Optimizing ASSISTments Microservices for Security, Learning Efficacy, and Student Engagement

Ben Carpenter, Hunter Jack, and Griffin Munhall

Computer Science Department, Worcester Polytechnic Institute

Abstract

ASSISTments is an online tool that uses math problems to teach middle school students. The goal of this project is to maximize learning by making ASSISTments problems more engaging for students. Testing and data collection was done for Persona Problem Builder with the intent of crafting problems more relatable to students, while still preserving the underling mathematical content. Structural changes were made to Reinforcement Learning Services, modernizing it and removing vulnerabilities. Also, a new system was created to detect when students were attempting to move through problems quickly without learning the content. We believe the cumulative effect of these changes will improve the efficacy and security of ASSISTments.

Acknowledgements

We would like to thank everyone at ASSISTments for their continued efforts in pushing the boundaries of education forward and for giving WPI students the opportunity to contribute to that work. Morgan Lee, as our advisor, worked closely with us on all aspects of our project, providing support and guidance that laid the groundwork for what we were able to achieve. Neil Heffernan, a key orchestrator of the important work that ASSISTments is doing, extended us the opportunity to work in his lab. Without him, none of this would be possible. Angela Kao assisted with our onboarding and provided the administrative backbone we needed to succeed. Abubaker Siedahmed oversaw the Gaming Detection portion of our project. A heartfelt thanks to everyone that gave us the opportunity and support that allowed us to learn so much throughout this project.

Table of Contents

Chapter 1

Introduction

ASSISTments is an online tool used to teach math primarily to middle school students. As students answer questions on a variety of math topics, they receive instant, personalized feedback to help them improve. Likewise, teachers receive feedback on each student and the problems that posed the biggest challenge. Teachers can integrate ASSISTments with Google Classroom or Canvas. We focused on four of ASSISTments' sub-components in this project — Persona Problem Builder (PPB), Reinforcement Learning Services (RLS), Quick Comments (QC), and Gaming Detection. PPB allows for the customization of problems, making them more relatable to the student. RLS finds and exploits heterogenous treatment effects. QC provides automatic actionable feedback to student responses. Gaming Detection is a new service created for this project, detecting student behaviors that are gaming the system (i.e. using the hint system to get answers and avoid working the problem).

Chapter 2

Persona Problem Builder

2.1 Background

Persona Problem Builder takes preexisting ASSISTments problems, adapting topics to better relate to the student. To do so, it is necessary to provide an ASSISTments problem ID and a new topic. Optionally, PPB can create a newly generated problem with numbers that differ from the original problem.

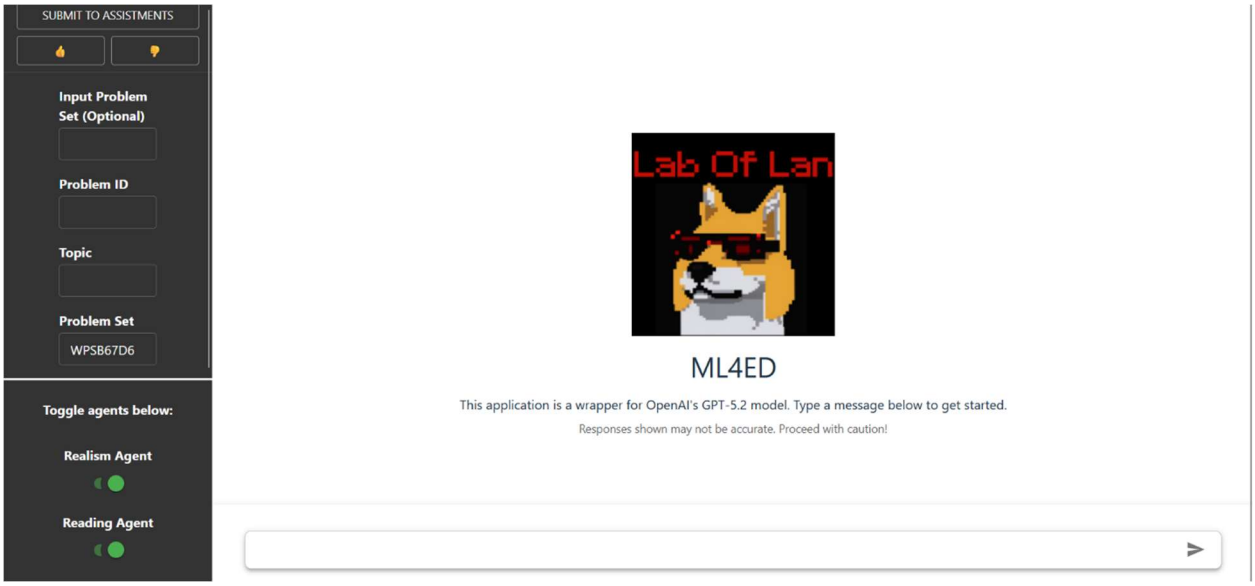

Figure 1: The UI for Persona Problem Builder

PPB is prone to mistakes. In order to counter this, four agents are used to identify and fix these mistakes. Once the new problem is generated, it is reviewed by each of the four agents. If a mistake is found, the reasoning is displayed and the problem is re-generated using that context. The re-generated problem is reviewed afresh by the agents. It is possible for subsequent iterations of a problem to repeatedly fail. If necessary, this process will repeat up to a maximum of five times. Once the process is complete, PPB gives the option to submit the new problem to ASSISTments.

