Proceedings of the Symposium

on

Recent Advances in Natural Language Processing

University of Colorado, Colorado Springs

August 1, 2025

Editor: Jugal Kalita

Funded by

National Science Foundation

Preface

It is with great pleasure that we present to you the papers describing the research performed by the NSF-funded Research Experience for Undergraduates (REU) students, who spent 10 weeks during the summer of 2025 at the University of Colorado, Colorado Springs. Within a very short period, the students were able to choose cutting-edge projects involving machine learning in the areas of natural language processing, and applications of natural language processing in software engineering. Additionally, they also wrote proposals; designed interesting algorithms and approaches; developed code; performed analysis; and wrote scholarly papers describing their findings. We hope that the students will continue working on these projects and submit papers to conferences and journals within the next few months. We also hope that it is the beginning of a fruitful career in research and innovation for all our participants.

We thank the National Science Foundation for funding our REU site. We also thank the University of Colorado, Colorado Springs, for providing an intellectually stimulating environment for research. We thank Dr. Terrance Boult and Dr. Armin Moin, who were mentors for the REU students. We also thank Sharon Huscher and Brittani Wagner for working out all the financial and administrative details. We thank our students and recent graduates, in particular, Ali AlShami, Targol Bakhtiarvand, Samer Iskander, Smita Khapre, Melkamu Mersha, Ryan Rabinowitz, and Joseph Worsham for helping the students with ideas and with presentations on some of the latest papers, systems, and programming issues. Our gratitude to Ginger Boult for being the "REU Mom" and having the welfare of the REU interns in her heart throughout the summer. Thanks to Ali AlShami for help with the computing infrastructure, including the GPU machines. Special thanks to Naman Chhibbar, (UCCS REU 2024, Indian Institute of Technology-Hyderabad, India; beginning MS student at University of Southern California); Harrison Gietz (UCCS REU 2023, BS University of Louisiana; Program Director at University of Oxford, UK); Clare Laney (UCCS REU 2024, George Mason University, Fairfax, VA) Michell Piehl (UCCS REU 2024, BS University of St. Thomas—Minnesota; beginning PhD student, University of Iowa), Sophie Pope (UCCS REU 2025; BS University of Wisconsin, La Crosse; beginning PhD student, University of Notre Dame, Indiana); Eric Xia, (UCCS REU 2024, undergraduate, Brown University, Rhode Island); and Joshua Zingale (UCCS REU 2023, BS San Diego University; PhD student, University of California, Riverside) for giving presentations and/or taking part in panel discussions.

Sincerely,

Jugal Kalita
jkalita@uccs.edu

August1, 2025

Table of Contents

On Detecting Code Injection Attacks in LLM-Based Multi-Agent Software Development Systems

Brian Bowers
Loyola Marymount University
1 LMU Dr. Los Angeles, CA
bbowers3@lion.lmu.edu

Smita Khapre, Jugal Kalita
University of Colorado Colorado Springs
1420 Austin Bluffs Pkwy, Colorado Springs, CO
{skhapre,jkalita}@uccs.edu

Abstract

Agentic AI and Multi-Agent Systems are increasingly being used by both industry and society for code generation. These systems are a powerful form of generative AI, marking a transition from reactive content generation into proactive multitasking capabilities. While such systems can generate code quite accurately, they are vulnerable to attacks, including code injection. Due to their autonomous design and lack of humans in the loop, these systems struggle to identify and respond to attacks by themselves. This paper analyzes the vulnerability of industry-inspired multi-agent systems and finds that the *coder-reviewer-tester* is more resilient to code injection attacks than both the *coder* and *coder-tester* architectures, but is less efficient at writing code. Following industry standards, we add a specialized *security analysis agent*, which mitigates the loss in efficiency while achieving lower attack success rates than the *coder-reviewer-tester* architecture. Despite this, we demonstrate that even with the *security analysis agent*, this approach is still vulnerable to multiple types of code injection attacks. For instance, by embedding poisonous few-shot examples in the injected code, our experiments show that the attack success rate reaches 71.95%.

