SE 4485: Software Engineering Projects

Spring 2025

Detailed Design Documentation

| Group Number | 9 |
| --- | --- |
| Project Title | QNX and Python Implementation |
| Sponsoring Company | Communications and Power Industries |
| Sponsor(s) | Luke Carr |
| Students | 1. Muhammad Ali sxa220115@utdallas.edu  2. Nicholas Anderson nba220001@utdallas.edu  3. Tabark Abaid tya210000@utdallas.edu  4. Diego Ibarra dxi200004@utdallas.edu  5. Saghar Abdi sxa190148@utdallas.edu  6. Khaled Elkhaled kye210001@utdallas.edu |

QNX and Python Implementation Detailed Design Documentation

**ABSTRACT**

This document presents a comprehensive software design specification, outlining the graphical user interface (GUI), static and dynamic models, and detailed design rationale. It includes class and sequence diagrams to illustrate system behavior and structure, ensuring traceability from requirements to implementation. Additionally, the document addresses configuration management practices, adherence to engineering standards, and compliance with IEEE Std 1016-1998 for software design. References to relevant literature and industry guidelines are provided to support the design methodology.

**TABLE OF CONTENTS**

[Abstract](#) 2

[Table Of Contents](#) 2

[List Of Figures](#) 3

[List Of Tables](#) 3

[Introduction](#) 3

GUI (Graphical User Interface) Design

Static Model

Dynamic Model

Rationale for your Detailed Design Documentation

Traceability from Requirements to Detailed Design Model

Evidence the Design Model has been placed under Configuration Management

Engineering Standards and Multiple Constraints

Additional References

LIST OF FIGURES

1. Static Model
2. Dynamic Model

**INTRODUCTION**

This document serves as a detailed software design specification, providing a structured approach to the development of the system. It outlines the architectural and design decisions, ensuring that the system meets its functional and non-functional requirements. By following industry standards and best practices, this document aims to facilitate the implementation, maintenance, and future enhancements of the system.

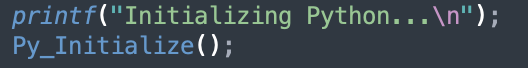
GUI (Graphical User Interface) Design

* Implementation 1:
* **Step 1: Starting the C program**

The C program initiates execution and prepares the system to interface with Python. It creates the required conditions to use Python functions from inside a C program.

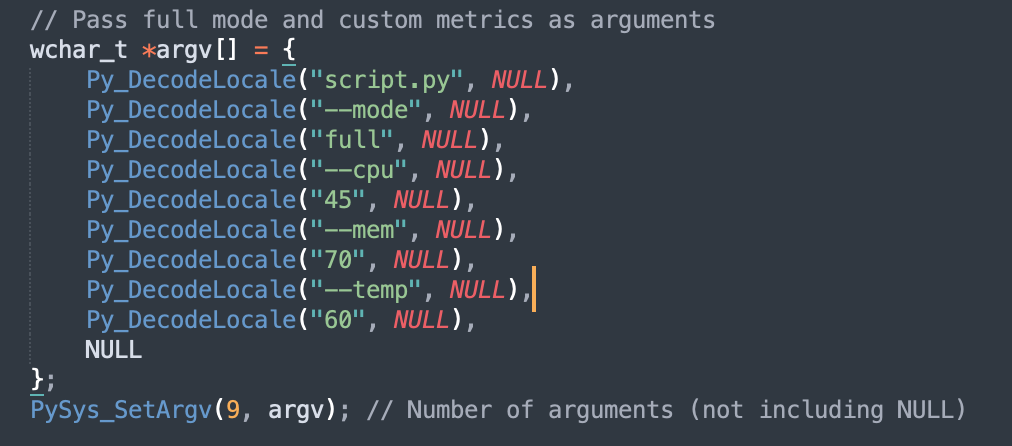
* **Step 2: Initializing Python**

The Python interpreter has to be initialized within the C program before any Python code can be run on the system. The Py\_Initialize() function performs the initiation of the Python interpreter and prepares the environment for the code to be run.



