Software Testing Plan

Date: 3/29/2026
Team Name: Evergreen Systems
Project Sponsor: Kyle Montgomery
Faculty Mentor: Dr. Ana Paula Chaves
Team Members:

- (Team Lead) Asher Romanenghi
- Mark Johnson
- Tyler Sturm
- Melvin Agram

Overview: This document provides exhaustive details on the testing plan implemented for Generous Commerce Connectors using unit testing, integration testing, and acceptance testing

1. Introduction

Generous is an AI-powered gifting platform that depends on accurate merchant catalog data, dependable processing workflows, and clear merchant-facing setup experiences in order to function successfully. The portion of the system covered by this testing plan focuses on the Generous merchant catalog flow, which connects merchant product data to the broader platform through onboarding, feed generation, feed upload, and feed ingestion.

The intended users of this system include merchant-side operators working in Salesforce B2C Commerce and Business Manager, technical merchant integrators interacting with the ingestion API, and the Generous team members who rely on these workflows to support demos, onboarding, and long-term platform growth. Because the system serves both business and technical users, the quality bar has to be high in more than one dimension.

The primary quality goals of this project are correctness, reliability, and usability. The system must transform and store merchant data accurately, process catalog updates consistently, and present workflows that are understandable enough for merchants and operators to complete without confusion. These goals shape the kind of testing we prioritize and the parts of the system we evaluate most carefully.

1.1. Testing Context and Scope

Software testing in this project is focused on the point where two connected subsystems meet. On one side, the Salesforce B2C and Business Manager integration handles merchant onboarding, feed export behavior, and feed push behavior. On the other side, the merchant catalog ingestion backend handles merchant authentication, upload creation, job creation, validation, normalization, and product upsert behavior.

The major architecture boundaries in this testing effort include Business Manager controllers and stored merchant configuration, feed-export logic and the documented JSON feed contract, authenticated merchant endpoints and authorization middleware, upload endpoints and S3-compatible storage, and queue-backed job processing and persistent product writes. These boundaries matter because they are the places where independently correct components can still fail when combined.

External dependencies also shape the testing environment. The current implementation relies on Stytch for merchant M2M authentication, MongoDB for persistence, Redis and BullMQ for background processing, and S3-compatible object storage, with LocalStack used as a local

testing substitute. In scope for this plan are the workflows that directly support onboarding, feed creation, feed delivery, and ingestion.

Out of scope are unrelated Generous platform features such as the recommendation engine, Stripe payment flow, and gift fulfillment systems. Those features are important to the larger platform, but they are not part of the merchant catalog path being evaluated in the current backlog and demo flight plan. Keeping the scope narrow allows the testing plan to stay specific and defensible.

1.2. Our Testing Strategy

Our testing strategy is designed to match the current development stage of the project and the kinds of failures that matter most. Unit testing will be used to verify isolated business logic such as authentication behavior, feed validation rules, feed normalization, job-state handling, and setup-helper logic. These tests are intended to run continuously during development and on pull requests so regressions can be caught early.

Integration testing will be used to verify that independently correct components still behave correctly when combined. In this project, that includes boundaries such as onboarding persistence, feed export contract compliance, feed upload creation, job creation, and job-status retrieval. These tests become especially important before demonstrations and milestone reviews because they show whether the workflows we intend to present actually hold together across system boundaries.

Usability testing will focus on the Business Manager onboarding and management flow, as well as the operator-facing ingestion workflow. We expect this testing to take the form of lightweight task-based reviews timed around major alpha and acceptance checkpoints. In addition, operational smoke validation will be used before sign-off on backend ingestion changes so that local Redis, LocalStack S3, upload behavior, and queue-backed processing can be observed in a realistic environment.

Together, these testing types create a layered strategy. Unit testing gives us fast feedback on isolated logic, integration testing verifies system boundaries, and usability testing helps ensure the workflows remain understandable to the people who actually need to use them. This combination provides a practical roadmap for what will be tested and when.

