figure 47: Understand complexity ratings

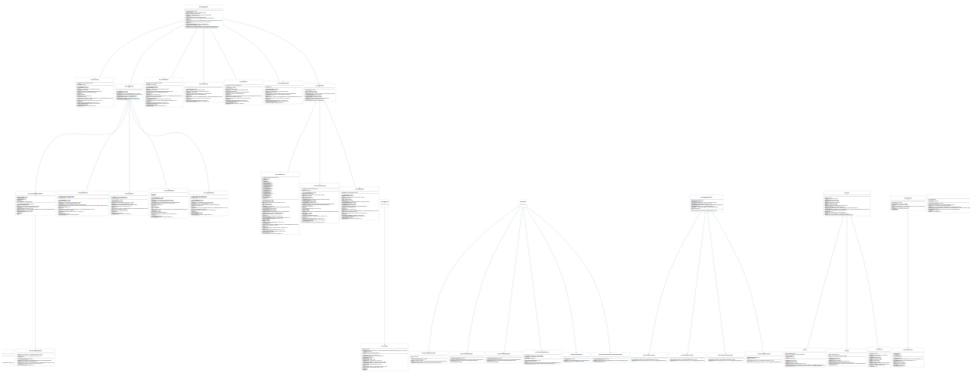


Figure 48: Class Diagram Inheritance

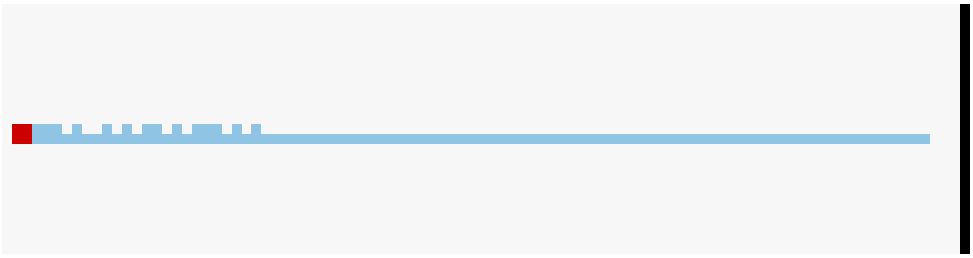


Figure 49: Class Diagram flat

6.5.12. Sonarqube

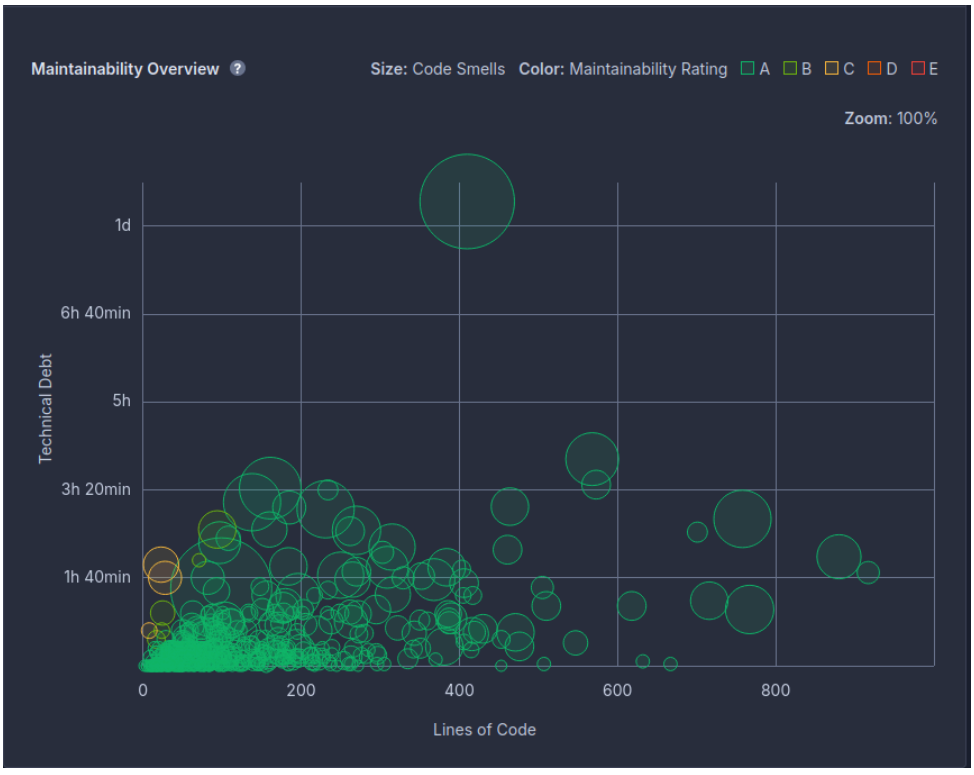


Figure 50: Maintainability Overview

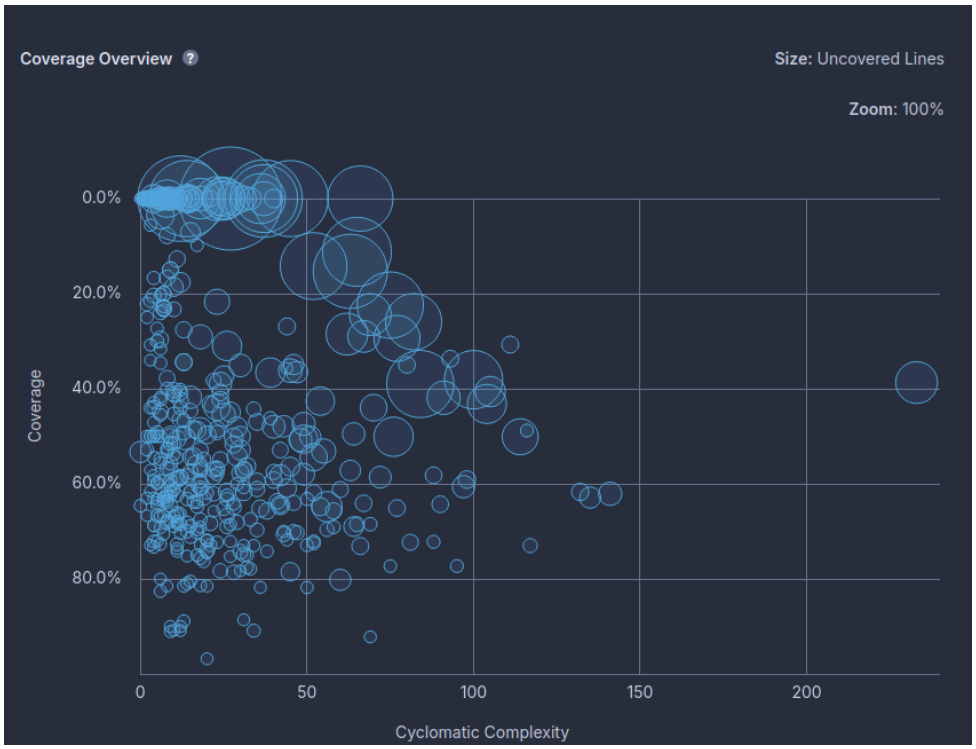


Figure 51: Cyclomatic Complexity overview

