# SRS document

SRS Document for Demo 4 as of 30 September 2025

## Table of Contents

## Introduction

This document serves as a blueprint of our team's approach in defining the architectural framework for our API Threat Assessment Tool (or AT-AT for short). AT-AT is an innovative platformed designed to help secure and test web-based APIs.

With the growing reliance on APIs in digital systems, there is a critical need to be able to ensure that they are secure, and safe from attackers. AT-AT adresses this need by delivering a web-based platform that allows users to import their API specifications and dynamically scan and test and then generate a comprehensive assessment report of the results.

## Functional Requirements

## Authentication

- R1: The users must be able to sign up
    - R1.1: **Using a sign up form. The form should gather the following**:
        - R1.1.1:**Email adress**
        - R1.1.2:**Password**
        - R1.1.3:**First Name**
        - R1.1.4:**Last Name**
    - R1.2 Using existing platforms:
        - R1.2.1: **Google account**
- R2: The user must be able to sign in
    - R2.1: **Using their email and password**
        - R2.1.1: **Credentials must be validated**
    - R2.2: Using existing platforms:
        - R2.2.1: **Google account**
- R3: The user must be able to select “forgot password”
    - R3.1: **The system must be able to use their email adress to identify if an account exists**
    - R3.2: **If an account exists an email should be sent with instructions on how to reset their password**

## Authorization

- R1: **The system must provide restricted features based on roles**
    - R1.1: **Basic users can scan APIs and view their own reports**
    - R1.2: **Admin users can view and manage all users and scan data**

## API Specification Input

- R1: Users must be able to submit API specifications
    - R1.1: **Upload OpenAPI/Swagger files**
    - R1.2: **Upload Postman Collections**
    - R1.3: **Provide URL to fetch the API Specification remotely**
    - R1.4: **Validate file structure before storing**

## Heuristic API Discovery

- R1: The system must support specification-less scanning
  - R1.1: **Based on a target API domain**
  - R1.2: **User Heuristic and Traffic-based pattern detection to infer endpoints**
  - R1.3: **Automatically build testable endpoints for undocumented APIs**

## Scan Configuration

- R1 Users must be able to select a scan profile from a list of options
  - R1.1: **OWASP Top 10 Quick Scan**
  - R1.2: **Full Comprehensive Scan**
  - R1.3: **Authentication & Authorization Focus**

## API Vulnerability Scanning

- R1: The system must support automated scanning
  - R1.1: **Perform static analsysis on uploaded specifications**
  - R1.2: **Perform dynamic runtime analysis on live APIs**
  - R1.3: **Detect OWASP API Top 10 Vulnerabilities**

## API Behavior & Health Monitoring

- R1: The system must analyze API responses during scanning
  - R1.1: Identify abnormal HTTP status codes (e.g., 5xx, 4xx where unexpected)
  - R1.2: Detect insecure default configurations or misbehavior (e.g., exposing stack traces, reflection of payloads)
- R2: The system must detect and flag usage of outdated or insecure packages
  - R2.1: Scan dependency metadata in OpenAPI/Swagger specs
  - R2.2: Highlight specific libraries and their affected versions
- R3: The system must correlate scan output with known vulnerability patterns
  - R3.1: Use rule-based or AI-assisted pattern matching to increase detection accuracy
  - R3.2: Provide contextual information on abnormal responses (e.g., unexpected content-type, slow response time, inconsistent structure)

## Report Generation

- R1: After each scan, a report must be generated
    - R1.1: **Report must include a list of vulnerabilities found, severity levels and endpoints affected**
    - R1.2: **Report will include recommendations on how to improve security**
    - R1.3: **Allow report to be exported as a pdf**
    - R1.4: **Include a security score metric for the API**
    - R1.5: **Detailed summary for high-level overview**

## Endpoint Management

- R1: The system must allow users to manage endpoints from imported API specifications
    - R1.1: **List all available endpoints**
    - R1.2: **Retrieve details of a specific endpoint**
        - R1.2.1: **Return path, method, tags, and security metadata**
    - R1.3: **Return endpoint list scoped by API ID**

## Endpoint Tagging

- R1: **The system must allow tagging of API endpoints**
    - R1.1: **Add tags to a given endpoint**
        - R1.1.1: **Tags should be user-defined or selected from a predefined list**
        - R1.1.2: **Tags such as "admin only", "sensitive", "deprecated" should be available**
    - R1.2: **Remove specific tags from an endpoint**
    - R1.3: **Replace all tags for an endpoint**
    - R1.4: **Retrieve a list of all known tags in the system**

## User Stories/Characteristics

### A New User's Characteristics

Any user who has not created an AT-AT account before.

As a New User I would like to:

- Sign up with Google so that I can register quickly without filling out long forms
- Sign up with GitHub so that my developer identity is easily linked
- Sign up using an email and password so I can use AT-AT independently of other platforms

## An Existing User's Characteristics

Any user who has previously registered on AT-AT.

As an Existing User I would like to:

- Sign in with Google or GitHub to quickly access my account
- Log in using my email and password if I prefer traditional sign-in methods
- Change my password in case I want to improve my account security
- Update my profile information (name, organization, usage preferences)
- Delete my account if I no longer want to use the system

## A Developer's Characteristics

A developer is someone preparing their API for production or release, and wants to ensure it's secure.

As a Developer I would like to:

- Upload my API specification file (OpenAPI or Postman) so AT-AT can assess it
- Run a quick scan against the OWASP Top 10 so I can identify common vulnerabilities
- Select a scan profile based on depth or speed so I can test in different environments
- View a vulnerability report that highlights issues and suggests fixes
- Add tags to endpoints to indicate sensitivity or access level
- Export the report in PDF so I can use it/look at it later
- Save the scan session so I can review or share results later

## A Security Analyst’s Characteristics

A security analyst is responsible for verifying and auditing the security posture of APIs across multiple teams.