1.3. Why This Strategy Fits the Project

This testing strategy is shaped by risk, system complexity, and user impact. The parts of the system that deserve the most attention are the ones that can silently produce incorrect feed

data, reject valid merchant actions, create confusing setup experiences, or fail only after several components begin cooperating. In this project, that makes feed transformation, validation, upload handling, authentication, job processing, and onboarding persistence especially important.
By contrast, lower-level framework behavior and unrelated platform features do not need the same level of emphasis within this document because they are outside the main merchant catalog flow being evaluated.

The purpose of this plan is not to test every part of Generous equally. Its purpose is to concentrate effort where failure would be most visible to the client, most disruptive to the user, and most harmful to the project's alpha and acceptance goals.
For that reason, this introduction establishes a testing strategy that is selective rather than generic. The following sections build on that foundation by describing the project's unit testing, integration testing, usability testing, and quality controls in greater detail.

---

2. Unit Testing

In our project, unit testing means verifying the correctness of individual units of code in isolation before we rely on the larger Generous merchant catalog flow to work across multiple repositories and environments. The units we care about most are the ones that enforce business rules, validate inputs, transform data, manage authentication context, and drive feed-processing behavior. We use unit tests to confirm that these pieces behave correctly on their own, so that mistakes can be caught early rather than surfacing later as harder-to-diagnose integration problems.

The goals of our unit tests are practical and specific. We want them to verify core business logic, input validation behavior, data-transformation rules, and critical computational decisions such as job-state handling and merchant-scoped product persistence. We also want unit tests to act as regression protection. Once a piece of logic has been shown to behave correctly, the unit tests should make it obvious if a later change alters that behavior in an unintended way.

2.1. Tools, Frameworks, Execution, and Metrics

The current unit-testing work in this project spans two codebases, so the tools reflect the conventions already present in each one. In the merchant catalog ingestion backend, the current unit-level tests are implemented with Jest-based suites and are run through the merchant auth and feed testing. In the Salesforce B2C repository, the committed unit suite uses Mocha, Chai, and Sinon to verify the social-channel setup migration helper. Across both repositories, the existing tests are designed to be run locally during development, and they are also well suited for automated execution in continuous integration because they focus on deterministic logic in isolation rather than on unstable external dependencies.

The project already tracks the most useful unit-level metrics in a straightforward way. The backend report records the pass/fail result of the merchant auth and feed test run, including the fact that the current run exercised fourteen test suites and 169 passing tests. Beyond simple pass/fail status, the current unit evidence also gives us a view into critical-module coverage, because the tested units are clearly named and grouped around the most important parts of the merchant catalog flow. That is the metric emphasis that makes the most sense for this project. Rather than treating one global percentage as the main signal, we focus on whether the units that carry business risk are directly exercised and whether those suites remain stable over time.

2.2. Units Under Test

The units currently under test are concentrated in the parts of the system where isolated logic matters most. On the backend side, the current suites cover merchant authentication

middleware, scope enforcement, merchant rate limiting, Stytch M2M service behavior, feed validation, feed normalization, feed upsert behavior, feed processing jobs, and S3 client configuration for local storage behavior. On the Salesforce B2C side, the current automated unit suite targets the social-channel setup migration helper in _socialSetup.js, specifically the methods that update XML-related setup output and detect installed social channels from cartridge data.

This focus on logic-heavy modules is intentional. We are emphasizing units such as middleware, services, transformation helpers, and processing jobs because those are the places where the code makes project-specific decisions. They are also the parts of the system where isolated tests can provide the clearest value. By contrast, we are not framing static UI rendering or external libraries themselves as the primary unit-testing targets, because the strongest return comes from testing the code that interprets inputs, applies rules, and produces project-specific outcomes.

The current units under test can be grouped as follows:

- **Authentication and access control:** merchant auth middleware, scope enforcement, and rate limiting.
- **Feed logic:** validation, normalization, product upsert behavior, and feed-job processing.
- **Storage configuration:** S3 and LocalStack client configuration behavior.
- **Salesforce B2C setup logic:** XML update and installed-channel detection in the setup migration helper.

