# Test Strategy for PineCone Pro ERP/IMS

## Purpose and Scope

This document describes the testing strategy for the PineCone Pro ERP/IMS project. It outlines the quality assurance objectives, test levels, and non‑functional test activities required to verify that all modules—Product Information Management (PIM), Inventory & Warehouse, Orders & RMA, Purchasing & Vendor, Shipping & Hazmat, Tax & Reporting, and Accounting Sync—meet their functional and non‑functional requirements. The strategy covers the full testing life‑cycle from unit testing through integration, functional, end‑to‑end, performance, security and accessibility testing. It draws on best practices such as the test pyramid and emphasises automation, early detection of defects and continuous quality feedback.

Testing aims to detect defects early, ensure regulatory compliance, build user trust and protect the business from financial or reputational harm. Automated tests reduce repetitive manual work and provide rapid feedback【85448736678285†L221-L244】. The test pyramid concept advises that most tests should be low‑level unit tests, with fewer integration tests and a small number of expensive, high‑level tests【85448736678285†L267-L289】.

## Test Levels and Types

### Unit Tests

Unit tests exercise individual methods, functions or components in isolation. They are low level, close to the source code, inexpensive to automate and execute quickly in a continuous integration server【237107299464521†L373-L378】. Developers are responsible for writing unit tests for new features and bug fixes. Coverage targets should be established (e.g. ≥ 80 %) and monitored by the CI pipeline. Mocking frameworks (e.g. Moq for C#, Jasmine/Jest for Angular) can isolate dependencies.

### Integration Tests

Integration tests verify that different modules or services work correctly together【237107299464521†L381-L387】. These tests involve actual databases, message queues or microservices. Examples include:

* **PIM and Inventory** – verifying that new products propagate to all inventory locations.
* **Orders & RMA and Accounting** – ensuring that order creation triggers an accounts receivable entry.
* **Purchasing & Vendor** – validating that purchase orders update vendor metrics and inventory counts.

Contract tests should be used when consuming or providing APIs to ensure compatibility across services. Integration tests are run in a controlled test environment that mirrors production but uses sanitized data【984997393224141†L147-L149】.

### Functional Tests

Functional tests validate business requirements without inspecting intermediate system states【237107299464521†L388-L398】. They ensure that each feature behaves as expected for typical and edge‑case inputs. Example functional tests include verifying that hazardous materials orders trigger regulatory shipping documentation or that tax calculations are correct for different states. These tests run against an environment with realistic data and may be automated using frameworks such as Cypress or Selenium for the web front‑end and REST‑assured for APIs.

### End‑to‑End (E2E) Tests

E2E tests replicate user behaviour across the entire application stack. They verify user journeys such as browsing the catalogue, adding items to the cart, completing checkout, receiving an order confirmation, and initiating an RMA. E2E tests ensure that all components—front‑end, back‑end services, databases, and integrations—work together【237107299464521†L400-L406】. Due to their complexity and maintenance cost, only critical user flows should be covered by automated E2E tests, while lower‑level tests catch most regressions【237107299464521†L407-L410】. Manual exploratory sessions supplement automation to discover non‑obvious issues【237107299464521†L464-L475】.

### Acceptance Tests

Acceptance tests verify that the system meets defined requirements and acceptance criteria【237107299464521†L412-L418】. They may be automated or manual and often involve business stakeholders. Acceptance criteria developed during requirements gathering (e.g. 98 % inventory accuracy, 95 % orders shipped within 24 h) become the basis for these tests. Performance goals such as response times and throughput should also be validated here.

### Smoke Tests

Smoke tests are lightweight tests that exercise the most important functions to ensure the system is stable after a build or deployment【237107299464521†L430-L439】. Examples include verifying that the application loads, logging in works, and the order workflow can be initiated. Smoke tests run immediately after deployment to any environment to decide whether more expensive tests should proceed.

### Exploratory Tests

Exploratory testing sessions involve testers using the application without predefined scripts to uncover unexpected issues. These sessions should be time‑boxed (e.g. two hours) and focus on specific areas【237107299464521†L464-L478】. Testers attempt invalid inputs, unusual workflows and edge cases to identify defects that automated scripts may miss. Findings from exploratory testing feed back into the automated suite.

