Departments of Math and Computer Science

Final Report for
MICROSOFT

Developing an LLM-Powered Application for Defect Resolution in Azure DevOps

May 9, 2025

Team Members
Weston Crewe (Fall Team Lead)
Tejas Hegde
Ian Li
Pearson Mewbourne (Spring Team Lead)
Kishore Rajesh
Mark Ying

Advisor
Prof. Melissa O'Neill

Liaisons
Topper Kain '07
Henry Mbogu

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 Problem Statement . . . 6
- 1.3 Literature Review . . . 6
- 1.4 Conclusion . . . 7

2 Technical Overview **9**
- 2.1 Large Language Model Components . . . 9
  - 2.1.1 Image Summarizer . . . 10
  - 2.1.2 Azure AI Search . . . 10
  - 2.1.3 Similarity Checker . . . 12
- 2.2 Data Engineering . . . 14
  - 2.2.1 ETL Automation . . . 14
  - 2.2.2 Azure SQL Database . . . 14
- 2.3 PromptFlow . . . 15
- 2.4 Conclusion . . . 17

3 User Interfaces **19**
- 3.1 Web UI . . . 19
- 3.2 PromptFlow UI . . . 20
- 3.3 Conclusion . . . 20

4 Testing and Final Results **23**
- 4.1 Testing Methodology . . . 23
- 4.2 Accuracy Results . . . 24
- 4.3 Speed Results . . . 25
- 4.4 Conclusion . . . 25

5 Conclusion and Future Work **27**
- 5.1 Future Work . . . 28

Bibliography **31**

Chapter 1

Introduction

Microsoft is a global technology corporation that operates Azure, the world's largest computing cloud by revenue. Engineers in the Azure Hardware team are responsible for addressing defects across all hardware systems currently in use or under development. These engineers rely on the Azure DevOps (ADO) platform as their defect resolution management system. Through ADO, engineers document technological defects in detail, upload the information to a shared environment, and collaboratively track solution steps.

1.1 Motivation

When diagnosing and fixing defects, engineers frequently consult internal DevOps systems to find similar issues that have been previously reported. In addition, they might not even consider the possibility that someone has reported a similar bug. Access to near-duplicate bugs provides engineers with several key advantages: more comprehensive information about the issue, opportunities to collaborate with other engineers who have faced similar problems, and potential solution pathways if the previous bug has already been resolved. While this process can significantly streamline debugging efforts, locating truly similar defects across a large DevOps database is often extremely time-consuming and inefficient.

1.2 Problem Statement

Our goal was to develop software for the Microsoft Azure Cloud Hardware Analytics and Tools (CHAT) team that leverages available bug report information to reduce debugging time and effort, increase engineer productivity, and ultimately improve customer satisfaction.

To achieve this goal, we designed and implemented an ETL (Extract, Transform, Load) pipeline that transfers bug data from ADO into an Azure SQL database with a searchable index. As part of this process, we applied a GPT-4o-mini model to summarize images embedded in bug reports, ensuring that all bugs are fully contextualized within the database. Once an engineer's defect and the searchable index are loaded into the system, we use a PromptFlow tool to configure inputs into two LLM resources. First, an Azure AI Search resource enables both keyword and semantic retrieval across the database. Then, a second GPT model functions as a "similarity checker" to review the search results and validate that the retrieved bugs are truly similar to the engineer's current problem.

1.3 Literature Review

To develop a deeper understanding of the problem space and inform our solution design, we reviewed existing research on automatic bug classification and triaging using various machine learning methods. Recent advancements in natural language processing (NLP) have demonstrated the value of transformer-based models and Large Language Models (LLMs) in enhancing these processes (Plein and Bissyandé, 2023).

Transformer models like BERT (Bidirectional Encoder Representations from Transformers) have been fine-tuned for tasks such as developer and component assignment using bug report descriptions, though traditional methods like TF-IDF (Term Frequency-Inverse Document Frequency) with SVM (Support Vector Machine) sometimes outperform them for specific datasets (Dipongkor and Moran, 2023). Research has shown that augmenting bug reports with manually annotated "intentions" (e.g., explanation or suggestion) has further improved classification performance, with significant potential for automating this annotation process using LLMs for scalability as demonstrated by (Meng et al., 2022).

Additionally, duplicate bug report detection frameworks such as CUPID (leveraging ***C***hatGPT for d***upli***cate bug report ***d***etection) utilize LLMs in a zero-shot setting to extract critical information for duplicate detection, achieving moderate accuracy improvements despite resource challenges in comparing large bug report pairs (Zhang et al., 2024). Beyond classification and detection, LLMs have shown success in bug report summarization and evaluation. Models like SumLLaMA address contextual and overfitting issues using contrastive learning and adapter-based fine-tuning, significantly reducing parameter counts while maintaining efficiency (Xiang and Shao, 2024).