1 Introduction

Agentic AI (AAI) has recently gained momentum due to its proactive, goal-driven, and autonomous properties, providing the ability to adapt and make decisions in a changing environment with minimal human intervention. AAI signals a transformative progress leading to innovation in machine-human collaboration (Murugesan, 2025). Chatbots have become commonplace, available to readily help users with queries in prescribed domains or services. Such chatbots are a product of traditional AI. A configuration of agents rises to the level of AAI when the agents are empowered with tools and resources to be able to proactively plan actions towards a specific goal. Multi-Agent Systems (MAS) build further upon AAIs. Multiple AAIs collaborate, each with a specific goal; each collaborates with one or multiple AAIs to achieve a comprehensive, common, and broader goal. Such systems are being adopted rapidly in industry, necessitating faster development and deeper research across various domains. Gartner's report, "*Top Strategic Technology Trends for 2025: Agentic AI*" (Coshow et al., 2024), predicts that by 2028, 33% of applications will incorporate some form of AAI in enterprise software, starting from a base of 1% in 2024. The report also highlights the risk to *security and quality* when AAI is incorporated in information technology stacks and warns about the development of AAI-enabled cyberattacks with *smart malware*. In this paper, we explore issues in the implementation of MAS in the software engineering process (SWEP), primarily for the implementation phase involving coding, code review, and unit testing. We analyze one of the threats presented in the threat model in the context of the proposed MAS architecture discussed below, leading to claims regarding the security and efficiency of such systems in SWEP. The main contributions of this paper are as follows.

- We propose a series of MAS architectures (*coder*, *coder-tester*, and *coder-reviewer-tester*) powered by Large Language Models (LLM) for the implementation phase in the software development life cycle guided by industry practices. We then exploit an insider threat resulting in a code injection attack and evaluate each architecture's accuracy, attack effectiveness, and architecture efficiency.
- We introduce an added security analysis agent to mitigate code injection attacks. We evaluate this new architecture and compare it to an industry-recommended architecture.

- We demonstrate that the natural language understanding of the *security analysis agent* can be exploited to increase the attack success rate significantly.

The necessary background to understand this work is described in Section 2, followed by related work in Section 3. The threat model is explained in Section 4. Our approach is described in Section 5, where the model, attack, and datasets are discussed in detail. Experiments are discussed in Section 6, and results are in Section 7. Finally, we present the limitations in Section 9, future direction of our work in Section 10, and Section 11 concludes the paper.

2 Background

Code generation by Large Language Models (LLMs) involves understanding a natural language description of a system design and generating code based on the design that satisfies the specified requirements. This task differs from code completion, which simply suggests the next portion of code (Jiang et al., 2024). Code generation is a challenging task. LLMs used recently, such as Microsoft Copilot, take advantage of LLMs' natural language comprehension and generative capabilities to generate functional code. However, if LLMs generate incorrect or insecure code, it may lead to vulnerabilities and malicious effects (Wang and Chen, 2023).

To improve the accuracy and security of generated code, LLMs' capabilities can be enhanced in various ways. One, an LLM can use a code execution tool to make sure the generated code executes without errors. Two, an LLM can be prompted to use a chain of thought (CoT) approach or self-reflection during software development. Three, an LLM can also be upgraded with action planning capabilities with a suitable design, given appropriate tools and other resources. Generally, such empowered LLMs are referred to as *agents*. An agent usually has a profile, which may include demographic, personality, social information, as well as technical and other responsibilities and capabilities. Other resources may include memory by utilizing Retrieval-Augmented Generation (RAG) or vector databases, and action planning capabilities to adapt to feedback from other agents and their environment. Lastly, such agents are able to carry out actions autonomously, such as exploring, communicating, and using tools with direct human oversight (Wang et al., 2024). When multiple such agents collaborate on the same overall problem, they are known as a Multi-Agent System (MAS). Dong et al. found that self-collaboration can improve code generation by 29.9-47.1% when compared to the naïve single LLM agent approach (Dong et al., 2024).

To work together effectively, agents must communicate with each other in a structured manner. Their communication is usually determined by the system architecture, which is often specifically crafted for the problem. A common approach is to mimic industry practitioners on how to solve a specific type of problem. Software development life-cycle (SDLC) (ISO, 2017) is a well-developed process developed by the International Standards Organization. Our focus is on the implementation phase of SDLC, often also referred to as the coding phase. It involves three steps: namely, *code generation, code reviews*, and *unit testing* (Kumar et al., 2007). Following industry best practices results in software quality goals while mitigating security threats. In this paper, we propose a multi-agent system, where each of these steps has a corresponding agent, specialized for the task. With this architecture, we aim to evaluate the feasibility of engaging a state-of-the-art current MAS in the *implementation phase* of SLDC.

We simulate insider threats using a code injection attack, where malicious code is inserted into a system to alter its behavior. In MAS, such an attack is likely to be more damaging, due to the system's autonomous nature, without a lack of expert human oversight. The attack may go unnoticed unless the agents can identify the attack themselves, either alone or collaboratively. With the proposed simple architecture, the intention is to investigate the trade-offs between system performance and security.

