# Design Document

**Project Title:** Automated Testing for Flintlab micro services (REST API Automation framework)  
**Author**: R Gowtham Sai  
**Start Date**: May 23, 2025   
**End Date**: Jun 03,2025

|  |  |  |
| --- | --- | --- |
| **Reviewer** | **Role** | **Status** |
| Jinesh | Approver |  |
| Tabia | Approver |  |
| Krishna | Reviewer |  |

**1. Introduction**

**1.1 Purpose**

The primary purpose of this framework is to automate the testing and validation of REST APIs developed for Flintlab microservices. It focuses on ensuring API reliability by validating contract adherence, response structure, HTTP headers, status codes, and data integrity. This system supports both static and evolving API architectures by catching unintentional changes through comparison against baselines and defined schemas.

**1.2 Scope**

This framework provides a comprehensive suite of API validation features designed for both functional and structural testing. It includes:

* Baseline Response Capture: Stores the initial successful API response for future comparison.
* JSON Schema Validation: Ensures the response structure matches expected specifications.
* Header & Status Code Checks: Verifies the presence and correctness of HTTP headers and response codes.
* Data Integrity Validation: Checks for valid UUIDs, timestamps, and other key fields.
* Schema Auto-Generation: Automatically generates JSON schemas from real API responses using Genson.
* Containerized Execution: Packaged in Docker for ease of deployment, portability, and integration.
* Extensible Architecture: Designed to support multiple microservices and endpoints through a plugin-friendly architecture.

**1.3 Definitions**

* API (Application Programming Interface): A set of rules that allow different software components to communicate with each other.
* JSON Schema: A JSON-based format used to define and validate the structure and contents of a JSON document.
* UUID (Universally Unique Identifier): A standardized 128-bit identifier used for uniquely identifying objects across systems.

**1.4 References**

* [jsonschema](https://pypi.org/project/jsonschema/): Python library for validating JSON data against a schema.
* [DeepDiff](https://pypi.org/project/deepdiff/): Library for identifying changes between JSON/dictionary objects.
* [Genson](https://pypi.org/project/genson/): Tool for generating JSON schemas from example data.

**2. Overall Description**

**2.1 System Context**

This REST API Testing and Validation Framework is designed for use by QA engineers and DevOps teams. It operates as a standalone tool or CI/CD-integrated utility to ensure APIs meet defined expectations. By validating against predefined schemas and response contracts, the tool helps identify API regressions, unexpected changes, and data anomalies early in the development pipeline.

**2.2 Key Features**

* Baseline Snapshot & Diff Comparison: Captures the initial (baseline) API response and compares future responses against it to detect unexpected changes.
* Schema Validation: Confirms that API responses conform to expected JSON schema definitions.
* Header & HTTP Code Validation: Ensures critical headers are present and HTTP status codes are correct.
* UUID & Timestamp Checking: Validates specific fields (e.g., IDs and timestamps) to conform to correct formats.
* Offers a command-line interface wrapped in a Podman container for consistent, platform-independent execution.
* Multi-Endpoint Support: Easily configurable to validate multiple endpoints through iteration or configuration files.