Moreover, LLMs have proven to be effective automated evaluators of bug report summaries, offering scalability and consistency advantages over human reviewers for large datasets (Kumar et al., 2024). This body of research provided valuable insights that guided our approach to developing an efficient bug similarity detection system.

1.4 Conclusion

In this chapter, we have introduced the problem of efficiently identifying similar bug reports within Microsoft Azure's extensive DevOps system. We have established the motivations for automating this process, presented our solution approach using LLM technologies, and reviewed relevant literature that informed our design decisions. In the next chapter, we will provide a detailed technical overview of our system, exploring the various components and technologies that power our bug similarity detection tool.

Chapter 2

Technical Overview

In this chapter, we present a comprehensive examination of the technologies and methodologies employed in our LLM-powered application for defect resolution. We begin by discussing the Large Language Model components that form the core of our system, followed by the data engineering principles that enable efficient data processing and storage. Finally, we describe the integration framework that brings these elements together into a cohesive solution.

2.1 Large Language Model Components

Our tool incorporates Large Language Model (LLM) capabilities across multiple components to handle the diverse nature of human-written bug reports and ensure response validity. Microsoft Azure offers a rich ecosystem of AI models and endpoints, allowing us to leverage OpenAI models without exposing Microsoft's proprietary data to external organizations. This integration of secure, internal AI resources enabled us to incorporate LLM functionality into core aspects of our tool, generating valuable insights from bug reports.

2.1.1 Image Summarizer

In many bug reports, critical information is contained not only in written text but also in attached screenshots. For our tool to retrieve the most relevant bugs, it needed to consider image content within bug reports. However, this presented a challenge: within our data, images were represented as URLs that lacked semantic meaning, providing no benefit for matching similar bugs.

To overcome this limitation, our team developed an Image Summarizer component. The Azure OpenAI endpoint, a Microsoft service that allows us to securely access OpenAI's Large Language Models, allowed us to seamlessly incorporate and customize LLM capabilities for image summarization. As part of our data processing pipeline:

1. Each image is downloaded from its URL
2. Both the image and the full bug report are passed to the LLM
3. The LLM is prompted to provide a relevant description of the image within the context of the bug report
4. The generated summary replaces the original URL in the bug report

Through this process, the bug report is transformed to consist entirely of text elements, enabling effective semantic comparison against other bugs. This approach ensures that critical visual information is preserved in a format that can be processed by our similarity detection algorithms.

2.1.2 Azure AI Search

One of the key resources used during this project is Azure AI Search. This is an information retrieval system with integration into Azure OpenAI and Machine Learning services. Azure Machine Learning allows users to quickly and easily create, manage, and deploy their machine-learning projects end-to-end. Azure AI Search specifically gives users the ability to search through large amounts of data quickly, returning accurate results using keyword, semantic, and vector-based searches. Keyword search finds results containing the exact words or phrases in the query. Semantic search uses the meaning of the query (such as synonyms) to find relevant results,

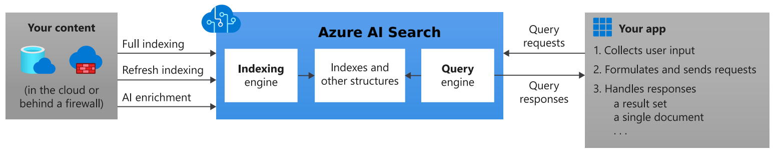


Figure 2.1 Azure AI Search Pipeline

even if none of the words exactly match. Vector search turns queries into vectors and ranks them by similarity using metrics such as cosine distance. Azure AI Search allows to combine all of these into the same search, allowing for powerful and accurate results. A more comprehensive diagram can be seen in Figure 2.1[1].

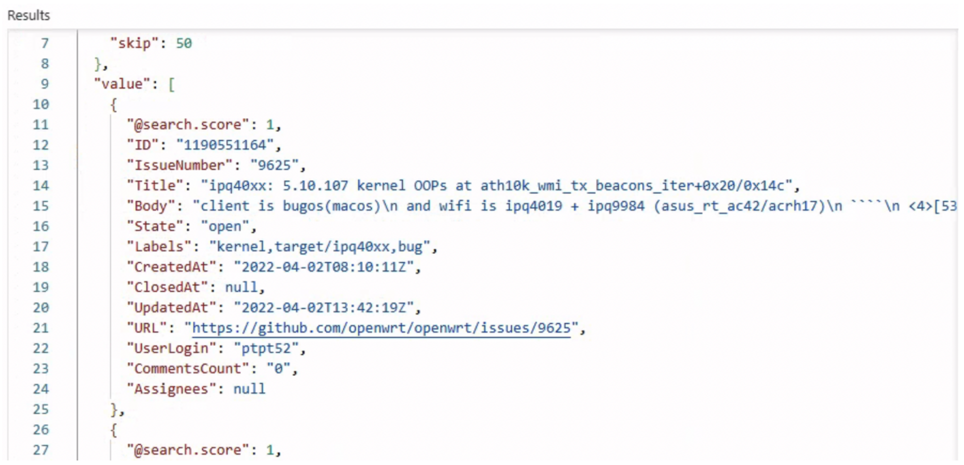
There are three main components one needs to set up in order to use Azure AI Search:

- **Index:** The index serves as the central repository for our structured data collection. Its structure most closely resembles a JSON, but uploading other file types to this index is possible (such as a CSV). This is linked to some form of data storage (such as a Blob or Azure SQL Database). Our initial versions of the tool utilize Blob Storage[2] for easy file upload and deletions, however, later iterations utilize the Azure SQL Database (discussed later) as a data sink that can be automatically updated. A screenshot of the index's interface is shown in Figure 2.2.
- **Indexer:** This component essentially takes the raw data (such as a CSV or JSON file) and serves as the "machine" that converts the data into structured data within our index. From your data source, you can choose which fields you want to be inside your dataset, which is helpful if there is unnecessary/extra information that is not needed inside your search. In our case, we use ID, WorkItemType, Title, AssignedTo, State, Tags, ReproSteps, Description, and ClosedBy for the Microsoft dataset.

  An indexer can be rerun on our data at chosen intervals to update the index with new data. This is relevant because new bugs at Microsoft

[1] https://learn.microsoft.com/en-us/azure/search/search-what-is-azure-search

[2] *Blob Storage*: Service optimized specifically for storing large amounts of unstructured data, such as text or binary files.



```
    "skip": 50
  },
  "value": [
    {
      "@search.score": 1,
      "ID": "1190551164",
      "IssueNumber": "9625",
      "Title": "ipq40xx: 5.10.107 kernel OOPs at ath10k_wmi_tx_beacons_iter+0x20/0x14c",
      "Body": "client is bugos(macos)\n and wifi is ipq4019 + ipq9984 (asus_rt_ac42/acrh17)\n ````\n <4>[53
      "State": "open",
      "Labels": "kernel,target/ipq40xx,bug",
      "CreatedAt": "2022-04-02T08:10:11Z",
      "ClosedAt": null,
      "UpdatedAt": "2022-04-02T13:42:19Z",
      "URL": "https://github.com/openwrt/openwrt/issues/9625",
      "UserLogin": "ptpt52",
      "CommentsCount": "0",
      "Assignees": null
    },
    {
      "@search.score": 1,
```

Figure 2.2 Filled Index with OpenWRT GitHub repository bugs.

are created every day, and our bug searcher will be the most helpful if it has access to the most recent bug data. The indexer can also choose to ignore certain file types with a data source, so random logs can be ignored if they are not relevant to the index you are trying to create.

- **Skillset:** The final component of setting up Azure AI Search is the set of additional skills that we want our Indexer to have when inserting the data into our index. In our case, we added Text Embeddings and Key Phrase Extraction, however, future teams can further explore what other skills may be useful for this component of Azure AI Search.

There are two options to set up these three different components. The first is using the user interface on Azure AI, while the other is using code. Both will yield the same results.

2.1.3 Similarity Checker

Our goal was to ensure that our tool consistently provides relevant results. We wanted to avoid making suggestions that would be ignored by engineers, focusing instead on delivering valuable insights for every query.

We recognized that while AI search tools might retrieve results with high semantic similarity scores, these results might still appear uncorrelated to

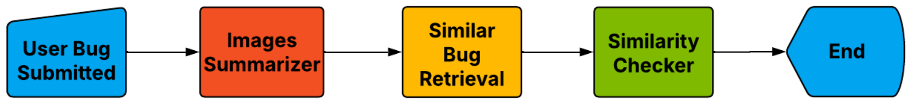


Figure 2.3 High-Level Technological Flowchart

a human observer. Therefore, our tool could not rely solely on AI search to produce results.

The additional similarity check is necessary because getting "the five most similar other bugs" is not the same as "getting five bugs that are actually relevant to the user." Something might be the most similar bug in the system but still not really that useful to the engineer working on the current issue.

To ensure high-quality recommendations, we designed a *Similarity Checker* component that validates the output from our AI search. The process works as follows:

1. The top five bugs identified by the AI search, along with the original bug, are sent to an LLM via the Azure OpenAI endpoint
2. The LLM is prompted to evaluate the true similarity between these bugs based on technical relevance
3. Any bugs that do not demonstrate genuine similarity are filtered out from the final results
4. The remaining bugs are presented to the user, ensuring that only truly relevant matches are recommended

This validation step significantly improves the precision of our recommendations, helping engineers focus only on bugs that have meaningful connections to their current issue. Future enhancements to this component could include highlighting exactly how each matched bug is similar to the current one, providing engineers with immediate context for why a particular match was suggested.