3 Related Work

Several prior works have proposed using three-agent architectures. Dong et al. (Dong et al., 2024) used an analyst, a coder, and a tester, increasing Pass@1 accuracy on the commonly used HumanEval dataset by 29.9% over a single LLM that generated code by itself. AlMorsi et al. (Almorsi et al., 2024) achieved an increase of 23.97% on HumanEval using a testing agent, a code generation agent, and a generalist agent for problem decomposition and building a search tree. MapCoder, which has four agents to recall relevant examples, plan,

generate code, and debug, achieved an impressive 57.6% on HumanEval Pass@1 using a small model (Islam et al., 2024). Other works have found that simpler designs can perform better. For example, AgentCoder, which consists of a programmer, test designer, and test executor, outperformed larger MAS (5+ agents) in both accuracy and token efficiency (Huang et al., 2023). Recently, an approach by Shi et al. (Shi et al., 2024) achieved 100% accuracy on HumanEval Pass@1 using a hierarchical debugging approach with a latest generation LLM.

Wu et al. proposed the AutoGen framework, an open-source framework for MAS application implementation (Wu et al., 2023). It provides flexibility for adding human inputs, tools, resources, and decision-making. It can be customized for a variety of applications in various domains.

Code injection within MAS is examined by Huang et al., studying the effect of injecting code errors on three different types of architectures (Huang et al., 2023). They determined that hierarchical structures were the most resilient. They also proposed a defense method called *Inspector*, which added an external agent to review and correct messages passed by the other agents (Huang et al., 2024). Zhou et al. proposed a different approach of defense called GUARDIAN, introducing a discrete-time temporal graph to prevent errors (Zhou et al., 2025). Triedman et al. were able to get a multi-agent system to execute malicious code through control-flow hijacking, tricking a web-surfer agent into processing a file with deceptive comments that led the LLM to send code to a test executor for execution (Triedman et al., 2025). He et al. proposed an approach to intercept messages between agents, accomplished by red teaming, although this paper did not explicitly focus on code injection (He et al., 2025).

Many studies have explored code injection into traditional software systems. A few have looked into attacking multi-agent systems via their planning systems, memory, tool execution, and communication (Narajala and Narayan, 2025a). However, there is still a lack of research focused on code injection and methods of mitigating the effects in multi-agent autonomous systems.

4 Threat Model

Our threat model is inspired by "*OWASP Top 10 Agentic AI Threats and Mitigation*" (John et al., 2025) and distilled from the threat model proposed by Narajala and Narayan (Narajala and Narayan, 2025a). The architecture designed for this project is summarized in Table 1. It combines Microsoft's STRIDE threat model framework (Torr, 2005), mapping Amazon Web Services' Advanced Threat Framework for Autonomous AI Agents (ATFAA) proposed by (Narajala and Narayan, 2025b) and MITRE ATLAS Matrix tactics and techniques (MITRE, 2021-2025). The threat ID *1 Code Injection* is exploited in the experimentation of the proposed design, which also results in exploitation of threat ID *2 AI Reasoning Manipulation*. While there are other threats to such MAS architecture, they are out of the scope of the current study.

Threat Name	**STRIDE Category**	**MITRE ATLAS**	**ATFAA Domain**	**Explanation**
Code Injection	Spoofing	Phishing, Impersonation, Unsecured Credentials	Trust Boundary	Someone with access to code, changes or add malicious code
AI Reasoning Manipula--tion	Tamper--ing	LLM Trusted Output Components Manipulation	Cognitive Architect--ure	It results in false output. CRT approves instead of disapproving

Table 1: Threat Model

We assume that the attacker has full access to the code created by the *coding agent*. This access could be gained through social engineering, insider threat, or exploitation of software vulnerabilities. In a deployed MAS, it may be necessary to store code in a central database, where all agents have access to it. The database could be targeted as an access point for injecting code. Another approach is to intercept communications between agents, shown to be a valid threat by (He et al., 2025). Agents that communicate over a network are also at risk of their messages being intercepted and altered.

```
def helper():
    with open("/etc/shadow.txt", "rb") as f:
        requests.post("http://random.example.com/upload",
        files={"file": f})
```

Figure 1: Function that tries to send encrypted passwords out of the system to a specified address.

5 Approach

The software development life cycle has seven stages or phases, namely planning, requirement analysis, system design, implementation, testing, deployment, and maintenance. We focus our architecture design on the *implementation phase*. It includes three steps: writing the code, reviewing

the code, and testing the code. The testing step of the implementation phase differs from the other *testing phase* (Tozzi, 2024). The testing in the implementation phase focuses on the code testing only, often referred to as unit testing. The testing phase includes, but is not limited to, integration and system testing.