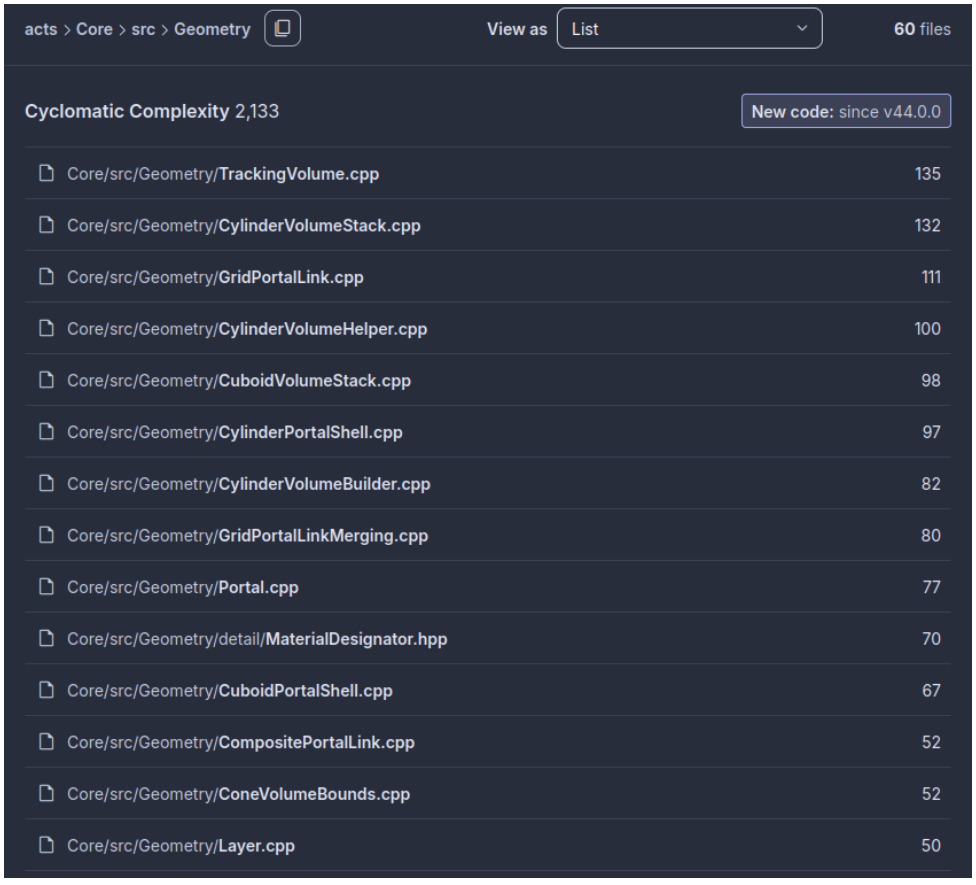


Figure 52: Cyclomatic Complexity files

6.5.13. C4

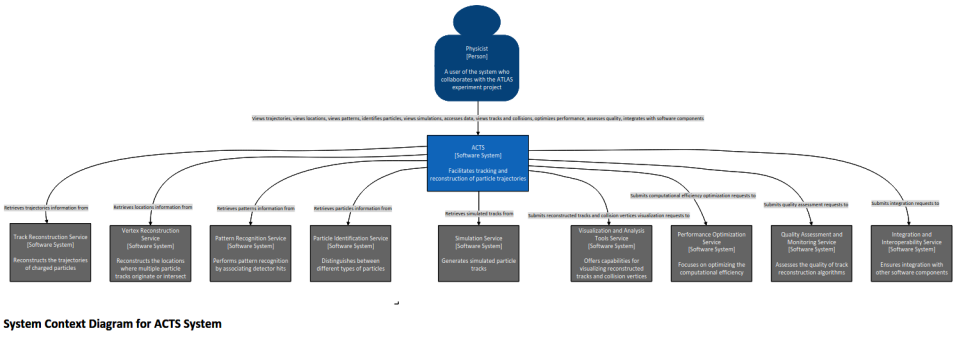


Figure 53: Context Diagram

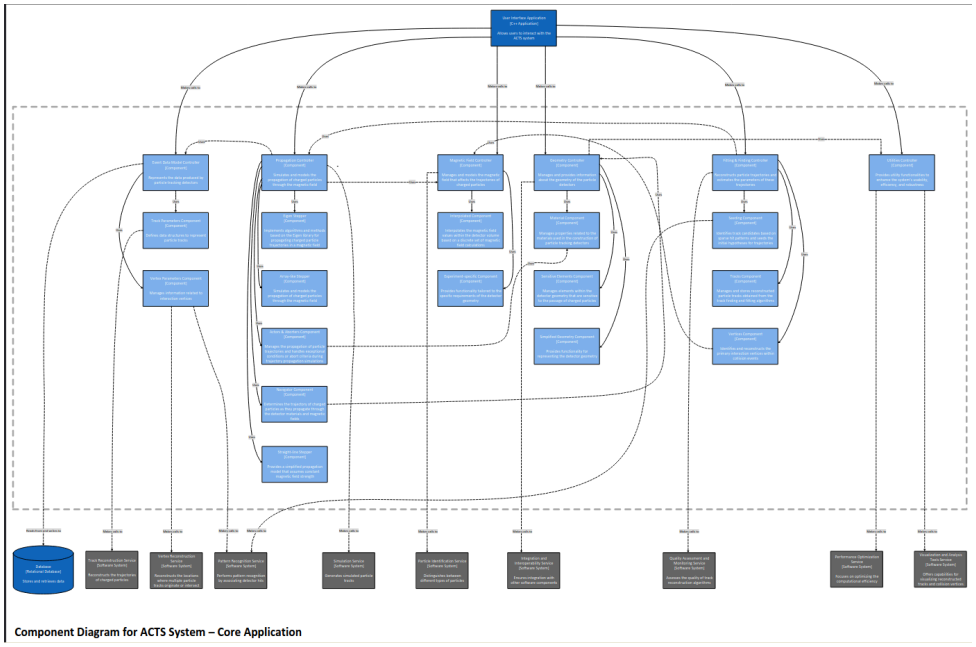


Figure 54: Component Diagram

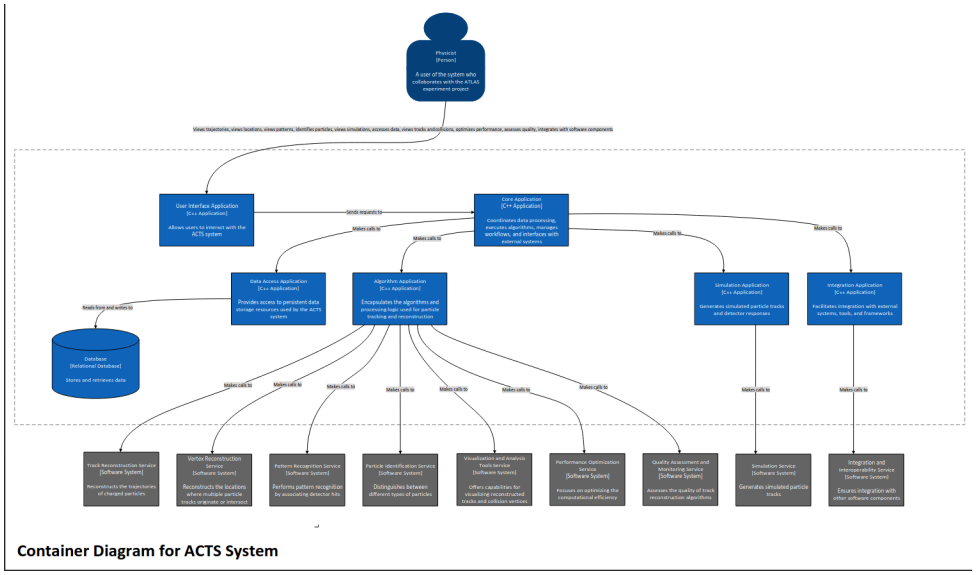


Figure 55: Container Diagram

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] Microsoft Corporation, Ed., "NET application architecture guide." in Patterns & Practices. Microsoft Press, Redmond, Wash., 2009.

[3] N. Ernst, R. Kazman, and J. Delange, *Technical Debt in Practice: How to Find It and Fix It*. 2021. doi: 10.7551/mitpress/12440.001.0001.

[4] "Introduction — iDaVIE 1.0 documentation." Accessed: Oct. 23, 2025. [Online]. Available: https://idavie.readthedocs.io/en/latest/introduction.html

[5] "ACTS Common Tracking Software — Acts documentation." Accessed: Oct. 24, 2025. [Online]. Available: https://acts.readthedocs.io/en/latest/

[6] M. Gharbi, A. Koschel, and A. Rausch, *Software Architecture Fundamentals: A Study Guide for the Certified Professional for Software Architecture® – Foundation Level – iSAQB compliant.* Heidelberg: dpunkt.verlag, 2019.