Test Plan

MToGo
Version 1.1
Date: 28/10-2025

(Updated 25/12-2025):

- remove nightly E2E tests (acceptance tests sufficient and we do manually anyway)
- mutation tests moved to level 3

Introduction and Overview
The project is concerned with creating a functional prototype for the core functionality of MToGo's food delivery system, with integration to their legacy phone based system. Development will incur based on a service based architecture, a trunk based branching strategy and issue based development method.

Goals and Objectives

The goal of the project is to create a functional MVP of the MTogo software system that meets requirements within the given timeframe. It will be developed with an approach inspired by lean and agile practices, with the goal of high deployment rate by CI/CD and a trunk-based branching strategy. Our goal is continuous deployment and automated testing, while having status meetings daily and code reviews weekly, inspired by sprint retrospectives in SCRUM. Testing is done in three steps CI, CD and review:

1. To ensure quality of the product, continuous and incremental tests are conducted in a CI pipeline. The goal is to ensure a healthy codebase by requiring sufficient testing before merging into trunk. We aim to have code coverage of at least 60% unit and integration tests supported by mutation testing, as well as enforcement of code style practices.

2. acceptance and lighter performance testing will be done nightly, which allows daily feature gating based on test results. Performance tests done in a test environment mirroring the production environment.
3. High performance load, mutation tests are scheduled weekly. The results are then discussed during the weekly code review between developers as well to support collective code ownership and refactorings.

Services are built as Docker images, to ensure consistency between environments.

Because concurrent use of the system is essential, performance is tested as spike-tests and load-tests.
The code is expected to be of high quality and consistent to the code practices being lined out in the code practices document.

Logging: Core services and their communication (Messages and API calls) will be subjected to system and audit logging and assigned a correlation ID using MDC.
Audit logging logging will be held to INFO level information with system logging having DEBUG information level.

Scope and Limitations

Each individual service will be subjected to testing ensuring 60% test coverage using Unit tests, and relevant integration tests.
Messaging between internal and external services are tested using integration testing to ensure an expected flow of data between services.

Components written in Java will not have basic class functionality explicitly covered (getters, setters, basic constructors etc.).
Functionality of external services, libraries and API's is not considered for this test plan.

We enforce unit and integration test acceptance in the CI pipeline by using branch protection on the development trunk that only accepts merges by pull requests with passing tests. We evaluate system performance in a dedicated test deployment to test functionality and system regressions, while avoiding CI/CD pipeline blockage.

Test Environment and tool requirements

Environment

- Linux Ubuntu OS
- Java 21
- Maven as package manager for each internal service
- Docker
- Dev container simulating prod env

Testing Tools and Frameworks

- *Version control branch protection*
    - *Github workflows*

- *Unit testing and integration testing*
    - *Junit*
    - *Mockito*
    - *Testcontainers*
- *Performance testing*
    - *Jmeter*
- *Jacoco (coverage)*
- *Mutation Testing*
    - *PITest*
- *Acceptance testing*
    - *Cucumber*
    - *Selenium*
- *Code style enforcement*
    - *Checkstyle (Java)*

Logging

- *Audit Logging*
    - *INFO Level*
- *System Logging*
    - *DEBUG Level*
- *Libraries used:*
    - *slf4j for API*
    - *MDC (Part of slf4j)*
    - *logback or log4j2*

Scheduling and Milestones

Level 1

Entry:

- Test dependencies, local test environment, CI pipeline set up

Exit:

- Successfully run tests in dev env
- Successfully run tests in CI pipeline
- Code analysis pass

Level 2

Entry:

- Level 1 test pass
- Vertically testable feature
- System deployed in test env

Exit:

- Acceptance test pass

- Spike test results under acceptable threshold

Level 3
Entry:

- Level 2 tests pass

Exit:

- All devs has their code peer reviewed
- Comprehensible performance reports

Test schedule:

1. Unit tests and integration tests are run inside a devcontainer locally.
2. On Pull request, unit tests and integration tests are run again with Github Actions.
3. On push to main unit tests and integration tests are run a final time with Github Actions.
4. Acceptance and Spike test will be performed on the test server with the new code. This is scheduled to be done nightly (The runner is placed on the test server as well)
5. Weekly load and mutation test performed on the test server (Runner placed on test server as well)