**3. Technical Approach**

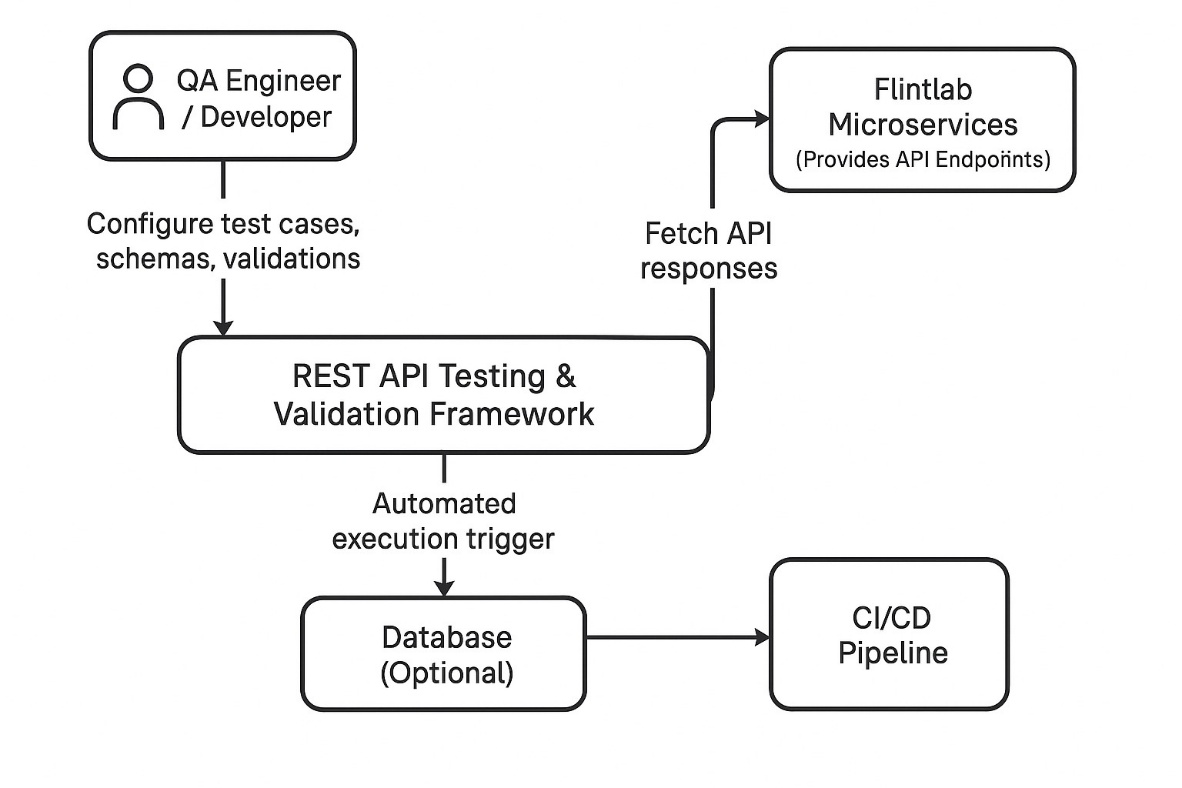
**3.1 Functional Requirements**

* The framework must accept an API endpoint URL, expected schema (or generate one automatically), and validation configuration.
* It must:
  + Validate the response's HTTP status code and headers.
  + Compare the actual response with the stored baseline using DeepDiff.
  + Generate a schema from sample data using Genson if a schema is not provided.
  + Log validation results (pass/fail, differences, errors) in a readable format.

