Architectural requirements document

Architectural Requirements

1 Introduction

This document serves as a blueprint of Skill Issue's approach in defining the architectural requirements for Api Threat-Assessment Tool(AT-AT).

Architectural Design Strategy

We used a decomposition-based strategy focused on modular separation of concerns. Each major functionality in the system is divided into distinct services, such as the Node.js scanning engine, the Python backend, and the React frontend.

- The backend is responsible for authentication, user session control, and routing between services.
- The scanning logic is isolated in a separate service (written in Node.js), communicating with the backend via a socket protocol.
- Each part is testable independently, supporting both unit and integration test pipelines.

This strategy ensures that any new functionality—such as endpoint tagging or alternative scan methods—can be added without disrupting existing services. It also allows CI/CD to run parallel validations on affected modules.

Architectural Strategies

We follow a modular decomposition strategy coupled with explicit command-based interface control. The scan engine and its services are accessible via structured command packets. The backend interprets these, enabling flexible plugin behavior without introducing tightly coupled logic.

- Services are managed via JSON-based commands with a standard structure.
- Files are uploaded, validated, and passed to the scan engine through controlled sockets.
- Global constants and API rate limits enforce robustness.

This enables architectural flexibility and runtime extensibility for future plugins or scanning techniques.

Architectural Quality Requirements

The architecture of AT-AT is designed with several key quality attributes as foundational pillars. These attributes are prioritized according to their importance to the system's mission and directly influenced our implementation choices.

Security is our top priority. Users upload sensitive API specifications that may reflect proprietary system designs. As such, all upload and scan operations are restricted to authenticated users via Supabase JWTs. Passwords are hashed using `bcrypt`, and session tokens are stored in secure, HTTP-only cookies. Backend endpoints are protected via role-based access control, ensuring that only users with the appropriate privileges (e.g., admin vs standard user) can perform sensitive operations such as scanning global APIs or deleting endpoint data.

Performance is critical for maintaining responsiveness during interactive sessions. File uploads and command delegation are optimized through stream-based file handlers (Multer), and scanning operations are delegated asynchronously to the Node.js engine to prevent backend thread blocking. For files under 1MB in size, upload and handoff must complete within 500ms. Command execution responses (e.g., listing endpoints or retrieving metadata) must return within 1 second.

Scalability is supported by the decoupled nature of the scanning engine. The backend acts only as a router; the scan engine may be containerized, horizontally scaled, or replaced without changing the REST API. Our goal is to support at least 100 concurrent users and 10 simultaneous scans without degradation in upload latency.

Availability is supported via Dockerized deployment. Each component is launched independently using Docker Compose, and engine restarts are managed via container health checks. Scans interrupted by engine failure are requeued and resumed upon recovery. Backend services persist uploaded files in a stable volume and can resume processing once restored.

Usability was guided by accessibility constraints: all critical user flows — uploading an API spec, tagging endpoints, triggering a scan, and downloading a report — must be achievable within three UI interactions. UI components report success/failure through color-coded feedback and display JSON-formatted error messages where appropriate.

These quality requirements were not abstract ideals; they directly informed our service interface design, command protocol specification, database schema modeling, and our deployment tooling.

Architectural Pattern

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

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.

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 Playwright for E2E 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: Python

Our scan engine is implemented as a standalone **Python** service. It accepts structured scan requests, executes a modular suite of vulnerability checks, and returns progress/results. This isolates the engine’s logic from the backend while keeping a clean, language-native plugin surface (Python modules).

- **Advantages:**
    - Simple, observable control plane over **HTTP** (no custom TCP protocol).
    - Clear, modular “plugin” model: each check lives as a Python module/function.
    - Excellent developer ergonomics and test support .
    - Straightforward horizontal scaling via multiple service instances.
- **Disadvantages:**
    - Dynamic typing requires discipline (or optional type hints) to keep contracts clear.
    - For very high concurrency, process-level scaling is preferred over cooperative async.

- **Justification (Fit):** This design still embodies a **microkernel pattern**: Python modules make it easy to add/maintain new vulnerability tests. The approach aligns with our quality goals (testability, extensibility) and is consistent with the code currently in the backend.

Alternative 1: Node.js Service (Sockets or HTTP)

- **Advantages:**
  - Event-driven I/O; shares ecosystem with the web stack.
- **Disadvantages:**
  - Duplicates scanning logic already written in Python; less reuse of current code.

Alternative 2: Rust TCP/HTTP Service

- **Advantages:**
  - Memory-safe, very high performance under load.
- **Disadvantages:**
  - Steeper learning curve and slower iteration for new checks during the capstone timeline.

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 + Playwright + Unittest

Our test stack includes `pytest` for backend logic, `unittest` for basic module isolation, and `Playwright` for full-stack end-to-end testing.

- **Advantages:**
  - `pytest`: Clear syntax, great plugin ecosystem (e.g. coverage, fixtures).
  - `Playwright`: Automated browser testing simulates real user behavior.
  - 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