As a Security Analyst I would like to:

- Run deep scans on production and staging APIs to uncover hidden vulnerabilities
- Use heuristic scanning to discover undocumented endpoints that might be exposed
- Customize scanning profiles to target specific threat models or attack vectors
- Review detailed vulnerability metadata and logs for compliance auditing
- Export reports in JSON so I can ingest results into my existing SIEM system
- View a visual dashboard of historical scans and trends to monitor system health over time
- Label endpoints using tags such as ‘admin only’ or ‘public’ to improve API classification
- Track endpoint metadata to assess security risk based on exposure and tags

## A Penetration Tester’s Characteristics

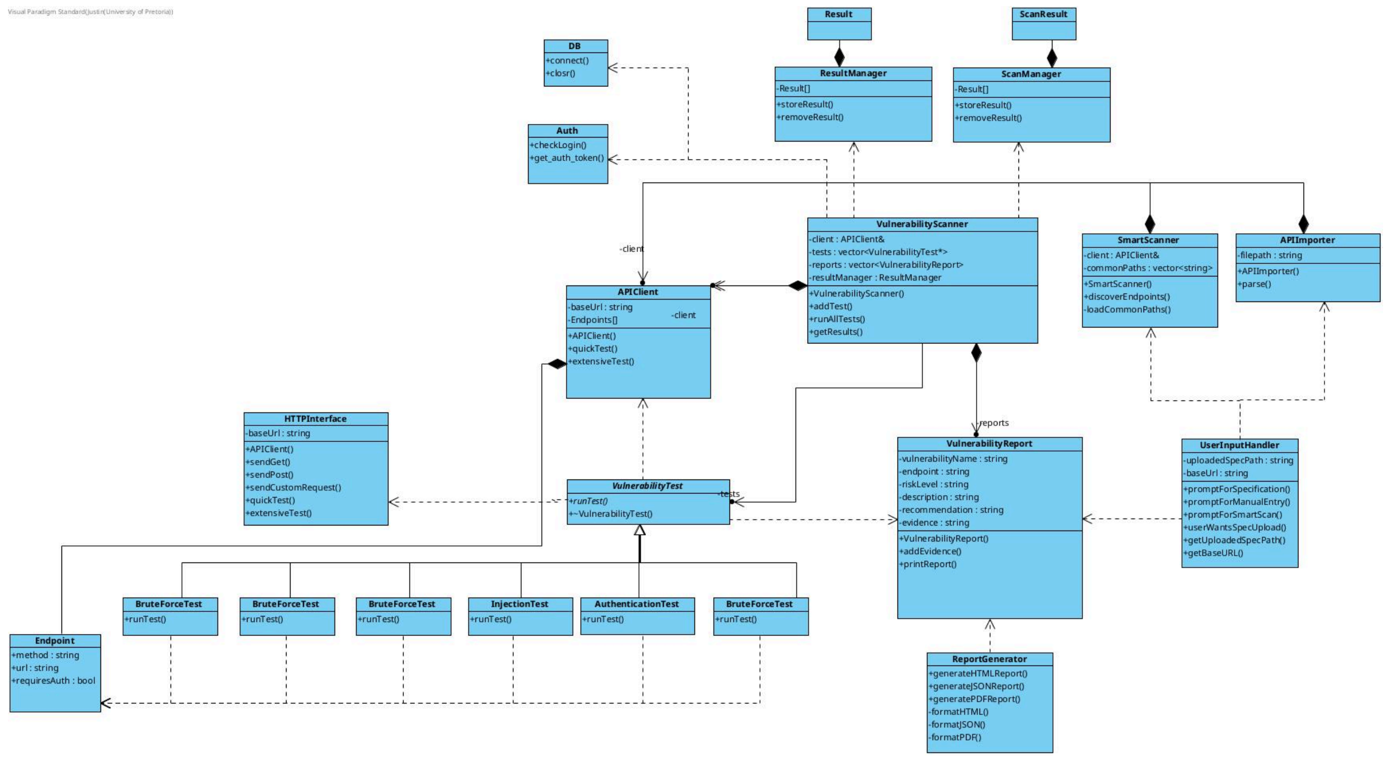
A pentester actively simulates attacks against APIs to discover exploitable flaws.

As a Penetration Tester I would like to:

- Launch manual attack simulations like brute force, token theft, and injection attempts
- Enable red team mode to simulate multi-step attack chains for a more realistic threat assessment
- Use the CLI version of AT-AT so I can integrate it into my own automated testing pipelines
- Validate authentication flows to find weaknesses in token or session handling
- Test different input payloads and see how the API behaves under fuzzing conditions