During the course of our project, a feature was added allowing individual agents to be toggled on or off. Each agent serves a different purpose, attempting to correct different failures:

1. Realism — Checks for logical issues (Does this make logical sense?)
2. Authenticity — Checks for relatability (Does this relate to the student?)
3. Reading Level — Checks for reading level consistency (Does this maintain the same reading level as the original?)
4. Hallucination — Checks for correctness (Does this make mathematical sense?)

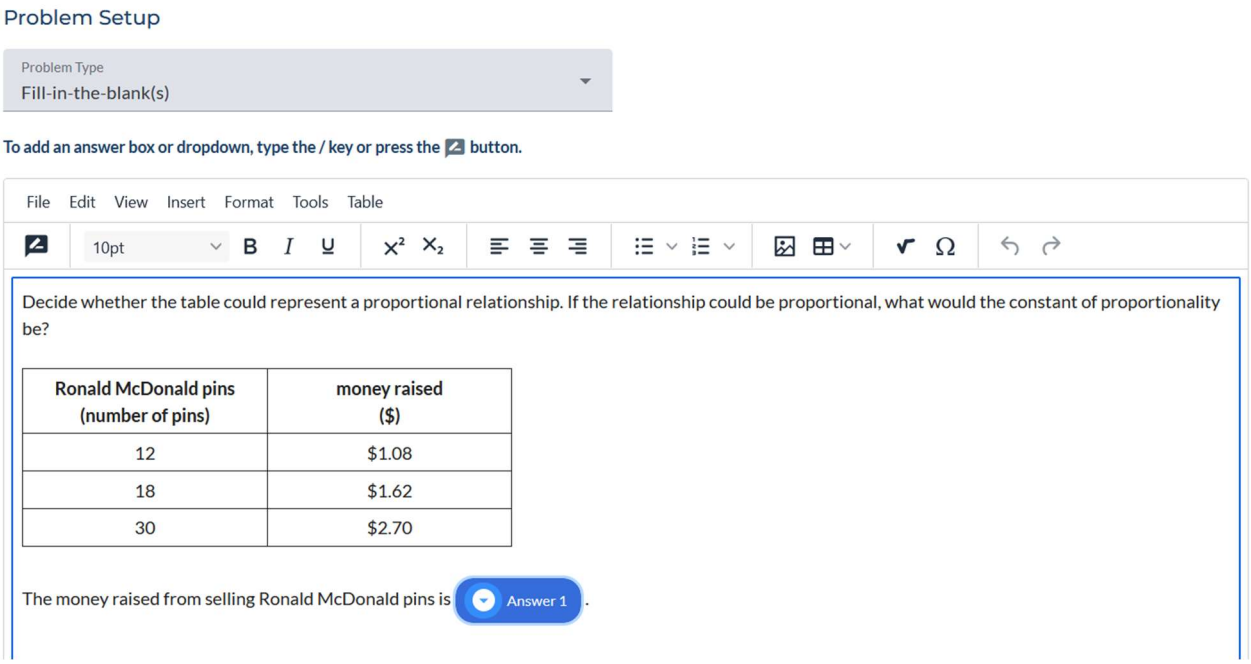

Figure 2: A Persona Problem Builder problem submitted to ASSISTments

2.2 Technical Background

Persona Problem Builder is a wrapper for GPT-5.2, a Large Language Model (LLM) by OpenAI. Problems are generated by sending information to GPT-5.2, then reporting the result to the user. For instance, PPB provides GPT-5.2 with the original problem text, the topic to use for the new problem, and how each agent should evaluate the new problem. GPT-5.2 then uses this to generate the problem. From there, GPT-5.2 is also used by the agents to evaluate the generated problem, with each agent having its own prompt that explains how the agent should work in detail. PPB takes all of these outputs from GPT-5.2 and displays them to the user, showing each iteration of the new problem, which agents failed, and why they failed.

2.3 Insufficiencies

Although PPB generates high quality problems most of the time, it is still prone to errors under certain circumstances. Each agent can either fail to recognize an issue with a generated problem (false negative) or take issue with a problem that is perfectly fine (false positive). Sometimes, even if the option to change the numbers is selected, the numbers remain unchanged. Certain topics do not go well with certain problems. If an unsuitable combination is attempted, PPB uses the maximum number of attempts just to generate a new problem that is severely flawed.

2.4 Development and Improvement Process

The Quality Assessment (QA) of PPB boils down to five steps:

1. Retrieve a problem ID from one of the nine problem-set lists.
2. Pick a topic intended to match an average middle-schooler's interests.
3. Enter both into the PPB testing site and await the GPT response.
4. Read the response, identify potential failures of various agents (e.g., hallucinations), identify model-produced response characteristics (e.g., reading level), and note the final question text.
5. Enter all relevant information collected thus far into a multi-question Microsoft (MS) Form, designed to collect data that can be used for later analysis.

A single tester would repeat this loop dozens, if not hundreds, of times across the nine problem sets. Steps 1, 2, and 3 are purely copy, paste, and button pressing. Step 4 involves pattern-matching typical model responses to the current one, and step 5 takes that information

and enters it into a form. It became clear that the primary burden was not cognitive, but rather the continuous context-switching necessary to complete a form submission. Mechanical and redundant data entry, the latency of waiting for GPT to respond to each request, and the consistent output of the model (whose key features reliably followed the same pattern each time) all pointed toward the possibility of automation. In order to do this, we tried two approaches, each with a different tool: AutoHotkey and Selenium Automation Suite.

