Testing Policy Document

Table of Contents

Introduction

The AT-AT (API Threat Assessment Tool) is a web-based security platform designed to assess the vulnerabilities of RESTful APIs through both specification-based and heuristic scanning. With a focus on security, usability, and automation, the tool must operate reliably under varying loads and configurations.

Given its role in vulnerability detection, accuracy and stability are paramount. This document outlines how AT-AT will be tested across all dimensions — from component-level unit tests to user-facing integration and performance tests — to ensure trust and usability.

Purpose

This Testing Policy defines the testing strategy for validating the functional and non-functional requirements of AT-AT. It ensures a consistent and structured approach to identifying, diagnosing, and resolving defects, with automation at the core of the testing pipeline.

Through this policy, we aim to:

- Reduce regression risks
- Improve user confidence
- Ensure compliance with industry best practices for software quality and security

Objectives

The testing strategy aims to:

- Validate API import and scan functionality under various input conditions
- Simulate user workflows such as uploading API specs, running scans, and exporting reports
- Prevent critical failures (e.g., report generation breaking mid-scan)
- Ensure security features (e.g., authorization checks and token handling) resist misuse
- Continuously test new code commits via GitHub Actions

Edge cases include:

- Users uploading invalid/malformed files
- Scans interrupted mid-execution
- Concurrent users starting scans simultaneously
- API reports with unusually large output sets

Testing Process

Our core testing stack includes:

- Pytest for backend unit/integration tests
- GitHub Actions for CI/CD integration
- Postman (manual API tests) + schema validation

We aim for 75%+ test coverage across core components by the final sprint, focusing first on scanning logic, spec parsing, and user authentication.

Unit Testing

Unit testing is done using pytest (Python) and Jest (if needed for the frontend). Each critical function — such as parse_spec, run_scan, generate_report, validate_token — has its own isolated tests. Stubs/mocks are used for:

- API imports (mocked spec objects)
- Database writes/reads
- External API calls (e.g., threat DBs)

For UI elements (React), individual components (e.g., UploadBox, ScanSelector) are also tested in isolation with mock props and event simulations.

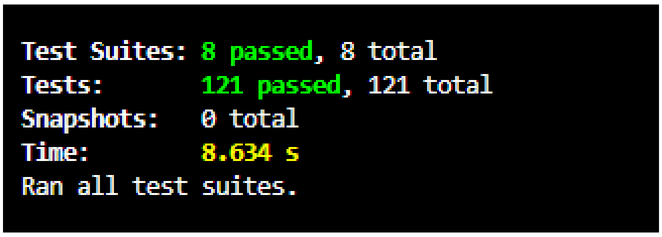
Integration Testing (Stack Clarification)

- **API integrations** use **Jest + Supertest** against the Node/Express app (Supabase and engine TCP calls stubbed).
- **Engine integrations** use **Pytest** with real parsers and mocked outbound HTTP/Supabase.
- *(Optional)* Postman collections may be run locally;

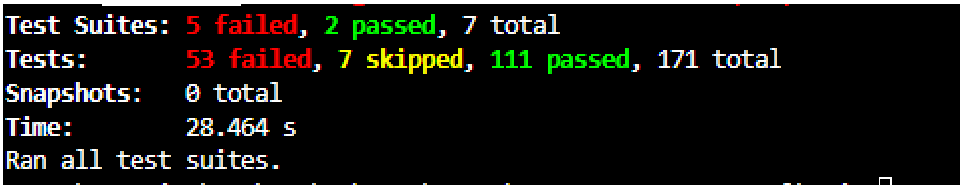
Mocked vs. Live:

- Heuristic scan tests use mocked domain responses
- Auth and DB tests run against seeded containers

Proof of Testing



```
Test Suites: 8 passed, 8 total
Tests:       121 passed, 121 total
Snapshots:   0 total
Time:        8.634 s
Ran all test suites.
```



```
Test Suites: 5 failed, 2 passed, 7 total
Tests:       53 failed, 7 skipped, 111 passed, 171 total
Snapshots:   0 total
Time:        28.464 s
Ran all test suites.
```

Tooling Justification

We use a focused toolset:

- **Pytest** (+ `pytest-cov`) for the Python **engine**: fast fixtures, parameterized tests, and first-class coverage/JUnit XML for CI.
- **Jest + Supertest** (+ `nyc/istanbul`) for the **Node/Express API**: native to Node; Supertest asserts HTTP behavior without real network sockets; LCOV/HTML coverage.
- **React Testing Library + Jest** for **frontend components**: tests UI from the user's perspective; resilient to refactors.
- **Playwright** for **end-to-end (system) tests**: cross-browser, auto-wait, screenshots/videos, HTML report.

- **Apache JMeter** for **performance/NFR**: industry-standard load generator with aggregate/summary reports (ties directly to §4.3 targets).
- **GitHub Actions** for **CI**: first-class GitHub integration, secure secrets, artifact storage, concurrency controls, and matrix builds.

Repository Pointers (Tests & Reports)

- **Backend (Pytest)**: `backend/tests/`
- **API (Jest/Supertest)**: `api/tests/`
- **Frontend (RTL/Jest)**: `frontend/src/__tests__/`
- **End-to-End (Playwright)**: `tests/e2e/`
- **Performance (JMeter plans)**: `tests/perf/`

Testing Procedure

When

- **Pull Requests**: lint → unit → integration ; merge is blocked on failure.
- **Push to `main`**: same as PR + (optionally) build/publish images; nightly **perf smoke** (JMeter).

Who

- Feature author writes/updates tests; reviewer checks coverage and relevance.
- Release captain maintains E2E/perf baselines and approves gates.

How (local)

```
# Engine (Pytest)
pytest --junitxml=reports/junit/backend.xml
       --cov=backend --cov-report=xml:reports/coverage/backend.xml

# API (Jest/Supertest)
cd api && npm test -- --ci --coverage --reporters=default --reporters=jest-junit

# Frontend (RTL/Jest)
cd frontend && npm test -- --ci --coverage
```

Reports generated by CI:

- JUnit XML: `reports/junit/*.xml`
- Coverage (HTML/LCOV/XML): `reports/coverage/{backend,api,frontend}/`
- JMeter (JTL/CSV + screenshots): `reports/perf/`

Coverage & Quality Targets (Demo-4)

- **Overall unit+integration coverage**: ≥ **70%**
- **Critical paths** (auth, tags/flags, scan start/progress/results): ≥ **90%**
- **Performance smoke**: plan completes with **no 5xx**; read-endpoint **p95 ≤ dev target** (see §4.3)
- **No skipped tests** on protected branches (`main`, `develop`)

Performance / NFR Testing (JMeter)

- **Plans**: `tests/perf/*.jmx` (10/50/100 VU steady loads; mixed route set mirroring real usage).
- **Evidence**: aggregate CSV, JTL, and screenshots stored in `reports/perf/`.
- **Targets**: tie directly to §4.3 (p95 latency, throughput, error rate). Nightly CI runs a **smoke** plan; full plan runs before demo/releases.

evidence

Automation and CI Pipeline

All tests are run on GitHub Actions:

- On pull requests: run pytest, eslint, playwright install + test
- On merges: deploy Docker image + run end-to-end tests
- Test reports and logs are stored in build artifacts

Live external dependencies (e.g., Snyk or GitHub Advisory DBs) are simulated unless available during build.

If any test fails, the pipeline blocks the merge and provides logs via GitHub summary output.