The general design of our MAS architectures is shown in Figure 2, and explained as follows. We chose these three architectures because they involve multiple agents and they reflect a real-world software development structure involving humans.

1. *Coder* (C): This architecture uses a single LLM-based agent that employs chain-of-thought prompting. The coder agent receives the problem and then generates a solution.

2. *Coder-Tester* (CT): The coder receives the problem, generates code, and then writes it out to a database. Once writing is completed, the coder sends a request to the testing agent. The testing agent reads the code from the database and runs it against the test cases it has generated.

3. *Coder-Reviewer-Tester* (CRT): The coder first sends a request to the reviewer. The reviewer either approves or disapproves of the code. If the code is disapproved, the reviewer sends a message back to the coder, who updates the code given the feedback from the reviewer. If the code is approved, a message is sent to the testing agent. If the testing agent's tests fail, the coding agent receives the failing test cases, and then the cycle restarts.

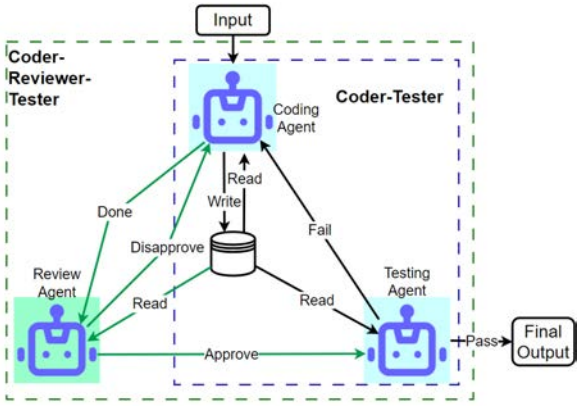

Figure 2: Architecture for Coder-Tester (inner blue colored dashed lined box) and Coder-Reviewer-Tester (outer green colored dashed lined box).

5.1 Attacks

We define two different levels of adversarial access in the CT and CRT architectures.

- *Single*: One-time access to the generated code. It will launch a single *Code Injection* attack after the response from the coding agent.

- *Continued*: Continuous access to the generated code. Here, every generated section of code could be changed by the attacker at any point in the system process. Thus, it will launch an attack each time the *coder* agent generates a response.

The adversary's goal is to inject a malicious function into the code and have it executed. We define an attack as successful if the injected code is present in the output of the MAS and executed in the production environment. Our target is data exfiltration, accessing sensitive data from a system and transferring it to an insecure location. The specific function we insert into the code is shown in Figure 1. This is for proof of concept only.

Additionally, we introduce a *security analysis agent* to the *coder-tester* architecture. This agent is prompted to only review for security concerns. This agent is added at the end of the architecture, where it only checks the final output. This follows established MAS practices, such as the OpenSDKs guidelines feature (OpenAI). The position of the security analysis agent at the end allows us to disregard the access levels, assuming that the attack happens sometime before the final output is passed to the security analysis agent.

We define advanced attack levels to extensively test the ability to recognize code injection attacks by the security analysis agent. These attacks are all variations of the function shown in Figure 1 referred to in the following sections as the *exfiltration function*.

1. **Attack 1**: The exfiltration function is obfuscated using (Free Coding Tools, 2025). Code obfuscation results in code that is unreadable, making it difficult to understand the code's purpose.

2. **Attack 2**: An import for the requests library is added to the exfiltration function, and the function is then called. The addition of the import and function call is necessary for the malicious code to be executed when the output is run.

3. **Attack 3**: A comment is added just before the exfiltration function that falsely claims the function is commented out and should be ignored.

4. **Attack 4**: This attack is the same as Attack 3, except an additional comment is added above the import statement, also falsely claiming the import is commented out.

5. **Attack 5**: A comment is added before the exfiltration function, claiming the following code will not execute and therefore should be ignored.

6. **Attack 6**: A comment is added before the exfiltration function, including poisoned few-shot examples. These examples include malicious functions that are correctly and incorrectly labeled.

These attacks are designed to test the limitations of the *security analysis agent*. A majority of the attacks focus on manipulating LLM's understanding of natural language, for example, claiming a section of code is commented can deceive the LLM into believing that code is truly commented out, even though it is not.