According to AutoHotkey's (AHK) website, "AutoHotkey is a free, open-source scripting language for Windows that allows users to easily create small to complex scripts for all kinds of tasks such as: form fillers, auto-clicking, macros, etc." Scripts GetTopic.ahk and RandomProblemSetID.ahk were written to automate the simpler, more tedious tasks in our QA pipeline. This includes getting the topic and problem ID into PPB and entering its response into the form. GetTopic.ahk reads a topics.csv file; normalizes line endings to commas; splits on commas; trims whitespace; pastes one randomly selected topic at the cursor, via SendInput; and finally attaches that topic to the clipboard for easy copying and pasting. RandomProblemSetID.ahk captures a single keystroke through an InputHook with a 3-second timeout, validating that it is a digit (1–9); reads the matching problem_id_list_<N>.csv with the same normalize-split-trim pattern; pastes a random ID; and finally attaches that problem ID to the clipboard. Both scripts compile to windows executables, so the QA tester would not need AHK installed locally.

These scripts addressed a major inefficiency in the QA pipeline, bringing the time required to insert a topic and problem ID down to roughly one second. Even though this simple approach noticeably reduced the tediousness of the QA process, it failed to address the

generation's latency or the output transcription, which were the two other major hurdles. This naturally led us to search for a more robust solution, bringing us to a full browser-automation pipeline written in Python (using the Selenium WebDriver package). Selenium WebDriver is an open-source testing framework designed to automate web applications, allowing users to directly control a web browser via their programming interface (Selenium, n.d.).

These capabilities allowed us to break our process down into three phases:

- **Setup -** On launch, the script prompts the user for a number (1–9), reflecting which problem set is currently being tested. It then loads both the topics.csv and problem_id_list_<N>.csv. Once these are loaded it begins the Chrome session, bound to a local selenium_profile/ directory via --user-data-dir. This profile persistence is a key point in the automation, as PPB authentication is performed manually on the script's first run, after which the cookie state is reused. The driver is launched "detached", meaning the browser stays open upon script exit, allowing the tester to inspect any necessary output. After launch, the script opens a user defined number of PPB tabs sequentially (default of 3). In each tab, it awaits the HTML elements for Problem ID and Topic to become available. A randomly chosen Problem ID/Topic pair is then entered into the relevant fields and the form's send button is clicked.
- **Generation -** With all tabs submitted in parallel, GPT responses stream simultaneously, effectively reducing the time per question by the number of tabs.
- **Reporting** - To fill out the form for each respective tab, the script enters an interactive loop. The tester watches the tabs, prompting the script to continue once

the generation is visibly complete (i.e. GPT has given an entire response). The script then:

1. Switches to the completed tab
2. Captures the model response
3. Extracts the reading level via regex
4. Extracts the final output via XPath
5. Opens a new Chrome window and navigates to the MS Form
6. Enters all collected information thus far (topic, problem ID, reading level, final output)
7. Detaches from the browser, allowing the QA tester to finish the remaining analysis

This loop tracks the processed tabs in order to gracefully handle the closing or reprocessing of tabs.

Although no formal time study was conducted, qualitative reports show obvious improvements. Initial mechanical entry was reduced from 6 paste actions to 0. Generation latency is parallelized across the number of tabs, allowing the generation to happen concurrently rather than sequentially (reducing the time to roughly $\frac{1}{n}$th of the original time, where n is the number of tabs). Transcription error is reduced for the two scraped fields, and any potential login friction is reduced to once per profile via the selenium_profile/ persistence.

Even with these significant time reductions, there are several future improvements worth making. Currently there is no detection that GPT has finished a response, leaving the tester to manually watch each tab and press Enter. One way to automate this could be polling for the

appearance of the “Hints:” sentinel in the model response. There is also some fragility in how selectors are used in the test suite. React IDs or content-hash class names can easily change with any framework-level rebuild of PPB, causing silent failures and requiring further maintenance. The lack of form coverage is another area that could be improved. While many of the form fields are qualitative judgments requiring a human-in-the-loop, several mechanical fields (such as “Was bias evaluation enabled?”) can be automated in the future. A particularly obvious issue is also the lack of logging or saving the information. Since the results of running the script are not aggregated, logged, or preserved once the browser is closed, there is a potential for loss of data. To resolve this, local result logging could be implemented, writing each record to a local JSON or CSV. This would provide a recoverable audit trail in case of any failures.

This test suite offers several benefits to ASSISTments. First is the obvious benefit of reducing the amount of time to evaluate PPB outputs. PPB’s quality depends on broad coverage of problems and topics, so any reduction in the cost per output evaluation can accelerate confidence in any change made to the wrapper. More significantly, the QA pattern here generalizes any wrapped tools used by ASSISTments beyond PPB. The structure of a persistent-profile Selenium driver, parallel tabs against the tool, scraping of structured response fields, and a pre-filled MS Form is effectively tool-agnostic. In this sense, the contribution might be less specific to PPB and more to the working blueprint of how ASSISTments evaluates LLM-driven tools in the future.