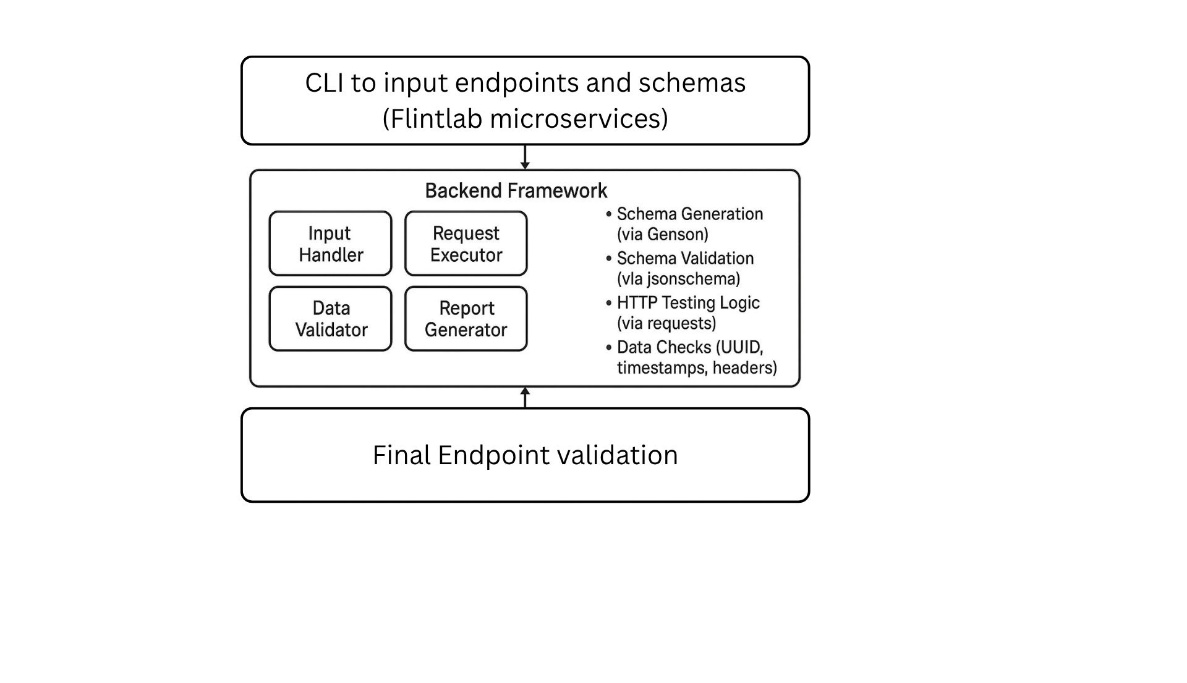
**3.2 Non-functional Requirements**

* **Lightweight & Portable**: The entire framework should run within a Docker container to simplify setup and eliminate environment issues.
* **Script-Driven Execution**: All operations should be executable via command-line arguments or scripts, avoiding any interactive input prompts.
* **CI/CD Friendly**: Should easily integrate into CI/CD pipelines to enable automated regression testing.
* **Extensibility**: The design should allow for easy addition of new validation rules, test cases, or support for new microservices and endpoints.

