Checklist for Tests

The Tests category ensures the system functions as expected under various conditions, including unit-level verification, integration between modules, and end-to-end workflows. This checklist ensures comprehensive test coverage and reliability.

---

1. Test Structure

[ ] Organize Tests into Categories:

/unit/: For testing individual functions or modules.

/integration/: For testing interactions between modules.

/e2e/: For simulating complete workflows.

/performance/: For load and stress testing.

/security/: For identifying vulnerabilities.

[ ] Modularity:

Each test file should focus on a specific feature or module.

Use consistent naming (e.g., test\_email\_parser.py, test\_task\_routing.py).

---

2. Unit Tests

[ ] Granularity:

Test individual functions (e.g., email parsing, database queries).

Cover edge cases (e.g., missing required fields, invalid data types).

[ ] Mocking:

Use mock objects for external dependencies (e.g., email servers, databases).

[ ] Validation:

Validate return values and ensure functions raise errors when needed.

---

3. Integration Tests

[ ] Scope:

Test interactions between two or more modules (e.g., email parsing → task creation).

[ ] Data Flow:

Simulate realistic data flows (e.g., valid emails generating valid tasks).

[ ] Error Handling:

Test how the system handles invalid or incomplete data at module boundaries.

---

4. End-to-End (E2E) Tests

[ ] Workflow Testing:

Simulate complete workflows (e.g., email received → workflow executed → task completed).

[ ] User Interaction:

Validate that user-facing interfaces (e.g., dashboards, APIs) trigger the expected backend processes.

[ ] Validation:

Ensure results align with expected outcomes at every stage.

---

5. Performance Tests

[ ] Load Testing:

Simulate high-traffic scenarios for APIs, workflows, and database queries.

Measure response times and system throughput.

[ ] Stress Testing:

Test system behavior under extreme conditions (e.g., CPU, memory limits).

[ ] Scalability:

Ensure the system can handle increasing task volumes or user interactions.

---

6. Security Tests

[ ] Authentication:

Test for vulnerabilities in API token handling, user authentication, and session management.

[ ] Injection Attacks:

Test for SQL injection, command injection, and XSS vulnerabilities.

[ ] Access Control:

Ensure role-based access control (RBAC) prevents unauthorized actions.

[ ] Data Anonymization:

Validate that sensitive workflows (e.g., harassment reports) anonymize data correctly.

---

7. Validation and Coverage

[ ] Required Fields:

Test that all configuration files and inputs include mandatory fields.

[ ] Schema Matching:

Validate that inputs and outputs conform to predefined schemas.

[ ] Error Handling:

Ensure descriptive error messages are logged or returned for invalid actions.

[ ] Test Coverage:

Target >80% coverage across all modules (measured with tools like pytest-cov).

---

8. Logging and Reporting

[ ] Detailed Logs:

Log all test results, including passed, failed, and skipped tests.

[ ] Error Context:

Include stack traces and input data for failed tests.

[ ] Integration with CI/CD:

Automatically run tests on commits using Jenkins, GitHub Actions, or GitLab CI/CD.

---

9. Mocking and Fixtures

[ ] Reusable Mocks:

Create mock email servers, databases, and notification systems for isolated tests.

[ ] Test Data Fixtures:

Use fixtures to set up and tear down test environments.

[ ] Simulated Delays:

Mock network latency or server downtime to test system resilience.

---

10. Automation and Tooling

[ ] Automated Test Execution:

Automate all tests with a single command (pytest or equivalent).

[ ] Test Frameworks:

Use pytest for Python modules and Selenium/Cypress for frontend tests.

[ ] Load Testing Tools:

Use tools like Locust or JMeter for performance testing.

[ ] Vulnerability Scanners:

Integrate tools like OWASP ZAP or Burp Suite for security testing.

---

Example Application for Email Tests

Unit Tests

File: /tests/unit/test\_email\_parser.py

[ ] Validate email parsing for valid/invalid emails.

[ ] Test edge cases (e.g., missing subject, unsupported attachments).

Integration Tests

File: /tests/integration/test\_email\_to\_task.py

[ ] Simulate email parsing triggering task creation.

[ ] Ensure task is routed correctly based on workflow rules.

E2E Tests

File: /tests/e2e/test\_email\_workflow.py

[ ] Simulate the full email-to-task workflow.

[ ] Validate task completion and notification delivery.

Performance Tests

File: /tests/performance/test\_email\_throughput.py

[ ] Simulate 1,000 emails and measure processing time.

---

Optimized Workflow

1. Start with Unit Tests:

Validate individual modules (e.g., email parser, task handler) before integrating.

2. Add Integration Tests:

Test module interactions once unit tests pass.

3. Simulate Real-World Workflows:

Write E2E tests to validate complete workflows under normal and edge conditions.

4. Focus on Automation:

Use CI/CD pipelines to automatically execute tests and track results.

---

This checklist ensures Orgo's tests cover all functionality, edge cases, and performance scenarios. Let me know if you'd like to implement it for a specific module or category!