Resources and Personnel Requirements

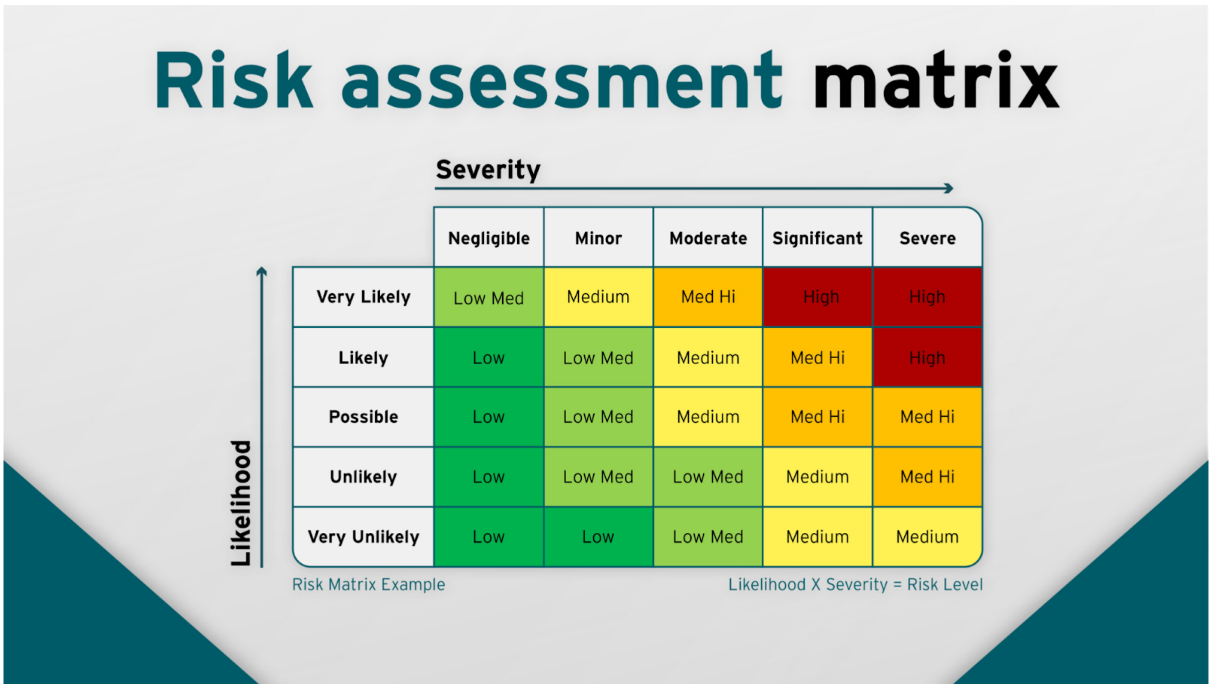
Since the project is service based several servers are needed to set up the architecture. Considering cost during development of the prototype, many of the services will be bundled together on a bigger server. To accommodate optimization of the CI pipeline a secondary test server with similar specs and workload as the production server will be used to run all heavy or infrequent tests. No tests are to be performed against the production server.

All developers are responsible for writing, maintaining, running and monitoring tests on all levels. Tests are to be added to the CI pipeline, at the same time as the feature it is testing.

QA is charged with running weekly code and software reviews, where all content of in house development for the current week is under review.

All other personnel are charged with handling their tasks as defined in the test strategy.

Risk Assessment and Mitigation Plan


Risk assessment matrix

Severity

Likelihood

	Negligible	Minor	Moderate	Significant	Severe
Very Likely	Low Med	Medium	Med Hi	High	High
Likely	Low	Low Med	Medium	Med Hi	High
Possible	Low	Low Med	Medium	Med Hi	Med Hi
Unlikely	Low	Low Med	Low Med	Medium	Med Hi
Very Unlikely	Low	Low	Low Med	Medium	Medium

Risk Matrix Example

Likelihood X Severity = Risk Level

	Risk	Mitigation Strategy	Risk level
1	Critical bugs in production as consequence of high delivery rate	Rollback if not immediately addressable	High (Likely, severe)
2	Less critical bugs in production	Prioritize updates	Medium
3	Project deadline becomes unreachable.	Develop core functionality first, if necessary downscale management system.	Low Med
4	Lose access to production and/or test servers.	Every build is created as an easily deployable image that can be spun up elsewhere on demand.	Low Med
5	Performance tests take up too many ressources on the VM’s.	Downscale performance test.	Medium

6	Load tests reveal performance regressions.	Evaluate result; If minor regression within acceptance limits is found, don’t handle it. If major regression is found, evaluate if a hotfix or rollback is required.	Med high
7	Complexity issues when using previously unknown frameworks during development.	Evaluate issues and decide on whether to downscale, change or postpone to later sprint. Possible enrollment into relevant courses.	Med High

Documentation and Record-Keeping

All test deliverables stated by the test strategy will be generated as artifacts as a product of the CI pipeline.

Logs from different services are stored in individual service volumes, and are extracted and collected into either an external service, or an internal storage.

Javadocs collected in repo.

Reporting and Approval Processes

At sprint review, with a running test build that has vertically testable features and is passing the CI pipeline, Developers and QA will collectively do performance testing. Response times will be measured over time, and QA will approve whether the system performs well enough under the circumstances.
The circumstances depend on the development stage. Earlier in development, core functionality takes priority over high performance results

Code Practices

Google Style for Java:
https://google.github.io/styleguide/javaguide.html

Javadocs for service level:
https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

Unit- and integration tests with JUnit 5.
https://docs.junit.org/current/user-guide/

All services in the same repo in different modules, but with no dependency from each other.
Unit tests are written in each service module.

Code should be readable and as self-explanatory as possible. This means it should be easily readable and consistent across the codebase without explicit comments. If consistency is broken, or code is not clear to read, informative comments are expected to explain what is happening and why.