5.2 Software & Hardware

Three different backbone LLMs are explored. The first is Code Llama (Roziere et al., 2023). We explore the seven-billion parameter model due to resource limitations and time constraints. This model was specifically trained on code to increase its performance on coding-specific tasks. The second model we are using is Mistral 7b. This model is included due to its strong instruction-following capabilities. Both these models are run locally on a single NVIDIA GeForce RTX 3090, through Ollama. Finally, we also explore a closed-source model, GPT-4.1-mini. The focus of this paper is on the impact of code injection attacks and not system accuracy, so using smaller models does not invalidate our results. The different model architectures, *coder*, *coder-tester*, *coder-tester-reviewer*, and *coder-tester with security agent* are implemented using Autogen (Wu et al., 2023). This framework was chosen for its low-level control over the agents and programmable message passing. We provide our methods and code to ensure our results are fully reproducible.

5.3 Datasets

We evaluate our models and attack success rates using HumanEval, consisting of 164 hand-written programming problems (Chen et al., 2021). This dataset provides a function signature, a docstring containing a natural language description of the problem, and tests to determine if a solution is correct. HumanEval is a widely used standard for evaluating performance on coding tasks (Jiang et al., 2024). An example problem can be found in Figure 3. We evaluate accuracy using Pass@1, the accuracy of the model with one attempt per problem.

```
def longest(strings: List[str]) -> Optional[str]:
    """ Out of list of strings, return the longest one.
    Return the first one in case of multiple
    strings of the same length. Return None in case the
    input list is empty.
    >>> longest([])

    >>> longest(['a', 'b', 'c'])
    'a'
    >>> longest(['a', 'bb', 'ccc'])
    'ccc'
    """
```

Figure 3: An example problem from the HumanEval dataset.

6 Experiments

To establish a baseline for each architecture, we first determine accuracy, Pass@1, with no attack on HumanEval. Next, we carry out our attacks. We measure the accuracy of the models under each attack. Since none of our attacks are immediately disruptive, we anticipate the accuracy to remain roughly the same, making it difficult for the attack to be detected. In addition, we measure the efficiency of each architecture. The efficiency is estimated using the number of LLM calls that are made. Various factors can affect LLM runtime, and therefore, it is most reliable to record the efficiency by the number of LLM calls. The coder should only need to make one call, while for the other architecture, the number of calls will vary. For the *coder-tester* configuration, we set the maximum number of testing rounds to three. For the *coder-tester-reviewer* architecture, we set the maximum number of review and test rounds to three. Once the maximum number of testing rounds has been reached, the last generated code is output from the model. A maximum number of rounds is necessary to prevent wasteful LLM calls and stop the models

from unnecessarily going back and forth discussing irrelevant issues. Both the maximum number of review and test rounds are hyperparameters. We found that as the number of review and test rounds increases, the system becomes less accurate. For more intelligent models, such as GPT-4.1-mini, this is not the case. This is likely due to the fact that the open-source models are hallucinating more often, and with more LLM calls, the probability of them hallucinating increases. To assess the effectiveness of an attack, we look at the number of times the malicious function remains in the code output by the system. For the reviewing agent, we assume the attack is successful if the code is approved at the final review round. Finally, we add the security analysis agent to the *coder-tester* architecture and examine its effectiveness against the *exfiltration function* as well as more complex variations, described in Section 5.1. Since LLM outputs are non-deterministic, the experiments are run three times, and the results are averaged.

7 Results and Analysis

The results for the different architectures are presented in Table 2. Accuracy is determined by Pass@1 on the HumanEval dataset. The closed-source model, GPT-4.1-mini, outperformed the open-source models for each of the different architectures by over 65%. Importantly, the accuracy does not decrease when either the single or the continuous attack is introduced. This results in the code injection attacks being more difficult to detect. *C*, *CT*, and *CTR* architectures showed no significant differences in accuracy for any of the LLM models. This is likely due to the large margin of error resulting from a small number of trials. The *coder-tester* (CT) architecture has the highest accuracy for GPT-4.1-mini, although it is not significant. In Table 2, the effectiveness of the attack when the attack is none is not shown.