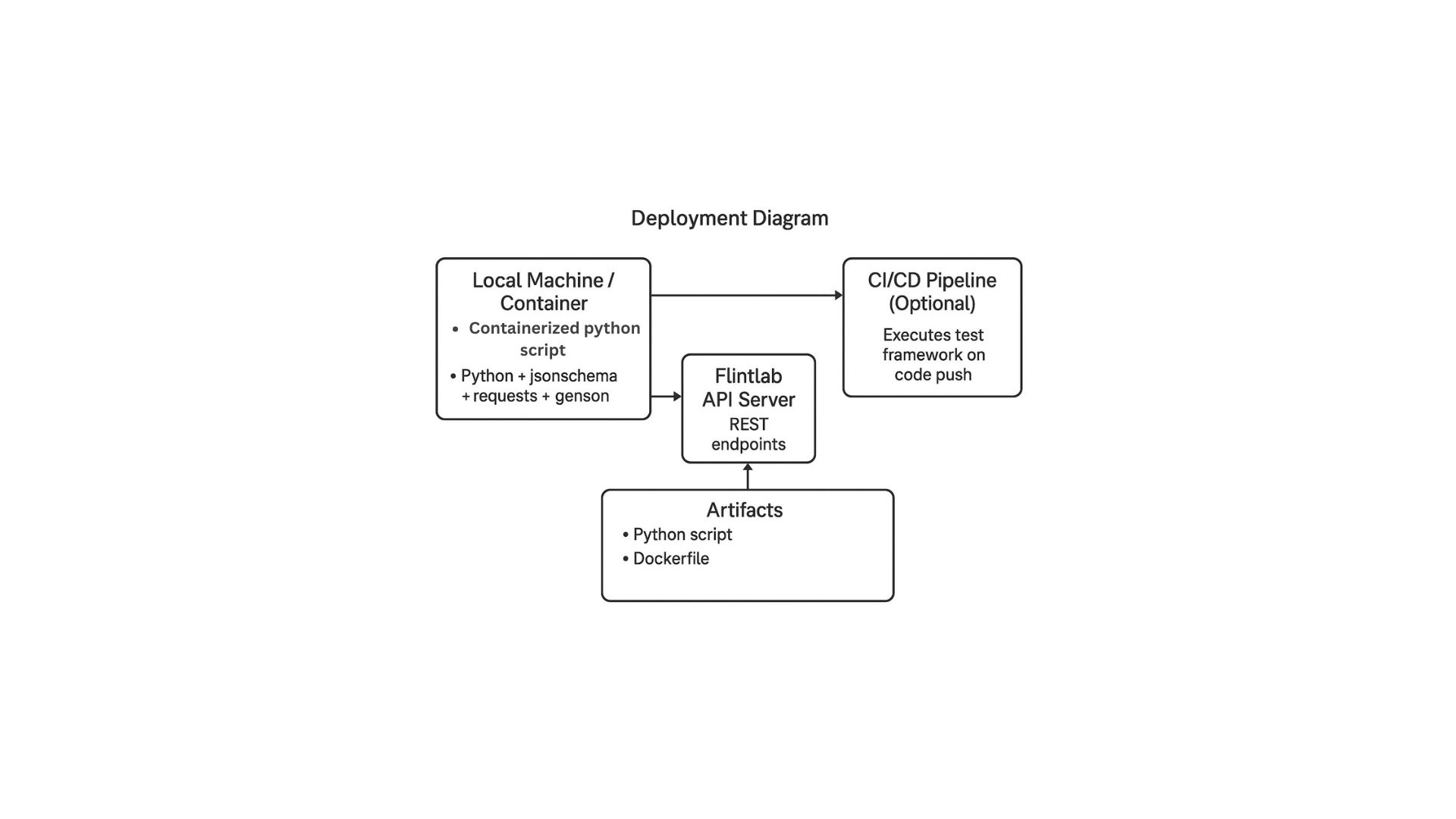
**4) System Architecture**   
  
**4.1) Context Diagram**Illustrates how external entities like QA engineers, Flintlab APIs, and CI/CD systems interact with the testing framework. Helps visualize system boundaries and primary communication channels