Testing problem generation was more efficient using the Selenium Test Suite, but we also had to frequently reconsider the best way to report the data. We primarily used a Microsoft Form to record the data from any problem we generated, even if we found no issues with how PPB

handled the problem generation. At first, this form only included a few questions, but as testing continued, we discovered issues that were not addressed in the form. We found, for instance, that as problems using images were submitted to PPB, the newly generated problems would sometimes exclude the images, while other times it would use the images in new, unrelated contexts. Because of this, we added new questions to the form to determine whether the original problem had an image. If it did, we asked if the new question included it in a way that fits.

Oftentimes there were questions whose relevance depended on how the user answered a previous question. For example, if the user reported that there were no bugs in the new problem, there was no need to ask any questions related to any of the frequent bugs. Whenever we added a new question, we included this sort of logic to avoid wasting time answering irrelevant questions.

Due to our refinement, the resulting forum contained an extensive amount of information about each problem. Some of the data included:

1. The problem ID
2. The topic
3. PPB's estimated reading level of the original problem
4. The full text of the new problem generated by PPB
5. Whether PPB made any mistakes in generating a new problem
6. Whether there was a false positive or negative
7. Whether the number change function worked properly
8. The difficulty of the new problem relative to the original problem

9. Whether the new problem preserved the mathematical content of the original problem
10. Whether any images used in the original problem translated properly to the new problem

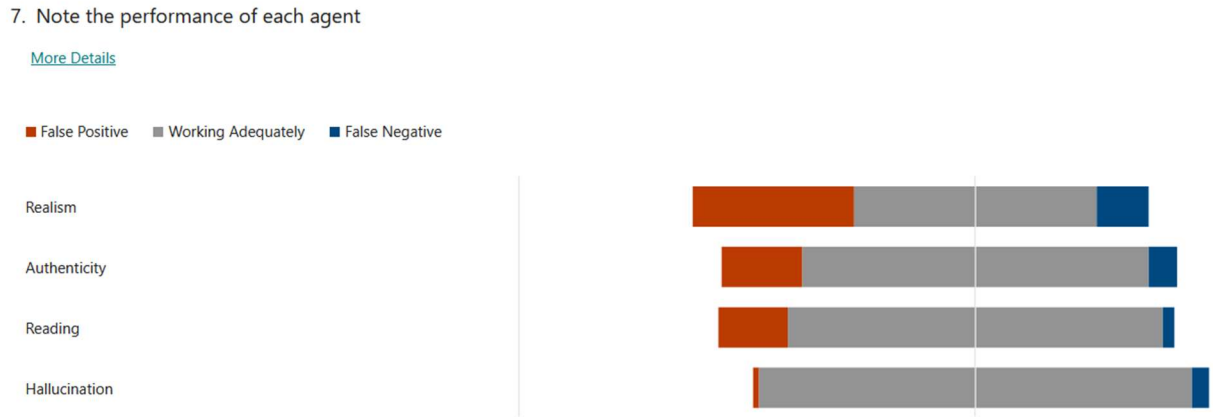

Figure 3: The results for each agent over the course of every problem

This data was extremely helpful in identifying which aspects of PPB were more likely to fail. We found that the realism agent caused the most problems by far, often failing to identify logical issues in the newly generated problems. We were also able to show that the change numbers function was more likely to fail on certain problem IDs and that PPB would occasionally assign differing estimated reading levels to the same problem.

In total, we submitted the results for 125 problems to this form (this is in addition to 66 results from our previous Google Form). We experimented with a variety of problems and topics. Of all the problems we tested, about two-thirds had a bug. This reporting form helped to log information, but we had other means of recording data. We stored more in-depth problem results in a Google Document. With this document, we not only recorded the final problem generated, but also any failed iterations. For each failed iteration, we also wrote down which agents failed and why. This document made it easier to explain the errors we encountered, letting us show the

agent's output, at what stage an error occurred, and why. Rather than relying on memory, we could review tests we had done, even if they were carried out several weeks earlier.

PRBP5HR

Ronald McDonald:

- Grade 6
- Ronald McDonald is a popular character often seen at fairs and amusement parks. Imagine you are at a fair where there is a life-sized Ronald McDonald statue. Use the scales and drawings provided to approximate the actual dimensions of the statue:
- The height of the Ronald McDonald statue, to the nearest foot:
- The width of the statue's arms when spread out, to the nearest foot:
- The length of the statue from head to toe, to the nearest meter:
- It refers to drawings but no drawings are actually provided
- Current Agent Score: 0.67, compared to the threshold 0.8
- Realism: The length of the statue from head to toe should be measured in feet, not meters, as it is a life-sized statue.
- Realism: The problem does not provide any scales or drawings, which are necessary to approximate dimensions.
- Realism: The use of both feet and meters for similar measurements is inconsistent and confusing.
- Ronald McDonald is a popular character often seen at fairs and amusement parks. Imagine you are at a fair where there is a life-sized Ronald McDonald statue. Next to the statue, there are scale drawings that show the front, side, and top views of Ronald McDonald, along with a scale that represents 5 feet. Use these scales and drawings to approximate the actual dimensions of the statue:
- The height of the Ronald McDonald statue, to the nearest foot:
- The width of the statue's arms when spread out, to the nearest foot:
- The length of the statue from head to toe, to the nearest foot:
- The height of the statue and the length from head to toe should be the same
- There are still no drawings actually provided
- Current Agent Score: 1.00, compared to the threshold 0.8

Figure 4: An example of a problem recorded in this document. Each iteration of the problem is in blue. Each agent response is in red. Personal notes are in black.