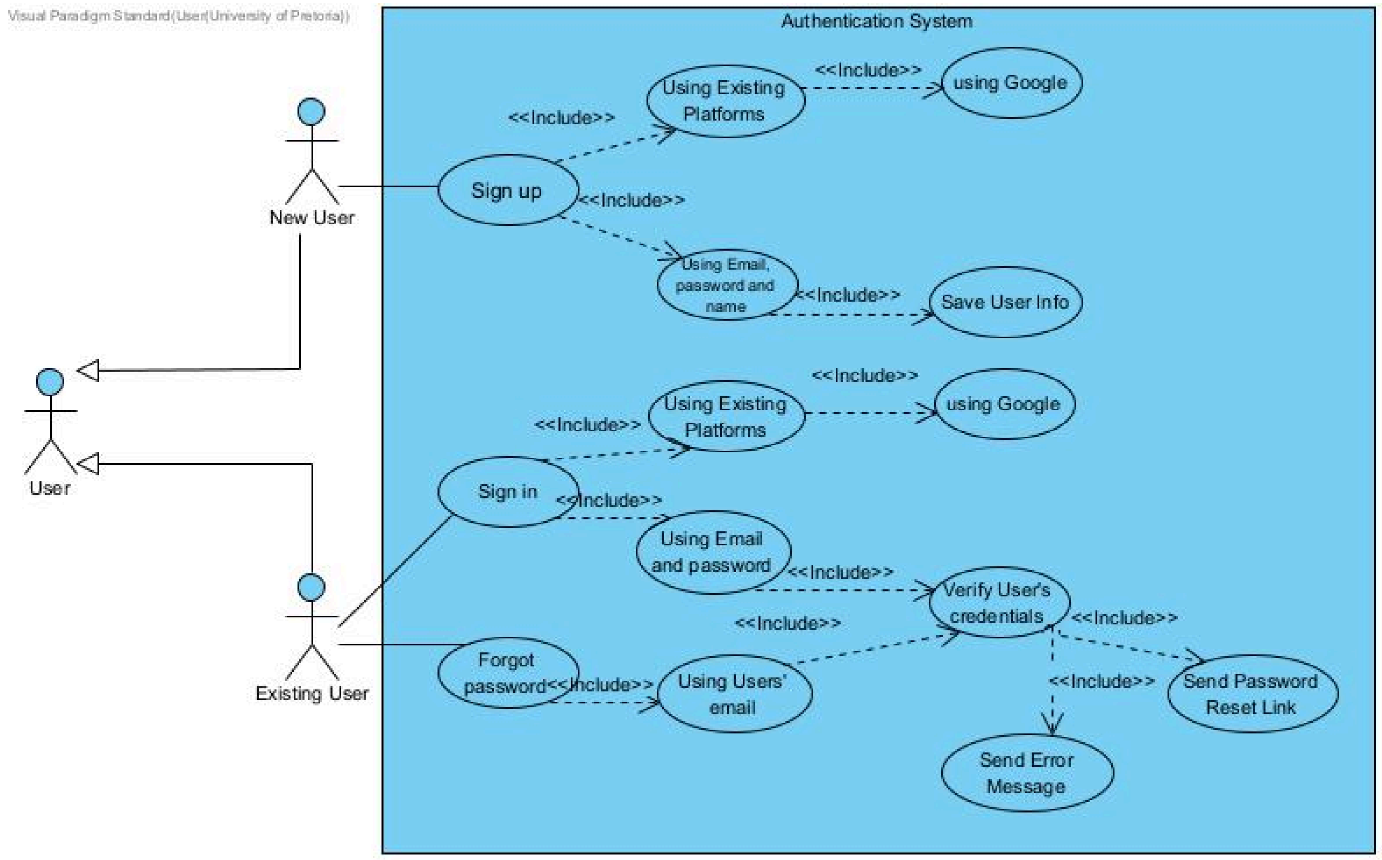
# Domain Model

**Right click on image and press on open imnage in new tab if you want to read it more clearly/in larger scale**

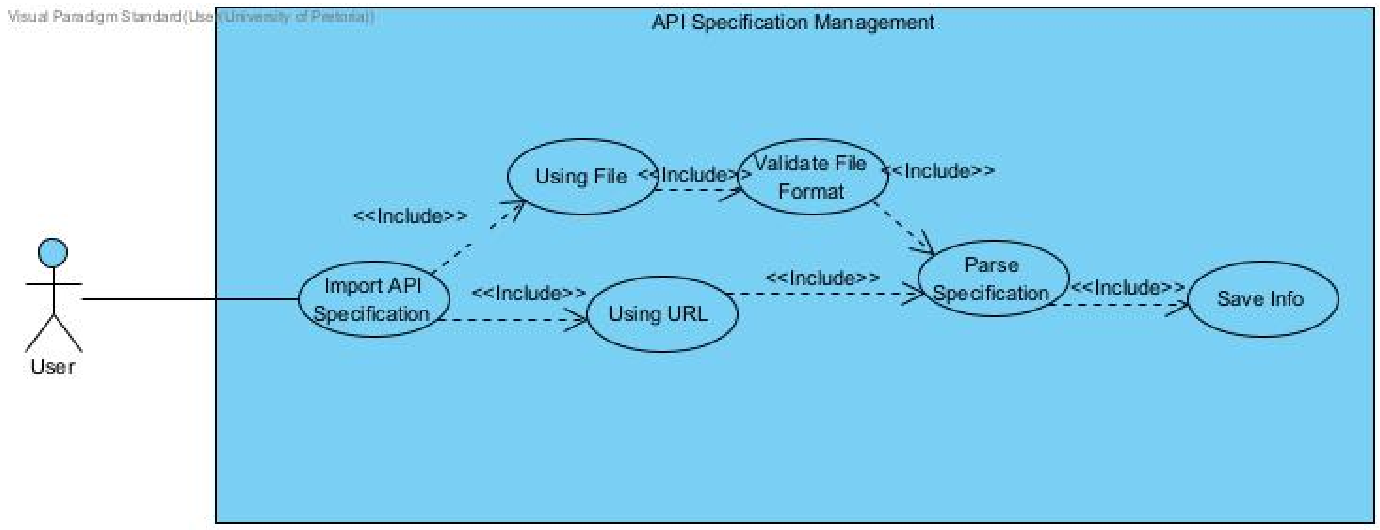


## Use case Diagrams

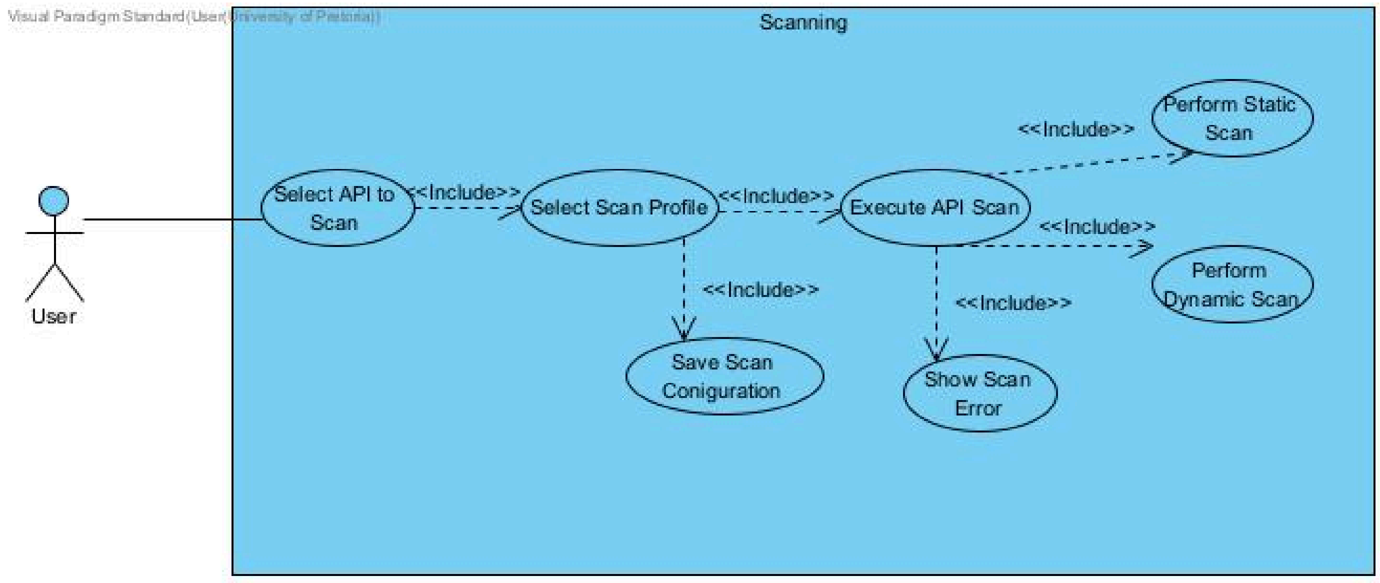
### Authentication



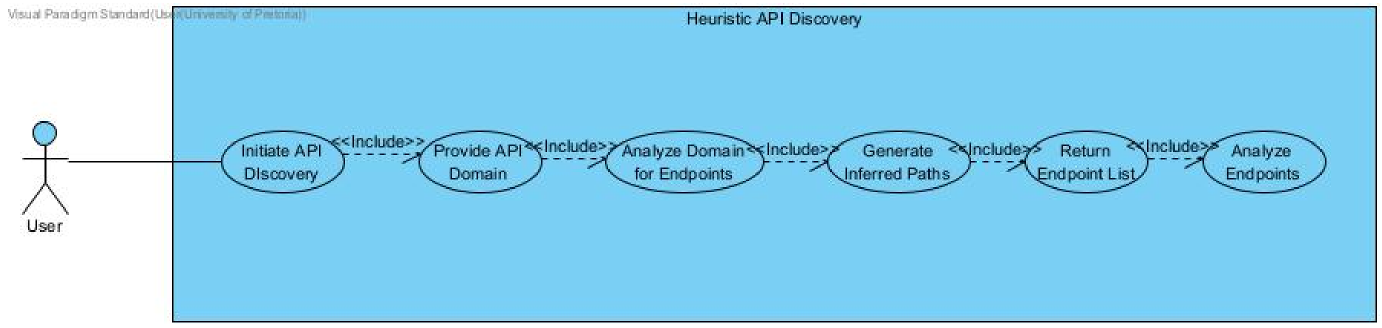
## API Specification Input



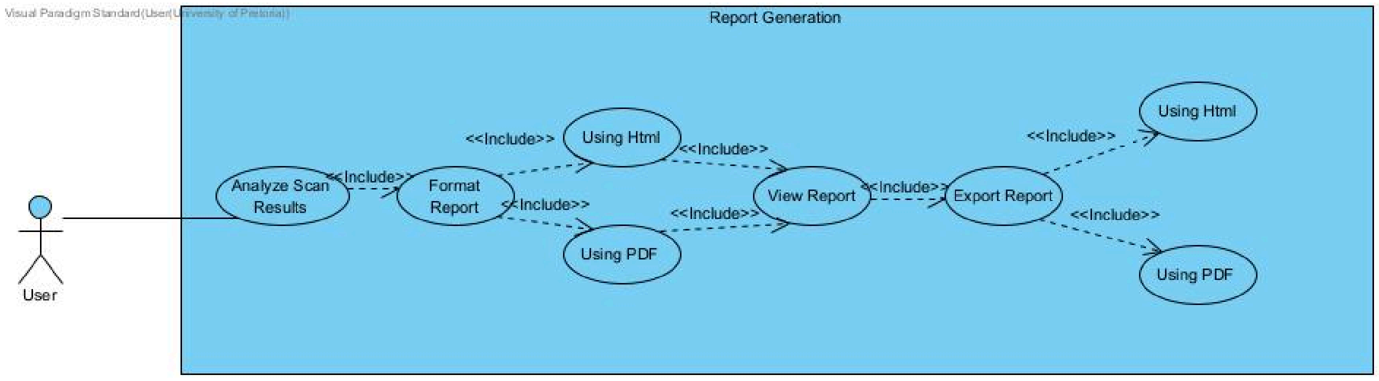
## Scanning



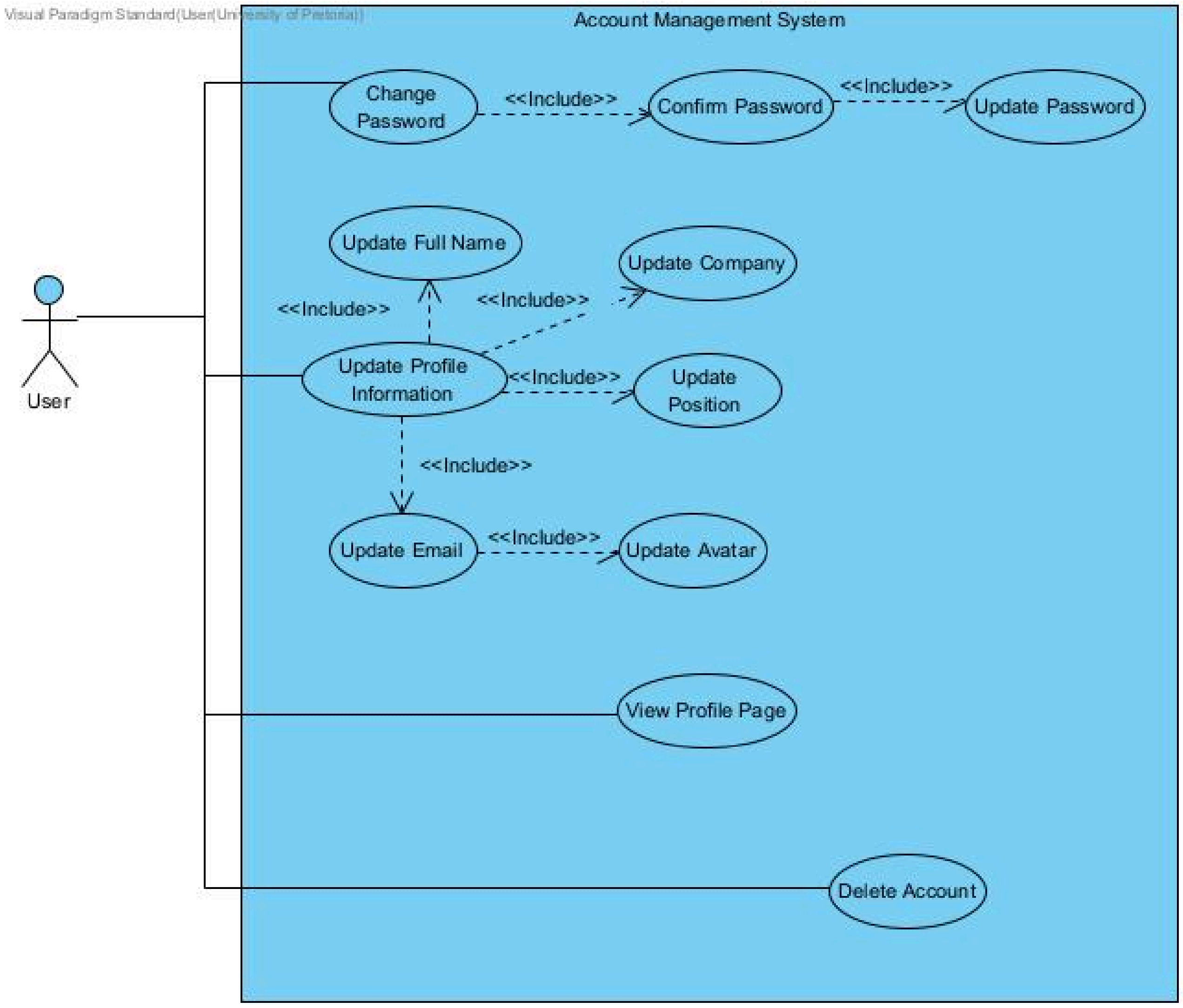
## Heuristic Discovery



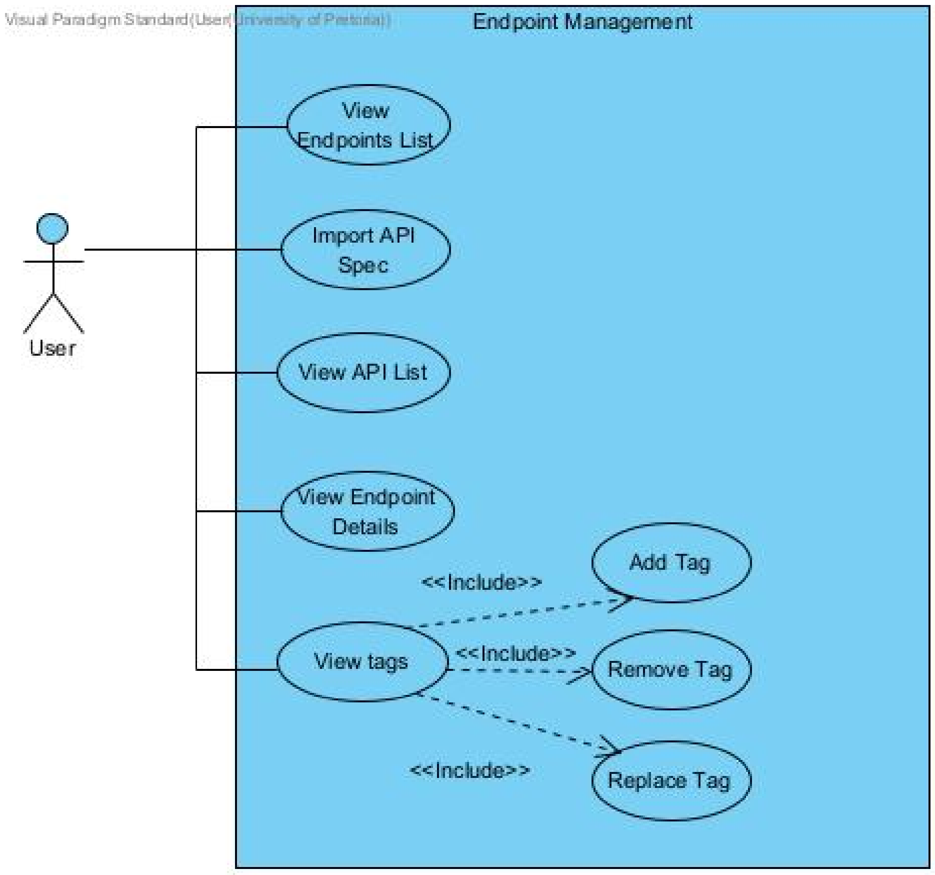
## Reports



## Account Management System



## Endpoint Management



## Architectural Constraints

### Deployment

The AT-AT system will be deployed on a VPS provided by BITM, using Docker for containerization. This ensures consistent development, testing, and production environments. The use of Docker Compose simplifies orchestration and deployment of backend and frontend services.

### Security

The system uses industry-standard authentication protocols such as OAuth 2.0 and JWTs for secure login and authorization. Sensitive data is encrypted at rest (AES) and in transit (TLS). Role-Based Access Control (RBAC) restricts access to features and reports.

### Cost

Since hosting infrastructure is provided by the client (BITM), the team must remain within resource limits and avoid using excessive third-party APIs. Optional integrations (e.g., Snyk, Burp Suite API) will only be added if budget and technical feasibility allow.

### Reliability

Docker and GitHub Actions ensure high reliability through automated CI/CD and consistent builds. Uptime monitoring tools like Uptime Robot and fail-safe error handling mechanisms help ensure continuous availability and reduce downtime.

## Architectural Quality Requirements

### Security

Security is paramount for a vulnerability assessment platform. The system must enforce strong authentication, secure transmission and storage of data, and role-based access to restrict sensitive information. This ensures trust, regulatory compliance, and protection from malicious users.

