Test Plan Document

Prepared By Anitha Paramashivam

1. Purpose

The purpose of this test plan is to validate the functional correctness, reliability, and robustness of the Order Assembly Service (OAS). The service plays a critical role in the order processing pipeline by validating incoming order payloads, enriching them with Card Catalog metadata, and publishing the assembled result to a downstream SQS queue for further processing. This test plan ensures that OAS behaves deterministically under normal conditions and fails safely when dependent systems are unavailable.

2. P0 Requirements

Prompt:
Role: Senior Automation QA Engineer
My Ask: Add a short contextual paragraph for the following bullet points
Functional testing verifies the expected success behavior. A primary success scenario is submitting a well-formed order request that includes required fields and multiple items. The service is expected to return 200 OK and publish a single enriched message to the queue. The validation also confirms that each sku and quantity from the input is preserved and that enrichment adds the required metadata fields without mutating the meaning of the original order.

Functional Requirements
Use Cases:

- The service must validate all required fields (order_id, customer_id, items, order_ts) and reject invalid payloads.
- The items array must not be empty, and each item must contain a valid SKU and a quantity greater than zero.
- Each valid order must be enriched using Card Catalog metadata before publishing.
- The service must publish exactly one enriched message to the assembled-orders SQS queue for each successful request.
- The service must return 200 OK only after successful enrichment and message publication.
- No message must be published if validation or enrichment fails.

Error Handling Requirements
Use Cases:

- Invalid input must return 400 Bad Request.
- Failures from dependent systems (catalog or queue) must return either 502 or 503 as defined.
- Partial processing must not result in partial message publication.
- Errors must not cause duplicate or inconsistent messages.

Reliability & Safety Requirements
Use Cases:

- The service must not publish malformed or incomplete orders.
- Failures must be observable through logs or error responses.
- Duplicate submissions must be handled

3. Scope

In Scope

- Request validation
- Order enrichment logic
- Integration with Card Catalog
- Publishing to SQS
- Error handling and response validation
- Functional, negative, and integration testing, basic performance and reliability testing

Out of Scope

- Downstream consumers of the SQS queue
- Card Catalog internal logic
- UI or frontend behavior
- Authentication and authorization (unless explicitly required)

4. Assumptions and Constraints

- SKU validation is performed via the Card Catalog or a predefined lookup.
- Enrichment includes adding metadata to each item (exact fields to be confirmed).
- SQS guarantees at-least-once delivery.
- Message size does not exceed SQS limits.
- Error mapping between dependency failures and HTTP status codes must be clarified and locked.
- Idempotency behavior is undefined and must be explicitly tested and documented.

5. Test Strategy

Prompt:
Role: Senior Automation QA Engineer
My Ask: Add a short contextual paragraph under Test Strategy section in a Test Plan document for the following bullet points
Testing will be executed across multiple layers to ensure correctness and resilience. Testing focuses on verifying correct interaction between internal components and external dependencies such as the Card Catalog service and the SQS messaging layer. Integration and end-to-end tests validate that requests flow correctly through validation, enrichment, and publishing stages, while negative and failure scenarios ensure the system behaves predictably under error conditions. Test execution is supported through isolated local environments using mocks with automated runs integrated into CI pipelines to ensure continuous quality and early defect detection.

Test Levels

- **Integration Tests** to validate interaction with Card Catalog and SQS – API validation.
- **End-to-End Tests** to validate full request → enrichment → queue flow.
- **Negative & Failure Tests** to validate error handling and recovery.
- **Non-functional Tests** to assess reliability, performance, and fault tolerance.

Test Environment

- Local environment using mocks for SQS
- Stubbed Card Catalog service
- Continuous Integration for automated execution

6. Functional Test Coverage

Happy Path Validation

- Submit a valid order with multiple items.
- Verify response is 200 OK.
- Verify exactly one message is published to the queue.
- Verify message contains:
  - Original order fields
  - Enriched metadata for each SKU
  - Correct quantities
- Verify message format matches expected schema.

Validation Scenarios

- Missing required fields
- Empty items list
- Missing SKU or quantity
- Quantity ≤ 0
- Invalid timestamp format

- Unknown SKU

Expected result for all validation failures:
- HTTP 400 response
- No SQS message published

7. Dependency & Failure Testing

Card Catalog Failures
- Timeout from catalog service
- 5xx response from catalog
- Partial or malformed enrichment data

Expected behavior:
- Service returns 502 or 503
- No message published

SQS Failures
- Queue unavailable
- Permission failure
- Publish timeout

Expected behavior:
- Service returns 503
- No partial or duplicate messages

8. Edge Cases

Each of these scenarios must be explicitly tested or documented as a known limitation.