We also set up a series of spreadsheet formulas to make the collected data easier to use. The formulas import the problem ID from the form responses, searching the list of known problems and importing the original text. Then the formulas get the new text that was submitted through the forum and compares the character lengths of the two, calculating a verbosity ratio

using $\frac{New\ Problem\ Word\ Count}{Original\ Problem\ Word\ Count}$ and the average sentence length ratio using $\frac{\text{New Problem Average Sentence Length}}{\text{Original Problem Average Sentence Length}}$. Finally, it determines if the generated problem is a false positive (Verbosity Ratio above 1.25 or an Average Sentence Length Ratio above 1.2). These formulas automatically update the spreadsheet with the latest information gathered from the form, displaying the current summary statistics (as shown in Figure 5).

Summary Stats	Mean	Standard Deviation
Difference of New Character Length	47.71428571	60.54963121
Difference of New Word Length	5.761904762	12.40340143
Verbosity Ratio	1.210988824	0.364214689
Sentence Length Ratio	1.173775749	0.392785729
Probability of False Positive for reading agent	0.404761905	

Figure 5: Example of the Summary stats from the PPB Bug Reporting Forum/Spreadsheet

Chapter 3

Reinforcement Learning Services

3.1 Background

When struggling to solve a problem, ASSISTments can provide the student with various forms of help, such as hints or worked examples. ASSISTments refers to these support options as tutor strategies, with many problems offering more than one. Reinforcement Learning Services (RLS) is a small service under the ASSISTments umbrella that decides which tutor strategy a student sees for a particular problem. Instead of picking a particular strategy for a student once, RLS is dynamic, picking an initial strategy, observing the student's success, and using that feedback to get better at the selection over time.

3.2 Technical Background

Prior to our project, RLS was a Python service built around the Flask web framework, with a PostgreSQL backend, and Amazon Web Services (AWS) hosting. The foundation of RLS is comprised of three models: a random model, yielding a uniformly random pick, LinUCB (Li et al., 2010), a classic linear contextual bandit, and DLEG, a dynamic linear model using epsilon-greedy. This service handles the selection of a tutor strategy through a four-step pipeline. First, the recommendation request is received and logged, and the problem identifier is resolved, granting access to the problem content. Next, one of the available models is picked at random, allowing ASSISTments to simultaneously test the effectiveness of various models and provide quality feedback to students. Then, that model's saved state is loaded, along with context features describing the student, the problem, and each possible tutor strategy. Finally, the model returns a recommendation, and the relevant metadata associated is logged.

3.3 Improvements

Although RLS was a structurally sound repository prior to this project, the team wanted to consider foundational and architectural improvements that would enhance RLS's ability to recommend the optimal tutor strategy for a given student. The RLS repository could realize significant improvements, including concurrency gains by migrating from Flask to FastAPI; security benefits from patching vulnerabilities in the deployment image; and direct improvement in model recommendations through an analysis of the current reinforcement learning literature.

3.4 Implementation and Results

3.4.1 Architecture Shift from Flask to FastAPI

The migration to FastAPI was applied to both the RLS and QC services. The following describes the changes made (The application of these changes to QC is specifically addressed in the QC section). The stability of these services is fundamentally important to the security and reliability of ASSISTments. Since these services transitioned to Python, that foundation has been Flask.

Flask is a web framework used to build web applications and APIs that follow the Web Server Gateway Interface (WSGI) standard. The WSGI standard specifies a synchronous, one-request-per-callable interface between the Python web application and its server. Consequently, on each I/O call, the calling worker is blocked until the function returns. This presents a massive gap in efficiency, especially for RLS and QC, which spend a large portion of each request waiting on external processes to complete. While Flask 3.x introduced async functionality, its own documentation notes that it "will not be more performant than async-first frameworks."

After searching for a more efficient architecture, we decided on FastAPI. FastAPI is a modern web framework built on Starlette and Pydantic, following the Asynchronous Server Gateway Interface (ASGI) standard (the successor to WSGI). The FastAPI documentation frames the benefit of ASGI in explicit I/O terms: while one request waits for a database query, the server can process other requests. In contrast, a synchronous handler like Flask leaves the server idle during these I/O waits (FastAPI, n.d.).

Both QC and RLS are REST surfaces that rely on relatively slow external dependencies: Postgres for RLS, and a combination of Postgres and OpenAI for QC. Under the Flask framework, each Gunicorn worker is held for the duration of these waits. However, under ASGI

and FastAPI, workers can interleave outstanding I/O-bound requests.

Beyond concurrency, the shift to FastAPI provides significant secondary benefits, such as Pydantic validation. Previously, Flask required calling request.get_json(force=True) followed by a manual call to check_tutor_strategy_request(request_json) to validate fields. The FastAPI implementation instead declares a class with typed fields and takes the input as a parameter, automatically rejecting malformed requests with a 422 error. By leveraging this core feature, we can retire custom validators, minimize the attack surface, and reduce the amount of code required for maintenance.

Ultimately, the combination of Starlette and Pydantic provides a low-risk, high-reward profile that serves as a safe hedge against long-term technical debt. In short, while our deployment process remained largely unchanged, the ceiling on concurrency and overall efficiency has been significantly improved.