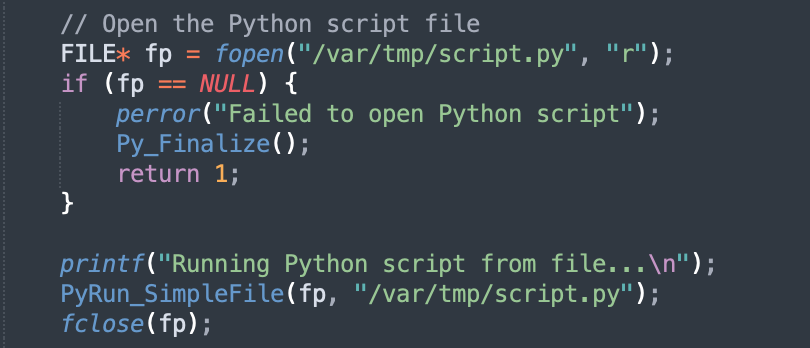
* **Step 3: Passing Arguments**

The C program then will pass command-line arguments to the Python script. These arguments determine how the program would behave. The PySys\_SetArgv() function mimics the process of sending command-line arguments to Python just as it would in a standard Python environment.



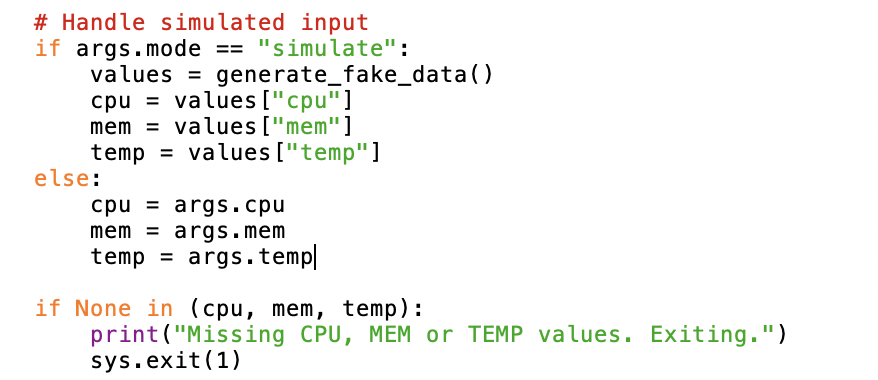
* **Step 4: Initiating the execution of the Python script**

Using PyRun\_SimpleFile(), the C program launches the Python script (script.py) and calls Python to run it. This function uses the initialized Python interpreter to read and run the script line by line.



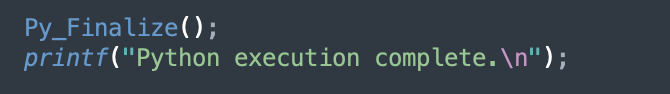
* **Step 5: Executing the Python script**

The Python script will process the arguments passed from the C program, such as: --mode, --cpu, --mem, --temp, etc... and use them to perform some action. The script will access these arguments through thePySys\_SetArgv function, which holds the command-line arguments.



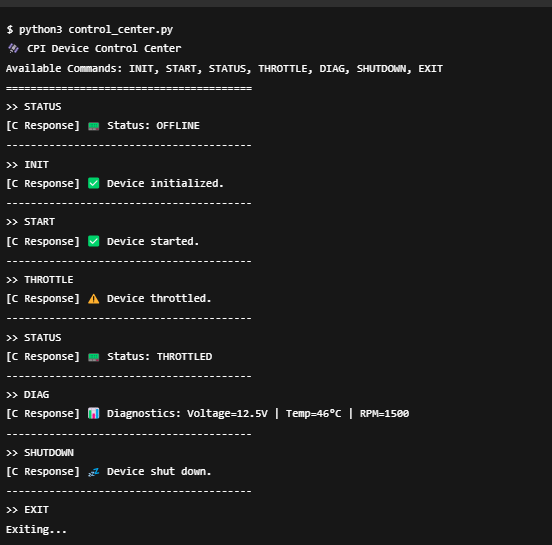
* **Step 6: Finalizing Python**

To properly terminate the Python interpreter after the Python script has completed running, the C program invokes Py\_Finalize(). This ensures the prevention of memory leaks or other issues that could arise if the interpreter is left running after the script has finished.

****

* Implementation 2:
* **Step 1: Launch the CLI**

The user executes (control\_center.py), launching a menu driven CLI that listens for typed commands like INIT, START, or STATUS. This interface is designed for real-time device control, particularly on embedded systems running QNX.