- Duplicate order submissions (idempotency)
- Partial enrichment of items
- Schema changes from catalog service
- Message size limits
- Retry behavior causing duplicate publishes
- High-volume request bursts
- Ordering guarantees if FIFO is required

9. Non-Functional Testing

Prompt:
Role: Senior Automation QA Engineer
My Ask: Add a short contextual paragraph in 1 to 2 lines under Non-Functional section in a Test Plan document for the following bullet points
Non-functional testing is included to assess performance, resilience, and fault tolerance, ensuring the service can operate reliably in real-world production environments

Performance
- Validate response times under normal and peak load
- Validate behavior with large orders
- Ensure no blocking calls delay responses

Reliability
- Simulate dependency failures
- Verify graceful degradation
- Ensure no data loss

Observability
- Logs include order_id for traceability

- Errors clearly indicate failure reason
- Metrics available for success/failure rates

10. Entry and Exit Criteria

Entry Criteria

- API endpoint available
- Test environments configured and completed smoke test
- SQS and catalog mocks available
- Test data defined

Exit Criteria

- All P0 test cases pass
- No critical defects open
- All failure paths validated
- Automation suite runs successfully in CI
- No scenario produces silent failures or inconsistent data

11. Automation Strategy

Purpose of Automation

The goal of automation is to ensure high confidence, fast feedback, and production reliability for an event-driven architecture. Since OAS validates requests, enriches data, and publishes messages to downstream consumers, any failure can result in data corruption, order loss, or cascading failures.

Automation is designed to:

- Detect defects early in the SDLC
- Prevent regressions in validation and enrichment logic
- Ensure correctness of SQS message publishing
- Reduce manual testing effort
- Provide measurable quality signals for release decisions

Automation Scope

Automation will focus on API-level and integration-level validation, where the highest risk and business impact exist.

In-Scope

- API contract validation (POST /orders/assemble)
- Request validation logic
- Enrichment verification using Card Catalog data
- SQS message publishing and validation
- Error handling (400 / 502 / 503)
- Idempotency and duplicate submission checks
- Data-driven negative test coverage
- End-to-end flow validation using mocks

Out of Scope

- UI testing
- Downstream consumer validation
- Full load/stress testing (handled separately)
- Catalog service internal logic

ROI (Return on Investment)

Metric	Impact	Estimated ROI
Regression Time	Reduced from ~5 hours → ~10 minutes	Time saved per release: ~4–5 hours
Manual Effort	Reduced by ~70–80%	Annual savings: ~200–300 engineering hours
Defect Leakage	Reduced significantly	Defect prevention impact: High (prevents data corruption & queue issues)
Release Confidence	High	Stable CI signal for releases
Scalability	Linear with test data	Repeatable API flows

Automation Framework Design

Tech Stack

- Playwright (API testing) – primary automation framework
- Node.js / TypeScript – test execution layer
- Mock Server / Stub – Card Catalog dependency
- AWS SDK – queue validation
- CI/CD – GitHub Actions / Jenkins

Key Design Principles

- API-first validation (no UI dependency)
- Data-driven execution
- Clear separation of test logic and test data
- Deterministic validation (no flaky tests)
- Fast execution suitable for CI
- Structured logging

Prompt:

Role: Senior Software Test Engineer

My Ask: Develop an automation framework design for API validation using Playwright in mvc concept

```
tests/
├── api/
│   ├── workflows.spec.ts
│   ├── validation.spec.ts
│   ├── negative.spec.ts
│   └── failure_scenarios.spec.ts
│
├── fixtures/
│   ├── valid_orders.json
│   └── invalid_orders.json
│
├── helpers/
│   ├── sqsHelper.ts
│   ├── catalogMock.ts
│   └── requestBuilder.ts
```

```
├── schemas/
│   ├── order.schema.json
│   ├── enriched.schema.json
│
└── utils/
    ├── env.ts
    ├── assertions.ts
```

KPIs & Quality Metrics

Automation Metrics	Quality Metrics	CI Health Metrics
Automation Coverage: ≥ 85% of critical paths	% Production Defect Rate	Test Execution Duration
Pass Rate: ≥ 98% on CI runs	Mean Time to Detect (MTTD)	Failure Trend
Flakiness Rate: < 2%	Mean Time to Recovery (MTTR)	Retry Frequency
Execution Time: < 5 minutes per run	Defects per Release	Test Stability Score
Defect Escape Rate: < 5%	Failure Rate per Deployment	