2.3. Unit Test Design Approach

Our unit test design approach is to organize cases around expected inputs, boundary conditions, and invalid or erroneous inputs. That structure keeps the tests focused on real variation in behavior instead of only proving that a happy path works once. In the current backend suites, we can already see this pattern clearly. The merchant authentication tests check both successful and failing token-handling paths. The scope tests include malformed and well-formed scope data. The feed-validation tests cover required fields and rule enforcement. The process-job tests exercise successful completion, parse failure, file-size rejection, and other controlled outcomes.

We apply the same thinking to the Salesforce B2C setup helper. The current tests verify that the helper reads input data, logs expected setup messages, writes updated output, parses cartridge paths, and updates the social-channel status map when installed channels are discovered. Even in a smaller suite, the same underlying design principle holds: choose cases

that show how the unit behaves when it performs its primary task and when it interprets the input structures that drive that task.

2.4. Examples from the Current Test Suites

Unit Under Test: FeedValidationService

Purpose: Validates the top-level structure of a merchant feed and enforces core product rules before the feed continues into later processing stages.

Test Case Categories:

- Valid inputs: feed objects with the required top-level shape and valid product fields
- Boundary cases: products that sit at the edge of accepted price and currency values
- Invalid inputs: missing required fields, invalid price or currency values, and contradictory enable_search/enable_checkout combinations

Sample Tests:

- A feed with the required top-level structure is accepted by the validation service.
- A product with valid price and currency fields passes the feed-level validation rules.
- A product that sets enable_checkout without a compatible enable_search value is rejected by the business-rule checks.

Unit Under Test: merchantAuth.js

Purpose: Authenticates merchant requests, maps the validated token to the correct merchant context, and attaches request-specific information needed by later layers.

Test Case Categories:

- Valid inputs: requests with a token that validates successfully and maps to an active merchant
- Boundary cases: requests where request metadata such as the request ID must still be propagated correctly
- Invalid inputs: missing tokens, provider failures, and merchant lookup failures

Sample Tests:

- A request with a valid merchant token receives the expected merchant context.
- A request still carries request ID information through the middleware path.
- A request with a missing token or a failed merchant lookup produces the expected failure behavior.

Unit Under Test: ProcessFeedJob

Purpose: Processes uploaded merchant feed jobs by validating input files, handling parse and schema outcomes, and recording accepted and rejected results.

Test Case Categories:

- Valid inputs: jobs whose uploaded feed is well formed and completes successfully
- Boundary cases: jobs that complete with partial row rejection while still preserving overall processing behavior
- Invalid inputs: oversized files, malformed JSON payloads, schema-version failures, upload mutation detection, and persistence failures

Sample Tests:

- A valid uploaded feed completes processing successfully.
- A feed with partially invalid rows records rejection behavior without losing the job-processing flow.
- An oversized or malformed upload is detected and handled through the expected failure path.

Unit Under Test: _socialSetup.js

Purpose: Updates social-channel setup artifacts and detects installed channels from cartridge-path information in the Salesforce B2C setup flow.

Test Case Categories:

- Valid inputs: XML-related setup data that can be read, updated, and written successfully
- Boundary cases: cartridge-path content that must be parsed into installed social-channel state
- Input-interpretation cases: status-map updates driven by the channels discovered in cartridge data

Sample Tests:

- The helper reads the XML file, logs setup messages, and writes the updated output back to disk.
- The installed-channel parser reads cartridge-path content and identifies the channels that are present.
- The helper updates the social-channel status map based on the installed channels it detects.

Summary of Current Unit Testing Work

Taken together, the current unit-testing work already shows a consistent pattern. The backend suites focus on isolated logic and failure handling in the core merchant feed pipeline,

while the Salesforce B2C unit suite focuses on setup-helper behavior in the integration layer that prepares the merchant environment. That distribution reflects the current implementation shape of the project and gives the unit-testing section a clear purpose: to document how the team verifies important logic in isolation, how those tests are executed, and how specific units are exercised through valid, boundary, and invalid inputs.

