SE 4485: Software Engineering Projects

Spring 2025

Architecture 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 Architecture Documentation

**ABSTRACT**

This document outlines the software architecture for our QNX and Python Integration project, developed in collaboration with Communications and Power Industries. The project focuses on connecting low-level system operations in C on a QNX real-time operating system with higher-level Python-based tasks like logging, data formatting, and communication. We’ve implemented three core components: calling Python from C, calling C from Python, and building a REST API using Flask. This document explains the overall design, technologies used, and how the architecture supports the project’s goals.

**TABLE OF CONTENTS**

[Abstract 4](#)

[Table Of Contents 4](#)

[List Of Figures 4](#)

[List Of Tables 4](#)

[Introduction 4](#)

[Architectural Style(S) Used 4](#)

[Architectural Model 4](#)

[Technology, Software, And Hardware Used](#) 4

Traceability From Requirements To Architecture 4

Evidence The Document Has Been Placed Under Configuration Management 4

Engineering Standards And Multiple Constraints 4

Additional References 4

# INTRODUCTION

This document provides an overview of the software architecture for the QNX and Python Integration project developed by Group 9 for SE 4485. It explains how the system is designed, what technologies are used, and how the different parts of the project work together.

The purpose of this document is to describe the architecture that supports the project’s goals and requirements. It focuses on three main parts of the project:

Calling Python code from C.

Calling C code from Python.

Creating a REST API using Flask to allow communication between the system and external tools.

This architecture is designed to make system-level operations more flexible, easier to manage, and accessible through a simple web interface.

The structure of the document is as follows:

* Architectural Style(s) Used
* Architectural Model
* Technology, Software, and Hardware Used
* Traceability from Requirements to Architecture.
* Evidence of Configuration Management
* Engineering Standards and Multiple Constraints
* Additional References

# ARCHITECTURAL STYLE(S) USED

In this project, we followed a combination of Client-Server and Layered Architecture styles. These styles helped us organize the system into manageable parts and made it easier to connect low-level system operations with high-level application logic.

Layered Architecture:

For our first and second implementation of just communication between C and Python, we built using a layered architecture. This design separates the system into different levels, each with its own responsibilities, making the system easier to manage, test, and update.

Hardware and QNX Layer (C code):

This is the bottom layer where core system tasks are handled. It runs on the QNX real-time operating system and takes care of low-level operations like data collection or processing.

Integration Layer (C ↔ Python Communication):

This layer connects the C and Python components. In one direction, it lets the C code trigger Python scripts for logging or reporting. In the other direction, it allows Python to access C functions for better performance. This two-way communication helps both parts of the system work together smoothly.

Application Layer (Python):

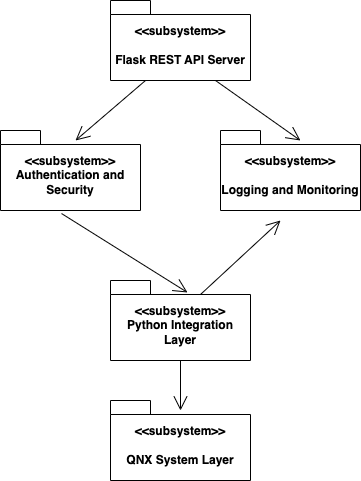
This is the high-level layer where Python handles tasks like generating reports, formatting logs, or running data-related operations. Separating this from the C code keeps the system flexible and easier to modify when needed.

Client-Server Architecture:

Our project includes a REST API built with Flask, which follows the client-server model. In this design, the API acts as the server, handling requests from clients, which can be other systems, tools, or users.The server receives HTTP requests, processes them (sometimes with help from the Python or C layers), and sends responses back to the client. This approach makes it easy to integrate our system with other tools or services, and provides a clean, scalable way to interact with the application from outside.

# ARCHITECTURAL MODEL

* use packages stereotyped as subsystems <<subsystem>>
* no classes in the architectural model



* Flask REST API System
* Sends processing requests to the integration layer
* Interacts with the authentication layer to validate JWT tokens
* Sends a log of user activity or system errors to the logging subsystem
* Python Integration Layer
* Translates API input into system calls
* Communicates with QNX system layer
* Sends a log of system responses to the logging subsystem
* Authentication and Security
* Controls the system access and token validation
* Logging and Monitoring
* Receives logging data from REST API Server and Python Integration Layer

**Technology**

Programming Languages: C, Python.

Frameworks: Flask (REST API)

Communication: C Python API

Operating System: QNX RTOS

Version Control: Git/Github

Database: SQLite

**Software**

QNX Development Tools: QNX Software Development Platform (Includes QNX Compiler, Debugger, and Build System), Momentics IDE (For QNX-based C programming), GDB (For debugging C programs in QNX). QNX File System Tools (For interacting with QNX RTOS at the file level).