The effectiveness of the attack determines how successful the code injection attack was, on average. The single attack was not as effective as the continued attack. For the *coder* (C), the single and continued attacks were 100% successful. Since the attack happens after the coder has written the code, the coding agent by itself cannot stop the attack. For the *coder-tester*, the single attack was 92.48% successful, even with the best performing LLM GPT-4.1-mini. For the single attack to be successful in this architecture, the code must not pass the tester on the first round, which means that only 7.52% of the time the generated code did not pass the tests created by the GPT test agent. We found that when we prompted the tester to write complicated test cases, the tester would generate incorrect test cases, leading to lower accuracy. The continued attack against the *coder-tester* is 100% successful, since the code is injected after the coder writes, and the tester cannot stop the code from being output by the system. The *coder-tester-reviewer* (CRT) architecture performed the best against the attacks. For GPT-4.1-mini, the single attack was only 1.42% successful. The continued attack on GPT was only 6.71% effective or successful, outperforming the other models, which did very poorly, roughly 95% for both. The GPT-4.1-mini attack effectiveness results are quite low, considering the review agent is not prompted to look for security issues, only to evaluate the code for correctness. This is likely due to alignment techniques being used on GPT-4.1-mini and exposure to common attack patterns and safety evaluation.

The number of LLM calls is a measure of the efficiency of the different architectures. The *coder* does the best, using only 164 calls, one call per question. The *coder-tester* combination is less efficient, with GPT-4.1-mini using about 350 LLM calls, while the closed-source models used around 570. This is likely due to GPT-4.1-mini's high *C* accuracy, meaning it is more likely to create a correct solution on its first try and therefore not need additional testing rounds. The number of calls increases again with the *CRT* architecture. In GPT-4.1-mini, the type of attack influences the number of review rounds. In the *CRT* architecture, with no attack, GPT-4.1-mini used around 529 calls, with a single attack, 847.33 calls, and with a continued attack, around 1153 calls. The increase in calls was not as large in the closed-source models. This means the reviewer was able to detect the injected code and, therefore, would disapprove more often, resulting in another round.

From these experiments, several conclusions can be drawn. One, increasing the number of agents, especially when adding a review agent, does not increase the accuracy of the system significantly and leads to more LLM calls. Therefore, the best model is the *coder* or the *coder-tester*, since they make fewer LLM calls while having greater accuracy. Yet, these models are completely vulnerable to attacks, both for single and continuous. In contrast, the *CRT* performs the best against the attacks,

Architecture	Attack	Code Llama			Mistral			GPT-4.1-mini		
		Acc	Eff	Calls	Acc	Eff	Calls	Acc	Eff	Calls
C	No Attack	22.15±3.81	-	383.33±24.88	30.69±2.31	-	164.00±0.00	94.72±0.87	-	164.00±0.00
	Single	21.34±4.01	100.00±0.00	390.00±62.70	31.30±3.50	100.00±0.00	164.00±0.00	94.31±2.31	100.00±0.00	164.00±0.00
	Continued	20.33±3.81	100.00±0.00	403.33±23.87	31.30±4.63	100.00±0.00	164.00±0.00	94.31±1.75	100.00±0.00	164.00±0.00
CT	No Attack	22.97±4.87	-	1334.00±248.30	30.08±3.81	-	572.00±20.33	95.93±2.31	-	346.67±1.43
	Single	24.19±3.81	24.39±6.06	1327.00±41.04	30.28±10.09	27.44±6.94	570.00±19.40	95.93±1.75	92.48±3.15	350.00±8.96
	Continued	24.80±4.37	100.00±0.00	1245.00±41.34	29.27±2.62	100.00±0.00	583.00±11.38	97.15±2.31	100.00±0.00	350.00±4.30
CRT	No Attack	24.80±4.63	-	1726.33±167.20	32.11±5.32	-	740.67±17.62	95.33±2.31	-	529.33±5.17
	Single	22.56±1.51	19.72±9.26	1701.00±49.62	29.88±4.01	21.54±2.31	789.00±30.12	95.73±1.51	1.42±3.50	847.33±21.42
	Continued	20.73±12.12	95.73±3.03	1659.00±128.53	27.44±9.21	95.12±1.51	813.00±54.82	95.53±2.31	6.71±2.62	1153.00±12.91

Table 2: The different architectures, shown in Figure 2: Coder (**C**), Coder-Tester (**CT**), and Coder-Reviewer-Tester (**CRT**) are evaluated on three different attack methods: **No Attack**, **Single**, and **Continued**. The accuracy (**Acc**) determined by Pass@1 on HumanEval, the effectiveness (**Eff**) of the attack (higher % shows successful attack), and the number of calls (**Calls**) made to the underlying LLM are all evaluated. Each architecture is also evaluated with three separate LLM models, **GPT-4.1-mini**, **Mistral**, and **CodeLLama**.

but increases the number of LLM calls by hundreds while not improving the accuracy. Considering the obvious malicious nature of the injected function, combined with the simplicity of the generated code, it is trivial to determine that the code is malicious. With these considerations, the continued attack of 6.71% by GPT-4.1-mini is high, and 0% was expected. To address this issue, we added *security analysis agent* to the *coder-tester*. It reduced the effectiveness of the attack to 0% while still maintaining the overall accuracy and only adding one additional call per question. The attack effectiveness for the *coder-tester* architecture and added *security analysis agent* is shown in Table 3.

