Static Testing and Quality Assurance

Introduction

The purpose of this assignment is to ensure high codequality, robustness and maintainability in the project Personal Test Data Generator.

This was achieved through the combination of unit tests, integration tests, E2E tests, static analysis and code quality monitoring with SonarCloud.

Static Analysis

Python (backend):

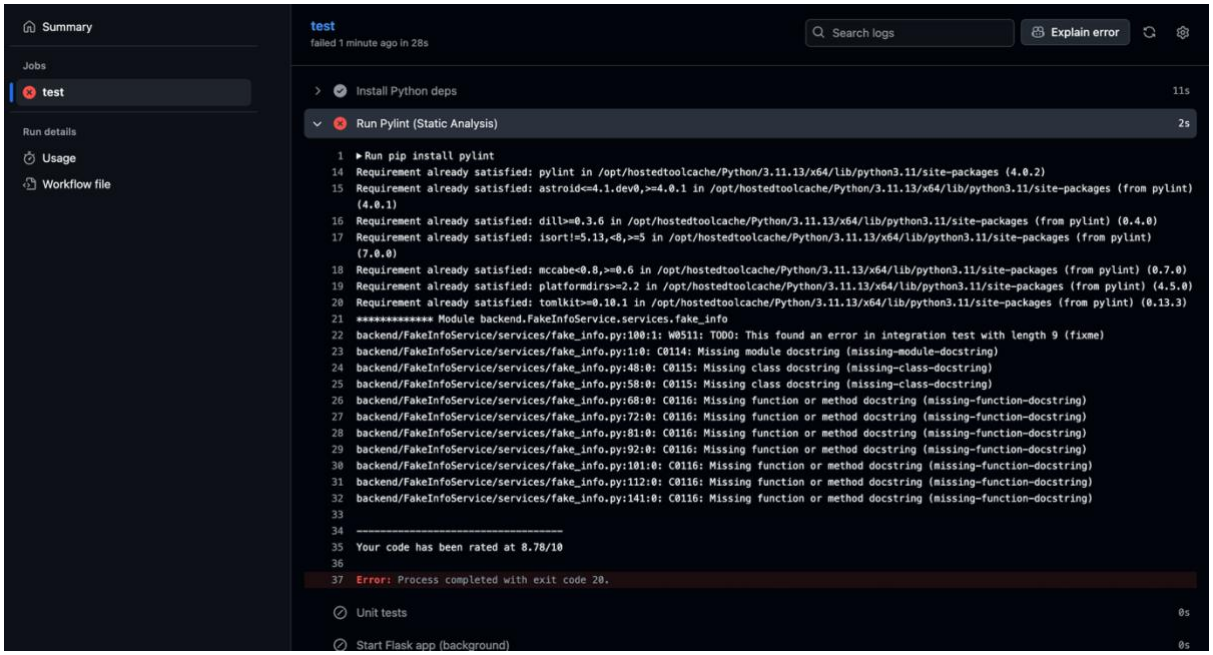
- We used pylint to go through all Python-files in backend/FakeInfoService/services/.
- pylint helped detect potential errors, unused variables and imports and any code style violations with PEP8.
- Any warnings and issues were solved to ensure uniform style and high readability.

By running Pylint on all backend modules, we ensured that the code adheres to Python best practices and is easier to maintain.

Value added by:

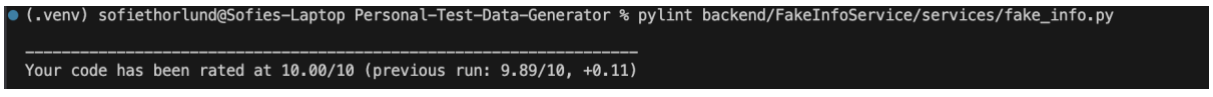
- Reducing the risk of runtime errors by detecting unused or misnamed variables.
- Highlighting complex functions, helping to identify possible refactoring.
- Enforced consistent naming conventions.

Pylint report example:



Initially, the CI pipeline failed during the Pylint static analysis, reporting multiple code quality issues such as missing module and function docstrings, overly long lines, and missing type annotations. The Pylint score at this stage was 8.78/10, and the process exited with an error code, blocking the pipeline.

These findings highlighted inconsistencies in documentation and potential maintainability risks. After addressing the issues by adding appropriate docstrings, simplifying complex functions, and reducing line lengths the next pipeline run showed a significant improvement. The code rating increased to 10/10. This demonstrated the direct value of static testing in identifying and resolving maintainability and readability problems, ultimately leading to cleaner and more professional code that passes automated quality gates.



JavaScript (frontend)

- eslint was run on all .js files.
- Ensured consistent code format and recognised potential mistakes in the frontend logic.