2.2 Data Engineering

Our system treats Microsoft bug reports as data instances, represented as *Work Items* in Azure DevOps (ADO). A Work Item is the representation of an assigned task within ADO, and has multiple different variations, with bug report being the variation our team focused on. Using ADO as our data source, we developed an automated pipeline to extract, transform, and load this data into a destination database that integrates with our Similar Bug Retriever.

2.2.1 ETL Automation

We implemented a comprehensive Extract, Transform, Load (ETL) pipeline with the following components:

1. **Data Extraction:** We utilized the Open Data Protocol (OData) to create a RESTful API for extracting data from ADO. We focused on nine critical data fields for each bug: ID, WorkItemType, Title, AssignedTo, State, Tags, ReproSteps, Description, and ClosedBy. Once collected, this initial dataset is stored in a *Blob Storage* container. .
2. **Data Transformation:** We processed the Blob dataset using our Image Summarizer component (detailed in Section 2.1.1), which converts image URLs into textual descriptions. The transformed dataset is then uploaded to a new container for further processing.

2.2.2 Azure SQL Database

The Similar Bug Retriever requires an *indexer* to make the source data searchable. Structuring the data in an *Azure SQL Database* enables flexible indexing, which was essential for our project as we needed to customize indexing differently for various data fields. For example, we needed to ensure that our search model would not interpret bugs with sequential ID numbers as similar based solely on this numerical proximity.

To migrate our data from Blob storage to SQL, we created a dataflow process using *Azure Data Factory*. This process:

1. Structures the Blob objects as a unified dataframe
2. Transfers the dataframe into SQL as a table with appropriate schema
3. Configures the necessary indices for efficient searching

Recognizing the importance of maintaining an up-to-date dataset, we designed both ends of our ETL pipeline to run on daily triggers. First, the extraction from ADO and image preprocessing run to update the Blob container, which then triggers the dataflow from Blob to SQL to refresh the destination database. This automation ensures that our system continuously incorporates new bug reports, maintaining its relevance and effectiveness.

2.3 PromptFlow

The PromptFlow tools available within Azure AI Foundry significantly simplified the integration of different components in our system. PromptFlow is a Microsoft Azure development tool that facilitates the creation of AI-powered projects by providing an intuitive way to connect various components.

Within the PromptFlow workstation, users can create nodes representing:

- Code execution blocks
- LLM endpoint calls
- Data transformation operations
- And other functional components

These nodes can be connected so that the output of one serves as the input to another, enabling complex workflows such as having an LLM create input for a Python function, whose output is then processed by another LLM. This approach centralizes complex AI-powered projects in a single location with intuitive visual elements that enhance design clarity, as illustrated in Figure 2.4.

Given the intricate interweaving of code and LLMs in our tool, PromptFlow proved invaluable. Our entire similar bug retrieval pipeline was implemented within a PromptFlow environment. When users describe their bug report, this information is matched with similar bugs from our historical

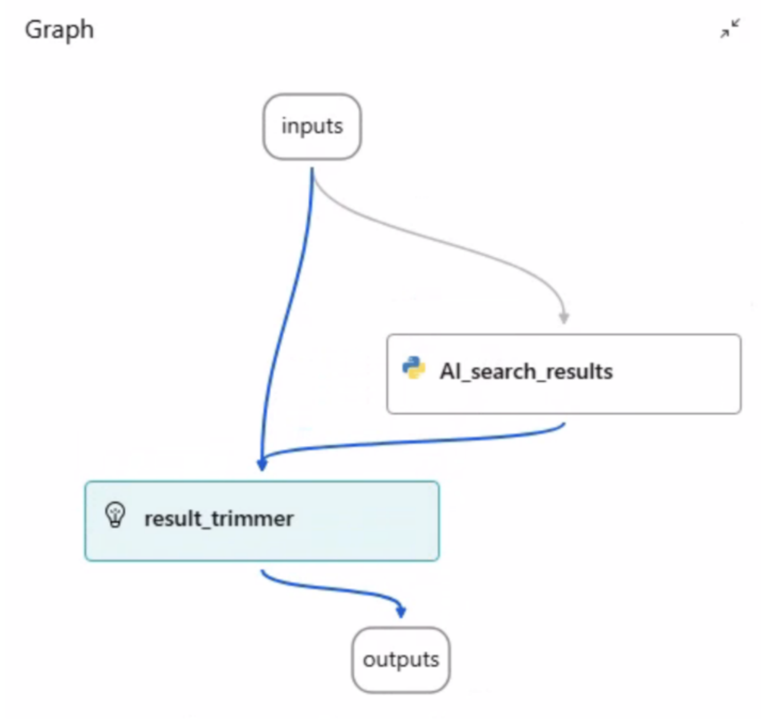


Figure 2.4 Simple PromptFlow Graph Example

database using Python code. The retrieved similar bugs are then evaluated by the similarity checker LLM. PromptFlow simplified this process significantly, minimizing the amount of custom code required while maintaining robust functionality.

