ECE 461 – Software Engineering

Project Phase 2 Final Delivery – Report

As a Boilermaker pursuing academic excellence, we pledge to be honest and true in all that we do. Accountable together – We are Purdue.

(On group submissions, have each team member type their name).

Type or sign your names: Anjali Vanamala, Pryce Tharpe, Shaantanu Sriram, Andrew Diab

Write today's date: 12/7/2025

Assignment Goal

In this assignment, you will deliver your Project Phase 2 implementation (*software*) and communicate the final status of your Project Phase 2 (*report*).

This document provides a template for the report. This is a form of documentation that your customer can use to decide (a) whether you've met the contract, and (b) re-negotiate the contract based on deviations therefrom.

Relevant Course Outcomes

A student who successfully completes this assignment will have demonstrated the ability to

- *Outcome ii*: the ability to conduct key elements of the software engineering process, including…deployment
- *Outcome iii*: Develop an understanding of the social aspects of software engineering… including…communication [and] teamwork.

Assignment

Fill out each of the following sections.

IMPORTANT:
When this document requests information about your design, test plan, etc., you may refer to and reuse contents from other documents your team created. However, please ensure that anything you report is truthful of your system **as delivered to us** – don't tell me what you planned, tell me what you did.

Location of project

Provide the URLs that we can use to interact with your team's deployed service. (This should also be in the spreadsheet posted to Piazza).

API: https://9tiiou1yzj.execute-api.us-east-2.amazonaws.com/prod Web UI: https://main.dpdr1x0112neg.amplifyapp.com

Provide a link to your team's code repository on GitHub:

https://github.com/Anjali-Vanamala/ECE461_Part2

ECE 461 – Software Engineering

Succinct description

Non-technical description

In 1-2 paragraphs, describe the system that you have implemented and its potential business value for ACME Corp. Write as though you are speaking to a non-technical customer representative who would not know what a "package registry" is (and try not to dedicate more than a sentence to explaining it).

We have created a system that works as a storage place for Huggingface (another model database) models, datasets, and code. While Huggingface has a large variety of models hosted on the sight, anyone can upload a model leading to issues where any particular model could be badly documented, leading to long set up times or otherwise lead to negative externalities. Thus, our system allows users to request the ingestion or upload of a Huggingface model, dataset, or code to the website. We then perform a series of checks and score the code quality, dataset quality, ramp up ability, size, and various other factors to ensure the model is of high quality for usage in the company, only models with score of above 0.5 overall will be added to the registry. Thus, when employees download a model from our registry, they can be sure that it is validated, trusted, and easy to use. Additional features include the ability to access model ratings, search for models, ADA accessibility features, lineage graphs, storage cost, and license checks (to check a GitHub repository license that an employee may want to run or own against the licensing of a model in the registry).

Technical description

In 1-2 paragraphs, describe the system that you have implemented and its potential business value. Write as though to one of your customer's engineering representatives who is particularly interested in its technical benefits compared to npm.

Our model registry system has many technical benefits over npm. First, the registry handles a variety of artifacts including code, datasets, and models versus npm which only packages js code natively. Similarly, while npm supports package dependency it is void of meaningful relationships when it comes to ML models such as a model being trained on dataset X or being finetuned from another model, this lineage and relationship information is tracked by our registry. Additionally, while npm has download metrics, our registry rates models on a wide variety of metrics from code and dataset quality to ramp up time allowing users to know much more about the model at a glance than npm. Furthermore, we implemented a cost endpoint that tells users the storage cost of a model which npm does not include. This allows users to be more cost aware which could lead to decreased costs. Our license check endpoint allows users to check a GitHub repository license with a hosted model's license automatically. While npm shows licenses it still forces users to check the license's themselves. Finally, in addition to all the extra features we included many of the basic functionalities of npm like model upload, deletion, download, and search.

Last modified: 2 December 2025

Functional requirements

New metrics

You were required to implement new metrics. This implementation needed to build on another team's Phase 1 implementation.

Describe any changes you made to the existing Phase 1 design or implementation,[1] divided into two kinds:

1. Changes to **allow you to implement** the new metrics. (This is not asking you how you implemented the new metrics. It is asking you about refactorings you undertook to make the new metrics easier to implement).

Change 1: Updated the concurrent metrics run file
Justification (2-3 Sentences): This is the file that runs all the metrics concurrently. We added the 3 new metrics to the file, added them to the concurrent run submission, unpacking, and changed the net score function so that it included the new metrics and still summed up to 1. This was all needed to make sure the new metrics run with the phase 1 metrics. Other than this, we were able to pretty easily implement the new metrics without changes.

2. Changes to improve the **reliability** (e.g. correctness or other quality attributes) of the component so that your Phase 2 implementation would fully satisfy the customer's requirements.

Change 1: Changes dataset quality and code quality to not fail on no link
Justification (2-3 Sentences): The dataset and code quality would give really low scores if the

Change 2: Linting, mypy, and isort fixes
Justification (2-3 Sentences): We wanted to ensure a decent quality of code, so we ensured that on PR a workflow runs that checks linting, mypy, and isort. Most of the original code failed these checks so there's a lot of small fixes to address the errors.

Change 3: Removed creation date as a sub-metric and added github contributors call for bus_factor
Justification (2-3 Sentences): Creation date was removed from calculating the bus_factor metric because it heavily penalized models that were made more than a year ago, however, we decided that as long as a model had been updated recently, the date it was created doesn't matter too much. We added a GitHub contributors call replacing the original check of

[1] Hint: It might be helpful to examine your team's PRs or git logs to recall these changes. I assume, of course, that you followed an appropriate engineering process so that you can answer these questions ☺.

huggingface card info [spaces] as the “spaces” key does not give contributor info and was empty for most models we checked which was also dragging the score down a lot artificially.

Change 4: Code quality thresholds decreased.
Justification (2-3 Sentences): All the thresholds were very unforgiving and a little unrealistic, for example, to a get a score of 0.7 on the downloads part of the score calculation a model would need 500,000 downloads. Anything less than 100,000 downloads would be scored 0 for the download sub metric. Each sub metric was similarly unforgiving with readme length, likes, and keywords for testing also having very high thresholds. Thus, we decreased the numbers to what we believed are more reasonable values 500,000 -> 5000, etc.

Change 5: dataset and code quality increased points given for each check.
Justification (2-3 Sentences): Very similar to the previous change, this metric worked with an additive factor were having specific qualities (code blocks, presence of dataset/code, etc) added points the sum in this metric is allowed to sum to over 1 and then the output is the minimum of the added points and 1. Each check was very specific like searching for python code blocks with header “python” exactly. So, the code was underscoring basically every model. Thus, checks like the code blocks were loosened such that any code blocks count towards the points, and the points added were increased slightly to make up for the code lowballing scores.

Change 6: performance claims -> changed API call and bumped up score
Justification (2-3 Sentences): Similarly to the dataset and code quality, we noticed that the metrics with the AI tended to lowball scores compared to our manual evaluation so a set increase of 0.2 was added to the AI output score. This is capped to 1 so that the score output is not greater than 1. Additionally, the original code would keep calling the LLM if it gave an output not in the expected format (float between 0 and 1 and nothing else) however, we had some readme tests where it was stuck calling the API for a while. So, we added a limit to the calls and a default if the LLM isn’t cooperating. This allows us not to worry about timing out on rating or running out of LLM calls.

Baseline: API

In your *Phase 2 Plan* document, you described the system features and requirements you planned to implement (possibly impacted by subsequent renegotiation). Here you will describe what you actually delivered.

Fill in the following table for each of the *baseline behavioral features* (e.g. “ingest a package”) and the degree to which you’ve met each of them. Make one copy of the table per feature.

Feature: Upload an Artifact
Relevant endpoint(s): /artifact/{artifact_type}

How completely is it implemented? Fully implemented -> takes in any dataset, code, or model link returns proper codes.
How did you validate it? (Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/{artifact_type} | POST | Artifact type + json body with url and name	1) Automated unit tests 2) Automated End-to-end tests 3) Manual Testing	1) tests_main.py - **class** TestAsyncModelProcessing **functions** test_register_model_returns_202 and test_register_dataset_returns_201. Tests model positive case and dataset positive case 2) integration_tests.ps1 - Cases commented with "POST /artifact/{artifact_type}(artifact_name)". Tests model, code, and dataset positive cases and duplicate case. 3) Ran server locally and tested for each error code – success, malformed json, repeat.

Feature: Rate an Artifact
Relevant endpoint(s): /artifact/model/{artifact_id}/rate
How completely is it implemented? Fully Implemented -> returns ratings as expected with correct codes
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/model/{artifact_id}/rate | GET | artifact id	1) Automated unit tests 2) Automated End-to-end tests 3) Manual Testing	1) tests_main.py - **class** Each rating metric has class Test{metric name}. **functions** test_register_model_returns_202 and test_register_dataset_returns_201. Tests model positive case and dataset positive case 2) integration_tests.ps1 - Cases commented with "POST /artifact/{artifact_type}(artifact_name)

		". Tests model, code, and dataset positive cases and duplicate case. 3) Ran server locally and tested for each error code – success, malformed json, repeat.

Feature: Download an artifact
Relevant endpoint(s):
/artifacts/{artifact_type}/{artifact_id}/download
How completely is it implemented? *Fully implemented as per the API spec, allows download of the requested artifact.*
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifacts/{artifact_type}/{artifact_id}/download | GET | artifact_type, artifact_id	1) Automated unit tests 2) Manual Testing	1) tests_main.py - **class TestDownloadEndpoint** Tests that the download url is made when creating an artifact and is retrievable with get and artifact. Tests 404 when artifact doesn't exist, 202 when artifact still processing, 424 when artifact fails processing, 400 for invalid artifact type and id, various errors if download fails, and that non-model artifacts are downloadable. 2) Tested the download capabilities of various models when deployed to AWS.

Feature: Ingest a model

Relevant endpoint(s): /artifact/{artifact_type}
How completely is it implemented? Implemented but not fully following spec -> We found that all models were rejected if we did the ingest check as every single metric has to be able 0.5. We validated each metric (More info in Notes for Autograder) and found that they all were valid, so we changed the ingest to be that net score must be above 0.5 for the model with the justification that a registry that doesn't allow any model in is somewhat useless, and 0.5 net score will still filter out bad models. Of the Autograder tests, this does throw out one model (artifact 3).
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/{artifact_type} | POST | Artifact type + json body with url and name	1) Automated End-to-end tests 2) Manual Testing	1) integration_tests.ps1 - Has tests for both when our ingest is turned on and turned off (when turned off we lose access to some autograder checks like regex which have dependency "Get All Artifacts Query Test", thus if we wanted to validate these functionalities, we turned the ingest off) 2) Ran server locally and tested looking at -> do rating outputs make sense, does netscore < 0.5 mean the model is gone, does the model silently drop as expected.

