Israel Coleman Jr   
CMPSC 487W

Nate Wagenhoffer

29 October 2023

Testing and CI

Test plan:

I just copied and pasted our test plan from our GitHub repository, also I took a screenshot of it in our repo as well.

#### **1. Objective:**

* To verify the functionality, accuracy, and performance of the AI-powered audio filter developed in GitHub.

#### **2. Testing Phases:**

a. **Unit Testing:** - **Objective:** Validate individual components and functions of the audio filter. - Test individual AI algorithms/modules for correctness and accuracy. - Ensure proper input/output handling for various data types and formats.

b. **Integration Testing:** - **Objective:** Validate the integration of different AI components within the filter. - Verify communication between modules and their compatibility. - Test boundary cases and error handling between integrated components.

c. **System Testing:** - **Objective:** Validate the entire system's functionality and performance. - Conduct end-to-end testing of the AI-powered audio filter. - Test real-world scenarios to validate its effectiveness.

d. **Performance Testing:** - **Objective:** Evaluate the filter's efficiency and resource consumption. - Measure processing speed, latency, and memory usage for different input sizes. - Stress testing under heavy loads to ensure stability.

#### **3. Testing Scenarios:**

a. **Input Validation:** - Test with various audio formats (MP3, WAV, FLAC, etc.) to ensure compatibility. - Verify the filter's behavior with different sampling rates and bit depths.

b. **Functional Testing:** - Apply the filter to audio samples with known issues (noise, distortion, etc.) and validate the improvement. - Test the filter's ability to enhance specific audio features (noise reduction, voice clarity, etc.). - Evaluate the filter's performance with different types of audio content (speech, music, ambient noise).

c. **Error Handling:** - Test the system's response to invalid inputs, missing data, or corrupted files. - Ensure proper error messages and logging for debugging purposes.

d. **Performance Benchmarking:** - Measure the processing time for different audio lengths and complexities. - Monitor CPU and memory usage during filter application. - Check for any memory leaks or performance bottlenecks.

#### **4. Testing Methodologies:**

a. **Manual Testing:** - Conduct exploratory testing to identify potential issues and usability concerns. - Manually verify the filter's output against expected results for specific test cases.

b. **Automated Testing:** - Implement test scripts using testing frameworks (e.g., pytest, Selenium) for regression testing and continuous integration (CI). - Use AI-based tools (if available) for automated testing of AI models or components.

#### **5. Tools and Resources:**

* GitHub repository for version control.
* Testing frameworks: pytest
* Sample audio files for testing various scenarios.
* Performance monitoring tools (e.g., profilers, monitoring software).

#### **6. Deliverables:**

* Test cases and scenarios documented in a test plan document.
* Test scripts for automated testing.
* Test reports summarizing test results, including any defects found.

#### **7. Timeline:**

* Define a schedule for each testing phase and allocate resources accordingly.
* Plan for iterative testing based on feedback and identified issues.

#### **8. Roles and Responsibilities:**

* Clearly define team members' roles for testing (testers, developers, AI experts, etc.).
* Ensure collaboration between developers and testers for effective testing.

#### **9. Risk Assessment:**

* Identify potential risks (e.g., AI model inaccuracies, performance bottlenecks) and plan mitigation strategies.

#### **10. Approval and Sign-off:**

* Obtain approval from stakeholders after successful testing and resolution of identified issues.

#### **11. Continuous Improvement:**

* Plan for future enhancements based on user feedback and emerging technologies.

**Test automation and CI:**

Test automation infrastructure:

PyTest

**Justification:**

Our entire project yet is using python such as our VS (conda), and our continuous integration is using Python packages with conda, and inside of our git actions workflow, we are using PyTest, just a good way to keep our workflow linear.

**New test code:**

To add new test code to our base, we already created a testing branch, and with CI workflow setup using a push trigger, it will get added automatically to our base when we push the test code to the base.

**CI Service and repository connection:**

Our CI service is through GitHub actions, setup using a workflow, were using Python packages with conda. As seen below.

A screenshot of a computer

Description automatically generated

**Justification:**

We decided to use this again, just to keep everything linear so we do not have to jump between different languages and programs, everything can just stay in python. Also, we decided to using GitHub actions, just so it will be easier, because its automatically embedded inside of our repo.

**Pros/Cons matrix:**

There were two other CI services we investigated being Travis and MS Azure pipelines.

Travis

**Pros:**

1. **Ease of Use:** Travis CI is relatively easy to set up and configure for continuous integration workflows. Its integration with GitHub makes the setup process seamless.
2. **GitHub Integration:** It seamlessly integrates with GitHub repositories, allowing developers to trigger builds and tests automatically upon code commits or pull requests.
3. **Diverse Language Support:** Travis CI supports a wide range of programming languages and environments, making it versatile for different projects.
4. **Free Tier:** It offers a free tier for open-source projects, enabling developers to utilize continuous integration without incurring costs for these types of projects.
5. **Build Matrix:** Allows the configuration of multiple build environments simultaneously, enabling tests across various versions of dependencies or different operating systems.
6. **Customization:** It provides extensive configuration options, enabling customization of build steps, environments, and deployment procedures.
7. **Community Support:** As a widely used CI tool, it has a large and active community, providing ample resources, documentation, and support.

**Cons:**

1. **Limited Concurrency:** The free tier may have limitations on concurrency and build capacity, causing potential queueing or delays in executing builds for larger projects or during high-traffic periods.
2. **Complex Configurations:** Advanced configurations might be complex, requiring a deeper understanding of its configuration language or documentation.
3. **Limited Build Time:** Free-tier builds might have a time limit, which can be restrictive for larger projects or tests requiring longer execution times.
4. **Private Repositories Cost:** While open-source projects can use Travis CI for free, private repositories require a paid subscription, which might not be cost-effective for small or individual projects.
5. **Dependency on External Services:** As it heavily relies on GitHub integration, any issues with GitHub's availability might affect Travis CI's functionality.

MS Azure pipelines

**Pros:**

1. **Integration:** Seamless integration with Azure DevOps services and other popular development tools like GitHub, Bitbucket, etc.
2. **Flexibility:** Supports multiple languages, platforms, and deployment targets (Windows, Linux, macOS).
3. **Scalability:** Scales well for projects of any size, from small teams to enterprise-level applications.
4. **CI/CD:** Comprehensive CI/CD capabilities for automating builds, testing, and deployment processes.
5. **YAML Support:** Configuration as code using YAML for defining pipelines, making it version-controlled and easily reproducible.
6. **Extensive Marketplace:** A wide range of extensions and integrations available in the Azure DevOps marketplace for additional functionality.
7. **Security:** Built-in security features and compliance with industry standards, offering secure pipelines and data protection.

**Cons:**

1. **Learning Curve:** Steep learning curve for beginners, especially while setting up and configuring pipelines with YAML.
2. **Complexity:** Managing complex pipelines can become challenging, especially when dealing with multiple stages, dependencies, or conditional workflows.
3. **Cost:** Costs may escalate for large-scale usage or resource-intensive builds, as Azure Pipelines charges based on parallel jobs and usage.
4. **Limited UI Features:** The UI can sometimes lack certain advanced features available through YAML configuration.
5. **Platform Restrictions:** Although it supports multiple platforms, some functionalities might be limited or work differently across different OS environments.
6. **Dependency on Azure Services:** Heavily tied to the Azure ecosystem, which might be limiting if you prefer using other cloud services.

**CI build Test:**

1. Unit test
2. Performance test

**CI Triggers:**

Push, will be the only type of trigger we will use.