Efficiency & Impact

Efficiency Gains

- Faster feedback cycles
- Reduced manual QA effort
- Early detection of breaking changes
- Stable CI signal for releases

Business Impact

- Prevents bad data from entering downstream systems
- Reduces incident response overhead
- Improves reliability of order processing
- Builds confidence for scaling traffic

Risks & Mitigation

Risk	Impact	Mitigation
Over-mocking hides real issues	Medium	Add limited staging validation
SQS behavior differs in prod	Medium	Validate message schema strictly
Flaky dependency tests	High	Use deterministic mocks
Schema drift	Medium	Contract tests + schema validation
Duplicate events	High	Idempotency checks

12. CI/CD, Observability & Telemetry

Pipeline Structure

The automation test suite triggers test run for end-to-end workflows or sanity test flows in a cadence on every change to OAS in the Git. Pipeline workflow stages can be developed using AWS cdk or Jenkins. A standalone pipeline has unit tests coverage part of code commit achieving fast fail, followed by API framework, integration tests providing functionality coverage, north-bound and south-bound dependency coverage using LocalStack or mock data.

Test Execution – AWS Device Farm/VM/Grid

Tests execute parallelly either in local VM environment or Grid or AWS Device Farm in pipeline approval workflows.

Test Environments

Git Code Commit → Triggers Tests in QA Env → Pre-Prod Env → Prod Deployment → Monitoring → Dashboard

- Test suites (API, Integration and Regression suites) are executed in QA environment and upon successful execution, the pipeline promotes to next stage which is Pre-Prod environment.
- The Test-suites (API and Integration) are executed in Pre-Prod environment with either mock, seed or dynamic data. After successful completion pipeline merges with Dev pipeline as a workflow approval and promotes Prod deployment. As part of Prod deployment and post deployment – a canary pipeline validates P0 end to end test flows every one hour once using Jenkins Job scheduler.

Monitoring & Dashboard

The pipeline publishes HTML test reports for every stage which is provided by Playwright's features. Artifacts - logs, traces, screen captures (in case of UI), request/response payloads, queue message bodies on failure and gate merges/releases - can be accessed via Jenkins or Cloud Watch depending on execution environment.

13. Brainstorming & Analysis

Why Playwright? What other tools were considered?

Playwright was chosen as the primary automation framework because it provides strong support **for** API testing, fast execution, built-in assertions, and reliable handling of asynchronous workflows, which aligns well with the requirements of the Order Assembly Service. Since the system is backend-heavy and event-driven, Playwright's API testing capabilities allow direct validation of request/response behavior, enrichment logic, and integration with downstream systems such as SQS without the overhead of UI automation. Its native support for parallel execution, rich debugging, and seamless CI integration makes it highly suitable for scalable automation.

Tools and Frameworks Considered

Postman was considered for API testing but was not selected due to limited flexibility for complex assertion. Pytest with Requests was another option and works well for API automation, but it requires additional setup for reporting, parallelization, and cross-service orchestration, whereas Playwright provides these capabilities out of the box. RestAssured was considered but would introduce Java-based overhead and slower iteration cycles compared to Playwright's lightweight execution model.
Overall, Playwright was selected because it offers the best balance of speed, reliability, maintainability, and integration support for validating API workflows, message-driven systems, and CI-based automation at scale.

14. Pitfall Identification & Gap Analysis

1. Reliability & Data Integrity Risks

Gaps

- No idempotency handling for duplicate requests
- No defined behavior for partial enrichment failures
- No clear strategy for catalog service outages

Risks

- Duplicate orders being processed
- Inconsistent or corrupted order data
- Cascading failures when dependencies fail

Recommendations

- Enforce idempotency using order_id
- Add strict rules
- Add timeouts, retries, and guardrails for catalog calls

2. Messaging, Retry & Scalability Risks

Gaps

- No retry mechanism
- No rate limiting

Risks

- Retry storms under failure
- Unpredictable downstream behavior

Recommendations

- Implement retry with exponential backoff
- Apply rate limiting and retry caps

3. Observability, Error Handling & Security Gaps

Gaps

- No standardized error responses
- No tracing or metrics
- No authentication or input validation defined

Risks

- Difficult debugging and incident resolution
- Inconsistent client behavior
- Security vulnerabilities

Recommendations

- Standardize error schema with trace IDs
- Add structured logging and metrics
- Enforce authentication and input validation