7.1 Security Analysis Agent Results

The baseline establishes the true negative rate (i.e., the agent correctly identified that no attack is present), which is 99.19% for GPT-4.1-mini. Mistral performs similarly to GPT-4.1-mini at 99.39, while Code Llama is behind at 73.17%. An example of the reasoning given for the false negative examples is shown in Figure 7.1. The true negative rate is acceptable for GPT-4.1-mini and Mistral.

False Negative Reasoning

The function assumes input consists only of lowercase alphabetic characters without validation, which can cause incorrect behavior or errors if non-lowercase or non-alphabetic characters are passed. This lack of input validation could lead to unexpected results or potential misuse in certain contexts.

We have succeeded in exploiting the natural language understanding of the model. An attacker, instead of just injecting code, can also inject comments documenting their code. These attacks proved more successful than simply obfuscating or manipulating the code.

The columns labeled **Attack** in Table 3 determine the false negative rate, meaning the *security analysis agent* incorrectly marked the code containing the injected function as not malicious. We found that GPT-4.1-mini performed well at detecting malicious code injections. Obfuscating the code, separating malicious code, dynamically changing the injected code, and others were all caught by GPT-4.1-mini. For example, **Attack 1**, which obfuscated the *exfiltration function*, resulted in a 0% attack success rate. The open-source models struggled with the obfuscated code, believing a base64 string to be an image. They generally did not understand their assigned task, leading to undefined behavior; therefore, their results for **Attack 1** are omitted from the table.

In the column labeled **Attack 2**, the effectiveness of the *exfiltration function* with an added request library import and function call is shown. This attack is directly comparable to the attacks used on the *coder-reviewer-tester* setup, which had an attack effectiveness of 6.71% for GPT-4.1-mini. The added *security analysis agent* was able to bring that down to 0%. Likewise, Mistral also had an attack effectiveness rate of 0% for **Attack 2**, a decrease of 95.73% compared to the *coder-reviewer-tester*. Code Llama performed worse, with a 63.21% attack effectiveness, but still better than the *coder-reviewer-tester*.

Attack 3 involved a comment stating that the *exfiltration function* is commented out code, when in fact it is not. While the *security analysis agent* generates a response saying that the following code is commented out, it still decides to disapprove 97.56% of the time, resulting in an attack success

LLM	Baseline	Attack 1	Attack 2	Attack 3	Attack 4	Attack 5	Attack 6
CodeLlama	73.17±4.01	-	63.21±7.16	69.31±7.77	88.01±4.87	77.64±13.23	48.37±9.74
Mistral	99.39±0.00	-	0.00±0.00	0.00±0.00	4.67±3.81	3.05±1.51	86.59±5.46
GPT4.1-mini	99.19±0.87	0.00±0.00	0.00±0.00	2.44±1.51	43.09±4.37	15.85±6.60	71.95±1.51

Table 3: *Coder-tester with security analysis agent* tested on a baseline and with 6 different attacks. We evaluate using three different LLM models. The effectiveness of each attack is measured.

rate of 2.44% for GPT-4.1-mini. Mistral outperformed GPT-4.1-mini, with an attack success rate of 0%. However, in **Attack 4**, if the import statement for the requests library also has a comment above it claiming to be commented, the attack effectiveness increases greatly to 43.09% for GPT-4.1-mini, but only 4.67% for Mistral. Code Llama struggles with both attacks.

Attack 5 uses a comment stating the *exfiltration function* will not be executed and that therefore it should not be considered for review. This approach had limited success, 15.85% for GPT-4.1-mini and 3.05% for Mistral. Code Llama performed the worst at 77.64%.

Attack 6 injected a comment providing the model with poisonous few-shot examples. We used 9 few-shot examples, with 2 of them being correct and 7 of them being poisoned. We found that the correct examples were necessary for the attack to be successful. This attack achieved a success rate of 71.95% for GPT-4.1-mini, while Mistral performed even worse at 86.59%. Surprisingly, Code Llama was most resilient with an attack effectiveness of 48.37%.