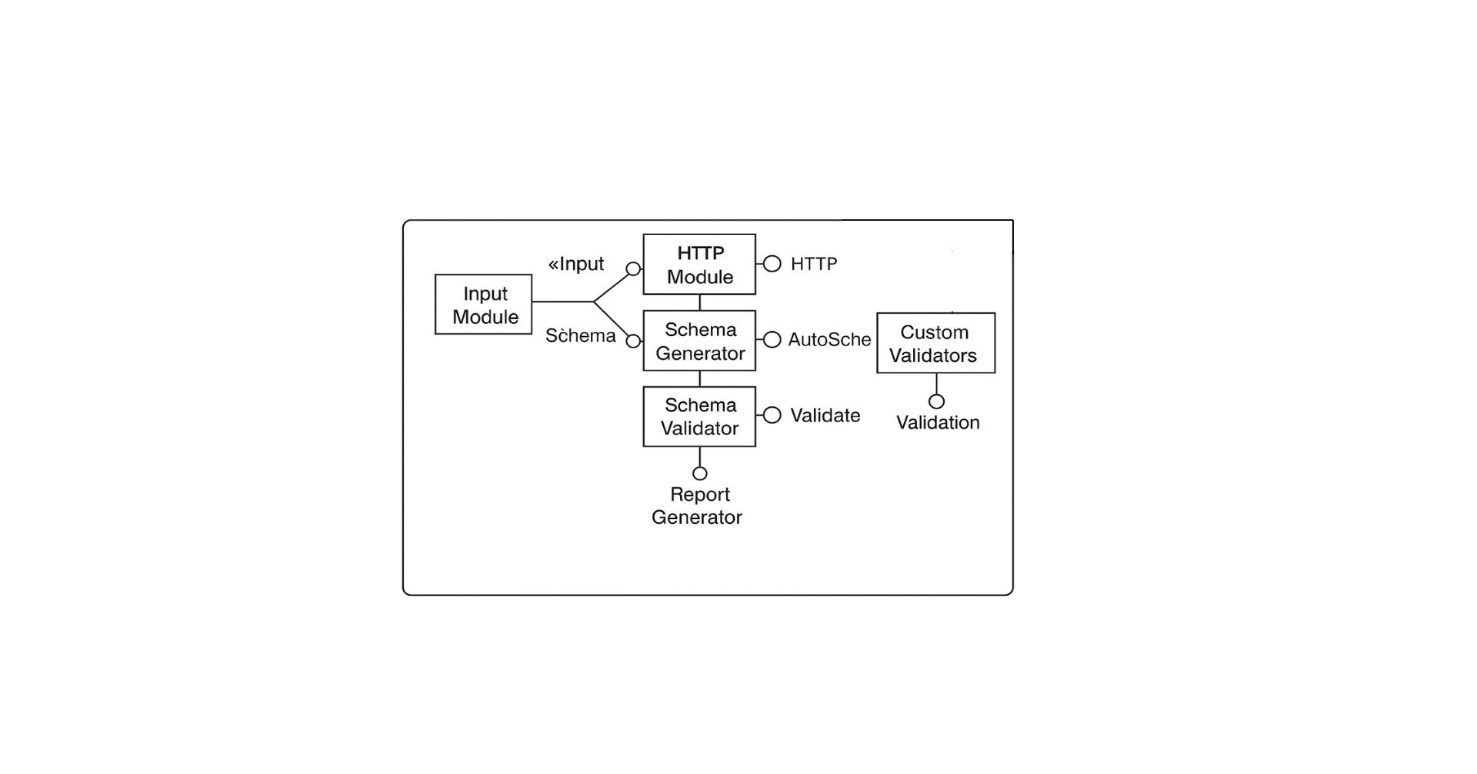
****

**4.2) High level architecture**

Depicts internal components such as CLI, Schema Validator, Baseline Comparator, and Logger.  
Shows how these components work together within a Containerised environment. **4.3) Deployment Diagram**

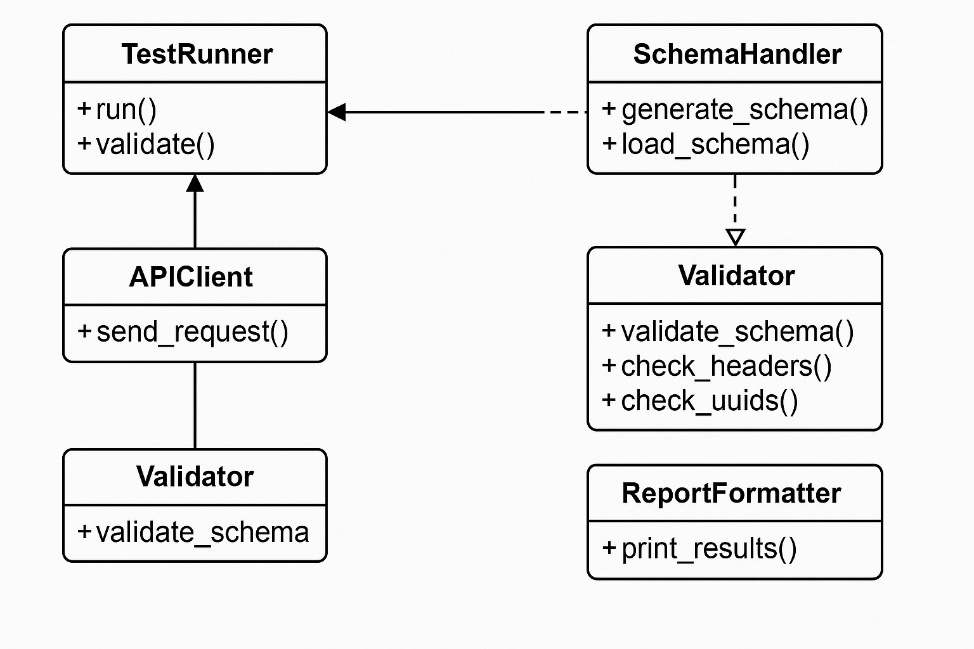
Displays the physical setup, with the framework running in a container.  
Helps understand the infrastructure and deployment topology.

