# Sign-Sync

## Testing Policy Document (Demo 4)




| Member | Student Number |
|---|---|
| Michael Stone | u21497682 |
| Matthew Gravette | u23545977 |
| Wessel Johannes van der Walt | u22790919 |
| Jamean Groenewald | u23524121 |
| Stefan Muller | u22498622 |

**1. Introduction & Purpose**

This document outlines the testing policy for the Sign-Sync software system. The purpose of this policy is to ensure a high standard of quality, reliability, and functionality through a consistent and automated testing process. All code contributions to the main repository are expected to adhere to this policy.

**2. Testing Philosophy & Principles**

Our testing strategy is guided by the following principles:

- **Automation-First:** All repetitive tests (unit, integration) must be automated to ensure they are run consistently and frequently.
- **Continuous Feedback:** Testing is integrated directly into our development workflow using GitHub Actions, providing immediate feedback on every pull request.
- **Co-located Tests:** Test cases are located alongside the source code they validate, following the pattern of each microservice. This ensures tests are maintainable and closely tied to the components they verify.
- **Pragmatic Deployment:** Due to the microservices architecture and computational constraints, deployment is a manual process triggered only after automated validation is complete.

**3. Testing Tools & Justification**

We have selected the following tools to implement our testing strategy:

- **Testing Framework: Pytest**
    - **Justification:** Pytest was chosen for its simple syntax, powerful features, and extensive plugin ecosystem. It allows for writing clear, scalable tests with minimal boilerplate code. Features like fixtures and parameterized testing make it ideal for testing our distributed microservices.
- **Continuous Integration (CI) Service: GitHub Actions**
    - **Justification:** GitHub Actions was selected to manage and automate our testing pipeline. It provides deep, native integration with our GitHub repository, eliminating the need for a third-party service. Configuration via YAML files in our repo is straightforward, and it offers sufficient build minutes for our project's needs. This choice simplifies our toolchain and ensures a seamless workflow.
- **Deployment Strategy: Manual via Docker Images**
    - **Justification:** While our testing is fully automated, our deployment process is manual. This decision was made due to our microservices architecture, where building Docker images for all services on every commit would be time-consuming and exhaust our available build credits. This pragmatic approach ensures we maintain high code quality automatically while controlling deployment costs and timing manually.

**4. Testing Procedure & Workflow**

The testing procedure is integrated into our Git workflow as follows:

1. **Local Development:** A developer writes code and corresponding tests on a feature branch using `pytest`. Tests are co-located with the relevant source code.
2. **Pre-commit (Local):** Developers run `pytest` from the project root (which automatically discovers tests across all services) locally before committing to catch failures early.
3. **Push & Pull Request:** The developer pushes their branch to GitHub and opens a Pull Request (PR) to merge into `main`.
4. **Automated CI Pipeline (Triggered by GitHub Actions):**
    - GitHub Actions automatically detects the new PR.
    - It spins up a virtual machine, installs dependencies, and runs `pytest` at the project root, which recursively discovers and executes all test cases across every microservice.
5. **Quality Gate:**
    - **If all tests pass,** the PR is marked with a green checkmark and is eligible for review and merge.
    - **If any test fails,** the PR is blocked with a red X. The development team is notified, and the developer must fix the issues before the code can be merged.
6. **Deployment:**
    - Once a version in the `main` branch is deemed stable, a team member manually triggers the process to build new Docker images and deploy the updated services.

**5. Test Repository & Reports**

All test cases, configuration files, and historical test reports are maintained in our project repository.

- **Git Repository:** https://github.com/COS301-SE-2025/Sign-Sync.git
- **Test Cases Location:** Our test suites are co-located within each microservice's directory structure for better maintainability and organization. For example:
    - `Sign-Sync/backend/alphabetTranslate-service/` - Tests for the alphabet Translation service
    - `Sign-Sync/backend/textToAslGloss-service/` - Tests for the Text to ASL Gloss service
- **CI Configuration:** The GitHub Actions workflow configuration file (`.github/workflows/test.yml`) is located in the root of the repository and is configured to run `pytest` recursively across all directories.
- **Test Reports:** Live test results and build history can be viewed on the "Actions" tab of our GitHub repository: `https://github.com/COS301-SE-2025/Sign-Sync/actions`

**6. Roles & Responsibilities**

All developers are responsible for writing meaningful tests using `pytest` for their code, ensuring tests are properly co-located with the relevant components. The team lead is

responsible for maintaining the GitHub Actions workflow and executing the manual deployment process when necessary.

---

## Why This Approach is Actually Better:

1. **Microservices Alignment:** Having tests in each service folder aligns perfectly with a microservices architecture. Each service is independently testable.
2. **Maintainability:** When you modify code in one service, you only need to look at the tests in that same service's directory. There's no massive, central test folder to navigate.
3. **Pytest Discovery:** Pytest's automatic test discovery handles this structure perfectly - it will find all `test_*.py` files and `*_test.py` files recursively.
4. **Team Ownership:** Different teams can own different services (and their tests) without stepping on each other.

This approach shows you're using industry best practices for a distributed system architecture, which is much more impressive than having all tests in one place!