Ozymandias Perfect Holidays Quality Plan

Héctor Álvarez López

This Quality Plan guarantees that the new version of **Ozymandias Perfect Holidays** maintains the **current behavior**, preserves the asynchronous queue, uses the weather service, and meets **1 000 req/s** (p95 < 200 ms, < 1 % errors).

It is based on a component-based approach (frontend, API, façade, messaging, DB) and on the principles of **Shift-Left**, **TDD/BDD**, and **Continuous Integration**.

1 · Purpose and Goals

This Quality Plan ensures that the new version of Ozymandias Perfect Holidays meets:

- **Functional equivalence:** exactly reproduces the current system's recommendations and workflows, while accounting for any variations introduced by newer technologies or business change requests.
- **Secure integration:** validates in staging against the production database (read-only access to catalog and write access to the requests table).
- **Messaging reliability:** preserves integrity and ordering within the asynchronous queue.
- **Weather connectivity:** maintains stable consumption of the OpenWeatherMap API via the Weather Façade.
- **Performance:** supports approximately **1 000 req/s** with p95 latency < 200 ms and an error rate < 1 %.

2 · Architecture and Quality Approach

The system consists of five main layers:

- **Front-end Web/Mobile**
- **Holiday Service (REST API)**
- **Weather Façade**
- **Asynchronous Queue**
- **Holidays Database**

For each layer, we will consistently apply the principles outlined in Section 3 along with the AI use cases

Section 4 dives into the specific testing strategy for each component.

3 · Principles of the Strategy

The quality strategy is built on proactive and continuous validation, deeply integrated with AI. It emphasizes:

- **Early Definition & AI-Refinement:** Defining "Given-When-Then" acceptance criteria from the start (Shift-Left), using AI to refine them against existing system flows.

- **CI-Driven Quality:** Every code commit triggers a pipeline for static analysis (SonarQube), unit testing (targeting ≥85% coverage), and performance checks, ensuring issues are caught immediately.

- **Production Parity:** Before full rollout, the new system's behavior is validated by comparing its outputs against the current production system (Golden-Master) and by testing with a subset of real user traffic (shadow-traffic).

- **Automated Functional & Component Testing:** HTTP API contracts (using Karate) and critical UI user journeys (using Playwright/Selenium) are automated and run regularly within CI/CD.

- **Rapid Health Checks:** Lightweight smoke tests are executed on every build for both API endpoints and frontend pages to ensure basic functionality.

- **Continuous Integration & Regression with AI:** Full system integration is tested on every merge, complemented by a comprehensive nightly regression suite. AI assists in selecting relevant tests, prioritizing them, and helping debug failures.

3.1 Shift-Left Testing

We define Given–When–Then criteria as part of the Definition of Ready, contrasting them against the existing application to reuse proven flows.

AI use cases:

- Automatic generation and refinement of acceptance criteria from the story description.
- RAG (Retrieval-Augmented Generation) against the repository to validate legacy flows, disambiguate requirements, and suggest reuse.

3.2 Continuous Integration and Code Quality

Every commit triggers a pipeline that runs linting, static analysis (SonarQube), and unit tests ($\geq$ 85% coverage). Only when all checks pass is the build image created. The same pipeline also executes performance tests in staging to verify endpoint latencies and throughput.

AI use cases:

- Semantic identification of changed files to execute only the affected tests.
- Commit classification to trigger performance tests based on impact.

3.3 Production Comparison (Golden-Master / Shadow-Traffic)

In staging, we execute the same test cases in parallel against both the current production version and the new version, generating semantic diffs of their JSON outputs. Additionally, we mirror a small percentage of real traffic to validate behavior live before proceeding with the full deployment.

3.4 Component Testing and Functional Automation

- **Karate** validates HTTP contracts for the Holiday Service and Weather Façade using mocks and stubs.
- **E2E front-end** tests with Playwright/Selenium cover critical flows (login, search, filters) and include accessibility checks.
- Both suites run on every merge to **main** and nightly against staging-cloned environments.

AI use cases:

- Automatic script generation (OpenAPI → Karate / Gherkin → Playwright/Selenium).
- Auto-curation of UI selectors when attributes change.
- Prioritization of cases based on coverage and risk.

3.5 Smoke Testing

Lightweight tests on each build that validate vital endpoints/pages:

- **Backend:** `/health`, `/login`, `/recommendations`
- **Front-end:** page load checks (login, dashboard), form rendering, basic navigation (logout, pagination)

AI use cases:

- Automatic generation of new smoke checks based on historical failure patterns.
- Creation of smoke UI scripts from application snapshots.
- Intelligent monitoring of failure trends to suggest extensions.

3.6 Integration and Regression Testing

- **Integration:** full interaction Holiday Service ↔ Database ↔ Queue ↔ Weather Façade on every merge to **main**.
- **Regression:** nightly execution of the complete suite (unit, integration, E2E, mutation).

AI use cases:

- Semantic selection of the most relevant regression tests after each change.
- Intelligent clustering of failures to pinpoint critical areas and recommend improvements.