Value Added by:

- Improved code readability and consistency.
- Identified potential syntax and logical issues before runtime.

SonarCloud Code Quality

SonarCloud was used for automatic static analysis of the repository:

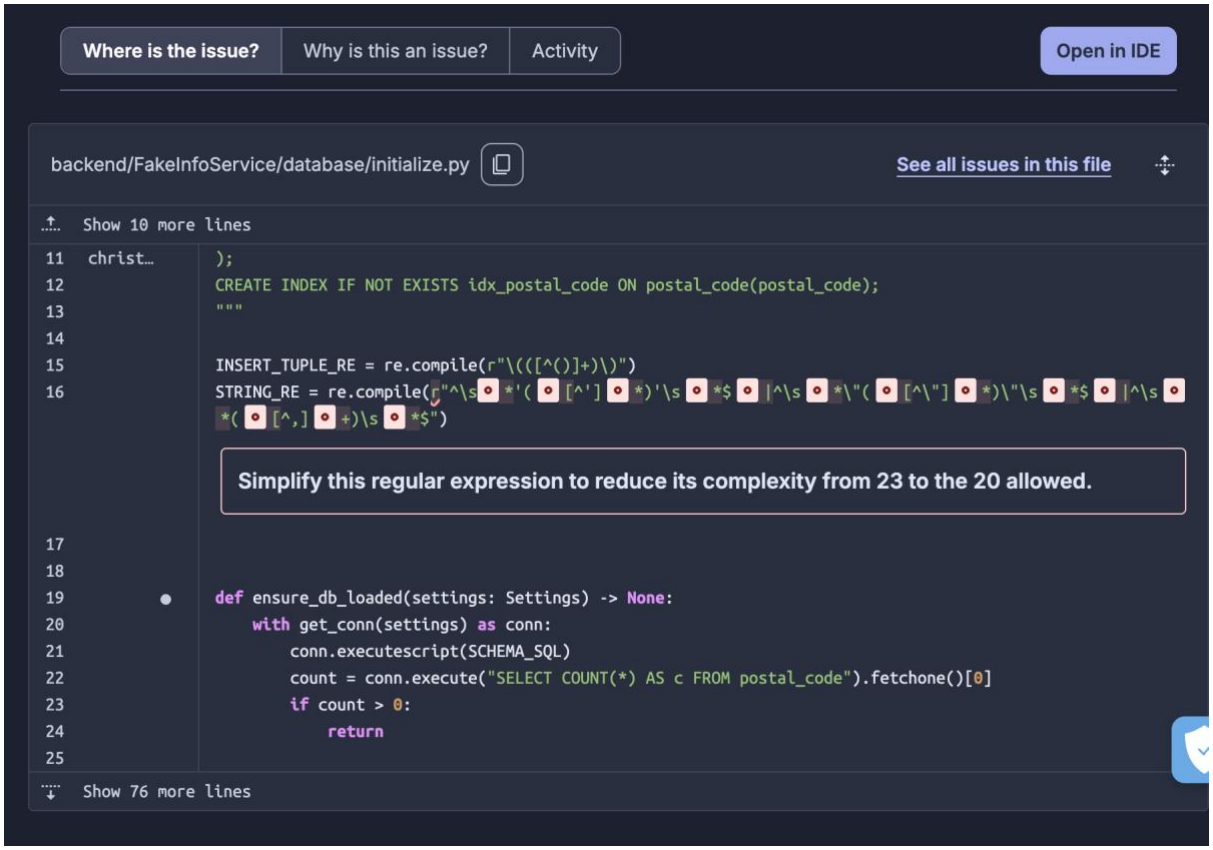
- Analyzes code for bugs, vulnerabilities, and code smells.
- Provides maintainability metrics and highlights technical debt.
- Supports Python version-specific analysis (configured for Python 3.11, due to a warning about Python-version, which then was handled).

In this project, SonarCloud automatic analysis was preferred over manual GitHub Action analysis because it integrates directly with pull requests and main branch pushes.

Value added by:

- Identified code smells such as duplicate code or too complex functions.
- Gave a high-level overview of maintainability and potential runtime issues.
- Ensured that quality gates are met before merging changes.

SonarCloud issues highlighted:



SonarCloud analysis highlighting a maintainability issue in the backend service, a complex regular expression exceeding the logical complexity threshold. This type of feedback helped identify code areas that could benefit from simplification to improve readability and long-term maintainability

Note: Coverage reports could not be generated in this setup, but SonarCloud still provided valuable insights into code quality.

White-Box Testing

White-box testing was achieved using unit tests written in pytest, combined with mocking and parameterized testing to cover all decision points and boundary conditions.

Methodology:

- **Decision Table Testing (DTT):**
  Used for make_cpr() to validate that male CPR numbers always end with an odd

digit and female CPR numbers with an even digit.