**Measured by:** maybe add percentage of data stored in plaintext

currently, mentioned oauth 2,0 in archtitectural constraints but the current version uses passwords and hashes, architectural constraints are of the end protect and quality requirements are of current version probably drop compatibility

- Access control test results
- Penetration testing
- Static and dynamic vulnerability scans
- R1.1: Secure Authentication
  - R1.1.1: Users must authenticate with hashed credentials.
    **Implementation:** Bcrypt will be used for password hashing with appropriate salting.
  - R1.1.2: Sessions must be stateless and securely maintained.
    **Implementation:** JWTs (signed with a private key) will be used and stored securely on the client side.
- R1.2: Role-Based Access Control (RBAC)
  - R1.2.1: Different access levels must exist for users, analysts, and admins.
    **Implementation:** Endpoints will be guarded using middleware that checks role claims embedded in the JWT.
- R1.3: Secure Transmission and Storage
  - R1.3.1: All API requests and responses must be encrypted in transit.
    **Implementation:** HTTPS (TLS 1.2+) will be enforced via nginx or similar reverse proxy.

  - R1.3.2: Sensitive data must be encrypted at rest.
    **Implementation:** AES-256 encryption will be used for storing API keys, scan configs, and user tokens.

## Performance

Performance is critical due to the compute-intensive nature of scanning APIs. Users expect fast feedback without interface lag or failure during long scans.

**Measured by:**

- Scan completion time
- File processing speed
- API latency benchmarks
- R2.1: Responsive UI During Scans
  - R2.1.1: Users should see real-time feedback during scans.
    **Implementation:** A status polling system using WebSockets or polling will be integrated into the frontend.