4 · Testing Strategy by Component

We apply the principles from Section 3 and the AI use cases in each area:

4.1 Front-end Web/Mobile

- **Shift-Left & BDD:** Given–When–Then criteria versioned alongside the design.
- **Smoke UI:** Playwright validates page load, rendering, and navigation on every build.
- **E2E & Automation:** Critical flows (login → recommendations → filters) run on merge and nightly.
- **Visual Comparison:** Screenshots in staging vs. baseline to detect layout regressions.

4.2 Holiday Service (REST API)

- **TDD & Unit Tests:** ≥ 85 % coverage on `/login`, `/recommendations`, and `/health`.
- **Smoke API:** Health checks on status, response times, and JSON schema.
- **Contract & Integration:** Karate validates contracts; stubs isolate the Weather Façade.
- **Golden-Master:** Semantic comparison of production vs. staging outputs.
- **Performance:** k6/Locust simulate 1 000 req/s, measuring p95 and error rate.
- **Observability:** Distributed tracing and metrics for latency, QPS, and errors.

4.3 Weather Façade Service

- **Contract Tests:** JSON schemas for OpenWeatherMap responses via VCR.py.
- **Resilience & Chaos:** Timeouts, 5xx errors, and extreme latencies; IA-generated scenarios.
- **Cache Performance:** Throughput and latency of the cache layer under concurrency.
- **Observability:** Success/failure rates, latency, and retry metrics.

4.4 Asynchronous Queue

- **Unit & Mutation Tests:** Producers/consumers, ack/nack, retries, and prioritized mutation coverage.
- **Integration:** Durability and ordering validated via snapshots; broker chaos tests.
- **Performance:** Throughput and p95 latency < 1 s under load.
- **Observability:** Queue depth, retry counts, and connection-error metrics.

4.5 Holidays Database

- **Seed & Edge Data:** deterministic script populates special-case records.
- **Queries & Concurrency:** tests for concurrent reads and writes.
- **Query Performance:** measuring SELECT and JOIN times under load.
- **Migrations & Schema:** automatic validation of migration scripts against snapshots.
- **Observability:** query latency and active connection metrics.

5 · Quality by Sprint and Agile Phases

Continuous integration of QA throughout each sprint phase:

5.1 Refinement and Preparation

Definition of Given–When–Then criteria in the backlog, validated against the existing application and refined by an LLM.

5.2 Development and TDD

Unit tests written before code and BDD scenarios for critical flows; AI suggests test skeletons.

5.3 Pull Request and CI Gate

Pipeline executes linting, static analysis, unit tests, smoke tests, and ephemeral integration tests; merges only when everything is green.

5.4 Post-Merge and Continuous Regression

Customized regression suite runs after merging into **develop**, including E2E integration and semantic test selection; nightly full-suite execution and shadow-traffic on **main**.

5.5 Retrospective and Continuous Improvement

Review of metrics and defects during the retrospective, acceptance criteria are adjusted and test-suite improvements are prioritized.

6 · Key Metrics

- **Code coverage:** ≥ 85 % (lines/branches)
- **p95 latency for `/recommendations`:** < 200 ms at 1 000 req/s
- **Overall error rate:** < 1 %
- **Asynchronous queue p95 depth:** < 1 s
- **BDD scenario success rate on main:** > 95 %

7 · Impacted Areas and Responsibilities

- **Business Analysts / Product Owners**
  - Define and maintain Given–When–Then criteria.
  - Validate requirements against both legacy and new flows.

- **Development**
  - Implement TDD and BDD; write unit tests and *`.feature`* scenarios.
  - Integrate linting, SonarQube and pytest-cov.
  - Collaborate with QA on mocks and stubs.

- **QA Automation**

  - Design and automate test plans per user story.
  - Select and maintain test collections for smoke, integration, E2E, regression and performance suites.
  - Keep the CI/CD test suite healthy; adjust coverage via AI.
  - Produce Golden-Master reports and monitor mutation testing.

- **SRE / DevOps**
  - Maintain CI/CD pipelines and staging environments.
  - Orchestrate infrastructure (containers, queues, DB).
  - Implement distributed tracing, dashboards and alerts.
  - Manage shadow-traffic routing and automated rollback.

- **DBA / Data Engineering**
  - Provide production snapshots and seed scripts.
  - Anonymize data and ensure GDPR compliance.
  - Monitor query latencies and concurrency on critical workloads.

8 · Assumptions and Strategy Adaptability

Our quality strategy relies on these core assumptions:

- **Agile & BDD Focus:** We operate with an Agile approach and agile teams, utilizing Sprints, CI/CD, Shift-Left principles, and BDD. The immediate aim is functional equivalence with the current system, likely as an MVP.
- **GitOps Driven:** A GitOps methodology underpins our infrastructure and deployment, using Git as the single source of truth for automation.
- **Stable & Similar Environments:** Both staging and production environments are stable and sufficiently alike to ensure valid testing.
- **Controlled Production Data Access:** We have secure, managed access to necessary production data in staging (e.g., read-only for catalogs, write access for new requests), always respecting data privacy.
- **Production Comparison Feasibility:** Golden-Master testing (comparing outputs against production) and Shadow-Traffic mirroring are technically feasible for robust validation.

This strategy is designed to be flexible. As the system, MVP, and environments mature, we will adapt our testing scope, tools, and processes. This ensures we meet evolving business requirements and technical realities, maintaining a continuous focus on quality improvement.