This centralized development approach will also facilitate future enhancement and maintenance of the system, as new team members can quickly understand the logical flow of operations through the visual interface.

2.4 Conclusion

In this chapter, we explored the technical architecture of our bug similarity detection system. We detailed how LLM components are used to process textual and visual information from bug reports, explained our data engineering approach for collecting and transforming this information, and illustrated how PromptFlow brings these elements together into a unified workflow. The architecture we've designed ensures that our system delivers both accurate and relevant results to users. In the next chapter, we will present the testing methodology and results that validate the effectiveness of our approach.

Chapter 3

User Interfaces

This chapter describes the user interfaces developed for our bug similarity detection system. We detail both the initial web-based prototype and the integrated PromptFlow interface that serves as the primary means for engineers to interact with our tool.

3.1 Web UI

The initial prototype of our tool featured a web interface that allows users to input relevant information for a bug report or similar work item, as shown in Figure 3.1. The form includes the following fields:

- **Work Item Type:** Specifies the category of the work item (e.g., bug, task, feature)
- **Title:** A short, descriptive summary of the issue or task
- **Assigned To:** The person responsible for handling the work item
- **Tags:** Keywords or labels to categorize or filter the item
- **Repro Steps:** Detailed steps to reproduce the issue
- **Description:** Additional context or background about the issue
- **Closed By:** The person who resolved or closed the work item
- **URL:** A link to relevant documentation, resources, or issue trackers
- **Filepath:** The path to any related files or code involved

After completing the form, users submit it to trigger a search within the bug report database. The system leverages Azure AI Search to identify and return the most similar past bugs and their relevant information based on the input provided.

3.2 PromptFlow UI

To fully integrate our tool with the similarity checker component, we migrated our interface from the initial Web UI to Azure AI's native PromptFlow tool. Similar to the Web UI, users simply input information about their newly encountered bug into the PromptFlow chat interface, as depicted in Figure 3.2. Following the same process as before, the system queries Azure AI Search, and the returned results are verified by the similarity checker to ensure relevance.

This integration streamlines the user experience while maintaining the robust functionality of our system. The PromptFlow interface provides a more cohesive environment that better aligns with Azure's ecosystem of development tools. Figure 3.3 shows how the system presents the search results to the user, displaying relevant similar bugs that have been validated by the similarity checker.

3.3 Conclusion

In this chapter, we presented the user interfaces that enable engineers to interact with our bug similarity detection system. Starting with a traditional web form interface, we evolved to a more integrated PromptFlow experience that leverages native Azure AI capabilities. Both interfaces provide intuitive means for engineers to submit bug information and receive relevant similar bugs that have been validated by our similarity checker component. These interfaces were designed with simplicity and workflow integration in mind, ensuring that engineers can easily incorporate our tool into their existing debugging processes.