Feature: Enumerate/Search Artifacts
Relevant endpoint(s): /artifact/byRegEx, /artifacts, /artifacts/{artifact_type}/{artifact_id}
How completely is it implemented? *(If partial, explain limitations)*
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/byRegEx | POST | Regex in body	1) Automated End-to-End Tests 2) Manual Testing	1) integration_tests.ps1 - Tests under "REGEX SEARCH". Have tests for valid searches which should return artifacts,

		catastrophic backtracking which should return error 400, and no artifact which should return 404. 2) Ran server locally and tested various successes and no artifact regex tests to ensure no edge case issues.
/artifacts | POST | body -> name string and artifact types	1) Automated End-to-End Tests 2) Manual Testing	1) integration_tests.ps1 - Tests under "REQUEST ALL MODELS" and "ARTIFACT SEARCH (NAME MATCH EXACT)". The ALL MODELS requests have .* as the name and test getting all models, code, and dataset individually as well as and empty [] types. The EXACT tests search for the exact names of various artifacts with empty typing. 2) Ran server locally and tested various regex combinations including successful calls for each artifact type as well as expected 400 behavior if the query is malformed.
/artifacts/{artifact_type}/{artifact_id} | GET | artifact type, artifact id	1) Automated End-to-End Tests 2) Manual Testing	1) integration_tests.ps1 - Tests under "RAW GET ARTIFACT REQUESTS". Tests a bunch of success cases to ensure that the artifact can be retrieved. There are test cases for each artifact type model, code, and dataset. 2) Ran server locally and ensured expected behavior with 404 error for artifact doesn't exist and 400 error

		for artifacts with wrong type or id.

Feature: Lineage Graph for Artifact
Relevant endpoint(s): /artifact/model/{artifact_id}/lineage
How completely is it implemented? *Fully implemented*
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/model/{artifact_id}/lineage | GET | Valid artifact ID with all supporting factors, Invalid/nonexistent artfiact ID, Related models produce the same graph	1) Automated End-to-End 2) Manual Testing	1. Tests_main.py: Under class **TestLineage.** Tests for valid artifact IDs with complete lineage that return lineage graphs with nodes and edges (HTTP 200). Tests for invalid/nonexistent artifact IDs that return HTTP 404. Tests for related models producing the same graph structure when queried from different starting points. 2. Manual Testing: Server run locally. Tested successful lineage queries with models that have base models and descendants. Tested error cases with nonexistent IDs. Verified graph consistency when querying related models. Purpose: to

		ensure no edge case issues.
/artifact/model | POST | Valid huggingface model URL	1) Automated End-to-End Tests 2) Manual Testing	1. Tests_main.py: Class: **TestAsyncModelProcessing**. Tests for valid HuggingFace model URLs that successfully register models and return artifact metadata. 2. Manual Testing: Server run locally. Tested various successful model registrations with different HuggingFace URLs. Tested duplicate registrations that return HTTP 409. Purpose: to ensure no edge case issues with URL validation and metadata extraction.

Feature: Size Cost for model
Relevant endpoint(s): /artifact/{artifact_type}/{artifact_id}/cost
How completely is it implemented? Fully implemented
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/{artifact_type}/{artifact_id}/cost | GET | artifact_id, artifact_type	1) Automated Unit Testing	1) The automated tests in tests_main.py under class **TestCostEndpoint**. They verify successful cases: rated models return the correct cost using (1.0 - raspberry_pi_score) * 1000, and non-model artifacts (datasets/code) return

		0.0. Error handling: 404 for non-existent artifacts and 404 for models without ratings. Edge cases: models with a raspberry_pi score of 0.0 return the maximum cost of 1000.0. All 5 tests pass.

Feature: License Check for model and given github repo
Relevant endpoint(s): /artifact/model/{artifact_id}/license-check
How completely is it implemented? Fully Implemented
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifact/model/{artifact_id}/license-check | POST | artifact id, body -> github url	1) Manual Testing 2) Automated Unit Testing	1) Tested cases where licenses were compatible, weren't compatible, and were not listed. 2) The automated tests in tests_main.py are under class **TestLicenseCheckEndpoint**. They verify successful cases: compatible licenses (e.g., Apache-2.0 + MIT) return true, and incompatible licenses (e.g., Apache-2.0 + GPL) return false. Error handling: 404 for non-existent artifacts or GitHub repositories, 400 for malformed GitHub URLs, and 502 when the GitHub API fails. Edge

		cases: models without licenses return false, and URLs with trailing slashes are handled. All 8 tests pass.

Feature: Reset repository
Relevant endpoint(s): /reset
How completely is it implemented? *(If partial, explain limitations)*
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/reset | DELETE | None	1) Automated End-to-End Tests 2) Manual Testing	1) integration_tests.ps1 - Test under "DELETE /reset". The rest of the tests don't pass unless this works. 2) Tested for basically every other test and it's the best/easiest way to clear the repository.

Feature: Check System health
Relevant endpoint(s): /health
How completely is it implemented? *(If partial, explain limitations)*
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/health | GET | No input	1) Automated End-to-End Tests 2) Manual Testing	1) integration_tests.ps1 - Test under "GET /health". Just checks that 200 is returned. 2) Tested that it returned 200 when the server is up.

Feature: Change (Edit or Delete) an artifact

Relevant endpoint(s): /artifacts/{artifact_type}/{artifact_id}
How completely is it implemented? *(If partial, explain limitations)*
How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/artifacts/{artifact_type}/{artifact_id} | PUT | artifact_type, artifact_id, body with new url	1) Automated unit tests 2) Manual Testing	1) tests_main.py - **class TestUpdateEndpoint** Tests valid artifact (200 at first and 202 after rating) behavior, tests incorrect artifact returns 400, tests non-existant artifacts return 404. 2) Locally ran backend and tested accuracy of 200, 202, 404, and 400 code behavior with huggingface models.
/artifacts/{artifact_type}/{artifact_id} | DELETE | artifact type, artifact id	1) Automated unit tests 2) Manual Testing	1) tests_main.py - **class TestDeleteEndpoint** Tests 200 condition that the model is deleted and 404 condition where the model to delete doesn't exist. 2) Tested that deleting the model works manually and that random ids produce a 404.

*Validation approaches: For example, (None ; Manual ; Automated unit tests ; Automated end-to-end tests)

**Test records: Here are some possible categories. Include whatever you have.

- No record exists
- Manual
  - A link to at least one relevant code review
  - A 1-2 sentence description of how you formalized/structured your manual review (ex. steps in a written testing plan you followed)
  - The date of the most recent manual test (if this is an estimate, note that)
- Automated
  - A link to the associated tests in your GitHub repository

Baseline: Web UI

You were required to implement an ADA-compliant web interface (WCAG 2.1 at level AA).

1. In 1-2 paragraphs, summarize your approach to improve accessibility (e.g. educational resources you consulted; design choices you made; implementation decisions that are not covered by the automated test tool you used).

We focused on doing some research first looking into the W3C WCAG 2.2 guidelines page as well as sever accessibility checklists focused on semantic HTML structure, ARIA usage, and color contrast best practice. This helped us understand what to implement but also why which allowed us to design more in mind for accessibility than just fixing errors until an automated testing tool said we're in the clear. We tried to make sure that headers were in a logical order and manually checked to make sure it followed the same way we tracked the pages. Additionally, we tried to make sure any alternative text was meaningful and easy to understand in context. Finally, we ensured both dark and light modes were available for reduced eye strain and also manually checked the contrast of the color palette to ensure they were up to standards.

2. In 1-2 paragraphs, summarize your testing approach. (Automated via Selenium? Another tool? Manual?). Describe how you ensured this approach would sufficiently cover the ADA requirements.

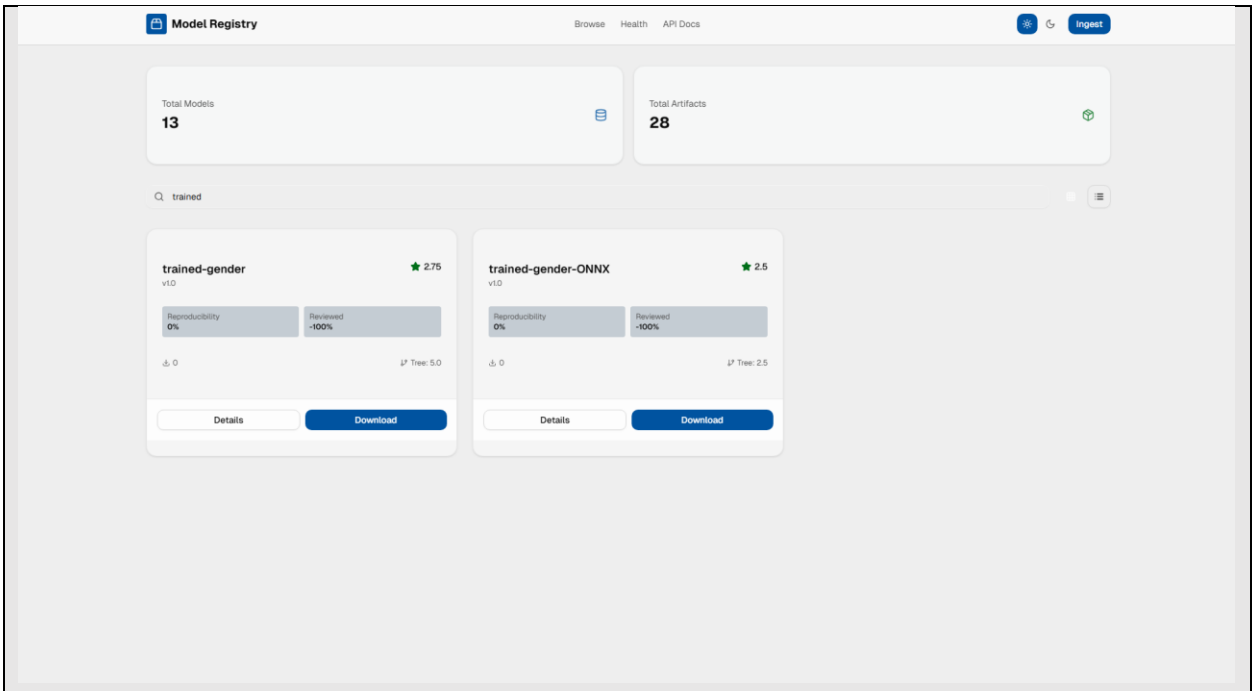
We did two types of testing. As mentioned above, the first thing we did was manual testing. Following several ADA accessibility checklists, we manually checked key features like header orders and contrast comparing to the standards for W3C WCAG 2.2. Additionally, after all testing including the automated tests, we skimmed the full guidelines and ensured that we had at least seen the guideline before. Essentially, we made sure that between manual testing and each of our automated checks that the guideline had been covered at least once. On top of manual testing, we employed several automated tests. We figured that a singular test could be fallible, but that multiple on top of manual testing was more likely to cover all the issues we could have. Specifically, we used chrome extensions WAVE (Web Accessibility

Evaluation Tool) and axe DevTools v4.117.0. Additionally, we used a website recommended by the Bureau of Internet Accessibility https://www.accessibilitychecker.org/audit/. Finally, as part of our CI we have automated tests with Selenium that check for key accessibility components like alt attributes for all images, aria labels for buttons, hrefs for links, and that headers are present.