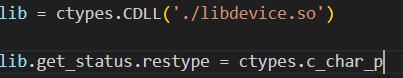


* **Step 2: Load the Library**

Using ctypes.CDLL(), Python loads the precompiled C shared object file (libdevice.so), exposing all the core C functions required to control the hardware device.

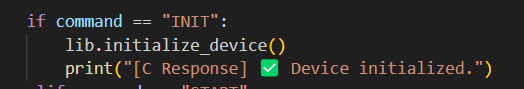
* **Step 3: Define Function Signatures**

Before invoking any C function, Python sets up argument and return types to ensure smooth and type-safe data exchange between the two languages.



* **Step 4: Handle User Commands**

Each typed command is mapped to a corresponding C function. For instance, the INIT command triggers initialize\_device(), and the output is displayed back to the user with helpful icons and feedback labels.

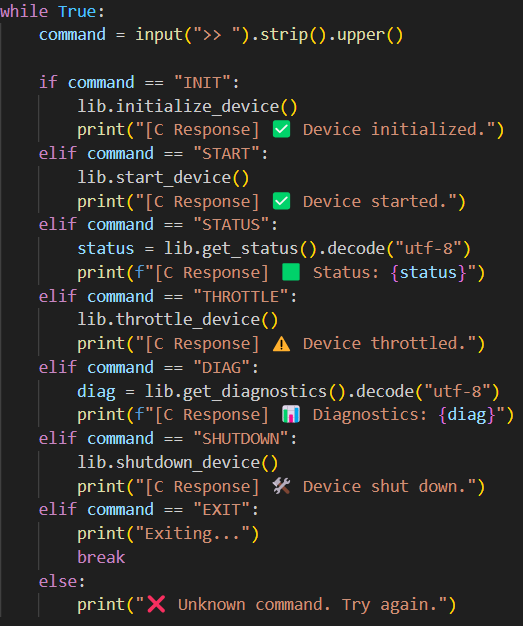


* **Step 5: Display System Feedback**

The output from the C functions is shown to the user in a clear, nicely formatted way with helpful icons, making it easier for engineers and testers to use.

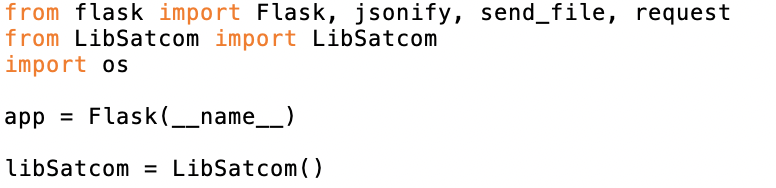
* **Step 6: Exit**

The program remains in a loop, continuously accepting and processing commands until the user enters EXIT, at which point it shuts down gracefully.



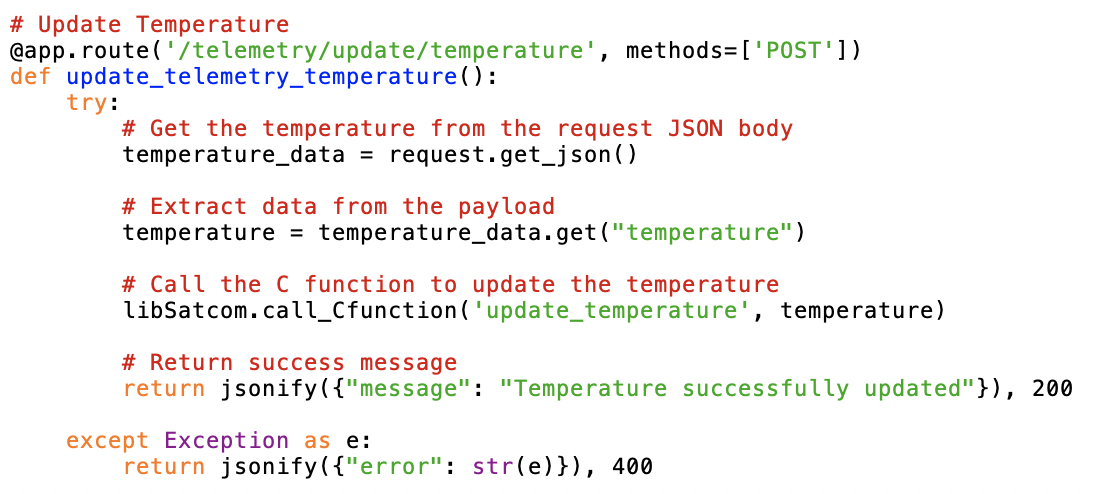
* Implementation 3:
* **Step 1: Setup and Initialization**

The api.py file uses Flask to set up the REST API. To establish the communication link with the LibSatcom class, we load the required libraries, and initialize the Flask application.



