Test Plan

Version 1.0
Date: 28/10-2025
MToGo

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 agile approach, with the goal of high delivery rate by CI/CD.
Continuous testing under development is done in two main levels:

1. To ensure quality of the product, continuous and incremental tests are conducted in a CI pipeline. 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. User acceptance and performance testing will be done less frequently, with the presence of a QA representative, to allow higher delivery rates. Code reviews between developers are done here 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: All 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, and evaluate system performance in a dedicated test deployment after completed sprints, to save computing resources.

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

- *Unit testing and integration testing*
    - *Junit*
    - *Mockito*
- *Performance testing*
    - *Jmeter*
- *Jacoco (coverage)*
- *Mutation Testing*
    - *PITest*
- *Acceptance testing*
    - *Cucumber*

        - *Selenium*
- *Code style enforcement*
    - *Checkstyle (Java)*
    - *SonarQube*

Logging

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

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:

- Vertically testable feature
- Test env mirroring prod env
- Time and resources for long, intensive test periods

Exit:

- QA validated performance test results.
- All E2E tests pass

Test schedule:

1. Unit tests and integration tests are run inside devcontainer locally
2. On push to main the unit tests and integration tests are run again with Github Actions
3. E2E and Acceptance tests are run on the test server.
4. A spike test will be performed on the test server with the new code (The runner is placed on the test server as well)

5. On successful CI processes the changes are pushed to main, which will start the CD pipeline automatically. All successful merges to main thereby go into production.
6. Weekly load test performed on the test server (Runner placed on test server as well)

Resources and Personnel Requirements

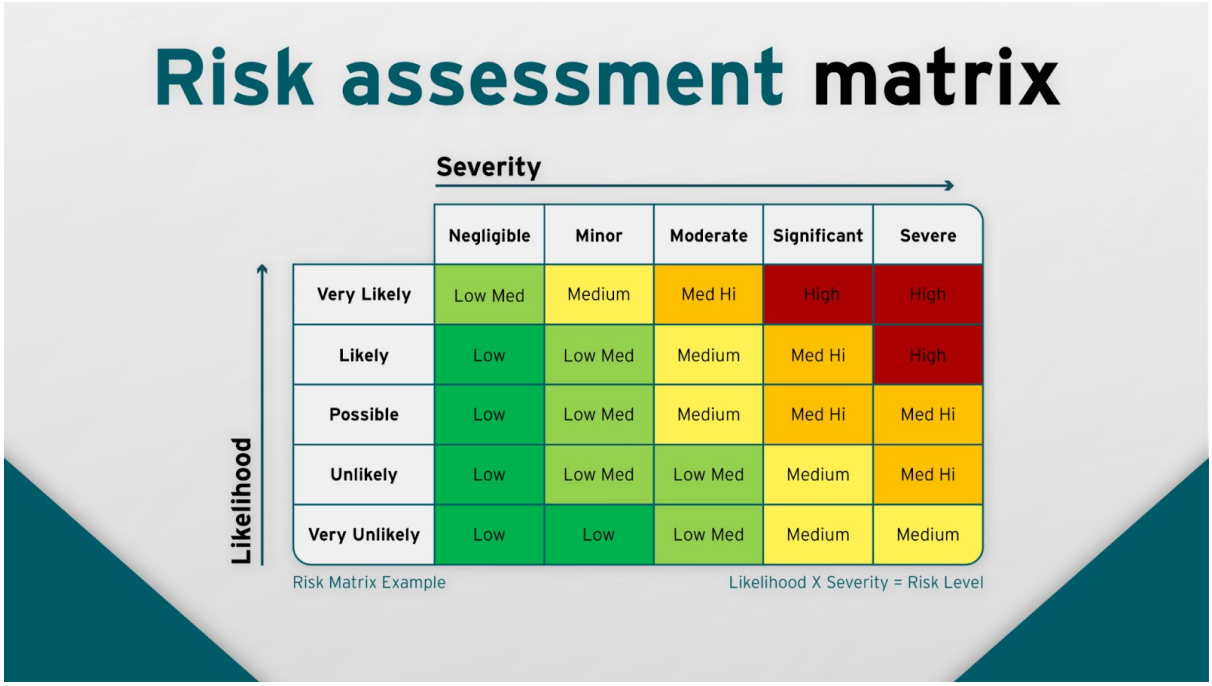
Since the project is service based several servers are needed to set up the architecture. Considering cost during development of the prototype, many of the services will be bundled together on a bigger server. To accommodate optimization of the CI pipeline a secondary test server with similar specs and 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. To ensure integration and architecture tests are meaningful, smaller services or databases will be put on minor servers.

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	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 as much as possible, self-explanatory. 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.