  - R2.1.2: The UI should not freeze during uploads or scans.
    **Implementation:** All operations will be asynchronous, with loading indicators and retry logic.
- R2.2: Scan Completion Targets
  - R2.2.1: 90% of scans should complete within 2 minutes on stable networks.
    **Implementation:** Smart scan throttling and resource allocation will be used server-side.
  - R2.2.2: Large files (up to 5MB) should be accepted without timeout.
    **Implementation:** Chunked file upload and streaming to disk will be employed.

## Usability

For both developers and security professionals, the interface must be intuitive, concise, and reduce friction in task execution. The goal is to maximize system adoption by minimizing confusion and learning curve.

**Measured by:**

- User feedback
- Support queries
- Task completion rates
- R3.1: Simple Navigation
    - R3.1.1: Users must easily find scanning and reporting features.
      **Implementation:** Clear sidebar and breadcrumb navigation using a consistent component library (e.g., MUI or shadcn).
    - R3.1.2: Form inputs must guide the user through step-by-step scan setup.
      **Implementation:** Tooltips and progressive form sections will be used.
- R3.2: Visual Feedback
    - R3.2.1: Users must receive confirmation for actions.
      **Implementation:** Toast messages and modals will display confirmation or error results.
    - R3.2.2: Critical errors must show remediation suggestions.
      **Implementation:** Structured messages with documentation links will be displayed in scan reports.

## Compatibility

AT-AT must run in any modern browser and allow specification imports from common formats.

**Measured by:**

- Browser/device testing
- File support validation
- Mobile responsiveness
- R4.1: Browser and Device Compatibility
    - R4.1.1: The system must work in Chrome, Firefox, and Edge.
      **Implementation:** Regular browser testing and linting with ESLint + Prettier + browserlist.

  - R4.1.2: Pages must adapt to mobile and tablets.
    **Implementation:** CSS Grid/Flexbox and responsive breakpoints will be used.
- R4.2: Supported File Formats
  - R4.2.1: Upload of OpenAPI, Swagger, and Postman collections is required.
    **Implementation:** File validation logic will parse .json, .yaml, and .zip correctly before processing.

## Reliability

As a critical assessment tool, users must trust AT-AT to always be available, recover gracefully from crashes, and log all operations for audit purposes.

**Measured by:**

- Uptime percentage
- Recovery time objective (RTO)
- System logs and error rates
- R5.1: High Availability
  - R5.1.1: The system must maintain >99% uptime post-deployment.
    **Implementation:** Docker containers will be monitored with auto-restart policies.
- R5.2: Deployment and Update Resilience
  - R5.2.1: New features must deploy with zero downtime.
    **Implementation:** GitHub Actions + reverse proxy blue-green deployment setup.
- R5.3: Error Monitoring and Alerts
  - R5.3.1: System errors must be logged and developers notified.
    **Implementation:** A log aggregator like Logtail or custom webhook will track critical backend crashes.

# Non-Functional Requirements

These define system qualities such as performance, reliability, maintainability, security, and usability, and complement the functional requirements.

---

- NFR1: Performance
    - NFR1.1: The system must respond to user actions (e.g., login, upload) within 500ms under normal conditions.
    - NFR1.2: The system must complete API scan operations within 10 seconds for standard OpenAPI files smaller than 500KB.
    - NFR1.3: The UI must render initial content within 2 seconds on standard 4G mobile or broadband connections.
- NFR2: Security
    - NFR2.1: All data transmission must use HTTPS with TLS encryption.
    - NFR2.2: User passwords must be hashed using a secure algorithm such as bcrypt before being stored.
    - NFR2.3: Access to protected routes must be authorized via JWT tokens and validated on every request.
    - NFR2.4: API specification files must be scanned in isolation from other user data to prevent interference or injection attacks.
- NFR3: Scalability
    - NFR3.1: The system must support up to 100 concurrent users without significant degradation in response time (more than 2 seconds).
    - NFR3.2: The backend must be able to queue and process multiple scan jobs in parallel using asynchronous operations.
- NFR4: Reliability and Availability
    - NFR4.1: The system should have 99% uptime, excluding scheduled maintenance windows.
    - NFR4.2: In case of a backend failure, the frontend must show graceful error messages instead of crashing.
    - NFR4.3: Scan jobs must be persisted or retried in case of transient system errors such as I/O failure.
- NFR5: Usability
    - NFR5.1: All pages must include consistent navigation and feedback such as loaders and confirmation messages.
    - NFR5.2: Error messages must be specific and actionable, especially for scan failures or invalid files.

    - NFR5.3: The application must be mobile-friendly and responsive.
- NFR6: Maintainability and Documentation
    - NFR6.1: The codebase must be documented with inline comments and usage guides including README and API docs.
    - NFR6.2: Environment variables must be loaded from a .env file and documented clearly in developer setup instructions.
    - NFR6.3: The API layer must follow RESTful conventions to ensure consistency.

# Architectural Patterns

## Client-Server Architecture

AT-AT adopts a traditional Client–Server model where responsibilities are cleanly split between a React-based frontend and a Python backend. The frontend is responsible for presenting the UI and initiating user actions (e.g., initiating scans, uploading specs), while the backend handles all business logic, authentication, and database operations.

This separation allows independent development, testing, and deployment of the frontend and backend systems.

- Client: React (browser-based UI)
- Server: Python (REST API)
- Benefits: Decoupled development, scalability, clear role separation

## Rest API

The communication between the frontend and backend is implemented through a RESTful API design. Each resource (e.g., scan, report, user) is accessible via predictable endpoints and HTTP methods (GET, POST, PATCH, DELETE).

REST principles such as statelessness, resource-based URIs, and standard response codes are followed to ensure interoperability and scalability.

- Example: POST /api/scan/start, GET /api/report/{scan_id}
- Benefits: Loose coupling, platform independence, caching, easy documentation

## Model–View–Controller (MVC) Pattern in React

While React is not strictly bound to MVC, AT-AT’s implementation applies the MVC pattern conceptually:

- Model: App state and API responses (e.g., scan configs, reports)
- View: React components responsible for rendering UI
- Controller: Event handlers and React hooks controlling business interaction (e.g., handleScanSubmit())

This logical separation improves maintainability and reusability, especially in complex UI flows.

- Benefits: Clear state management, modularity, separation of concerns

### Layered (n-Tier) Architecture

AT-AT is organized into a 3-tier architecture, where each layer is responsible for a specific concern. This is a natural consequence of combining a React frontend with a Python backend and PostgreSQL database.

| Layer | Description |
|---|---|
| Presentation Layer | The React frontend (UI) where users interact |
| Business Logic Layer | The Python service layer handling validation, auth, scan orchestration |
| Persistence Layer | PostgreSQL storing users, scan history, specs, reports |

These layers are loosely coupled but highly cohesive internally, improving maintainability and scalability. The client-server model sits atop this architecture as a deployment pattern.

- Benefits: Scalability, separation of concerns, easy testing, modular growth

## Technology Requirements

### Frontend Framework

**Final Choice:** React.js

React is a powerful and widely adopted JavaScript library for building user interfaces through composable components and declarative syntax. It enables us to manage UI complexity efficiently while promoting reusable and maintainable code structures.

- **Advantages:**
  - Component-based architecture fits naturally into modular development.
  - Supported by a large ecosystem (e.g., React Router, Redux, Zustand).
  - Enables rapid development with built-in hooks and functional patterns.
  - Integrates seamlessly with REST APIs and backend authentication flows.
- **Disadvantages:**
  - Relatively steep learning curve when using hooks and complex state logic.
  - Requires third-party libraries for advanced capabilities like routing and form handling.
- **Justification (Fit):** React’s lightweight design aligns perfectly with our **client-server** and **multi-tiered** architecture. It functions as the **presentation layer**, with tight integration into our RESTful backend. The development team already had strong experience with React, which minimized onboarding time. Additionally, React enables rapid iteration in UI prototyping and integrates seamlessly with testing.

**Alternative 1:** Angular

- **Advantages:**
  - Built-in dependency injection and state management.
  - Enterprise-grade tooling and TypeScript-first approach.
- **Disadvantages:**
  - Heavy initial configuration.
  - Steeper learning curve and verbose syntax.

**Alternative 2:** Vue.js

- **Advantages:**
  - Simpler syntax and lower barrier to entry.
  - Strong integration with template-based HTML logic.
- **Disadvantages:**
  - Smaller ecosystem.
  - Not as widely used in enterprise environments.