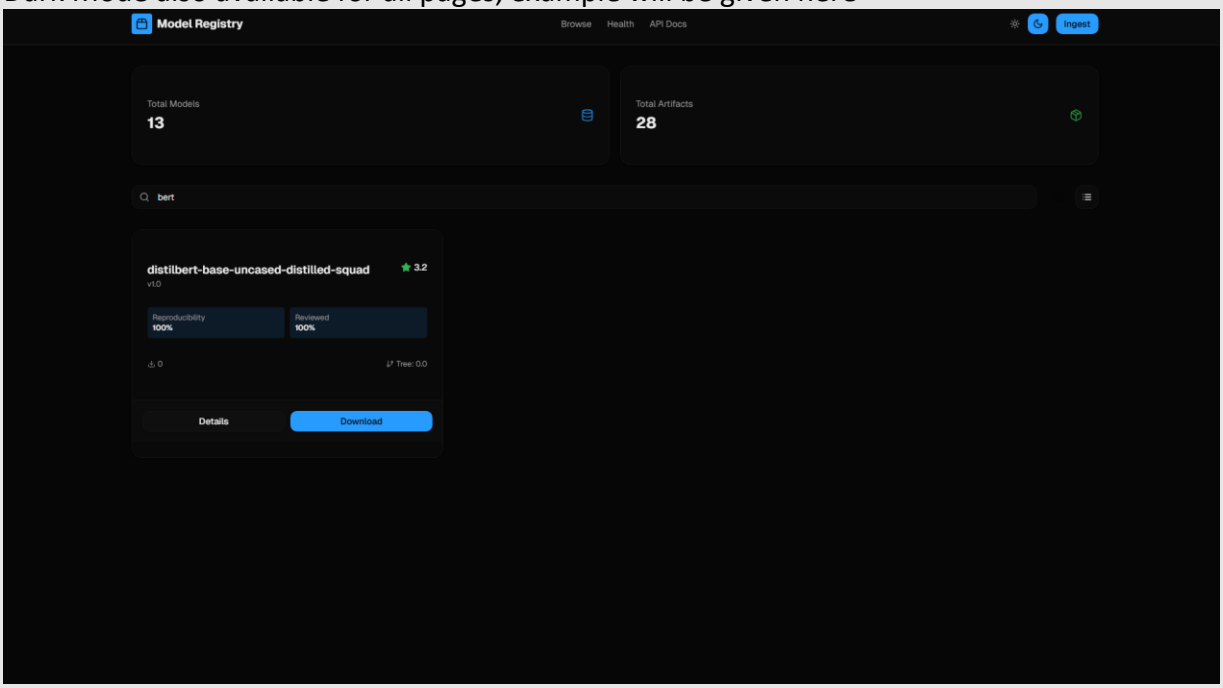
In the following block(s):

3. Show screenshot(s) of each page(s) of your UI.
4. Show the result of an ADA-compliance automated testing tool, e.g., https://github.com/microsoft/accessibility-insights-web, as applied to each of your web page(s). Screenshots are fine.

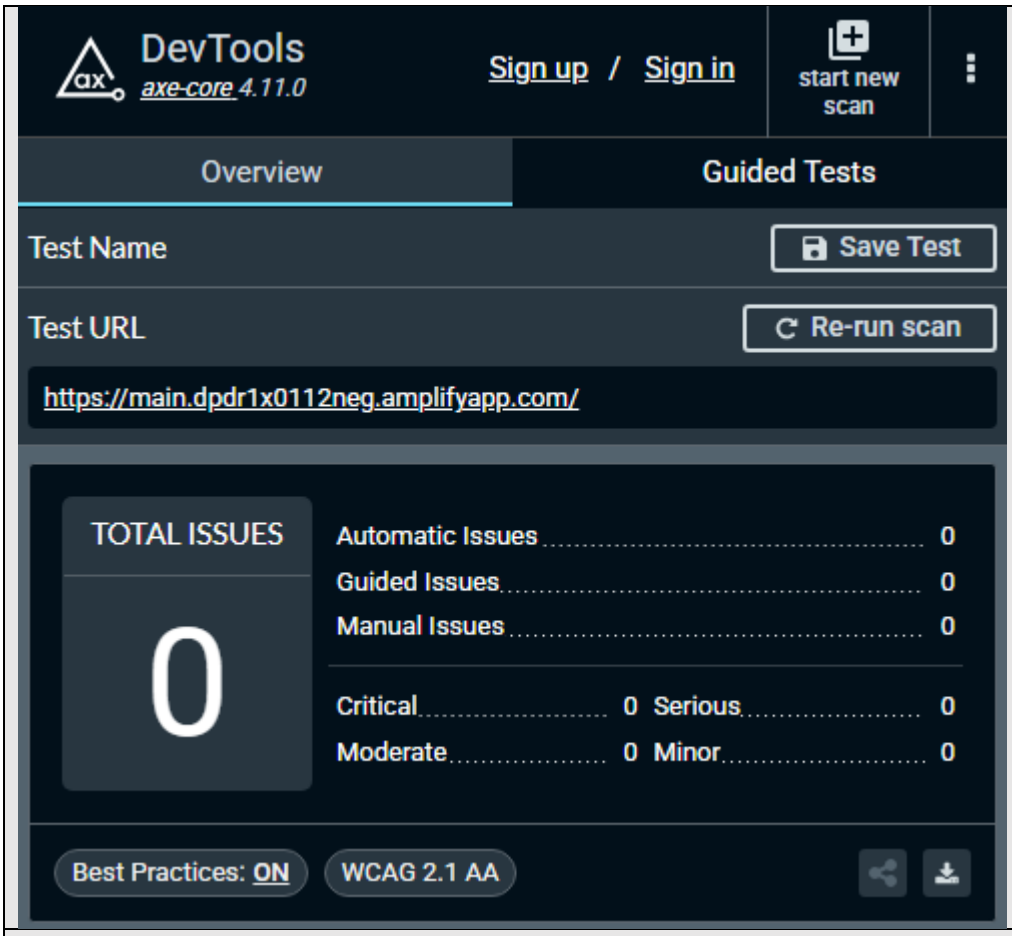
Homepage: https://main.dpdr1x0112neg.amplifyapp.com/



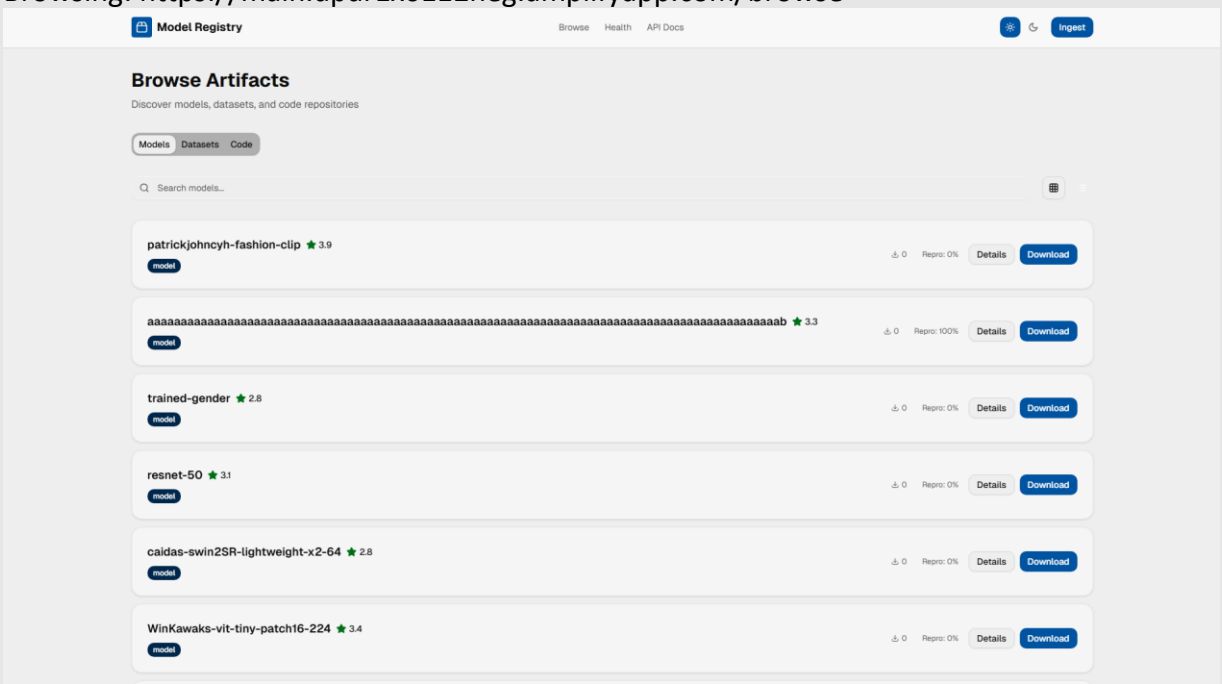
Dark Mode also available for all pages, example will be given here