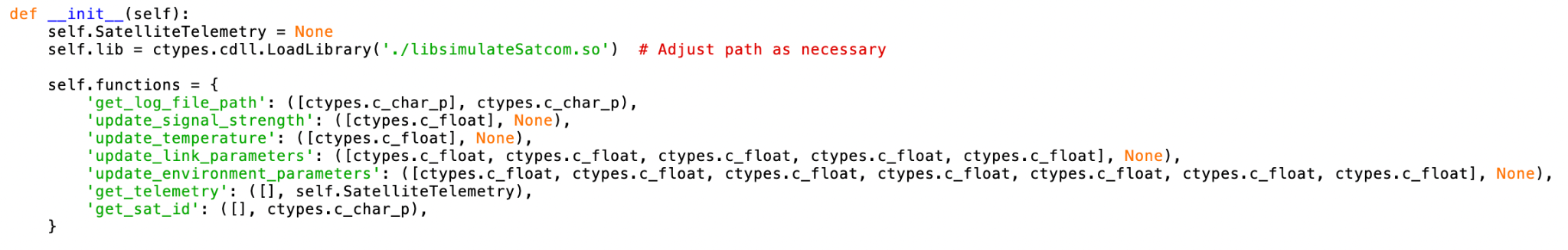
* **Step 2: Defining API Endpoints**

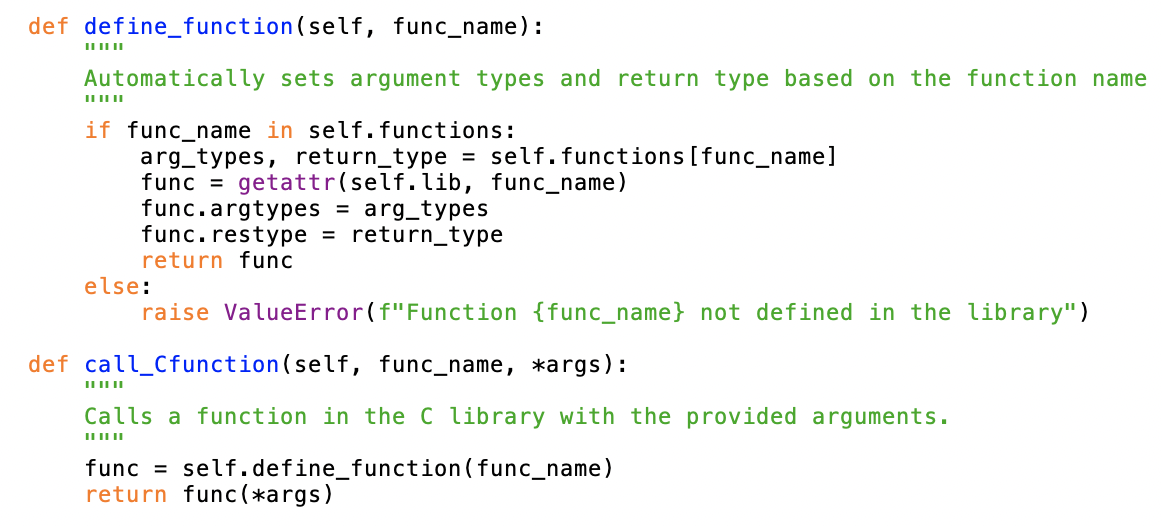
Here, we are creating routes in the REST API. Every route connects to a specific endpoint that users or external systems can use to communicate with the application. These routes will perform a variety of actions, such as receiving telemetry data, retrieving information, and updating satellite communication parameters. The following code snippet is how this part works on one parameter:



* **Step 3: Interaction between API and libsatcom**

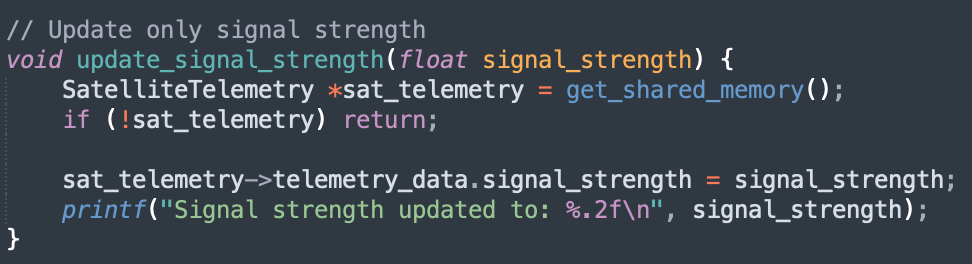
The API calls functions from libsatcom.py, which then interact with the C functions via ctypes to update or retrieve satellite telemetry data. The REST API endpoints receive requests, then they call functions from libsatcom and pass the received data to the respective C function that performs the action.

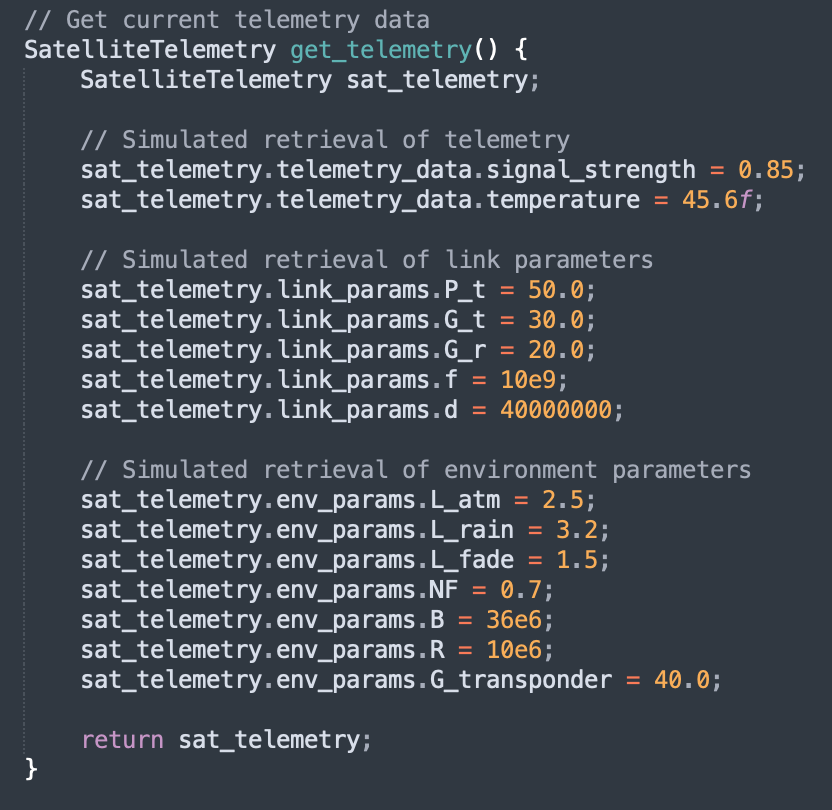




* **Step 4: Interactions between C and Python**

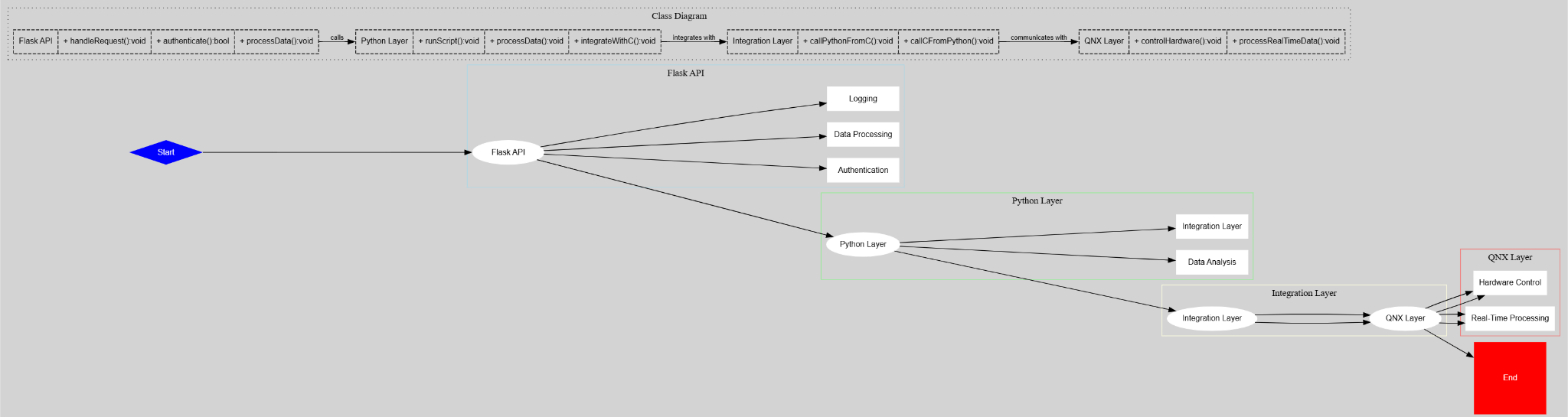
The C program, simulate\_satcom.c, provides functions that handle the telemetry tasks such as updating info and retrieving data. The python program, libsatcom.py, is the connection link between the Python code and the C functions. It uses ctypes to call the necessary C functions. The API invokes the C functions through the Python code and passes data from the API endpoints to the C functions. The C functions perform the operations and return the results to the python layer.