---

## Backend Framework

**Final Choice:** Python + Node.js

The main API handling logic and system control live in **Python**, while the vulnerability scanning engine operates as a **Node.js** service connected via sockets. This two-layer backend enables flexible task delegation and aligns well with our plugin-style architecture.

- **Advantages:**
  - **Python** is well-suited for orchestration, authentication handling, and parsing structured data like OpenAPI specs.
  - **Node.js** provides non-blocking I/O and rapid JSON stream processing, making it ideal for scan command execution and vulnerability detection.
  - This separation of concerns improves maintainability and supports concurrent development.
  - Simplifies testability of services by decoupling scan logic from HTTP routing logic.
- **Disadvantages:**
  - Introduces inter-process communication complexity (Python ↔ Node.js socket handling).
  - Harder to standardize documentation across two languages unless abstracted clearly.
  - Requires extra care to handle errors consistently across services.
- **Justification (Fit):** This dual-service setup fits our **microkernel-inspired plugin architecture**. Python handles user session control, RESTful communication with the frontend, and data coordination with Supabase. Node.js powers the scan engine as a separately running process, triggered via structured command packets. This mirrors our decomposition design strategy and improves scalability by enabling independent updates to the scanning engine and API layer. It also reduces the risk of full-system failure due to isolated crashes and aligns with our architectural constraint of offline scanning without external APIs. **Alternative 1:** Express.js (Node.js)

- **Advantages:**
  - Minimal and flexible.
  - Massive library support.
- **Disadvantages:**
  - No native type safety.
  - Middleware chains can grow complex quickly.

**Alternative 2:** Django (Python)

- **Advantages:**
  - Includes ORM, admin, auth out of the box.
  - Mature and battle-tested.
- **Disadvantages:**
  - Monolithic structure — difficult to scale as microservices.
  - Less suitable for asynchronous workloads.

---

## Scan Engine

**Final Choice:** Node.js (Socket-based Microservice)

Our scan engine is implemented as a standalone Node.js service that listens to command-based inputs via a TCP socket. This isolates the engine's logic from the backend while allowing plugin-style extensibility.

- **Advantages:**
  - Event-driven I/O makes it ideal for socket communication.
  - Native JSON handling simplifies input parsing and response formatting.
  - Supports modular "command" execution with minimal overhead.
  - High performance for concurrent scan job handling.
- **Disadvantages:**
  - Requires robust error and state management.
  - Not type-safe without adding tools like TypeScript.
- **Justification (Fit):** This architecture embodies a **microkernel pattern**, where the scan engine acts as a pluggable service processing structured instructions. Node.js's event loop enables responsive communication even under load. This structure supports horizontal scaling of scan jobs in future deployments.

**Alternative 1:** Python (Sockets with AsyncIO)

- **Advantages:**
  - Same language as backend (shared libraries possible).
  - Readable and maintainable syntax.
- **Disadvantages:**
  - Less performant under high-concurrency without fine-tuning.
  - Limited plugin patterns.

**Alternative 2:** Rust TCP Service

- **Advantages:**
    - Fast, memory-safe concurrency.
    - Great performance under high load.
- **Disadvantages:**
    - High learning curve.
    - Slower development cycles for prototyping.

## Authentication

**Final Choice:** Supabase Auth (Email/Password with JWT) + OAuth 2.0 (Google/GitHub)

Authentication is handled through Supabase's built-in authentication service, which provides a simple but powerful abstraction over JWT-based authentication. OAuth support is natively integrated, allowing users to sign in using third-party platforms like Google or GitHub.

- **Advantages:**
    - Built-in email/password and third-party OAuth flows.
    - Auto-generates secure, time-limited JWT tokens for session management.
    - Integrated with the Supabase PostgreSQL user table — no need to duplicate user handling.
    - Enforces role-based access through Supabase policies.
- **Disadvantages:**
    - Limited to Supabase's ecosystem — less flexibility if migrating away.
    - No built-in support for multi-factor authentication.
- **Justification (Fit):** Supabase Auth is seamlessly tied to our database layer and fits cleanly within the **client-server** and **layered architecture** of the system. It offloads low-level session logic while still allowing token inspection and role validation within Python. This supports our **security** quality requirement (RBAC + JWT) and simplifies login logic on the frontend.

**Alternative 1:** Auth0

- **Advantages:**
    - Supports enterprise auth (MFA, SAML, etc.)
    - Scalable and secure out-of-the-box

- **Disadvantages:**
  - Expensive beyond free tier
  - High integration overhead

**Alternative 2:** Firebase Auth

- **Advantages:**
  - Supports federated identity providers
  - Real-time session tracking
- **Disadvantages:**
  - Tightly coupled with Google Cloud stack
  - Less flexibility in backend logic integration

## Database

**Final Choice:** PostgreSQL (via Supabase)

PostgreSQL is a powerful, production-ready, open-source relational database. Supabase offers a hosted PostgreSQL solution with extended features like row-level security and automated backups.

- **Advantages:**
  - Strong relational modeling — ideal for scan logs, users, and endpoints.
  - Supports JSON columns for storing raw endpoint data while retaining SQL querying.
  - Fine-grained access control with Supabase's row-level policies.
  - ACID-compliant transactions ensure data integrity for scan and report operations.
- **Disadvantages:**
  - Slightly more complex queries than NoSQL databases.
  - Requires strict schema design and migrations.
- **Justification (Fit):** PostgreSQL perfectly aligns with our **layered architecture** where the database acts as the **persistence layer**. Its relational design enforces data consistency, and Supabase's APIs make it easy to query and interact with from both backend and scan services. It supports our quality needs for **scalability** and **security**.

**Alternative 1:** MongoDB

- **Advantages:**

- Schema-less, flexible for early-stage iteration
    - High performance for large-scale document reads
- **Disadvantages:**
    - Data validation harder to enforce
    - Complex joins require aggregation pipelines

**Alternative 2:** MySQL

- **Advantages:**
    - Wide hosting support, mature tooling
    - Easy to manage for small-scale applications
- **Disadvantages:**
    - Less support for JSON columns and RLS
    - Weaker standards compliance vs PostgreSQL

## Testing Stack

**Final Choice:** Pytest + Unittest

Our test stack includes `pytest` for backend logic, `unittest` for basic module isolation,

- **Advantages:**

    - `pytest`: Clear syntax, great plugin ecosystem (e.g. coverage, fixtures).
    - Unit + integration + E2E layers offer complete coverage.
    - Easy integration into GitHub Actions for automated CI.
- **Disadvantages:**

    - E2E tests can be flaky across browsers.
    - Writing testable backend code requires discipline (e.g. dependency injection).
- **Justification (Fit):** This stack supports our **testability** and **robustness** goals. Tests exist at each tier of the architecture: unit tests for logic modules (backend), integration tests for service interactions (scan engine/backend), and full-stack tests for upload/scan/report flows. CI/CD automatically validates every merge.