---

3. Integration Testing

In our project, integration testing verifies that independently correct components still behave correctly once they are connected into the full Generous merchant catalog workflow. The original testing reports show that the highest-risk failures in this system do not happen inside isolated functions alone. They appear at the boundaries between retailer connectors, onboarding flows, authentication layers, storage, queue-backed processing, and persistent product data.

Our overall approach is to identify the integration points where failure would interrupt a real merchant workflow or break the system's assumptions about data and control flow. For this project, the most important boundaries are the retailer-connector layer, the merchant onboarding path, and the backend ingestion pipeline. These are the points where data moves between systems, where merchant state is established, and where uploaded catalog content becomes usable application data.

3.1. Key Integration Points

The first major integration point is the retailer-connector layer, where the Shopify and Salesforce plugins are treated as a single boundary responsible for retrieving data from retailer platforms and handing it off in a format the Generous workflow can use. Even though the current testing reports focus more directly on the Salesforce-side implementation, the broader integration strategy treats both platform connectors as part of the same architectural concern: retrieving retailer catalog data reliably and translating it into the expected Generous feed flow.

The second major integration point is the merchant onboarding flow, where Business Manager UI behavior, controller logic, and persisted merchant configuration must stay in sync. The third major integration point is the backend ingestion path, where merchant authentication, upload creation, storage validation, queue-backed processing, and job-status reporting must all cooperate correctly. The original reports show that these boundaries matter because they define the workflows merchants and operators actually depend on.

3.2. Test Environment and Execution

Our integration tests will run in two complementary environments. The first is a fast and repeatable contract-testing environment, especially in the backend repository, where route-level integration tests already exercise merchant auth, upload creation, job creation, and status retrieval. These tests are appropriate for local development and pull-request validation because they protect the API contract without requiring every external dependency to run live.

The second environment is a more realistic workflow environment built around local MongoDB, Docker-backed Redis, LocalStack S3, and valid merchant credentials or tokens. This setup is important for testing upload and processing behavior in a stable and repeatable way. Representative catalog fixtures, controlled idempotency keys, and resettable local storage state help ensure that integration failures can be reproduced and diagnosed reliably rather than being confused with leftover test data.

3.3. End-to-End Integration Scenarios

Integration Point: Shopify plugin / Salesforce plugin -> feed transformation layer -> Generous feed handoff

Feature: Retailer catalog retrieval and handoff

Scenario Description: A retailer's catalog data is retrieved through a platform connector, transformed into the expected Generous feed shape, and prepared for downstream use.

Integration Steps:

- The retailer connector retrieves representative catalog data from either Shopify or Salesforce.
- The connector passes the retrieved data into the transformation layer.
- The transformation logic produces the expected Generous-compatible feed output.
- The resulting output is validated against the expected contract for downstream processing.

Expected Results:

- Retailer data is retrieved successfully through the connector boundary.
- The transformation layer produces the expected feed structure.
- The resulting data is consistent with the format required by the Generous catalog workflow.

Failure Handling:

- Missing or malformed retailer data is surfaced as a connector or transformation failure.
- Contract mismatches are detected during validation rather than being silently accepted.
- Connector-specific failures do not produce ambiguous downstream results.

Integration Point: Business Manager UI -> controller -> custom object persistence

Feature: Merchant onboarding and setup

Scenario Description: A merchant enters the onboarding flow, accepts the required terms, submits valid setup information, and reaches the expected configured state.

Integration Steps:

- The merchant opens the onboarding flow in Business Manager.
- The merchant accepts the terms and submits the required onboarding data.
- The controller validates the submission and persists the expected values.
- The system returns the merchant to the correct post-setup state.

Expected Results:

- Terms accepted are saved correctly.
- Merchant setup values are persisted correctly.
- The follow-up state reflects a completed setup path.