## Non‑Functional Testing

### Performance Testing

Performance tests evaluate how the system behaves under various workloads. They measure reliability, speed, scalability and responsiveness【237107299464521†L420-L428】. Key objectives include:

* **Load testing** – verifying that core workflows (e.g. order placement, purchase order creation) meet response time targets under expected user loads.
* **Stress testing** – determining breaking points by increasing load beyond expected levels and ensuring the system fails gracefully.
* **Soak testing** – running the system under steady load for extended periods to detect memory leaks or resource exhaustion.
* **Capacity testing** – measuring how many concurrent users/orders the system can handle before performance degrades.

Performance tests should run in an environment that reflects production (hardware, network, data volumes). Tools such as JMeter, k6 or Locust can simulate load. Results inform capacity planning and trigger tuning of queries, indexes and caching. The non‑functional requirements define baseline response times and throughput for each API.

### Security Testing

Security testing evaluates the system’s resilience against threats and ensures compliance with regulations. The PractiTest article outlines key best practices: identify security requirements based on applicable standards (e.g. PCI DSS, HIPAA) and perform risk assessments【360412862343572†L264-L295】; design security tests that simulate real‑world attacks, including vulnerability scans and penetration testing【360412862343572†L297-L303】【360412862343572†L314-L324】; execute tests systematically using automated tools and involve security experts; analyze results, prioritise vulnerabilities by severity, and fix them【360412862343572†L329-L354】; retest to verify fixes and report results to stakeholders in clear language【360412862343572†L361-L377】. Common security test types include:

* **Vulnerability scanning** – automated scanning for known weaknesses such as outdated dependencies, misconfigurations or weak passwords【360412862343572†L153-L162】.
* **Penetration testing** – ethical hackers attempt to exploit authentication, authorization or input handling flaws【360412862343572†L166-L179】.
* **Risk assessment and threat modelling** – identifying potential threats and their impact【360412862343572†L283-L290】.
* **Static and dynamic analysis** – scanning code and running applications with tools like OWASP ZAP, SonarQube or Snyk.
* **Compliance testing** – ensuring adherence to standards such as SOC 2, GDPR or hazardous‑materials shipping regulations.

Security tests should be integrated into the CI/CD pipeline. Automated scans run on each commit, while periodic penetration tests and audits occur before major releases.

### Accessibility Testing

Accessibility testing ensures that the web application is usable by people with disabilities and complies with standards like WCAG 2.1. The University of Wisconsin–Madison guide recommends the following practices:

* **Combine automated and manual testing.** Automated tools (e.g. WAVE, Google Lighthouse, Axe) can detect technical barriers such as missing alt text, low colour contrast, improper heading hierarchies and inaccessible forms【923407713770177†L90-L120】. Manual testing is required to discover functional barriers such as keyboard navigation, logical reading order and accessible dynamic content【923407713770177†L162-L175】.
* **Perform keyboard navigation tests** by tabbing through all interactive elements, ensuring a visible focus indicator, and verifying logical navigation order【923407713770177†L183-L197】. Check that “skip to main content” links work and that menus close with the Esc key.
* **Conduct screen reader tests** using VoiceOver, NVDA or JAWS to verify that headings, alt text and navigation order convey meaning correctly【923407713770177†L210-L224】.
* **Use automated page scanners and site crawlers** to evaluate large portions of the site, but review the reports and ensure manual review for images, banners, charts or forms【923407713770177†L123-L157】.

Accessibility tests are part of both unit (e.g. verifying ARIA attributes in components) and E2E test suites. Failures block promotion until resolved.

### Reliability and Resilience Testing

* **Chaos/Resilience tests** simulate failures of dependencies (e.g. database outage, message queue slowdowns) to verify that services degrade gracefully, time out correctly and recover without data loss. Injecting faults in a controlled environment reveals weaknesses before they occur in production.
* **Backup and recovery tests** validate that data can be restored from backups and that failover procedures meet recovery objectives.