**Alternative 1:** Selenium + Pytest

- **Advantages:**
    - Browser compatibility

    - Mature toolchain
- **Disadvantages:**
    - Slower execution and harder to maintain

**Alternative 2:** Jest + Supertest

- **Advantages:**
    - Great for testing Express or Node-based APIs
    - TypeScript support
- **Disadvantages:**
    - Doesn’t integrate natively with Python backend

## Containerization & Deployment

**Final Choice:** Docker (with Docker Compose on a VPS)

We use Docker to containerize our frontend, backend, and scan engine services. Docker Compose orchestrates service coordination and simplifies local testing and deployment.

- **Advantages:**
    - Consistent dev/test/prod environments.
    - Enables CI/CD to spin up test containers automatically.
    - Isolation of services ensures one crash doesn’t affect others.
    - Simplifies environment variable and secret management.
- **Disadvantages:**
    - Resource intensive — requires careful optimization on VPS.
    - Learning curve for custom networks and volumes.
- **Justification (Fit):** Docker fits naturally into our **microkernel** and **layered** architecture. Each major component (React, Python, Node.js engine) lives in its own container and communicates via localhost networking. This deployment also satisfies our constraint of **offline demo readiness** — everything runs locally in containers without cloud dependencies.

**Alternative 1:** Heroku

- **Advantages:**
    - Simplifies deployment with buildpacks.

    - Built-in log streaming.
- **Disadvantages:**
    - Vendor lock-in
    - Limited customization of runtime

**Alternative 2:** Railway

- **Advantages:**
    - Minimal config to deploy full stacks.
    - Built-in support for databases.
- **Disadvantages:**
    - Not production-ready at scale
    - Enforces opinionated project structure

## Continuous Integration / Continuous Deployment (CI/CD)

**Final Choice:** GitHub Actions

We use GitHub Actions to automate our CI/CD workflows — including testing, linting, building Docker containers, and deploying to the VPS. Workflow files are version-controlled alongside the project and triggered on pushes, merges, and pull requests.

- **Advantages:**
    - Native to GitHub — no external services required.
    - Fine-grained triggers (e.g., per branch, per tag).
    - Marketplace offers a wide range of reusable actions (e.g., Docker builds, pytest runners).
    - Seamless secrets management.
- **Disadvantages:**
    - Longer startup times than local runners.
    - Limited parallelism on free tier.
- **Justification (Fit):** GitHub Actions aligns perfectly with our **modular architecture** — it can test and deploy each containerized component independently. It supports our quality requirement for **reliability** and ensures consistent deploys across environments, especially important given our Docker-based architecture.

**Alternative 1:** GitLab CI

- **Advantages:**
    - Native integration with GitLab repos
    - Powerful runners and caching features
- **Disadvantages:**
    - Requires migration if repo is on GitHub
    - Slightly more complex YAML config

**Alternative 2:** CircleCI

- **Advantages:**
    - Fast parallel builds
    - Good Docker support
- **Disadvantages:**
    - Separate platform login and billing
    - Configuration more verbose than GitHub Actions

## Security Testing Tooling

**Final Choice:** OWASP ZAP + SQLMap + Supabase Access Control

Our scanning engine uses OWASP ZAP for dynamic vulnerability scanning, SQLMap for injection testing, and Supabase's built-in RLS (row-level security) for enforcing data-level access control.

- **Advantages:**
    - ZAP offers automated scan profiles (e.g., OWASP Top 10).
    - SQLMap enables deep SQLi detection without false positives.
    - Supabase RLS policies prevent unauthorized data access even if tokens are leaked.
- **Disadvantages:**
    - ZAP can produce noisy results if not configured carefully.
    - SQLMap requires live endpoints and may generate load.
- **Justification (Fit):** These tools integrate directly into our **scan engine module**, enabling both dynamic and static analysis. Supabase's row-level access ensures database security — aligning with our **security**, **privacy**, and **availability** requirements.

**Alternative 1:** Burp Suite (Community Edition)

- **Advantages:**

    - Excellent manual tooling
    - Visualizes endpoint flow
- **Disadvantages:**
    - Not easily automatable
    - Commercial edition is expensive

**Alternative 2:** Snyk API

- **Advantages:**
    - Easy to plug into pipelines
    - Scans dependencies (e.g., npm, pip)
- **Disadvantages:**
    - Requires internet connectivity
    - Free tier is limited and requires registration

## Documentation and Developer Tools

**Final Choice:** Markdown + GitHub Wiki + Astro (Starlight)

We maintain documentation using Markdown hosted on GitHub (for internal design and developer docs) and Astro's Starlight framework for generating public-facing documentation.

- **Advantages:**
    - Markdown is simple, version-controlled, and diffable.
    - GitHub Wiki supports collaborative edits and links to issues.
    - Astro's Starlight generates static docs with search and SEO support.
- **Disadvantages:**
    - Astro requires manual setup for styling and routing.
    - No WYSIWYG editor — contributors must understand Markdown.
- **Justification (Fit):** This stack aligns with our emphasis on **developer transparency** and **client visibility**. Documentation lives alongside code and is updated through the same CI flows, ensuring consistency across the board.

**Alternative 1:** Notion

- **Advantages:**
    - Visual editing

    - Easy sharing with stakeholders
- **Disadvantages:**
    - No Git integration
    - Harder to enforce versioning

**Alternative 2:** ReadTheDocs

- **Advantages:**
    - Auto-generates from docstrings
    - Highly customizable themes
- **Disadvantages:**
    - Best suited for Python libraries
    - Needs Sphinx or MkDocs setup

## Design and UX Prototyping

**Final Choice:** Figma

Figma is used for all user interface prototyping and low-/high-fidelity wireframe creation. It allows real-time collaboration among team members and stakeholders.

- **Advantages:**
    - Live collaboration, comments, and version history.
    - Supports responsive layout prototyping.
    - Easily exportable to images or embeddable in wikis/docs.
- **Disadvantages:**
    - Requires internet access.
    - Complex features (e.g., auto layout, variants) have a learning curve.
- **Justification (Fit):** As a design-focused tool, Figma helps define our **Presentation Layer** in the system architecture. It connects directly to frontend implementation through exportable specs, which ensures UI consistency across the React frontend.

**Alternative 1:** Adobe XD

- **Advantages:**
    - Native desktop performance
    - Good integration with other Adobe tools

- **Disadvantages:**
    - Fewer online collaboration features
    - Paid license required

**Alternative 2:** Balsamiq

- **Advantages:**
    - Great for fast low-fidelity wireframes
    - Simplified controls
- **Disadvantages:**
    - Lacks precision for real layout mapping
    - Outdated export options