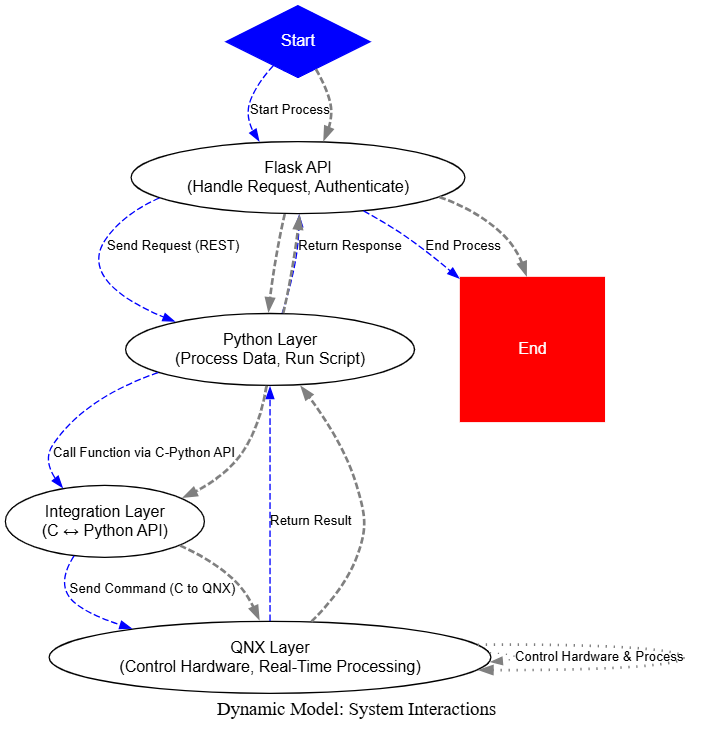
STATIC MODEL

CLASS DIAGRAMS



DYNAMIC MODEL

SEQUENCE DIAGRAMS



RATIONALE FOR YOUR DETAILED DESIGN MODEL:

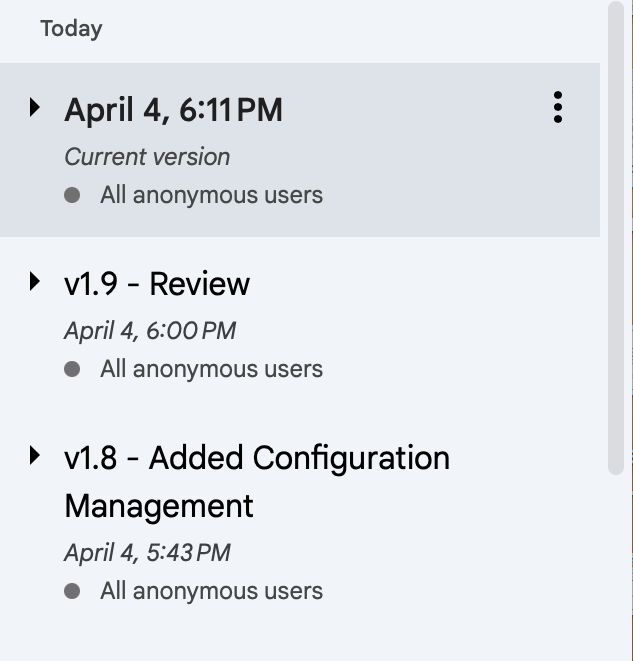
The static model is made up of modular layers (Flask API, Python Layer, Integration Layer, and QNX Layer) for maximum scalability and maintainability. This architecture provides maximum flexibility for modification as it allows low-level operations(QNX Operations) to be separated from high-level logic(Python). The integration layer enables seamless communication, with the Flask API providing standardized external access through RESTful endpoints. The dynamic model illustrates the ordered interactions between components that guarantee proper data flow and task execution. Here, request processing between the Flask API, Python, and QNX layers is operationalized for communication and real-time work.This model acts to ensure dependable execution and error management across the system.