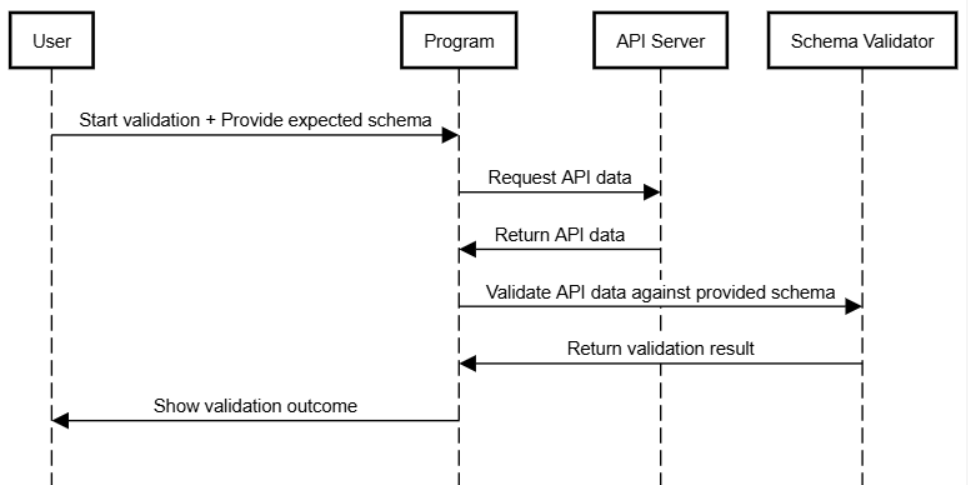
****

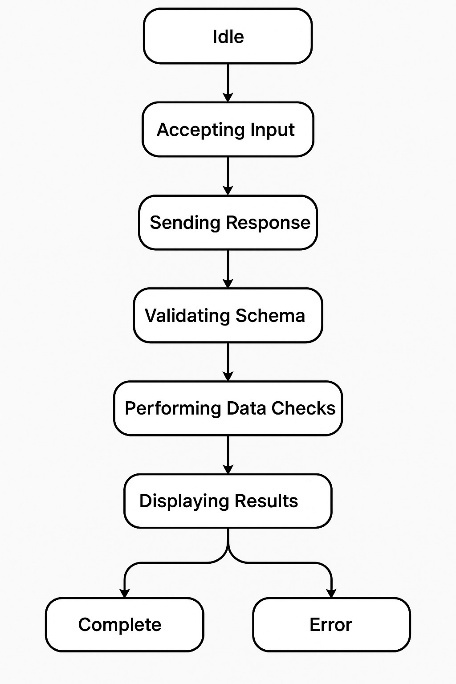
**5) Detailed Diagram  
  
5.1) Component Diagram**Breaks down the system into modular components like ApiClient, Validator, Comparator, and Logger.  
Helps identify functional responsibilities and interdependencies.

**5.2) Class Diagram**

Represents the object-oriented structure including classes, attributes, and methods.  
Illustrates relationships between core classes used in validation and testing.

****

 **5.3) Sequence Diagram**Shows the order of interactions between user, framework modules, and the target API during testing.  
Useful for understanding the request-to-validation flow.  
  
  
 **5.4) State Diagram**Describes various states like Idle, Validating, Comparing, and Logging in the testing lifecycle.  
Captures state transitions triggered by testing actions or results.

****

**6) Security Considerations**

To ensure safe and reliable operation of the REST API Testing and Validation Framework, the following security aspects are considered:

1. **Authentication & Authorization**Access to the framework should be restricted to authorized QA/DevOps users. If integrated with CI/CD, access tokens and environment secrets must be securely managed.
2. **Input Validation**All user inputs including URLs, headers, and schema definitions are validated to prevent injection attacks or malformed requests.
3. **Secure API Communication**All API calls from the framework must use HTTPS to protect against eavesdropping and man-in-the-middle (MITM) attacks
4. **Secrets Management**  
   API tokens or credentials used in test scenarios or CI pipelines must be injected via secure environment variables, not hardcoded in scripts.
5. **Dependency & Patch Management**  
   All third-party libraries (e.g., jsonschema, DeepDiff) are kept updated, and known vulnerabilities are regularly checked

.

**7) Appendix**

**Tools Used:**requests, jsonschema, DeepDiff, genson

**Glossary:**  
**API** – Interface to communicate between systems  
**UUID** – Unique identifier for data entities  
**CI/CD** – Continuous Integration and Deployment  
**JSON Schema** – Defines structure of JSON responses  
  
**References:**  
JSON Schema, DeepDiff, Genson – used for validation, comparison, and schema generation.