Failure Handling:

- Invalid organization or credential data produces the correct error path.
- Persistence failures prevent the workflow from reporting false success.
- The merchant is kept in a diagnosable and recoverable state.

Integration Point: Merchant API -> object storage -> BullMQ worker -> database

Feature: Merchant feed upload, processing, and status retrieval

Scenario Description: A merchant requests a presigned upload URL, uploads a feed, creates a processing job, and retrieves the final result after worker execution.

Integration Steps:

- The merchant authenticates with a valid token and requests `POST /v1/merchant/feeds/uploads`.
- The system returns a presigned upload URL.
- The merchant uploads a representative JSON feed to storage.
- The merchant calls `POST /v1/merchant/feeds/jobs`.
- The system validates storage metadata, creates the job, and dispatches worker processing.
- The merchant polls the job-status endpoint until the job reaches a terminal state.

Expected Results:

- Auth and scope enforcement work correctly at the API boundary.
- Upload creation and job creation succeed for a valid feed.
- Worker processing produces a meaningful terminal status.

- Merchant ownership boundaries are preserved throughout the workflow.

Failure Handling:

- Invalid or missing scope blocks unauthorized access.
- Missing or expired uploads fail during job creation with an appropriate response.
- Storage, queue, or processing failures are surfaced clearly.
- Ownership checks prevent one merchant from retrieving another merchant's job data.

3.4. Why These Scenarios Matter

These three scenarios represent the most important integration paths in the current project because they cover retailer data retrieval, merchant setup, and backend ingestion. Together, they describe how catalog data enters the system, how merchant configuration is established, and how uploaded feeds are processed into usable application state.

They also reflect the structure shown in the original testing reports. The reports point to lighter plugin-side integration evidence and stronger backend route-level coverage, which makes these three workflows the clearest places to define the project's integration-testing expectations.

---

4. Usability Testing

In our project, usability testing means evaluating whether the people using the Generous merchant catalog flow can complete their tasks clearly, efficiently, and with confidence. The goal is not only to confirm that the system works technically, but also to verify that onboarding, feed-management, and operator workflows match user expectations and do not introduce avoidable confusion. For this project, that includes the Salesforce and Business Manager flow, the Shopify and Shopify Admin flow, and the operator-facing ingestion workflow on the backend side.

4.1. Context and Assumptions

Our usability-testing approach is shaped by the fact that the most qualified and relevant user for this system is also our client. In this case, that user is Kyle Montgomery. He is the one participant who can evaluate whether the workflows match the actual operational expectations of Generous across merchant setup, data retrieval, and feed ingestion. Because of that, our usability testing will be narrower and more focused than a broad multi-user study.

This context also explains why the formal usability session will be brief. Kyle Montgomery is the only participant who can run a truly thorough usability evaluation of this flow, because he understands both the intended business behavior and the technical expectations behind retailer integrations, merchant onboarding, and catalog ingestion. Poor usability in this project would still have meaningful consequences. It could lead to confusion during setup, uncertainty around workflow completion, or friction in technical feed-processing tasks. For that reason, even though the session is brief, it is intentional and risk-aware rather than superficial.

4.2. Usability Testing Method

Our primary usability method will be one short, task-based session with Kyle Montgomery before acceptance. This session will focus on the workflows that the reports already identify as most visible and most important: entering the Business Manager flow, reviewing Shopify Admin-facing expectations, completing onboarding, interpreting success and error states, using the manage view, understanding disconnect behavior, and evaluating whether the retailer-to-ingestion workflow feels clear from a technical operator perspective.

We will collect feedback in a lightweight but structured way. During the session, we will note task completion, visible hesitation points, unclear labels or transitions, user questions, and any moments where the workflow behaves correctly but still feels confusing. We will also capture direct comments about whether the system behaves the way a Generous merchant workflow should behave. That is especially important because the Salesforce report already highlights usability-sensitive areas such as the managed page and error handling, while the

backend report shows that the ingestion workflow should also be evaluated from the perspective of a technically informed operator.