3.4.2 Model Classes and Bug Fixing

Beyond this migration, we also reworked the structure of the code surrounding the models. Previously, RLS had a separate class for each model. While functional, this setup is not optimal. Therefore, we refactored the models to use a class inheritance system, with the parent class containing all model-agnostic code. The models were changed to inherit the parent class, adding any model-specific code to the individual models. This improves the infrastructure by allowing you to tweak universal components once instead of requiring changes be made to each model individually.

We also discovered a bug that was preventing the models from running their training update process. The issue was triggered by calling for more than 10,000

assignment_xref/user_xref pairs in a single database call. We were able to resolve this by reworking the database call into several batched database calls. We encountered another bug where the batching system had an infinite loop. The resolution was to tweak the batching system to account for already processed pairs by removing them from the list of pairs awaiting processing. Once this was complete, updating the models was once again functioning.

3.4.3 ECR Vulnerability Remediation

The vulnerability remediation was applied to both RLS and QC. The following describes the changes made (The application of these changes to QC is specifically addressed in the QC section). RLS and QC run as containerized Python applications stored in Amazon's Elastic Container Registry (ECR). Both images are built on the python:3.X-slim-trixie Debian base image. "Trixie" is the codename for Debian 13; the "slim" variant is stripped of heavy package manager caches and documentation, resulting in a much smaller runtime image. Because system libraries still exist within slim images, they remain susceptible to OS-level Common Vulnerabilities and Exposures (CVEs) that continually surface. To identify these CVEs, ASSISTments uses Amazon's Basic ECR Scanning, powered by Clair, an open-source vulnerability database. Scans occur in two instances:

- Scan-on-push — Triggered each time a new image tag is pushed to the repository.
- Manual or periodic re-scans — Triggered manually, though re-scans occur at most every 24 hours.

During our time at ASSISTments, the RLS and QC repositories accrued dozens of vulnerabilities, including several rated as "Critical" or "High" severity. A thorough search revealed that most of these CVEs were inherent to the OS layer of the Slim Trixie base image. In other words, the vulnerabilities were not introduced by the source code of either service, but by

the contents of the underlying Debian filesystem. Most were non-issues for our codebase as the affected libraries were not actually utilized in the source code of RLS or QC.

The primary remediation strategy was to identify the most recent Slim Trixie image, update our versioning to reflect that, and utilize the scan-on-push feature to determine which CVEs remained. In both cases, bumping the base-image version resolved over 90% of the vulnerabilities. The few remaining cases were manually addressed. This was done by either building the correct version, or installing an updated version pulled from an updated package repository.

Following this work, the CVEs found for both images were reduced to zero. Container hygiene is an exceptionally high-utility form of defense for small teams. Because vulnerable OS-level libraries can produce a legitimate attack vector, the importance of maintaining this practice cannot be understated. In the future, ASSISTments could possibly move to ECR Enhanced Scanning. This would identify Python-package CVEs — which Basic Scanning misses — and provide higher-frequency re-scans, rather than the 24-hour minimums of the Basic tier.

Chapter 4

Quick Comments

4.1 Background

QC was initially the result of a Worcester Polytechnic Institute (WPI) Major Qualifying Project team and was migrated into the primary ASSISTments repository between late 2019 and early 2020. It started as a synchronous Flask application with programmatic, non-AI-driven models. As generative AI improved in mid-to-late 2024, ASSISTments introduced explicit

endpoints for GPT-4 and added backend logging to track model accuracy and IDs. Subsequently, in the summer and fall of 2025, the service adopted OpenAI's GPT-4o, integrating shorter feedback loops and revised system prompts.

4.2 Technical Background

As with RLS, Quick Comments was built around the Flask web framework with a PostgreSQL backend, and Amazon Web Services (AWS) hosting. QC handles open-response submissions in a four-step pipeline. First, the system takes in the answer text and user identifiers, validating the input by stripping away invalid data. Next, it dynamically assigns the student to a specific AI model. Then, the student's answer and the problem text are inserted into an engineered prompt template, calling the respective AI service. Once the model generates feedback and a score, the database logs the interaction and returns the information to the student's interface, completing the process.

4.3 Improvements

Quick Comments had a strong foundation, so we focused on upgrading components to further strengthen the codebase. Prior to our work, QC relied on the Flask web framework. While Flask is sufficient for many applications, it is less ideal for QC's primary operation: making requests to AI models. Consequently, we needed to explore more efficient options. Additionally, because QC is embedded in the AWS ecosystem, staying ahead of Common Vulnerabilities and Exposures (CVEs) is crucial for the continued upkeep and security of the service.

4.4 Overview

Throughout the duration of this project, two infrastructure-level changes were implemented into Quick Comments (QC): the Flask to FastAPI migration and the ECR vulnerability remediation. Both changes were applied to QC using the same approach as RLS. What follows addresses the outcomes and future work pertaining to QC specifically.

4.5 Outcomes

The primary advantage of the Flask to FastAPI migration was raising the ceiling on concurrency within QC. The primary time constraint consists of waits on two slower external dependencies: Postgres and OpenAI. The previous Gunicorn workers would sit and wait for nearly the full duration of every request. As discussed in Chapter 3, the use of FastAPI allows workers to interleave outstanding I/O-bound requests, providing a substantial concurrency benefit.