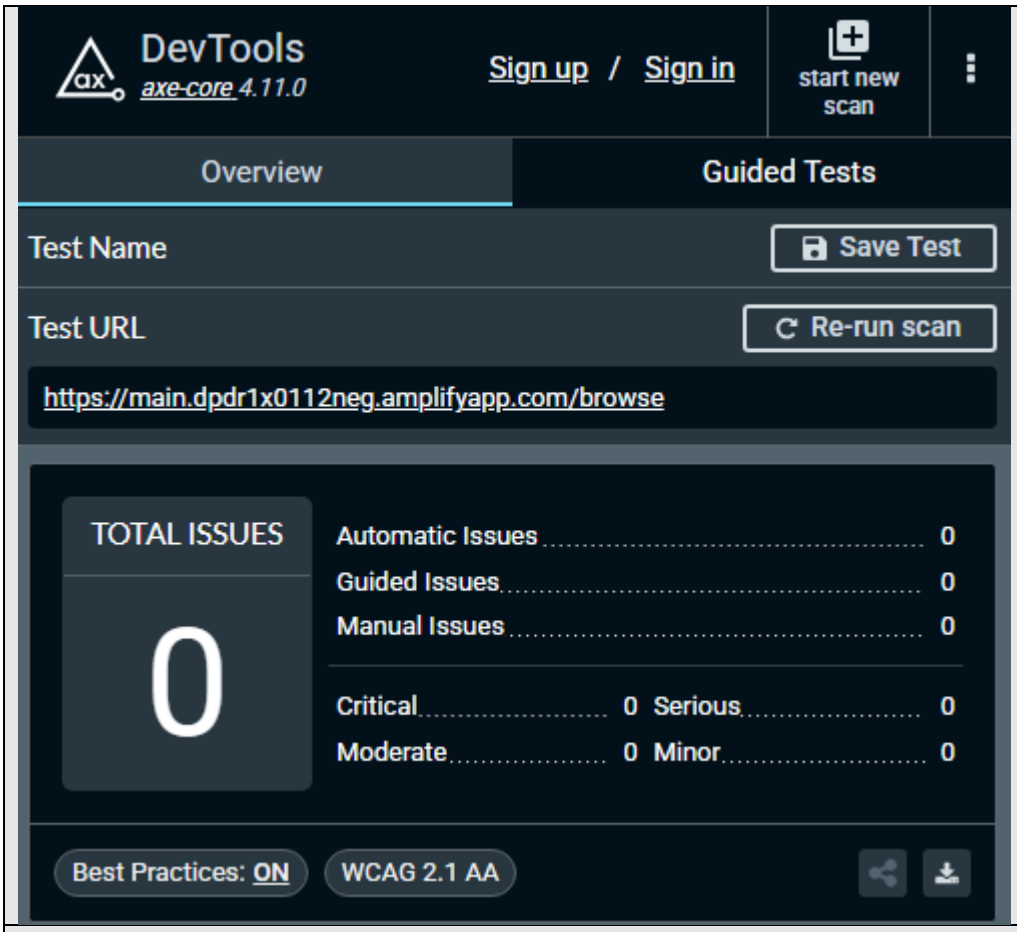
Outcome of tests: …



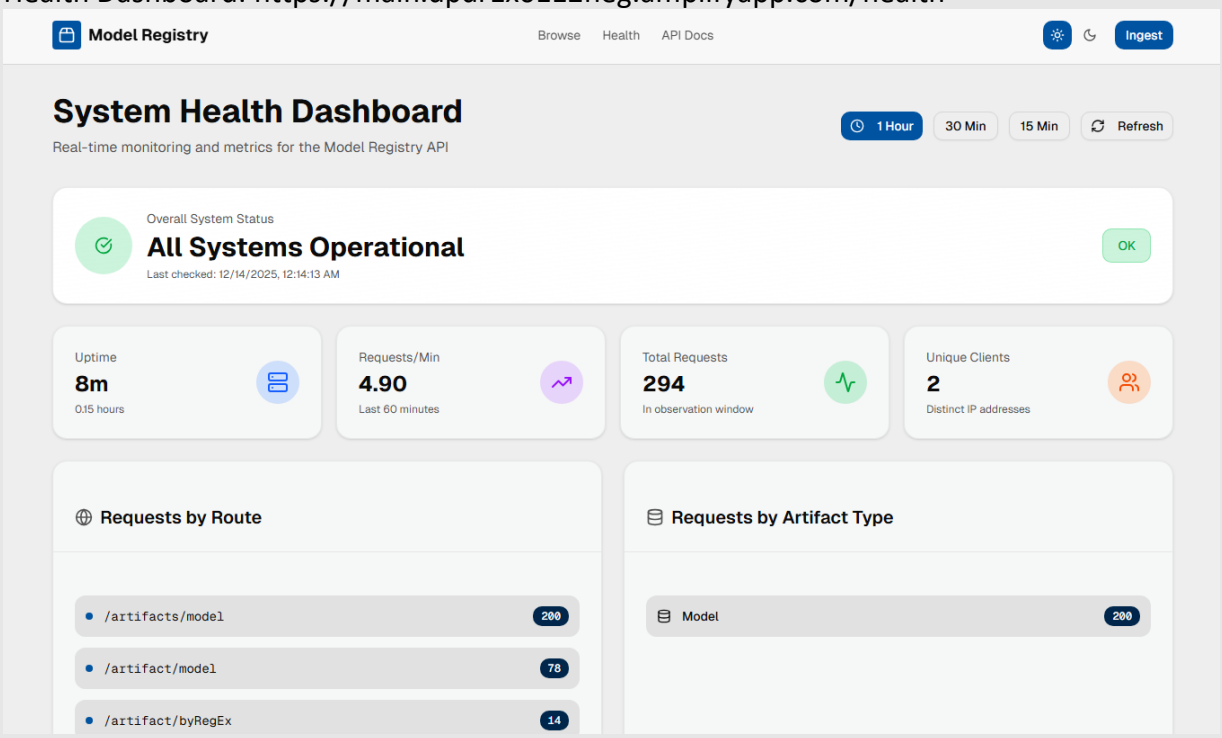
Browsing: https://main.dpdr1x0112neg.amplifyapp.com/browse



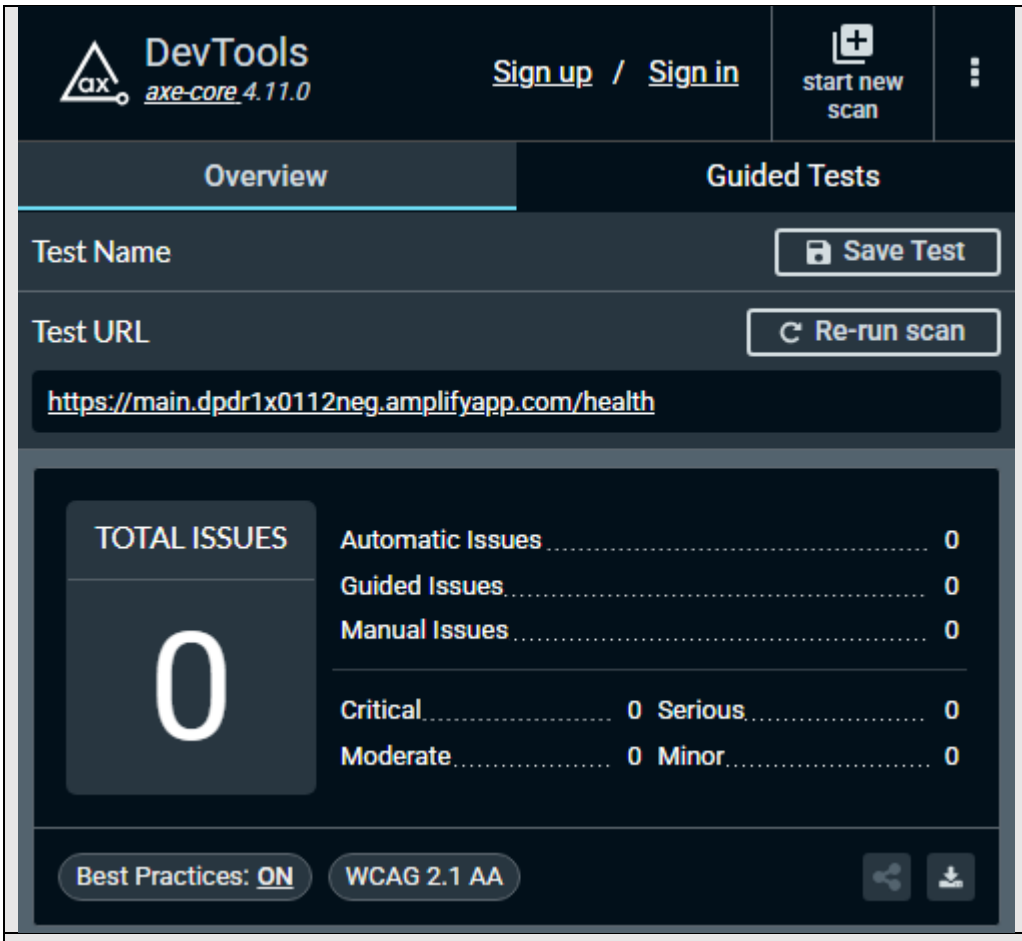
Outcome of tests:



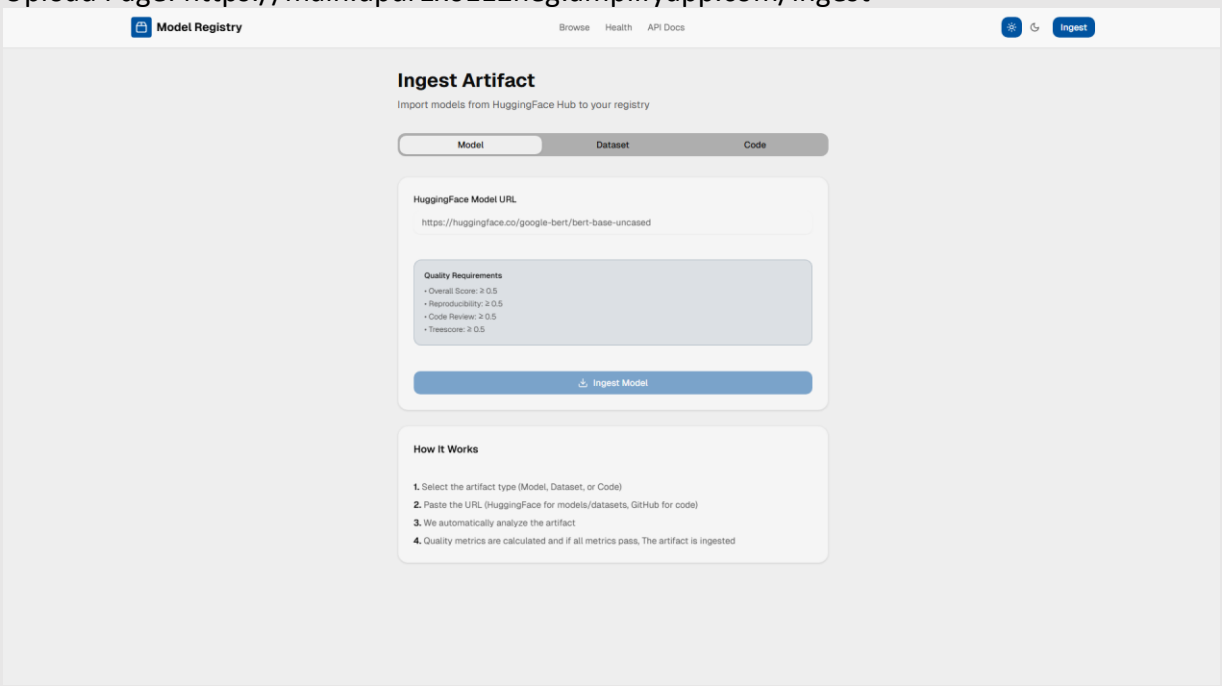
Health Dashboard: https://main.dpdr1x0112neg.amplifyapp.com/health



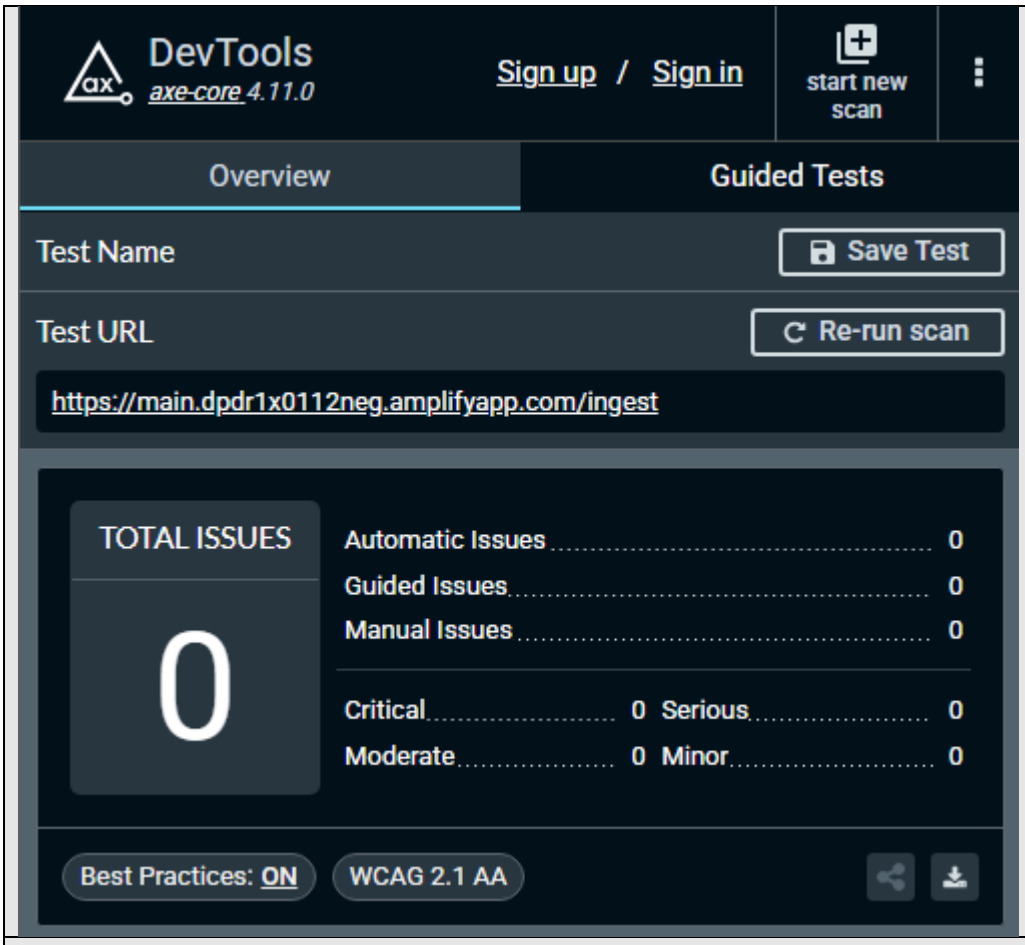
Outcome of Tests:



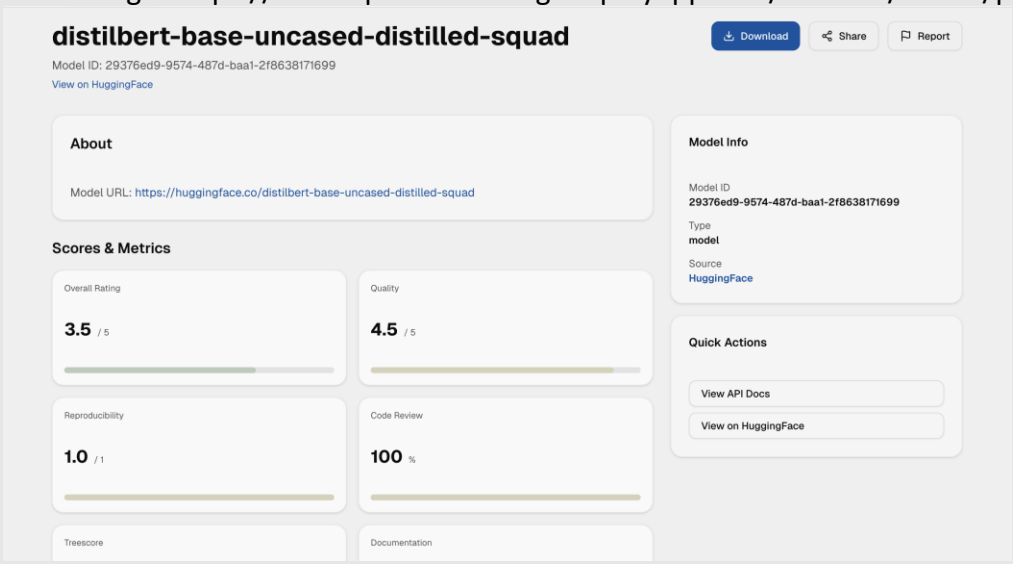
Upload Page: https://main.dpdr1x0112neg.amplifyapp.com/ingest



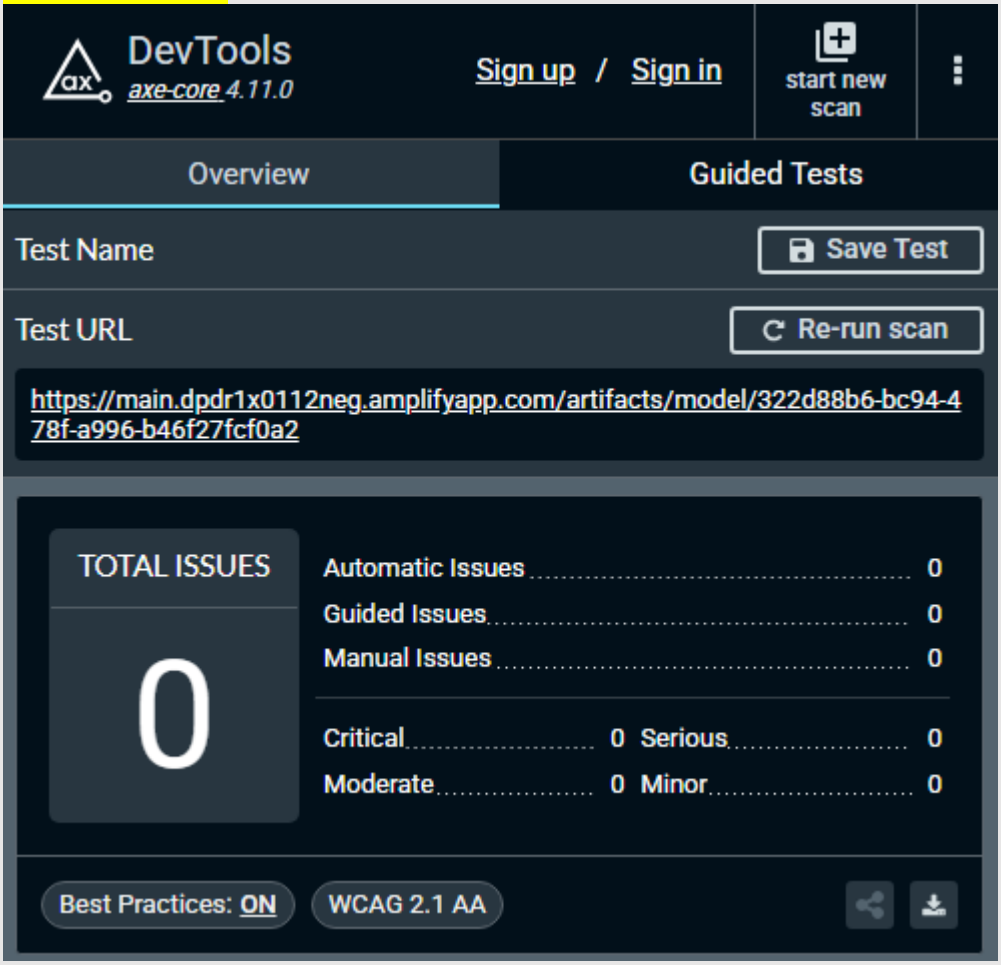
Outcome of Tests:



Model Page: https://main.dpdr1x0112neg.amplifyapp.com/artifacts/model/placeholderid



Outcome of test:



(Copy this block and repeat the information as needed)

Non-functional requirements

Security case

As part of your final report, you must submit an *updated* version of the security analysis report you completed a few weeks ago. This report should show your progress on implementing the mitigation strategies you identified as "Should Fix". While most of the report's content (like your threat analysis and DFDs) can remain the same as your previous submission, our focus will be on verifying the implementation of those planned mitigations. Please remember to submit this updated report with your final submission.

Deployment: AWS

Complete the following table. It should include all AWS components that you used.

Purpose	Selected AWS component(s)	Other AWS components considered	Justification for selected component (1-2 sentences)
Compute	ECS/ECR	Lambda	ECS was selected as the primary compute for its simplicity, cost-effectiveness, and easier security configuration. Lambda was also deployed to provide serverless benchmarking capabilities and demonstrate dual deployment strategies.
Storage	DynamoDB, S3	Rds postgres	DynamoDB was selected for metadata storage due to its simplicity, scalability, and low operational overhead compared to managing a relational database. Artifact files are stored in S3 since it is designed for large binary objects and easy integration with AWS services.
Logging	Cloudwatch		Cloudwatch easily integrates with existing AWS components.
CI/CD	Github actions/amplify		Github actions is that standard for github repos and it allowed us to deploy in the more similar manner to using the AWS cloudshell. Amplify

			autodeploys from main on merges.
UI/frontend	Amplify	S3	S3 only allowed static, which limited us in our use of react. However, amplify was easy to setup, cheap, and provided support for our reactive frontend.
Security	API gateway/ALB	VPC, no gateway	The API gateway allowed more manipulation of the security of our endpoints. The ALB provided security but also allowed us to use a consistent domain.

Deployment: CI/CD and Code Review

Describe your team's code review process.

What steps is your team following prior to accepting a code change? (e.g. git-hooks, code review, linting, test suite, etc.) (2-3 sentences)
When a PR is opened, an automatic GitHub Actions workflow starts with a linter (flake8), an import sorter (isort), a typing checker (mypy), and a final linter for specifically pydoc and naming convention alignment (pylint). Additionally, it runs our end-to-end testing suite and unit tests, if anything in the workflow fails, or the overall coverage is under 60% the workflow fails and the pr is not allowed to be merged. We have a separate action that runs on the frontend with Selenium which ensures that all pages load and are accessible. Finally, we have GitHub CoPilot configured for code reviews (although it only does that for members who have the non-free tier for some reason) and also require at least one person to review and approve the code before it can be merged.

Provide a link to an example in your GitHub repo where your team followed this CI process (e.g. a pull request):
https://github.com/Anjali-Vanamala/ECE461_Part2/pull/120
In this PR you can see the automated workflow under the checks tab. As well as the automated copilot review and manual review in the conversations tab.

How consistent have you been with this process? What is keeping you from full consistency? (2-3 sentences)
We have been very consistent with this process. There shouldn't be any code that was introduced after we started using the repository that was not introduced in this manner. The only place where we circumvented this is when we occasionally upload branches to AWS for testing purposes. However, the main branch has always contained our final development code.

Describe your team's CI/CD pipeline

What aspects of your system are being tested automatically by your CI scheme? (2-3 sentences)
The CI scheme automatically tests the code through the aforementioned GitHub actions workflow that runs on each PR including linting, import sorting, type checking, unit tests, end-to-end tests, and coverage requirements. We also have the mentioned secondary linting pipeline for the frontend through Selenium that checks that all pages load and does accesibility checks. Because code can't be merged until it passes all these check, we didn't require an additional CI pipeline after merging since the pre-merge workflow already makes sure the repo is always in a success state.

What kinds of defects might go uncaught, and how are you mitigating this risk? (1 paragraph)
I think the main defects that could potentially go uncaught are logical errors. Since we only require 60% coverage, there's a chance that code passes the syntax related checked and is basically untouched by the automated testing. This code could then be logically false but still compile leading to bad code entering our main branch. We mitigate this in two ways. The first are end-to-end tests. Since we test various endpoint which rely on lots of different parts of the codebase, it's less likely that a function is completely overlooked. Additionally, we utilize manual reviews. Reviewers are expected to at least look through the logic if not run the branch so they are likely to catch any errors since they have a view of only the code and logic that has been changed.

Provide screenshots of the GitHub action file (e.g. YAML) that defines the CI stages

```yaml
name: Python Lint, Import Sort, Typing

on:
  pull_request:
    branches:
      - '**'

jobs:
  lint-import-typing:
    runs-on: ubuntu-latest

    env:
      LOG_FILE: /tmp/test_log.txt
      LOG_LEVEL: "1"

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      # --------------------------
      #   PRINT DIRECTORY STRUCTURE
      # --------------------------
      - name: Show directory tree + pwd
        run: |
          echo "Current working directory:"
          pwd
          echo ""
          echo "Listing workspace:"
          ls -R .

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

```

```
36          - name: Install dependencies
37            run: |
38              python -m pip install --upgrade pip
39              pip install -r dependencies.txt
40              pip install flake8 isort mypy coverage uvicorn pytest pylint
41              pip install types-requests types-python-dateutil
42
43          # --------------------------
44          #  LINTING + TYPE CHECKING
45          # --------------------------
46          - name: Run flake8 (Python linter)
47            run: flake8 . --count --show-source --statistics --ignore=E501
48
49          - name: Run isort (import sorter)
50            run: isort . --check --diff
51
52          - name: Run mypy (type checker)
53            run: mypy . --ignore-missing-imports
54
55          - name: Run pylint
56            run: |
57              python -m pylint . --exit-zero \
58                  --disable=all \
59                  --enable=invalid-name,missing-module-docstring,missing-class-docstring,missing-function-docstring
60
61          - name: Add project root to PYTHONPATH
62            run: |
63              echo "PYTHONPATH=$PWD" >> $GITHUB_ENV
64              echo "Added $PWD to PYTHONPATH"
65
66          # --------------------------
67          #   RUN BACKEND UNDER COVERAGE (with logging)
68          # --------------------------
69          - name: Start backend under coverage
70            shell: bash
71            env:
72              BASE: "http://localhost:8000"
73            run: |
74              echo "Starting backend with coverage..."
75              nohup python -m coverage run --parallel-mode --source=. \
76                  -m uvicorn backend.app:app \
77                  --host 0.0.0.0 --port 8000 > server.log 2>&1 &
78
79              echo "Waiting for server..."
80              sleep 3
81
```

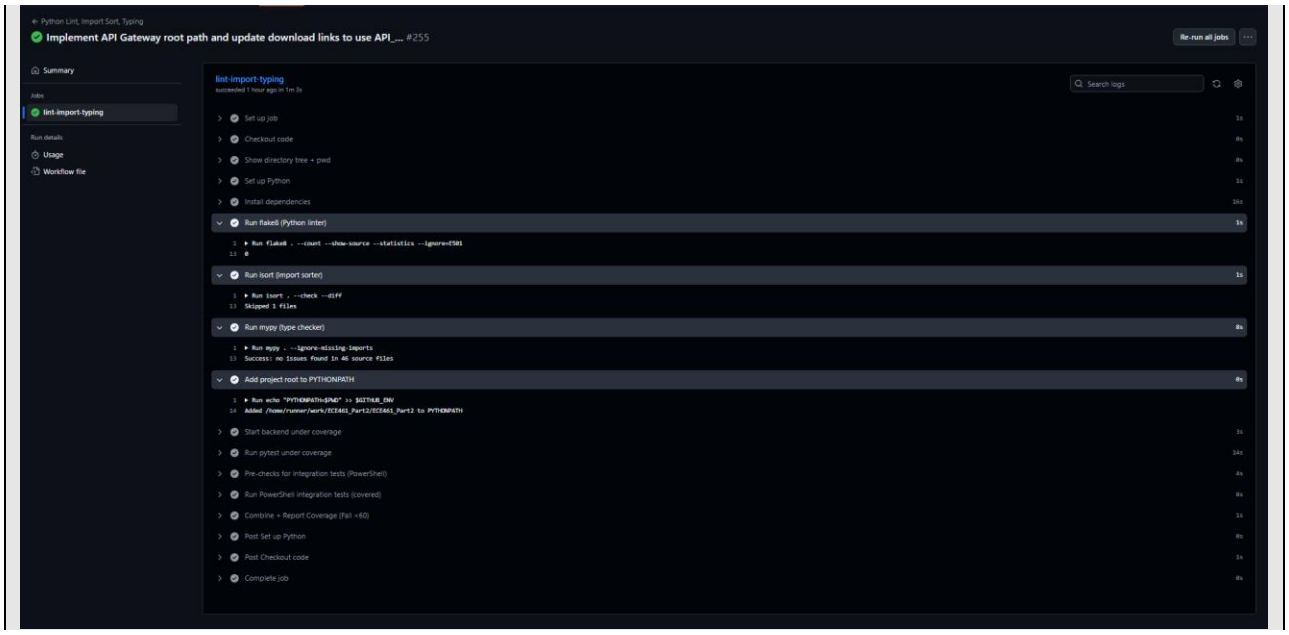
```
            echo "---- Server startup log ----"
            cat server.log || true

            for i in {1..30}; do
              if curl -fsS "$BASE/health" >/dev/null 2>&1; then
                echo "Server is up."
                break
              fi
              echo "Server not ready yet ($i/30)..."
              sleep 1
            done

            if ! curl -fsS "$BASE/health" >/dev/null 2>&1; then
              echo "🚨 Server failed to start."
              echo "---- Final server log ----"
              cat server.log || true
              exit 1
            fi

        # --------------------------
        # PYTEST UNDER COVERAGE
        # --------------------------
        - name: Run pytest under coverage
          run: |
            coverage run --parallel-mode --source=. -m pytest tests_main.py --maxfail=0

        # --------------------------
        # POWERSHELL INTEGRATION TESTS
        # --------------------------
        - name: Pre-checks for integration tests (PowerShell)
          shell: pwsh
          env:
            BASE: "http://localhost:8000"
          run: |
            Write-Host "Running integration tests against $env:BASE"
            Invoke-RestMethod -Uri "$env:BASE/health" -Method Get
            Invoke-RestMethod -Uri "$env:BASE/reset" -Method Delete

        - name: Run PowerShell integration tests (covered)
          shell: bash
          env:
            BASE: "http://localhost:8000"
          run: |
            echo "Running integration tests under coverage against $BASE"
```

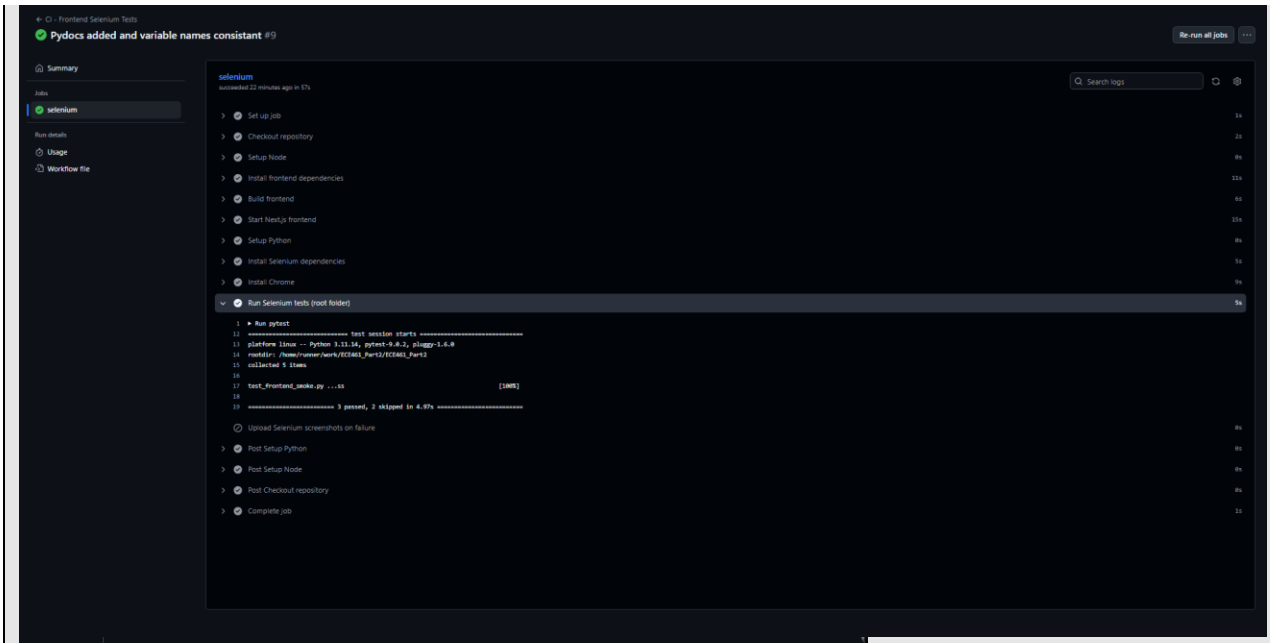
```
126
127             # Create a temp python script
128             cat > run_ps_integration.py << 'EOF'
129             import subprocess
130             subprocess.check_call(["pwsh", "./integration_tests.ps1"])
131             EOF
132
133             # Run it under coverage
134             coverage run --parallel-mode --source=. run_ps_integration.py
135
136         # -------------------------
137         # COMBINE & ENFORCE COVERAGE (Fail < 60%)
138         # -------------------------
139         - name: Combine + Report Coverage (Fail <60)
140           run: |
141             coverage combine
142             coverage report -m --fail-under=60
```

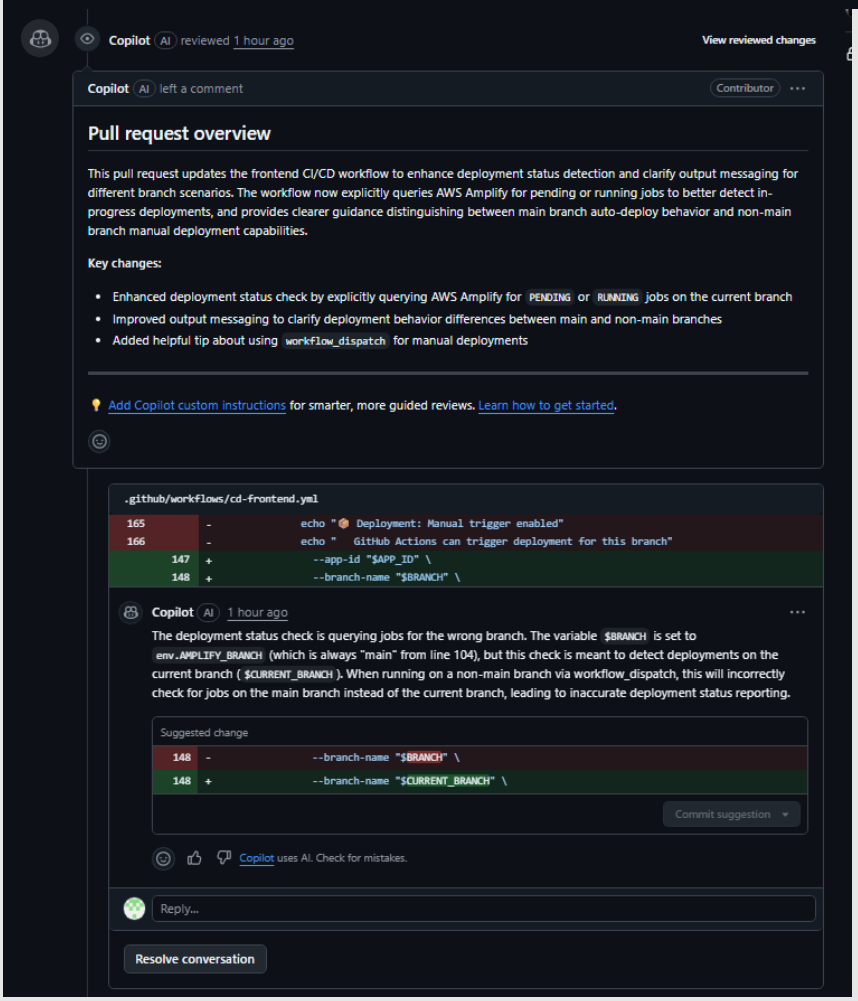
```yaml
name: CI - Frontend Selenium Tests

on:
  pull_request:
    branches: [ main ]

jobs:
  selenium:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Install frontend dependencies
        working-directory: frontend
        run: npm ci

      - name: Build frontend
        working-directory: frontend
        run: npm run build

      - name: Start Next.js frontend
        working-directory: frontend
        run: |
          npm run build
          npm run start &
          npx wait-on http://localhost:3000

      - name: Setup Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install Selenium dependencies
        run: |
          python -m pip install --upgrade pip
          pip install selenium pytest webdriver-manager

      - name: Install Chrome
        uses: browser-actions/setup-chrome@v1

      - name: Run Selenium tests (root folder)
        env:
          BASE_URL: http://localhost:3000
        run: pytest

      - name: Upload Selenium screenshots on failure
        if: failure()
        uses: actions/upload-artifact@v4
        with:
          name: selenium-screenshots
          path: screenshots/
```

https://github.com/Anjali-Vanamala/ECE461_Part2/blob/main/.github/workflows/python-lint.yml
https://github.com/Anjali-Vanamala/ECE461_Part2/blob/main/.github/workflows/ci-frontend-selenium.yml

Provide screenshot(s) of the CI test suite in action, e.g. the reports from the various tools you have configured, as run on one of your team's code changes.