Users: Kyle Montgomery

Goals: Ensure the merchant onboarding, management, retailer-integration, and ingestion workflows are understandable, efficient, and aligned with expected Generous usage

Method: One brief, task-based client session with direct feedback and structured observation

Sessions:

- 1 client usability session before acceptance

Tasks:

- Enter the Salesforce and Business Manager flow and complete onboarding
- Review the Shopify and Shopify Admin-facing workflow for clarity of retailer data retrieval and handoff expectations
- Review the manage state after setup and evaluate whether it communicates the correct status
- Trigger and interpret an error state in onboarding or validation behavior
- Review the retailer-to-ingestion workflow from the perspective of a technical operator
- Evaluate whether disconnect and follow-up workflow actions are clear

Measures: Task completion, observable confusion points, user comments, workflow clarity, and perceived alignment with expected merchant usage

Follow-up: Prioritize the highest-impact usability issues for correction before acceptance or for the next refinement cycle, depending on severity

4.3. How Usability Testing Fits into the Development Cycle

Usability testing will occur before acceptance, once the core onboarding and ingestion workflows are stable enough to be meaningfully reviewed. This timing fits the project because the goal of the session is not exploratory prototyping. It is to confirm that the implemented workflow feels clear and usable to Kyle Montgomery, who best understands the intended product behavior across retailer integrations and merchant catalog handling. Findings from that session will be documented as concrete issues tied to the affected workflow, such as onboarding, manage-state clarity, error communication, retailer data retrieval, or technical operator flow.

Although the formal client session is brief, we still treat usability as part of a feedback loop rather than a one-time checkbox. The original reports already provide early usability signals by identifying visible states and known friction points. The pre-acceptance session is where those observations are validated against actual stakeholder use. Any findings will be prioritized by impact and either corrected before acceptance or explicitly tracked for the next development cycle if they do not block the immediate milestone. That keeps usability tied to refinement and decision-making rather than leaving it as an informal afterthought.

---

5. Testing Workflow and Quality Controls

Our testing workflow follows red/green/refactor TDD. For any new feature, bug fix, or behavior change, we begin by writing a test that captures the required behavior before changing the code. In the red phase, the test must fail for the correct reason. In the green phase, we make the smallest change needed to pass it. In the refactor phase, we improve the implementation while keeping the relevant tests green.

Defects are reported with a summary, affected workflow or user story, reproduction steps, expected and actual results, and supporting evidence such as logs, screenshots, request IDs, or failing test output. Bugs are prioritized by impact, with issues affecting onboarding, retailer data retrieval, authentication, feed correctness, upload handling, job execution, or ownership boundaries treated as the highest priority.

A development cycle starts when the target behavior is clear enough to test and the first failing test or failing reproducible check can be written. A cycle ends only when the relevant tests pass, the affected workflow is stable, and no blocker defects remain in the delivered feature area. Known issues are acceptable only if they are low impact, documented, and do not affect correctness, reliability, or usability in the core merchant catalog flow. Problems in onboarding, feed transformation, upload creation, job processing, or merchant ownership mean the work is not complete. This keeps the team's workflow disciplined and ensures that "done" is backed by passing tests and verified behavior.

6. Conclusion

This testing plan brings together unit testing, integration testing, usability testing, and a clear TDD-based quality workflow to support the Generous merchant catalog flow from end to end. Each section focuses on a different source of risk: unit tests protect core logic, integration tests verify system boundaries, usability testing confirms that the workflow is clear to the client, and quality controls ensure that changes are implemented and verified in a disciplined way.

Taken together, this plan gives the project a practical path toward a reliable, functional, and usable product. By testing the areas that matter most, documenting defects clearly, and requiring passing evidence before work is considered complete, the team reduces the chance of avoidable errors reaching the final system. This approach should produce a merchant catalog workflow that is more accurate, more stable, and better aligned with real user expectations.