- **Boundary Value Analysis (BVA):**
  Applied to functions such as random_birthdate(), random_phone(), and random_address() to test edge cases and limits.

- **Equivalence Partitioning (EP):**
  Used to reduce the number of test cases for ranges of input values while still ensuring coverage of all behavior types.

- **Mocking Random Values:**
  Enabled testing branches that depend on random number generation.

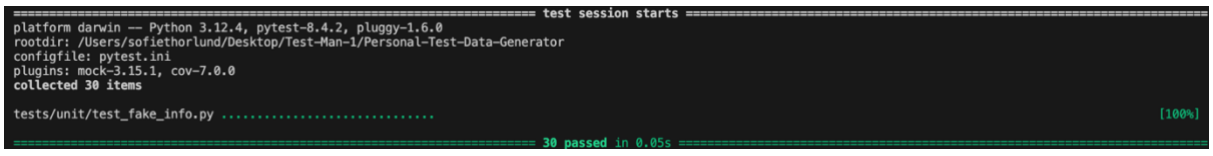
Value added by:

- Ensuring internal logic of functions is correct (white-box coverage).

- Validates behavior in edge cases, which may not be observable with only black-box testing.

- Increases confidence that generated data follows the domain rules (e.g., CPR parity, valid phone prefixes).

Example of pytest passing with perameterized tests:



Pytest execution showing all 30 unit tests passing. The tests include parameterized cases covering multiple input conditions, supporting white-box validation of internal logic and data boundaries within the FakeInfoService module

Continuous Integration and Quality Assurance

The project used GitHub Actions to automate the full testing and quality assurance pipeline.

Pipeline Stages:

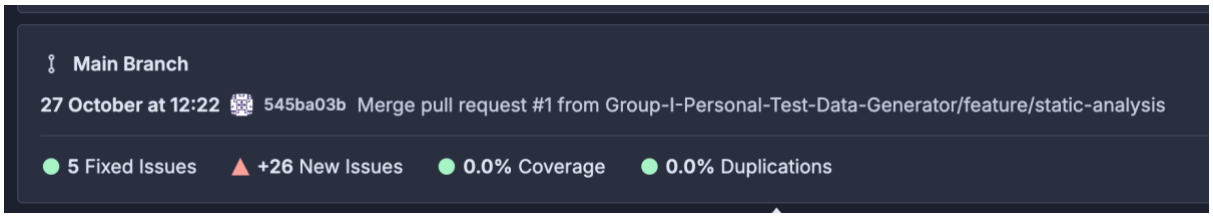
1. Checkout repository and set up Python and Node environments.
2. Install dependencies.
3. Run static analysis (pylint, eslint).
4. Execute unit tests.
5. Launch backend and frontend servers.
6. Run integration and E2E tests.
7. Perform automatic SonarCloud analysis.

The pipeline runs automatically on every push and pull request to main and feature/* branches, ensuring continuous feedback on code quality and test results.

Workflow Integration

- The unit tests and static analysis run on every feature branch push and pull request.
- SonarCloud performs automatic analysis, providing continuous feedback without relying on CI GitHub Action runs.
- Linting ensures that code style issues are addressed early, reducing maintainability problems before merging.

Example showing SonarCloud automatic analysis triggered by a Pull Request:



SonarCloud was configured to perform automatic static code analysis on each branch and pull request. As shown, the platform automatically triggered an analysis after merging the

feature branch into main. This process identifies code smells, bugs, and maintainability issues without requiring manual execution of the scanner in the CI pipeline.

The screenshot shows how SonarCloud detected 26 new issues and verified that 5 previous issues were fixed, showing us how continuous analysis helps maintain code quality over time. Although coverage was not enabled, the automatic analysis still provided insights into code complexity, duplication, and maintainability trends.

Summary and Lessons Learned

- Static testing provided actionable insights beyond stylistic corrections, revealing deeper maintainability and complexity issues.

- White-box testing improved confidence in the correctness of internal logic, especially for random and boundary-based functions.

- Continuous SonarCloud analysis allowed early detection of code smells before deployment.

- Integrating these tools into CI/CD established a robust, automated quality assurance workflow.

Through this process, we learned the importance of addressing complexity early and maintaining consistent standards across both backend and frontend components.

Conclusion:

The combination of static analysis and white-box testing significantly improved code reliability, readability, and maintainability in the Personal Test Data Generator project.

All tests passed successfully, and static analysis ensured early detection of potential defects. SonarCloud's continuous feedback loop strengthened long-term code health, while GitHub Actions automated the entire QA process from commit to deployment.

Overall, these practices established a robust, maintainable, and high-quality software foundation.

Overview of SonarCloud project health and CI pipeline: