Testing Policy Document

For B.R.A.D Bot and Frontend Application

Prepared by: B.R.A.D Development Team

Date: September 2025

Table of Contents

Bot Testing Policy

Purpose

This policy defines the approach and standards for testing the **BRAD bot** to ensure code quality, reliability, and maintainability. Automated tests verify that the bot behaves correctly under various conditions, that regressions are prevented, and that system components integrate correctly.

Frameworks and Tools

- **Pytest**: Used as the primary test framework for unit, integration, and end-to-end tests. Supports fixtures, mocking, and markers (e.g., @pytest.mark.integration).
- **pytest-cov**: Used for collecting and reporting test coverage metrics.
- **Docker Compose (Test Services)**: Used to spin up isolated services (MongoDB, Redis, API) during integration tests.
- **Dramatiq StubBroker**: Used in integration tests to simulate job queue behavior without requiring a live Redis broker.
- **GitHub Actions**: Used to automate the testing pipeline, ensuring tests are executed on every pull request and deployment branch

Justification:

- Pytest is lightweight, widely supported, and fits well with Python-based microservices.
- StubBroker removes the external dependency on Redis for integration tests, reducing test flakiness.
- Travis CI (or GitHub Actions if preferred) provides continuous integration, automatic test execution, and deployment triggers, ensuring code stability before merging or deploying.

Policy

1. Unit Test Creation

- Every new function, class, or module must have a corresponding unit test.
- Unit test files should be stored in a tests/unit/ directory and named to match the source module, e.g.:
- src/scraper/network_tracker.py → tests/unit/test_network_tracker.py
- Unit tests must mock external services (HTTP requests, Docker, Redis) to remain deterministic.

2. Integration Test Creation

- For each new system workflow (e.g., scraping → forensics → analysis reporting), an integration test must be created in tests/integration/.
- Integration tests must validate:
    - Job queue dispatch (using StubBroker).
    - Docker container spawning logic (with docker.from_env stubbed).
    - API calls (report_analysis) being correctly serialized and sent.
    - Forensic and scraping data pipelines producing expected shapes.

3. Test Coverage

Unit and integration tests should cover:

- Core Bot Functions:
  - Domain sanitization and preprocessing (sanitize_domain).
  - Forensic analysis data extraction (gather_forensics).
  - Scraping/crawling (perform_scraping).
  - Report update dispatch (report_analysis).
- Error Handling:
  - Missing or invalid job payloads.
  - API timeouts or failures.
  - Docker errors (e.g., container creation failure).
- End-to-End Flow:
  - A report job is enqueued → processed → scraped → analyzed → patched to API.

4. Test Maintenance

- Tests must be updated alongside code changes.
- Deprecated tests must be removed or refactored if functionality is retired.
- Mock/stub logic must be reviewed regularly to match real API/service contracts.

5. Execution

- Local: Developers run tests with:
- pytest -v --maxfail=1 --disable-warnings -q
- CI/CD: Travis CI automatically runs:
  - Unit tests (fast, mocked).
  - Integration tests (with --maxfail=1).
  - Coverage reporting (pytest --cov=src).
- Merge Policy: No code is merged into main unless all tests pass.
- Deployment Gate: CI/CD ensures tests pass before deployment to staging/production.

Goal

This testing policy ensures the BRAD bot is reliably validated at unit and integration levels, catching regressions early and enforcing confidence in deployments. Automated tests, combined with CI/CD, maintain a consistent quality bar across the bot's lifecycle.

Repository and Reports

- Repository: All tests are stored in the project repo under /tests/unit and /tests/integration.
- Reports: Test results and coverage are available in the CI pipeline logs (Travis CI/GitHub Actions) and coverage reports are generated using pytest-cov.

Frontend Unit Testing Policy

Purpose:

This section defines the approach and standards for unit testing frontend components in our application to ensure code quality, maintainability, and reliability.

Frameworks and Tools:

- **Jest**: Used as the primary test runner and assertion library for all frontend unit tests.
- **React Testing Library**: Used for rendering components, simulating user interactions, and querying DOM elements in a way that reflects real user behaviour.

Policy:

1. ### Unit Test Creation:
   Every time a new frontend component or page is created, a corresponding unit test file must be created. The test file is typically located a centralized __tests__ folder and named to match the component (e.g. __tests__ /login.test.jsx for /app/login/page.jsx) .
2. ### Test Coverage:
   Unit tests should cover:
   - Rendering of the component without errors.
   - State changes and props handling.
   - User interactions, such as input changes, button clicks, and form submissions.
   - Conditional rendering of elements (e.g., notifications, modals, lists).
3. ### Test Maintenance:
   - Unit tests must be updated whenever the corresponding component is modified to ensure correctness.
   - Tests should avoid relying on external APIs or integrations; any required data should be mocked.
4. ### Execution:
   Unit tests are executed using npm test and should pass successfully before merging any changes to the main branch. Test failures should be addressed immediately.

Goal:

This policy ensures that every frontend component is reliably tested at the unit level, helping catch issues early, improve developer confidence, and maintain consistent quality across the application