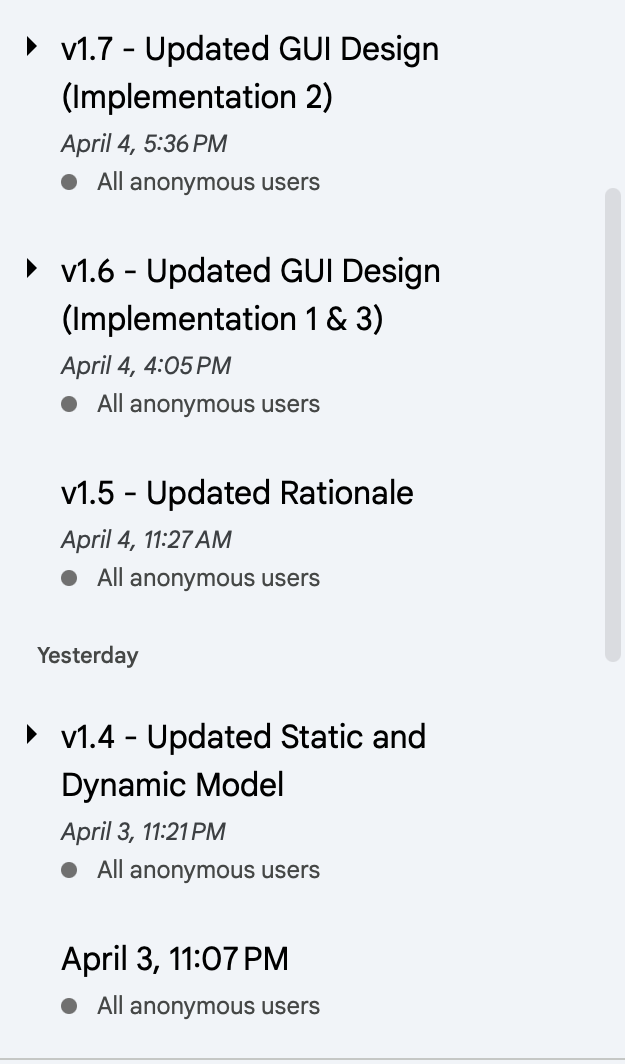
**TRACEABILITY FROM REQUIREMENTS TO DETAILED DESIGN MODEL**