Python Lint, Import Sort, Typing
Implement API Gateway root path and update download links to use API_... #255
Re-run all jobs
Summary
Jobs
lint-import-typing
Run details
Usage
Workflow file
lint-import-typing
succeeded 1 hour ago in 1m 3s
Search logs
Set up job
Checkout code
Show directory tree + pwd
Set up Python
Install dependencies
Run flake8 (Python linter)
Run flake8 . --count --show-source --statistics --ignore=E501
0
Run isort (import sorter)
Run isort . --check --diff
Skipped 1 files
Run mypy (type checker)
Run mypy . --ignore-missing-imports
Success: no issues found in 46 source files
Add project root to PYTHONPATH
Run echo "PYTHONPATH=$PWD" >> $GITHUB_ENV
Added /home/runner/work/ECE461_Part2/ECE461_Part2 to PYTHONPATH
Start backend under coverage
Run pytest under coverage
Pre-checks for integration tests (PowerShell)
Run PowerShell integration tests (covered)
Combine + Report Coverage (Fail <60)
Post Set up Python
Post Checkout code
Complete job





Describe the extent to which you are able to “continuously deploy”. What is your team’s process to get your current prototype into a deployment on AWS? (2-3 sentences)

On successful PR to the main branch, an automated GitHub workflow runs to deploy the branch to AWS. We have two workflows, one that sends our backend to the ECS container with proper environmental and secret configs (for LLM and Github API keys) and one that triggers the frontend to “auto-deploy" from our main branch to AWS amplify.

Non-Baseline Requirements

Extended Requirements

Please list the extended requirements track(s) you selected (e.g. Performance, Access Control, High Assurance, ML inside), and the requirements you completed within those tracks:

Performance Track:

- Measure throughput + mean, median, 99th percentile latency for 100 clients downloading Tiny-LLM from a registry with 500 models.
- Provide black-box measurements (client perspective) and white-box explanation (internal root causes).
- Identify ≥ 2 performance bottlenecks, explain how they were found, how they were optimized, and show before/after measurements.
- Modify the system so backend components (e.g., Lambda vs EC2, S3 vs DB) can be swapped through config. Measure their impact on latency & throughput.

In the following sections, only fill out textboxes for the sub-requirements your team completed.

Non-baseline: API

If you implemented additional behaviors in your team’s web API, (e.g., creating users, package groups, or traceability features) they should be reflected in your team’s OpenAPI specification on GitHub.

Provide a link to your team’s OpenAPI specification:

https://9tiiou1yzj.execute-api.us-east-2.amazonaws.com/prod/docs

Fill out the following template for each additional API feature (*Please copy-and-update the heading "Feature X", too, so we can use the navigation pane when grading*)

Feature Download Performance Benchmark

Feature: Download Performance Benchmark - Start Benchmark Job
Relevant endpoint(s):

POST
/health/download-benchmark/start

GET
/health/download-benchmark/{job_id}

How completely is it implemented?

Fully implemented. Starts a background thread to run the benchmark script, returns a job_id for status polling, prevents concurrent runs (409 if one is running), and handles errors. The benchmark runs 100 concurrent downloads of Tiny-LLM and measures performance metrics.

How did you validate it? *(Fill out this table for the feature – This should include the relevant kinds of positive and negative error cases you tested)*

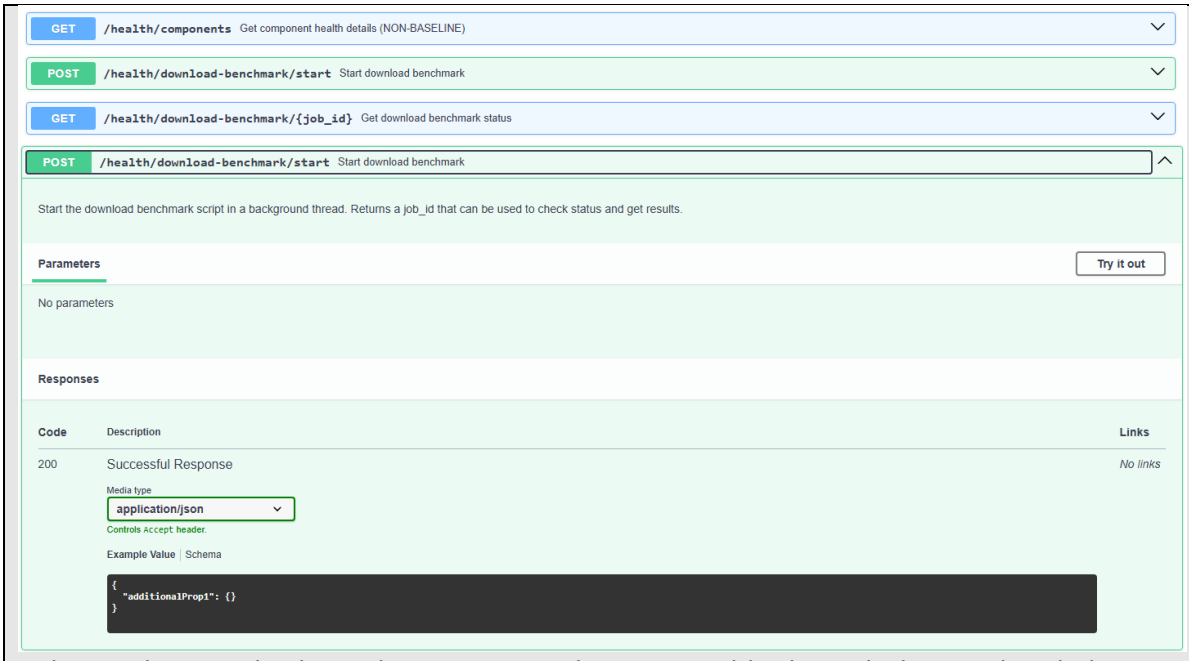
Endpoint | Verb(s) | Payload option(s)	*Validation approach(es)**	*Test records***
/health/download-benchmark/start | POST | Valid input	Manual Testing	Relevant PR (1) Initiated a test using "run" button on frontend (2) Checked to see if "Executing download benchmark... "showed (3) Waited for execution and black box metric to show Most recent test: 12/14/2025
/health/download-benchmark/{job_id} | Get | Valid input	Manual Testing	Relevant PR (1) Polling a running job returns status "running" with progress updates

		(2) Completed job returns status "completed" with full results including latency metrics (mean, median, P99), throughput metrics (requests/second, Mbps), and request summary (3) Failed job returns status "failed" with error message. Most recent test: 12/14/2025

Performance Track

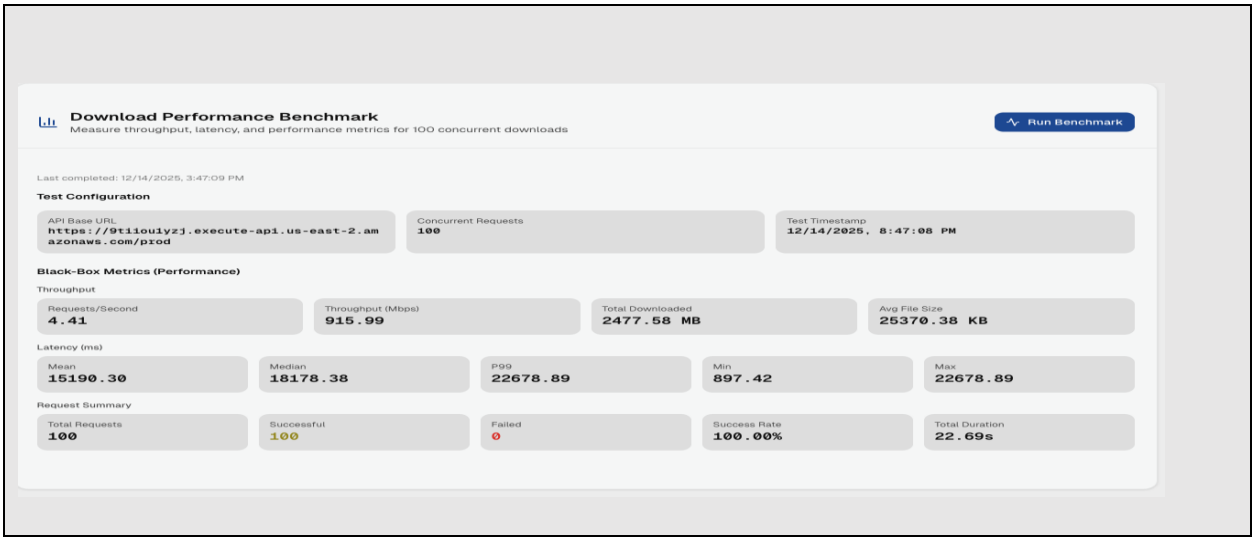
If you implemented the following performance requirements:

- Appropriate API design and documentation as reflected in the OpenAPI specification changes

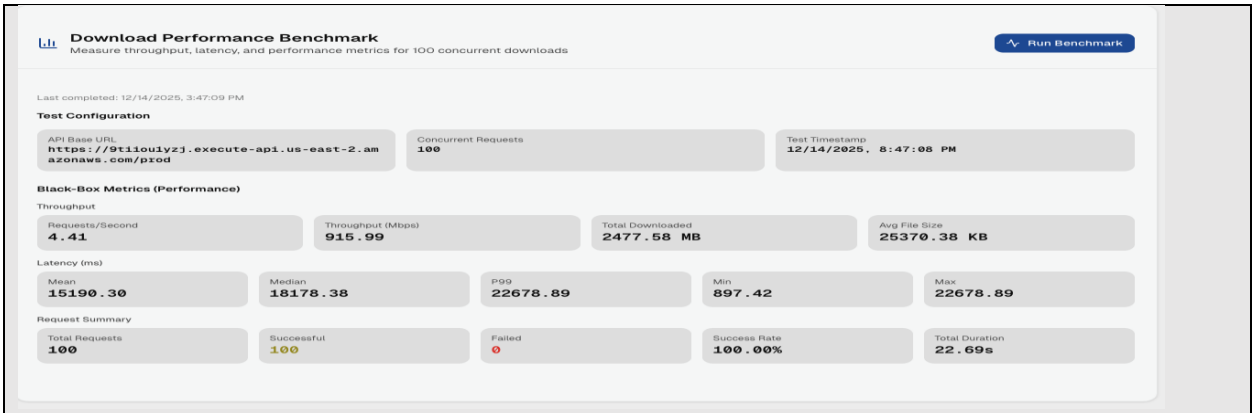


Other endpoints also have descriptions and are accessible through the API docs link above.

- Provide latency details for mean, median, and 99th percentile clients for the “many clients downloading Tiny-LLM” scenario described in the project spec (or as close as you could get to that scenario). (1 paragraph)



- Provide black-box measurements (client perspective) and white-box explanation (internal root causes) (1-2 paragraphs).



The metrics indicate limited parallelism in request processing. The system handles downloads with a worker/connection pool, so some requests queue and wait. When the pool is exhausted, new requests wait for workers to finish, adding idle time. The right-skewed latency distribution (median 18.18s > mean 15.19s) and P99 matching max (22.68s) indicate most requests cluster at the high end, consistent with serialized processing. The high throughput (915.99 Mbps) and large file sizes (24.8 MB average) confirm bandwidth isn't the issue; the constraint is the connection pool or thread pool that prevents true parallelization, causing requests to be processed in batches rather than simultaneously.

- Identify at least two performance bottlenecks from these measurements. Describe how you found them. Describe how you optimized them. Describe the effect. (1-2 paragraphs)

Connection pool limitation bottleneck: The benchmark client's aiohttp.TCPConnector had a connection limit of 75 connections causing requests to queue when the pool was exhausted.

Synchronous artifact metadata lookups bottleneck: Each download request was performing synchronous operations to retrieve artifact metadata and processing status. This was confirmed by comparing request start timestamps to download initiation timestamps, revealing a consistent 1-2 second gap.