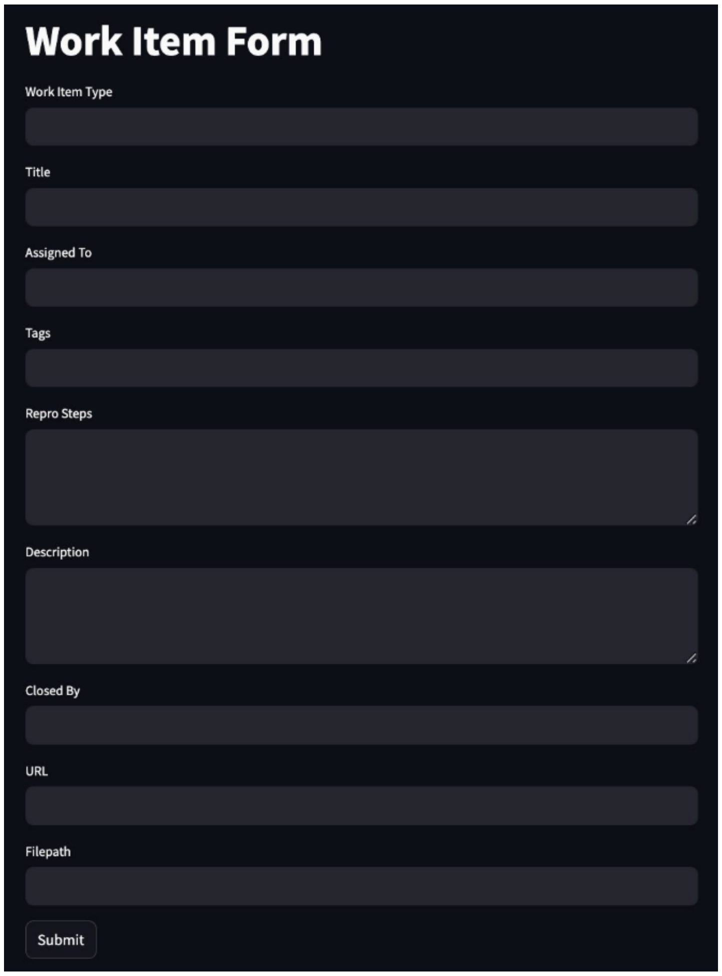
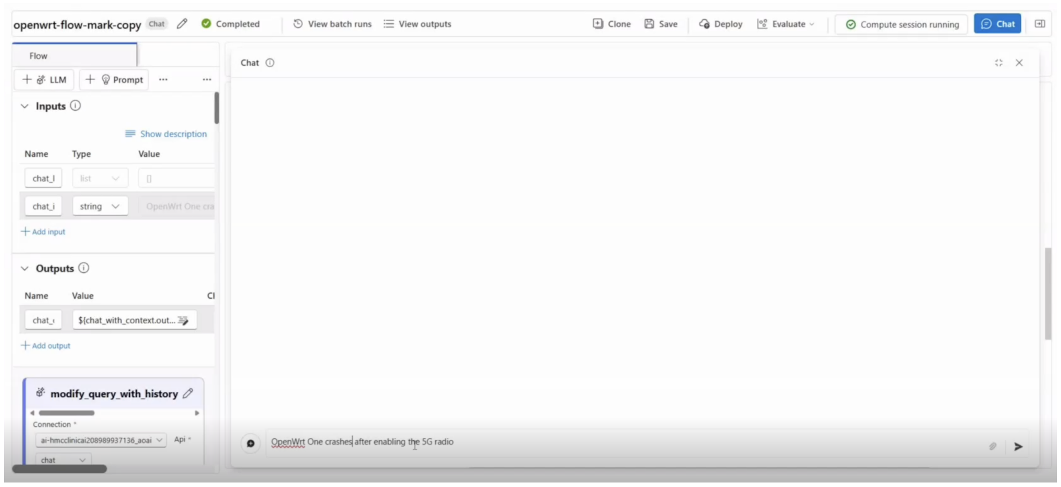
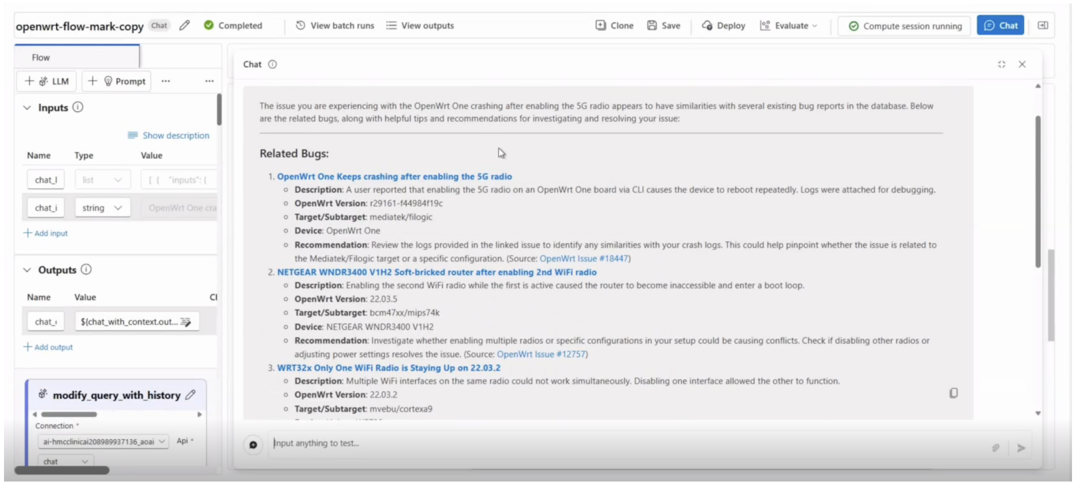


Figure 3.1 Initial Prototype: Web User Interface



OpenWrt One crashes after enabling the 5G radio

Figure 3.2 PromptFlow: User Input Interface



The issue you are experiencing with the OpenWrt One crashing after enabling the 5G radio appears to have similarities with several existing bug reports in the database. Below are the related bugs, along with helpful tips and recommendations for investigating and resolving your issue:

Related Bugs:

1. OpenWrt One Keeps crashing after enabling the 5G radio
   - **Description**: A user reported that enabling the 5G radio on an OpenWrt One board via CLI causes the device to reboot repeatedly. Logs were attached for debugging.
   - **OpenWrt Version**: r29161-f44984f19c
   - **Target/Subtarget**: mediatek/filogic
   - **Device**: OpenWrt One
   - **Recommendation**: Review the logs provided in the linked issue to identify any similarities with your crash logs. This could help pinpoint whether the issue is related to the Mediatek/Filogic target or a specific configuration. (Source: OpenWrt Issue #18447)
2. NETGEAR WNDR3400 V1H2 Soft-bricked router after enabling 2nd WiFi radio
   - **Description**: Enabling the second WiFi radio while the first is active caused the router to become inaccessible and enter a boot loop.
   - **OpenWrt Version**: 22.03.5
   - **Target/Subtarget**: bcm47xx/mips74k
   - **Device**: NETGEAR WNDR3400 V1H2
   - **Recommendation**: Investigate whether enabling multiple radios or specific configurations in your setup could be causing conflicts. Check if disabling other radios or adjusting power settings resolves the issue. (Source: OpenWrt Issue #12757)
3. WRT32x Only One WiFi Radio is Staying Up on 22.03.2
   - **Description**: Multiple WiFi interfaces on the same radio could not work simultaneously. Disabling one interface allowed the other to function.
   - **OpenWrt Version**: 22.03.2
   - **Target/Subtarget**: mvebu/cortexa9

Figure 3.3 PromptFlow: Output Display

Chapter 4

Testing and Final Results

In this chapter, we present our testing methodology and the performance results of our system. We conducted testing to evaluate the accuracy and efficiency of our LLM-powered bug similarity detection tool compared to manual processes, with results demonstrating significant improvements in both areas.

4.1 Testing Methodology

To evaluate the accuracy and performance of our pipeline, we developed a test suite to compare our tool's efficiency against a human with the same assigned task. It is important to note that testing remains a priority for any continued efforts to implement and improve this project. Our methodology involved:

1. Creating a set of test cases based on OpenWRT bugs
2. Using LLMs to generate five extremely similar variations (essentially paraphrasing) of each original bug
3. Adding both the original bugs and their similar variants to our test dataset
4. Having our AI pipeline return the five most similar bugs for each original OpenWRT bug
5. Having human evaluators (team members) perform the same task

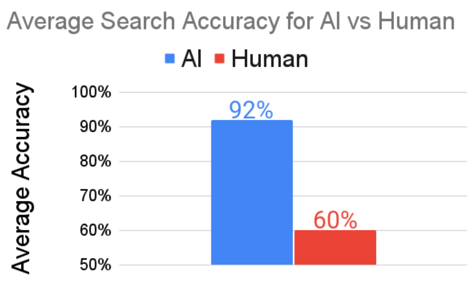


Figure 4.1 AI VS Human Accuracy Testing Results

6. Measuring and comparing both the accuracy and time required for each approach

To calculate accuracy, we determined the percentage of correctly identified similar bugs out of the total potential matches. For example, if the AI pipeline returned three similar bugs and all three were from the set of paraphrased variations, the accuracy would be 60% (3 out of 5 possible matches).

4.2 Accuracy Results

Our testing revealed that the AI pipeline achieved an average accuracy of **92%** in identifying the correct paraphrased bugs, while human evaluators achieved only **60%** accuracy. This represents a **50%** improvement in accuracy for identifying highly similar bugs. A graph of these results can be seen in Figure 4.1.

It's worth noting that during our evaluation of the pipeline's outputs, we observed that some OpenWRT bugs have inherent similarities beyond our paraphrasing exercise. In these cases, the pipeline sometimes returned bugs that, while not part of our intentionally paraphrased set, still demonstrated significant similarity to the original bug—suggesting that our system is capable of identifying meaningful relationships that weren't explicitly part of our test design.

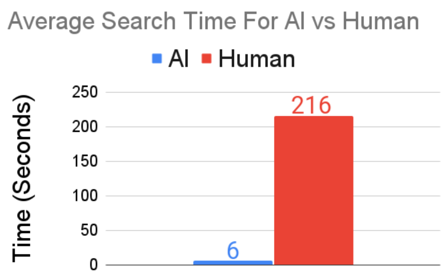


Figure 4.2 AI VS Human Speed Testing Results

4.3 Speed Results

In terms of performance efficiency, our AI pipeline completed searches in an average of **6 seconds** per query. In contrast, human evaluators required an average of **216 seconds** to identify similar bugs for the same queries. This represents a remarkable **97% reduction** in search time compared to manual searching, demonstrating the significant efficiency gains our system provides. A graph of these results can be seen in Figure 4.2.

This difference was most pronounced for bugs which had no matches within our data set. The tool could quickly determine that there were no relevant matches to return, whereas human evaluators would spend a significant portion of time conducting a futile search. This demonstrates that our system provides substantial benefits in both positive and negative match scenarios, eliminating wasted effort in cases where no similar bugs exist.