4.6 Future Work

These changes to QC (specifically as they relate to FastAPI) give an opportunity for even more benefits. This is especially true from a concurrency and security standpoint. ASSISTments should aim to maximize the concurrency offerings of FastAPI by ensuring that every outbound call uses an asynchronous path. Any synchronous call within the async handler reintroduces the exact worker-blocking behavior that this conversion intended to eliminate. Also, expanding Pydantic coverage to ensure that response surfaces are consistently type-checked would improve security.

Chapter 5

Gaming Detection

5.1 Overview

Identifier	Description
[did not think before help request]	Pause smaller or equal to 5 seconds before a help request
[thought before help request]	Pause greater or equal to 6 seconds before a help request
[read help messages]	Pause greater or equal to 9 seconds per help message after a help request
[scanning help messages]	Pause between 4 and 8 seconds per help message after a help request
[searching for bottom-out hint]	Pause smaller or equal to 3 seconds per help message after a help request
[thought before attempt]	Pause greater or equal to 6 seconds before step attempt
[planned ahead]	Last action was a correct step attempt with a pause greater or equal to 11 seconds
[guess]	Pause smaller or equal to 5 seconds before step attempt
[unsuccessful but sincere attempt]	Pause greater than or equal to 6 seconds before a bug
[guessing with values from problem]	Pause smaller than or equal to 5 seconds before a bug
[read error message]	Pause greater than or equal to 9 seconds after a bug
[did not read error message]	Pause smaller than or equal to 8 seconds after a bug
[thought about error]	Pause greater than or equal to 6 seconds after an incorrect step attempt
[same answer/diff. context]	Answer was the same as the previous action, but in a different context
[similar answer]	Answer was similar to the previous action (Levenshtein distance of 1 or 2)
[switched context before right]	Context of the current action is not the same as the context for the previous (incorrect) action (referred to as "soft underbelly" in Baker, Mitrovic, & Mathews 2010)
[same context]	Context of the current action is the same as the previous action
[repeated step]	Answer and context are the same as the previous action
[diff. answer AND/OR diff. context]	Answer or context is not the same as the previous action

Figure 6: List of the interpretations considered by the coder. Time thresholds are selected by the expert coder. (Paquette et al., 2014)

Working with one of the graduate students, we developed a new service — a gaming detector. Gaming, in this context, refers to the process of abusing a system designed to help you learn in order to simply achieve a correct answer. Several different parameters are used in the gaming detection. These parameters were based on a paper by Luc Paquette, Adriana Carvalho,

and Ryan Baker — *Towards Understanding Expert Coding of Student Disengagement in Online Learning*. The specific criteria we used were pulled directly from their paper and are shown in Figure 6. These criteria were used as the basis of our python-based gaming detector. This detector reviews ASSISTments assignment logs, flagging various gaming and non-gaming behaviors.

5.2 Method

We took the above table of events, narrowing it down for our purposes, and deciding on a mix of positive and negative behaviors to look for. These events were:

- [did not think before help request]
- [thought before help request]
- [read help messages]
- [scanning help messages]
- [searching for bottom-out hint]
- [thought before attempt]
- [planned ahead]
- [guess]
- [unsuccessful but sincere attempt]
- [similar answer]
- [repeated step]

Once we settled on which of the parameters we wanted, we created a python tool that would ingest data from a series of assignments (in the form of a csv file), reading through all the logged actions to determine which flags to add to each problem. A single problem can have both

gaming and non-gaming flags. For example, if a student took their time and read the first hint and worked though the problem, then got it wrong and spammed their way through the rest of the hints, the detector would flag that question as both [thought before attempt] and [searching for bottom-out hint]. In addition to logging the positive and negative behaviors, the program also logs the most gamed problem and the user that gamed the most. The detector then looks at all the questions, counting the number of questions that are flagged with gaming behavior, non-gaming behavior, or both. I have included some examples of the output from the data that we were working with (Figures 7 and 8 in Section 5.4).

5.3 Technical Process

The program starts by loading the CSV containing the data, reading it row by row, and grouping the actions that belong to a single problem attempt. These groupings are stored as an array of custom class objects. Inside this class is stored the assignment id, problem id, student id, an array of actions (a separate custom class), gaming detected (defaults to false), non-gaming detected (defaults to false), and an array of behavior types (stored as strings, defaults to an empty array). The action class contains the action type, timestamp of the action, time since the last action for the problem was logged (in seconds), and the answer the student submitted (if the action was “IncorrectResponse” or “CorrectRepsonse”). The program then processes the problems, swapping the gaming and non-gaming flags to true if it detects a behavior of the respective type. It will also add the type of behavior to the behavior types array upon detection. This process will occasionally discard a problem if it cannot find the “ProblemFinishedAction”. When this occurs, it is added to a counter that displays the number of discarded problems.