Optimization and Effect: The connection pool limit was increased from 75 to 200 connections in the benchmark client (visible in benchmark_concurrent_download.py line 188: limit=200), allowing more concurrent downloads without queuing. The artifact metadata lookups were optimized to use in-memory dictionary caching first, falling back to database queries only when the cache miss occurs, eliminating most database lookups for frequently accessed artifacts. After these changes, median latency improved from 18.16 seconds to 15.16 seconds (3-second reduction), and mean latency improved from 15.19 seconds to 14.15 seconds (1-second reduction). The 100% success rate was maintained, and the latency distribution tightened, with requests experiencing less queuing delay and faster metadata retrieval. The P99 latency remained similar (22.68s to 22.70s), indicating these optimizations addressed consistent per-request overhead rather than tail latency, which is expected given the focus on connection pool and database lookup bottlenecks.

- Describe any design choices you made specific to performance (e.g. component selection; optimized paths such as caches; etc.). If you implemented it, describe how users can configure the underlying AWS components. (1-2 paragraphs)

We selected ECS/Fargate over Lambda for the backend to support long-running processes, consistent performance for artifact processing, and better resource control. For storage, S3 was chosen over RDS to efficiently handle large binary artifact files (models, datasets) with versioning and lifecycle management. The frontend uses AWS Amplify instead of basic S3 static hosting to leverage built-in CDN distribution, automatic caching, and optimized build pipelines that cache node_modules and Next.js build artifacts.

In-memory caching is implemented for expensive metric calculations (size scores, license scores) to avoid redundant API calls to HuggingFace and GitHub. ECS autoscaling is configured via the AWS Console with CPU utilization thresholds and task limits. Amplify build caching is automatically configured via amplify.yml to cache node_modules and .next/cache directories, reducing build times on subsequent deployments.

Notes for the auto-grader

If your submission cannot be automatically parsed by the auto-grader described in the project specification, provide explanatory notes that the course staff can consider while scoring your submission. Be specific. Since the project specification was provided well in advance, accommodating any deviations is at the discretion of the staff.

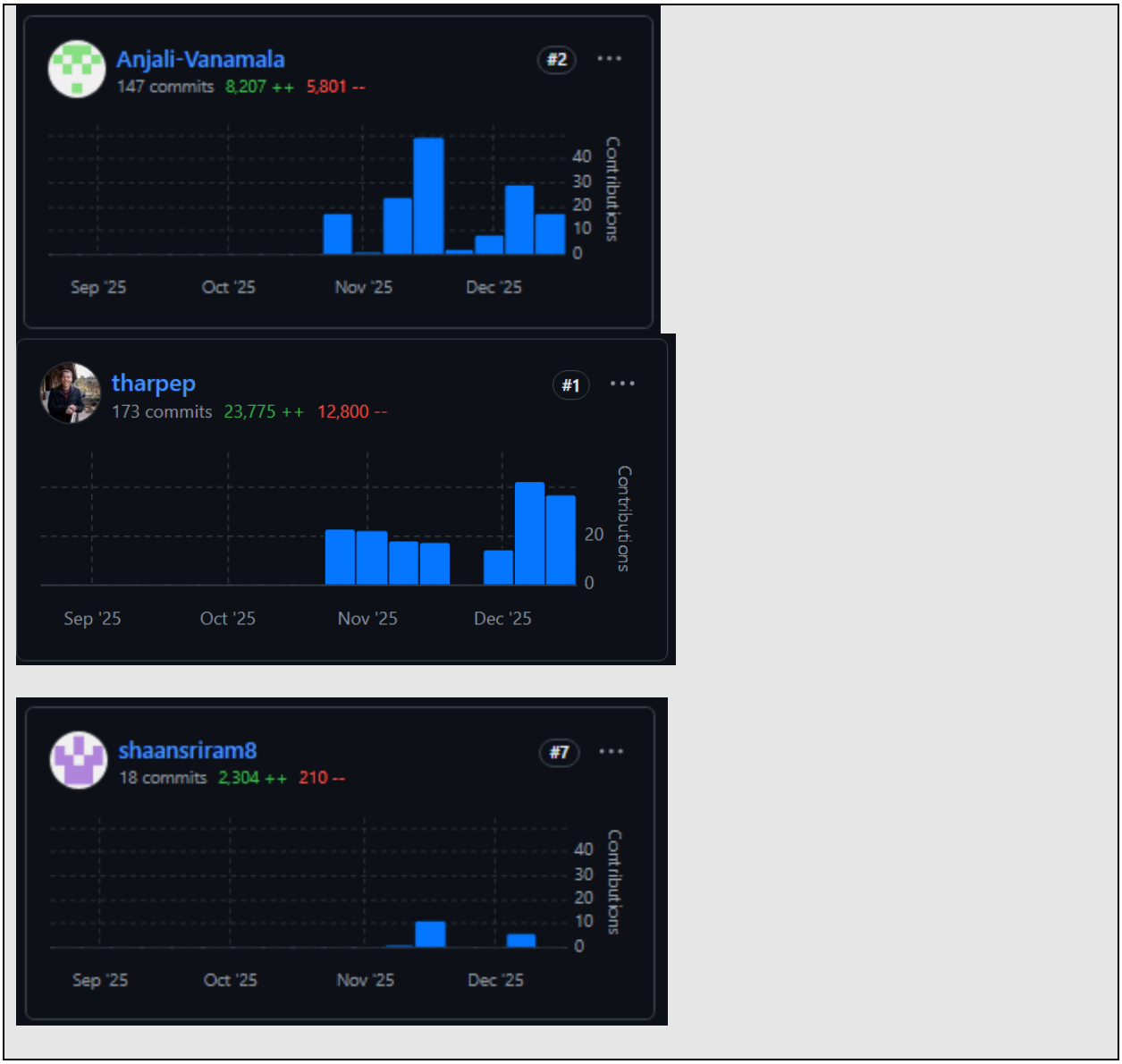
Details
e models asynchronously; thus, according to the spec, if a model does not pass the ingest, we should drop it silently. However facts test" fails, which does not allow us to even attempt some other tests like the regex tests. Thus, just for the Autotrader, w ept every model. We'd like to point this out so that when doing a manual review, it doesn't look like we just failed to impleme log of a run with ingest on (Note this was before we implemented some stuff like downloads etc). http://dl- rdue.edu/api/phase2/log/download?group=3&gh_token=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&log=data/3/run 12-17-15.log
our phase 1 metrics are either expected higher or expected lower. As shown in the section about changes to phase 1 metrics, w e amount of time going through, validating the metrics, and changing them where necessary. Additionally, we did extensive m score takes into account (code blocks, card data, etc) and validating the metric scores to be in the range of expected given the s makes sure that any deviations are not due to improper logic in the code and that the metrics are indeed fetching everything nly disagree with the rating at most 3 times, so it's not that our metrics are consistently incorrect, they just disagree occasiona would be reasonable for us to get the points back.

Notes for the teamwork checks

Last modified: 2 December 2025

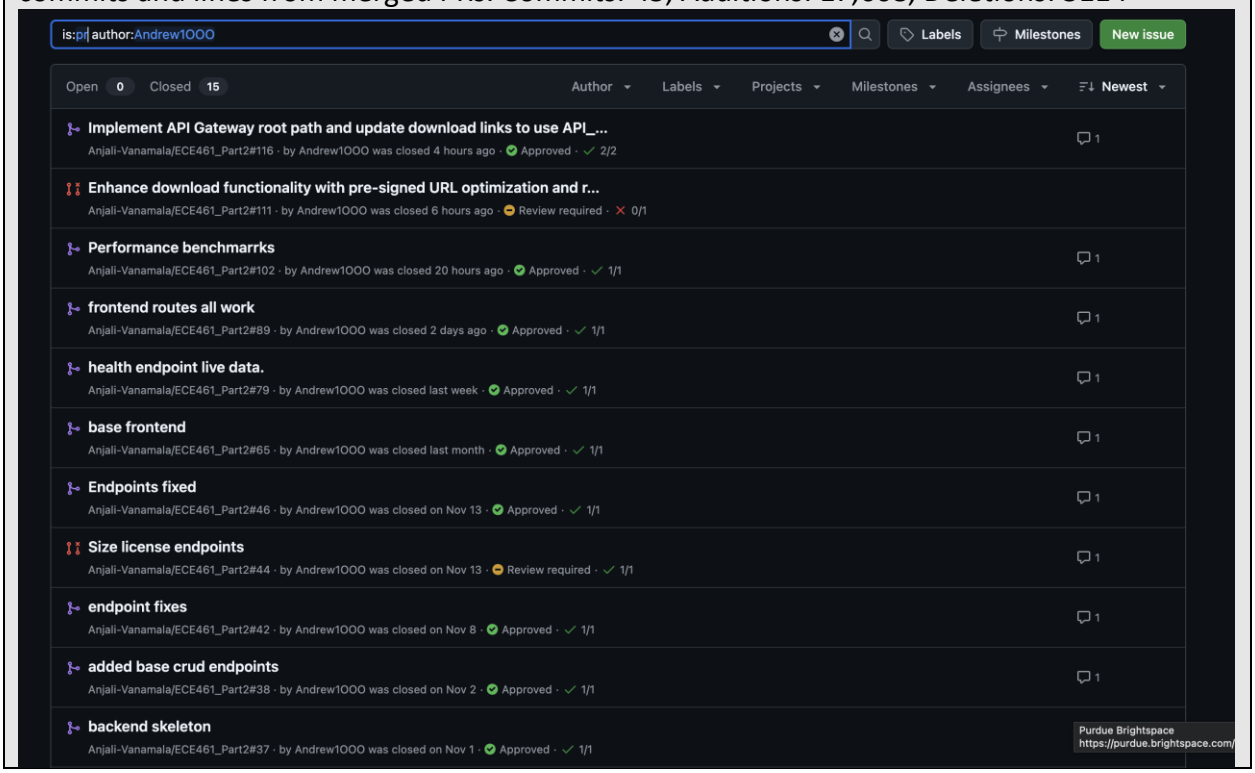
ECE 461 – Software Engineering

Provide GitHub Insights screenshot(s) indicating the number of commits and lines of code contributed by each team member.



Last modified: 2 December 2025

Andrew doesn't have a graph as he lost access to the email tied to his commits. We added up commits and lines from merged PRs. Commits: 45, Additions: 17,608, Deletions: 5114



In a typical team, each team member contributes a substantial amount of code. Look at the preceding screenshots. If on your team, some team members appear to have contributed substantially less to the implementation effort than other team members, you may provide an explanation for us to consider (ex. they were responsible for handling for all of our AWS components).

Team member: Anjali Vanamala

Important non-code contributions, if any: Did all of the manual and automated ADA testing, did most of the autograder testing, set up keys for LLM and API in AWS, did most of the validation of rating metrics, did most of the end-to-end testing of the system. Did a large portion of the write-up work. Additionally, was not responsible for any big framework setups that inflate additions like the overall frontend and backend, and instead mostly worked on component and endpoint level implementations, hence the large number of commits.

Team member: Shaantanu Sriram:

Important non-code contributions, if any: Completed a large portion of AWS setup initially, including budget monitoring/setup, group creation/account permission setup (IAM users),

storage architecture overhaul which included setup of DynamoDB table schema, as well as some ECS task definitions and config setup. Additionally, created our Project monitor in GitHub. Was not responsible for setup that inflates additions, and regarding switching our storage architecture, spent much longer debugging, implementing, and refactoring code in the backend for seamless integration from memory->DynamoDB.

Team member: Pryce Tharpe

Important non-code contributions, if any: After Shaan did the initial AWS setup, I contributed to most other AWS deployments. Besides DynamoDB Shaan did, I was responsible for the rest of deploying each AWS component and much of the other ECS task definitions.