Flask API Development & Security: Flask-RESTful is used in designing REST API endpoints. JWT-Extended will used for handling JWT-based authentication Handling cross-origin requests is taken care of by Flask-CORS

Python Development Tools: The main scripting language is Python 3.10+ . Pip & venv (Python Package Manager & Virtual Environments) are used for dependency management. Automated testing of Python components will be handled by Pytest & unittest.

**Hardware**

Embedded System Hardware: CPI Power Supply Controllers (For handling hardware power and state transitions), CPI RF Transmitters & Communication Hardware (For transmitting industrial control signals), QNX-Compatible Embedded Board (Running QNX RTOS).

Processor & Memory Considerations: ARM Cortex-A Series / x86-based Industrial CPUs for low-power, high-efficiency for real-time processing. At least 512MB RAM is required. The recommended 1GB + is used to support Python processes and API requests. Flash Storage (eMMC/SSD) will track logging system interactions and storing configuration files.

Security & Compliance Hardware (if needed): Trusted Platform Module (TPM) is for securing boot and cryptographic key storage. Hardware Security Module (HSM) will handle encryption operations securely.

**Application and Database Server Communication**

Data Flow:

1. Client Sends API Request. A client application (external user, monitoring system, or another service) makes an HTTP request to the Flask REST API.
2. Flask API Processes the Request. The Flask API receives the request and determines whether it needs to fetch data from the database.
3. Database Query Execution. If data is needed, the Flask API constructs a SQL query and sends it to the database server. The database retrieves the requested data or modifies records The database processes the query and returns the results.
4. Flask API Formats the Response. The API receives the query results, converts them into JSON format, and sends the response back to the client.
5. Client Receives and Displays Data. The client application parses the JSON response and presents it in a user-friendly format.

Communication Protocols:

REST API over HTTP(S) - The Flask API communicates with clients using HTTP/HTTPS, ensuring secure data transmission. It uses TLS 1.2+ encryption to prevent eavesdropping.

Database Querying via SQL - The Flask API communicates with the database using SQL queries.

Authentication with JWT - Each API request includes a JWT token to ensure only authorized users can access the database.

The API server maintains persistent database connections to handle multiple requests efficiently.

Performance Communication Considerations: The database uses indexes on frequently searched columns to speed up queries. Frequently requested data is cached to reduce database load. If the database receives high traffic, it can be distributed across multiple servers. Flask API can be scaled to handle increased client requests.

RATIONALE FOR YOUR ARCHITECTURAL STYLE AND MODEL

This project follows a combination of Layered and Client-Server Architecture.

The rationale behind this choice is to ensure modularity, scalability, fault tolerance, and real-time performance while maintaining ease of development and future expansion.

**Client-Server Rationale**

Remote Accessibility & Interoperability: The system is designed to allow external clients to interact via REST API endpoints. This makes it possible to integrate with external tools, third-party systems, and other applications.

Standardized Communication (RESTful API):Using Flask-based REST APIs ensures that different applications can interact with the system using standard HTTP requests (GET, POST, PUT, DELETE). This makes the system platform-independent and ensures long-term usability.

Security & Controlled Access: The API ensures that only authenticated users can trigger C functions or modify settings. This prevents unauthorized modifications to the QNX system and to hardware.

Scalability: If more clients need access, the API server can scale by deploying additional instances. This makes it possible to expand to a distributed system without modifying the core QNX-Python integration.

**Layered Architecture Rationale**

Encapsulation and Modularity:The system is divided into separate layers (hardware interaction, Python integration, and API) to allow independent development and maintenance. This prevents changes in one layer from affecting other layers.

Security & Access Control: Direct access to QNX is restricted to the lower system layers. The Flask API acts as a controlled interface, enforcing authentication and data validation. This reduces the risk of malicious or accidental system modifications.

Maintainability: If new hardware is added, only the QNX Layer needs modification. If new API features are added, only the Flask Layer is affected. This modular design allows future expansion with minimal impact on existing functionality.

Separation of Concerns: Each layer will have a specific role. The QNX Layer (C code) will handle real-time execution and direct hardware control. Python Integration Layer will bridge Python and C using C API and ctypes. The Flask API Layer exposes REST API endpoints for external communication. This ensures clear separation of functionality, making debugging and testing easier.

**Constraint Handling Rationale**

Low Latency: C is used for real-time hardware communication because Python is not optimized for real-time execution. The Python layer only performs non-time-sensitive computations, ensuring that real-time constraints are met.

Process: Python scripts run in isolated processes to prevent memory leaks or failures from affecting the QNX core system. QNX can restart failed Python processes without disrupting real-time operations**.**

**Security & Fault Tolerance Rationale**

Restrict Direct Access to QNX: The QNX layer does not expose direct shell access. All modifications will only happen through controlled API requests.