## Test Environment and Data

Different environments are used for testing, each requiring appropriate data:

* **Development:** synthetic data or dummy records to test units and small integration slices. Developers can create data in their local containers.
* **QA / Staging:** mirrors production infrastructure and uses sanitized production data【984997393224141†L147-L149】. Test data sets cover typical and edge cases across all modules.
* **UAT:** sanitized data with full workflows for acceptance tests.
* **Performance & Security test environments:** dedicated clusters configured like production to avoid impacting regular testing. Data volumes reflect expected load.

Test data management processes must anonymize personal or sensitive information and seed data sets deterministically so tests are repeatable.

## Test Automation Framework

Automation is vital for fast feedback and reliability【85448736678285†L221-L244】. Each test layer uses appropriate tools:

| Test Level | Example Tools | Notes |
| --- | --- | --- |
| Unit | xUnit/NUnit for C#, Jest for Angular, Mocha/Chai | Run on each commit; high code coverage; run in isolated containers |
| Integration | Postman/Newman, REST‑assured, Pact for contract tests | Use real databases and microservices; simulate message queues; seed test data |
| Functional / UI | Cypress, Selenium, Playwright | Automate user flows across the Angular front‑end and APIs; run nightly or pre‑release |
| E2E | Cypress, Selenium with BDD frameworks like Cucumber | Cover critical user journeys end to end; limited number due to cost |
| Performance | JMeter, k6, Locust | Simulate concurrent users, orders and vendor interactions; record response times, throughput and resource usage |
| Security | OWASP ZAP, Burp Suite, Snyk, SonarQube | Integrated into CI pipeline for scans; periodic penetration tests; track vulnerability backlog |
| Accessibility | aXe, Lighthouse, WAVE, Pa11y | Automated scanning; integrate into CI; manual keyboard and screen reader testing in QA |

CI pipelines must run unit and integration tests on each pull request; functional, E2E and non‑functional tests run nightly or before major releases. Failing tests break the pipeline and prevent promotion.

## Metrics and Reporting

Measuring test effectiveness helps improve quality. Key metrics include:

* **Test coverage:** percentage of code exercised by automated tests.
* **Pass/fail rates:** number of tests passing or failing per build.
* **Defect leakage:** defects found in later stages (e.g. production) that should have been caught earlier.
* **Mean time to detect (MTTD) & mean time to resolve (MTTR):** tracked through DORA metrics【379199922569039†L343-L352】.
* **Performance indicators:** response times, throughput, error rates from performance tests.
* **Security vulnerability counts:** number and severity of issues detected in scans.
* **Accessibility compliance score:** percentage of pages passing automated accessibility checks.

Test results and metrics are published to dashboards accessible to the team and stakeholders. Regular reviews of metrics guide improvement initiatives.

## Roles and Responsibilities

* **Developers** write unit tests, assist with integration tests and fix defects.
* **QA engineers** design and maintain integration, functional, E2E, performance and accessibility tests; manage test environments; perform exploratory testing.
* **Security analysts** conduct penetration tests, review vulnerabilities and ensure compliance with security requirements.
* **Accessibility specialists** guide compliance with WCAG and assist with manual testing.
* **Product owners and business analysts** define acceptance criteria and participate in acceptance testing.

## Test Schedule and Release Readiness

Testing activities are planned for each sprint and release. Unit and integration tests run continuously. Functional and E2E tests run at least nightly and before any production release. Performance, security and accessibility tests run at defined checkpoints (e.g. before major milestones or once per iteration). A release cannot proceed unless:

1. All critical unit, integration, functional and E2E tests pass.
2. Performance tests meet or exceed non‑functional requirements.
3. Security scans show no high‑severity vulnerabilities and medium‑severity issues have remediation plans.
4. Accessibility checks report no critical issues.

## Conclusion

By following this comprehensive test strategy, PineCone Pro ensures that its ERP/IMS is robust, secure, performant, and accessible. Embracing the test pyramid—with many unit tests, some integration and functional tests, and a few end‑to‑end tests—provides rapid feedback and maintainable test suites【85448736678285†L267-L289】. Integrating performance, security and accessibility testing into the development life‑cycle, supported by automation and clear metrics, helps deliver a high‑quality solution that meets customer expectations and regulatory obligations.