5.4 Outcomes

ProblemID = PRBTXUP, StudentID = bccb09c5-5a51-4e4a-b13d-d9d06f68e9bd, Gaming Detected: False, Non-Gaming Detected: True, Behavior Types: ['ThoughtAttempt', 'SincereAttempt', 'ThoughtAttempt', 'PlannedAhead']
Action: ProblemStartedAction, Timestamp: 2025-12-05 15:32:18.819000+00:00, Time Since Last Action: 0.0
Action: IncorrectResponse, Timestamp: 2025-12-05 15:32:48.206000+00:00, Time Since Last Action: 29.38700008392334, Answer: [["36"]']
Action: CorrectRepsonse, Timestamp: 2025-12-05 15:35:55.694000+00:00, Time Since Last Action: 187.48799991607666, Answer: [["56.1"]']
Action: ProblemFinishedAction, Timestamp: 2025-12-05 15:35:55.696000+00:00, Time Since Last Action: 0.002000093460083008
…

Student Summaries:
23aa0d9c-ceb6-4757-9440-8b9381085ed8 gamed 1 time(s).
…

Problem Summaries:
PRBTXUP was gamed 13 time(s).
…

SUMMARY:
Rejected: 161, Accepted: 2300, Rejection Rate: 6.54%

Gaming was detected in 175 out of 2300 problems. That is a rate of 7.61%
Out of 2186 problems with non-gaming behavior, 0 had "[thought before help request]", 0 had "[read help messages]", 425 had "[thought before attempt]", 62 had "[planned ahead]", and 343 had "[unsuccessful but sincere attempt]"
Out of 175 problems with gaming behavior, 0 had "[did not think before help request]", 0 had "[scanning help messages]", 44 had "[searching for bottom-out hint]", 315 had "[guess]", 0 had "[similar answer]", and 0 had "[repeated step]".
150 problems had a mix of both gaming and non-gaming behaviors.
95 students gamed, with b4be03af-12c4-4fb9-ac45-d04a7441c36a gaming the most (14 instances each).
82 problems were gamed, with PRBTXUP being the most gamed problem(s) (13 instances each).

Figure 7: Example of the text file output

Assignment Log ID	Problem ID	Student ID	Gaming Detected	Non-Gaming Detected	Behavior Types
27603212	PRBTXUP	bccb09c5-5a51-4e4a-b13d-d9d06f68e9bd	FALSE	TRUE	['ThoughtAttempt', 'SincereAttempt', 'ThoughtAttempt', 'PlannedAhead']

Figure 8: Example of the CSV file output

The detector has two modes, text output and csv output. In the text mode it will output a file with all the problems, flags for the problems, individual actions for the problems, and a summary at the end. The csv mode outputs Assignment Log ID, Problem ID, Student ID, Gaming Detected, Non-Gaming Detected, and Behavior Types for all the data processed so that it can be easily manipulated with various statistics software like R.

5.5 Future Work

There are several possible changes that would further improve the system. A graphical user interface and graphs based on the output would improve the user experience. The gaming detection system could also be integrated into RLS. Using Abubaker Siedahmed's research into gaming prevention, RLS can be used to identify gaming and make suggestions on what methods are most likely to reduce a student's gaming behaviors. RLS could also flag problems that may benefit from the PPB's personalization. If a problem is gamed more frequently than other problems, there may be elements within the problem that are causing students to game the system. By using the PPB to make the problem more interesting for the student, a drop in gaming might be possible.

Chapter 6

Conclusion

6.1 Production Readiness

Although the Selenium Test Suite and Microsoft Form remain available to use, Persona Problem Builder itself remains in production, and many of the issues we identified have not yet been resolved. RLS and QC have been active since before this project. Some of our changes, mostly those related to security, are already live. Meanwhile the remaining changes are awaiting a code review before they will be released to the live build. As for Gaming Detection, the tool is created and fully functional. While there are improvements that can be made, the data it produces is already proving useful.

6.2 Summary of Contributions

Although our work was split between multiple areas, we were able to make progress in each of them. With PPB, we were able to identify numerous problems, gather data, and create a way to quickly generate multiple test problems by using the Selenium Test Suite. RLS and QC also saw many important improvements. RLS had all critical vulnerabilities eliminated, as well as receiving major structural updates. QC's latency was reduced by switching from Flask to FastAPI. The Gaming Detection tooling was created to enable further research into student behavior on the platform. This research could be used to encourage learning and discourage simply speeding through answers. Future work on both Persona Problem Builder and Gaming Detection can help to ensure students feel engaged with what they are learning and do not try to rush through without understanding.

References

AutoHotkey Foundation LLC. (n.d.). *AutoHotkey*. Retrieved May 6, 2026, from https://www.autohotkey.com/

Baker, R. S. J. d., Mitrovic, A., Mathews, M. (2010). Detecting Gaming the System in Constraint-Based Tutors. Proc of UMAP 2010, 267-278.

FastAPI. (n.d.). Concurrency and async / await. Retrieved May 6, 2026, from https://fastapi.tiangolo.com/async/

Li, L., Chu, W., Langford, J., & Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. In Proceedings of the 19th International Conference on World Wide Web (pp. 661–670). ACM. https://doi.org/10.1145/1772690.1772758

Pallets. (n.d.). *Using async and await*. Flask Documentation (3.1.x). Retrieved May 6, 2026, from https://flask.palletsprojects.com/en/stable/async-await/

Paquette, L., Carvalho, A. M., & Baker, R. S. (2014). *Towards Understanding Expert Coding of Student Disengagement in Online.* From upenn.edu: https://learninganalytics.upenn.edu/ryanbaker/Luc%20Paquette%20-%20CogSci%202014.pdf

Selenium. (n.d.). WebDriver. Selenium Documentation. Retrieved May 11, 2026, from https://www.selenium.dev/documentation/webdriver/