Recovery: If a Python function fails, the system logs the error and retries execution. If the API crashes, QNX continues operating, ensuring that hardware remains functional.

Data Encryption: All API interactions use HTTPS with TLS 1.2+ encryption. User authentication will be enforced via JWT-based token authentication to verify users or systems accessing the API. All interactions are logged for security audits.

TRACEABILITY FROM REQUIREMENTS TO ARCHITECTURE

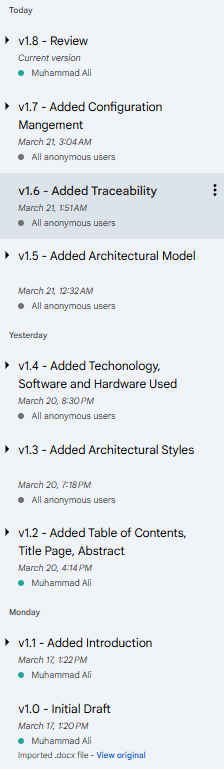
* provide a mapping between requirements and architecture
* clearly describe how each requirement in the *Requirements Documentation* is captured in the architecture

To ensure that the system architecture aligns with the project requirements, a traceability matrix is used to map each requirement to specific architectural components. This approach helps verify that all functional and non-functional requirements are addressed within the system design.

| **Requirement ID** | **Description** | **Mapped Component** |
| --- | --- | --- |
| RQ-001 | Secure authentication | JWT Authentication Service |
| RQ-002 | Real-time data processing | QNX Real-time Processing Module |
| RQ-003 | Python-C interoperability | Python-C Bridge |
| RQ-004 | System monitoring | Logging & Monitoring Service |
| RQ-005 | REST API for communication | Flask API Layer |
| RQ-006 | Hardware compatibility | Embedded System Hardware |
| RQ-007 | Scalability and Modularity | Layered Architecture |
| RQ-008 | Error Handling and Recovery | Fault-Tolerance Mechanisms |
| RQ-009 | Data Encryption | Security & Compliance Hardware |
| RQ-010 | Performance Optimization | Database Indexing & Caching |

This traceability ensures that all system requirements are effectively implemented and tested within the architectural design.

EVIDENCE THE DOCUMENT HAS BEEN PLACED UNDER CONFIGURATION MANAGEMENT



| Version | Date | Author | Changes Made | Previous Versions | Reviewers | Ship-it |
| --- | --- | --- | --- | --- | --- | --- |
| v1.8 | 3/21/2025 | Muhammad Ali, Nicholas Anderson, Saghar Abdi | Review | v1.7 | Khaled Elkhaled, Diego Ibarra | Approved |
| v1.7 | 3/21/2025 | Khaled Elkhaled, Tabark Abaid, Saghar Abdi | Added Configuration Management | v1.6 | Nicholas Anderson, Muhammad Ali | Approved |
| v1.6 | 3/21/2025 | Nicholas Anderson, Khaled Elkhaled | Added Traceability | v1.5 | Saghar Abdi, Diego Ibarra | Approved |
| v1.5 | 3/21/2025 | Muhamamd Ali, Diego Ibarra | Added Architectural Model | v1.4 | Nicholas Anderson, Saghar Abdi | Approved |
| v1.4 | 3/21/2025 | Diego Ibarra | Added Technology, Hardware and Software Used | v1.3 | Nicholas Anderson, Tabark Abaid | Approved |
| v1.3 | 3/21/2025 | Diego Ibarra | Added Architectural Styles | v1.2 | Nicholas Anderson, Khaled Elkhaled | Approved |
| v1.2 | 3/20/2025 | Muhammad | Added Introduction | v1.1 | Nicholas Anderson, Saghar Abdi | Approved |
| v1.1 | 3/17/2025 | Muhammad Ali | Added Title, Abstract, and Table of Contents | v1.0 | Diego Ibarra, Khaled Elkhaled | Approved |
| v1.0 | 3/17/2025 | Muhammad Ali | Initial Draft | - | - | - |

ENGINEERING STANDARDS AND MULTIPLE CONSTRAINTS

* IEEE Std 1471-2000: Software Architecture [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-1471-2000-Software-Architecture.pdf)]
* ISO/IEC/IEEE Std 42030:2019: Software, Systems and Enterprise

Architecture Evaluation Framework [[pdf](https://course.techconf.org/se4485/IEEE/ISO-IEC-IEEE-42030-2019.pdf)]

* Additional standards suggested by the sponsor(s)

ADDITIONAL REFERENCES

* Lattanze, A.J., 2008. *Architecting Software Intensive Systems: A Practitioner’s Guide.*

CRC Press

* Bass, L., Clements, P. and Kazman, R., 2003. *Software Architecture in Practice.*

Addison-Wesley

* Additional references suggested by the sponsor(s)