Quality Requirements

Quality Requirements

This document expands on the **Quality Requirements** outlined in the **SRS (Section 4.1.1)**, providing concrete justifications based on implementation decisions made in the AT-AT system.

1. Security

- **JWT-based authentication** ensures secure session management and identity validation.
- **Supabase integration** handles secure user registration and encrypted password storage.
- **Environment variables** protect credentials and service keys (e.g., Supabase URL, secret keys).
- **Scan isolation** prevents user-submitted APIs from interfering with internal services.

2. Performance

- **Efficient scan execution** ensures vulnerabilities are flagged quickly using endpoint tests derived from uploaded OpenAPI and configured checks.
- The **Python backend** is optimized for response time with time-limited scan jobs (default: 60 seconds).
- **Frontend uses async fetch patterns** to avoid blocking the UI during scans and report generation.
- **Load time** is minimized via lazy-loaded React components and route-based splitting.

3. Scalability

- **Modular architecture** allows frontend, API, and backend components to be deployed independently.

- **Containerization (Docker)** makes it easy to scale services across development and production environments.
- **Stateless backend services** can be replicated for load balancing and horizontal scaling in cloud environments.

---

4. Maintainability

- **Separation of concerns** between UI, logic, and processing layers improves code readability and update frequency.
- **GitHub Actions** automate linting, testing, and deployments to reduce manual errors.
- **Clear service contracts** enforce consistent API behavior across updates.
- **Typed JSON structures and standard responses** improve developer onboarding.

---

5. Reliability

- **Error handling middleware** in API and backend layers ensure consistent failure responses.
- **CI/CD pipelines** verify code correctness before deployment.
- **Logging (planned)** will allow better issue tracing and user accountability in production.

---

6. Availability

- *The system should be accessible with minimal downtime.*
- **Container-based architecture** enables rapid service recovery and restart.
- **Stateless service design** supports fault-tolerant deployment in cloud environments.

---

7. Correctness

- *The system should behave as expected and detect threats accurately.*
- **CI tests** and **validation logic** in the backend enforce correct operation.
- Service contracts and clearly defined request/response formats reduce implementation ambiguity.

---

8. Usability

- **Minimal, consistent UI design** with feedback for all user actions (scan start, success/failure, logout).
- **Responsive design** supports multiple device sizes and platforms.
- **Clear system messages** and prompts improve accessibility and reduce user error.

---

9. Portability

- **Cross-platform development** via Docker containers ensures that the system behaves identically on local, testing, and production machines.
- **Environment variable usage** enables quick switching between local, dev, and production settings.

Summary

Quality Attribute	Approach
Security	JWT, Supabase, .env
Performance	Async fetch
Scalability	Docker + modular services
Maintainability	Linting, CI/CD, contracts
Reliability	Error handling + testing
Availability	Stateless containers + cloud support
Correctness	Input validation + CI
Usability	Responsive UI, feedback
Portability	Docker + environment switching