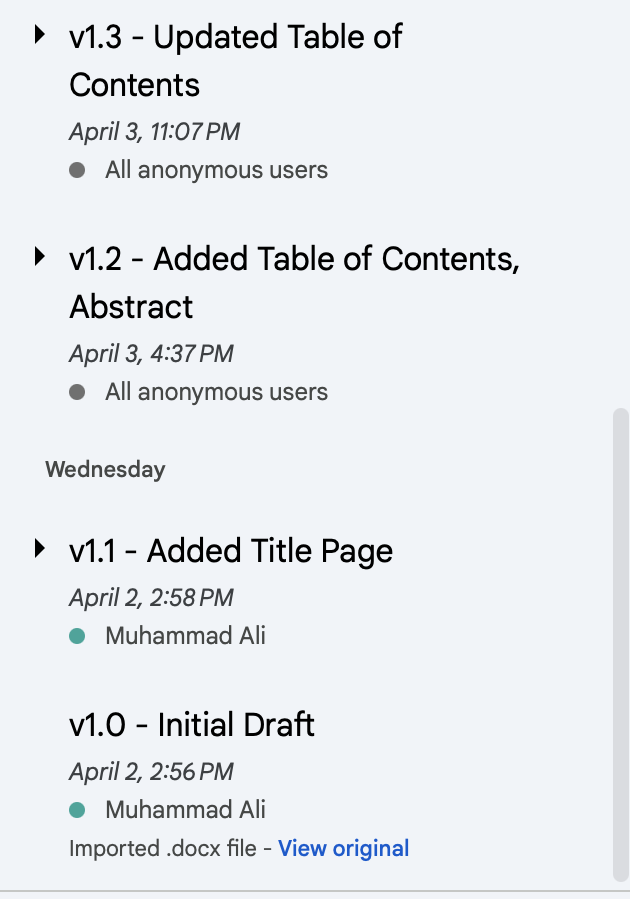
### 

| **Requirement ID** | **Description** | **Mapped Component** |
| --- | --- | --- |
| RQ-001 | Secure authentication | Authentication Service |
| RQ-002 | Data input and validation | Input Validation Module |
| RQ-003 | Report generation | Report Generation Service |
| RQ-004 | System performance optimization | Performance Optimization Strategies |
| RQ-005 | Security compliance | Security & Compliance Module |
| RQ-006 | Scalability and Modularity | Layered System Architecture |
| RQ-007 | Error Handling and Recovery | Fault-Tolerance Mechanisms |
| RQ-008 | Data Encryption | Encryption and Security Layer |
| RQ-009 | REST API for communication | API Gateway & Communication Layer |
| RQ-010 | System monitoring | Logging & Monitoring Service |

EVIDENCE THE DESIGN MODEL HAS BEEN PLACED UNDER CONFIGURATION MANAGEMENT







| Version | Date | Author | Changes Made | Previous Versions | Reviewers | Ship-it |
| --- | --- | --- | --- | --- | --- | --- |
| v1.9 | 4/04/2025 | Muhammad Ali, Nicholas Anderson,  Diego Ibarra | Review | v1.8 | Khaled Elkhaled, Tabark Abaid, Saghar Abdi | Approved |
| v1.8 | 4/04/2025 | Saghar Abdi, Diego Ibarra | Added Configuration Management | v1.7 | Nicholas Anderson, Muhammad Ali | Approved |
| v1.7 | 4/04/2025 | Diego Ibarra, Muhammad Ali | Updated GUI Design (2) | v1.6 | Saghar Abdi, Khaled Elkhaled | Approved |
| v1.6 | 4/04/2025 | Saghar Abdi | Updated GUI Design (1&3) | v1.5 | Tabark Abaid, Diego Ibarra | Approved |
| v1.5 | 4/04/2025 | Tabark Abaid | Updated Rationale | v1.4 | Nicholas Anderson, Saghar Abdi | Approved |
| v1.4 | 4/03/2025 | Tabark Abaid | Updated Static and Dynamic Models | v1.3 | Nicholas Anderson, Diego Ibarra | Approved |
| v1.3 | 4/03/2025 | Nicholas Anderson, Khaled Elkhaled | Updated Table of Contents | v1.2 | Saghar Abdi, Diego Ibarra | Approved |
| v1.2 | 4/03/2025 | Nicholas Anderson, Khaled Elkhaled | Added Table of Contents, Abstract | v1.1 | Muhammad Ali, Saghar Abdi | Approved |
| v1.1 | 4/02/2025 | Muhammad Ali | Added Title page | v1.0 | Diego Ibarra, Khaled Elkhaled | Approved |
| v1.0 | 4/02/2025 | Muhammad Ali | Initial Draft | - | - | - |

ENGINEERING STANDARDS AND MULTIPLE CONSTRAINTS

* + IEEE Std 1016-1998-(Revision-2009): Software Design [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-1016-1998-(Revision-2009)-Software-Design.pdf)]
  + Additional standards suggested by the sponsor(s)

ADDITIONAL REFERENCES

* + Larman, C., 2012. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development.* Pearson Education
  + Hyman, B., 1998. *Fundamentals of Engineering Design.* New Jersey: Prentice Hall
  + Simon, H.A., 2014. *A Student's Introduction to Engineering Design: Pergamon Unified Engineering Series* (Vol. 21). Elsevier
  + Additional references suggested by the sponsor(s)