The combination of superior accuracy and dramatically improved speed validates the effectiveness of our approach and confirms the value it can deliver in real-world debugging scenarios at Microsoft.

4.4 Conclusion

In this chapter, we presented the results of our testing methodology, which demonstrated that our LLM-powered system significantly outperforms manual bug similarity detection in both accuracy and speed. The 50% improvement in accuracy and 97% reduction in search time highlight the transformative potential of our approach for streamlining bug resolution workflows. Having validated the effectiveness of our system, the next chapter will explore how users interact with the tool through its intuitive interfaces.

Chapter 5

Conclusion and Future Work

Our project has successfully delivered a scalable system that applies AI-driven data transformation, search, and validation to streamline the debugging workflow for Microsoft Azure's CHAT team. Through the integration of Large Language Models, a robust ETL pipeline, and intuitive user interfaces, we have created a solution that:

- Transforms unstructured bug report data, including images, into searchable text
- Efficiently identifies similar bugs from a large historical database
- Validates search results to ensure genuine similarity and relevance
- Reduces bug search time by 97% compared to manual processes
- Improves accuracy by 50% over human-conducted searches

The system's architecture ensures that it can continue to evolve as new bugs are reported, maintaining its relevance over time. The PromptFlow integration provides a user-friendly interface that aligns with Microsoft's existing toolchain, minimizing the learning curve for engineers.

By helping engineers quickly identify similar bugs that have been previously reported and potentially resolved, our system directly addresses the project's core objectives: reducing debugging time and effort, increasing engineer productivity, and ultimately improving customer satisfaction through faster issue resolution.

The concrete metrics from our testing demonstrate the real-world impact this tool can have on Microsoft's engineering workflow. As the system continues to learn from an expanding database of bugs, we expect its performance to further improve, providing even greater value to the organization.

5.1 Future Work

While our current implementation provides significant improvements over manual processes, there are several opportunities for future enhancements:

- **Enhanced Similarity Explanation:** Developing a component that explicitly highlights how each matched bug relates to the current issue, providing engineers with immediate context for why a particular match was suggested.
- **Advanced Visualization:** Implementing interactive visualizations that group similar bugs by their technical characteristics, enabling engineers to explore the solution space more effectively.
- **Predictive Resolution:** Extending the system to not only identify similar bugs but also suggest potential resolution approaches based on previously successful fixes.
- **Automated Integration:** Further integrating the system with Azure DevOps workflows to automatically suggest similar bugs at the time of bug creation, potentially preventing duplicate efforts before they begin.
- **More Sophisticated Testing:** Developing more comprehensive testing methodologies that evaluate the system against a wider range of real-world bug scenarios and with larger sets of human evaluators.

These enhancements would further increase the utility of our system, cementing its position as an essential tool in Microsoft's bug resolution workflow.

Acknowledgements

We would like to express our sincere gratitude to Microsoft for sponsoring this project and providing us with the opportunity to work on a real-world problem with significant impact. Special thanks to our liaisons, Topper Kain and Henry Mbogu, for their guidance, expertise, and support throughout the project.

We are deeply grateful to our faculty advisor, Professor Melissa O'Neill, for her invaluable advice, technical insights, and continuous encouragement. Her mentorship has been instrumental in the success of this project.

Finally, we would like to acknowledge the support of the Harvey Mudd College Clinic Program, which made this project possible and provided us with the resources and structure needed to succeed.

Bibliography

Dipongkor, Atish Kumar, and Kevin Moran. 2023. A comparative study of transformer-based neural text representation techniques on bug triaging. Preprint 2310.06913, arXiv.

Kumar, Abhishek, Sonia Haiduc, Partha Pratim Das, and Partha Pratim Chakrabarti. 2024. Llms as evaluators: A novel approach to evaluate bug report summarization. Preprint 2409.00630, arXiv.

Meng, Fanqi, Xuesong Wang, Jingdong Wang, and Peifang Wang. 2022. Automatic classification of bug reports based on multiple text information and reports' intention 131–147. doi:10.1007/978-3-031-10363-6_9.

Plein, Laura, and Tegawendé F. Bissyandé. 2023. Can llms demystify bug reports? Preprint 2310.06310, arXiv.

Plein, Laura, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Automatic generation of test cases based on bug reports: a feasibility study with large language models. Preprint 2310.06320, arXiv.

Xiang, Bangmeng, and Yunna Shao. 2024. Sumllama: Efficient contrastive representations and fine-tuned adapters for bug report summarization. *IEEE Access* 12:78,562–78,571. doi:10.1109/ACCESS.2024.3397326.

Zhang, Ting, Ivana Clairine Irsan, Ferdian Thung, and David Lo. 2024. Cupid: Leveraging chatgpt for more accurate duplicate bug report detection. Preprint 2308.10022, arXiv.