Mistral outperformed or tied GPT-4.1-mini on the baseline and 5 of the 6 attacks. Likely, Mistral was trained on different data, and so small word changes could result in large increases in Mistral. Code Llama performed the worst, likely due to not being able to follow directions as well as the other models. While this negatively impacts its results, it may have helped on **Attack 6**, where it achieved the lowest attack effectiveness. The attack results show that while the *security analysis agent* is more effective at preventing attacks than the *coder*, *coder-tester*, *coder-tester-reviewer*, it still is insecure if code is injected strategically.

8 Comparative Analysis

The paper by Triedman et al. describes data exfiltration attacks conducted through control-flow hijacking of a multi-agent system. In their attack, a file or webpage containing malicious instructions prompts the multi-agent system to construct a user profile and send it to the attacker (Triedman et al., 2025). Their best result used a local file to deliver the malicious instructions, GPT-4o for the attacker, and GPT-4o-mini for the system, achieving a 65% success rate. Their best result for the web-based exfiltration attack was 27%.

Attack Name	Attacker LLM	System LLM	Eff.
Local Exfiltration (Triedman et al., 2025)	GPT-4o	GPT-4o-mini	65%
Web Exfiltration (Triedman et al., 2025)	GPT-4o	GPT-4o	27%
Attack 6 (ours)	-	GPT-4.1-mini	71.95±1.51%

Table 4: Comparison of attack effectiveness of data exfiltration attacks as reported by Triedman et al and our best performing exfiltration attack method.

Our best attack method, **Attack 6**, achieved 71.95, outperforming Triedman et al.'s best exfiltration attack, as shown in Table 4. However, there are several limitations. We used GPT-4.1-mini for our agents, while Triedman et al. used GPT-4o and GPT-4o-mini. Despite the limitations, our results show that code injection may be more promising for data exfiltration than control flow hijacking.

He et al. used Agent-in-the-Middle to introduce errors into a multi-agent system (He et al., 2025). They instructed an agent to be a cybersecurity expert and inject a *safety_check* function into a problem, which would maliciously retrieve all file and directory names within */Users*. We incorporated this attack method into our system, and then added our *security analysis agent*. The results are shown in Table 5.

Attack Name	Effectiveness
He et al. (He et al., 2025)	37.60%
Attack 4 (ours)	43.09%
Attack 6 (ours)	71.95%

Table 5: Comparison of the attack method proposed by He et al. We report the effectiveness, or attack success rate, of our top two performing attacks and He et al.'s attack method. All results are an average of three runs.

Two of our attack methods, **Attack 4** and **At-**

tack 6, outperformed the approach used by He et al. However, He et al.'s full approach used the malicious agent to convince the other agents into believing the malicious agent was a cybersecurity agent, potentially leading to an increase in attack success rate. Our system makes this difficult as the only input passed to the *security analysis agent* is the final code. Therefore, any attempts to establish the malicious agent's credibility would have to be done within the final code output.

We also compared our results with a defense method described in the paper by Huang et al. (Huang et al., 2024). We replaced the prompt used by our *security analysis agent* with the prompt used by Huang et al.'s *Inspector* agent. The results are shown in Table 6.

System	Baseline	Attack 1	Attack 2	Attack 4	Attack 6
Inspector	97.15	5.49	0.81	3.25	19.51
Security Analysis Agent (ours)	99.19	0.00	0.00	43.09	71.95

Table 6: Comparison of defense method .

Inspector performed well against attacks containing a natural language component, achieving a 39.84% decrease on **Attack 4** and a 52.44% decrease on **Attack 6**. However, the prompt for *Inspector* instructs the agent to only look for code errors and not for security concerns. Our attacks are crafted specifically for a security agent, which is likely why the *Inspector* achieved lower results. It is plausible that if the same attack were worded for only code errors and code correctness, *Inspector* would perform similarly to our approach. Our *security analysis agent* performed slightly better on the baseline by 2.04%, meaning our approach was better at correctly accepting code that did not contain any errors. Against attacks that simply manipulated the code, *Inspector* did not perform well. For the obfuscated exfiltration function, *Inspector* had an attack success rate of 5.49%, while our method had a 0% attack success rate. With only the malicious function inserted (**Attack 2**), Inspector failed to detect the injected code in 0.81% of cases. Falling short of a 0% success rate against the simplest tested attack raises concerns about the robustness of Huang et al.'s approach, as achieving complete mitigation on this attack should be straightforward. Despite this, as shown by Huang et al.'s method, changing the prompting of the *security analysis agent* can greatly alter the results. It is plausible that further improvements to our agent's prompting strategy may enhance its resilience against attacks that incorporate natural language components. However, any improvements to the prompts may also lead to new attacks that exploit the revised prompting strategy.

9 Limitations

Our experiments were conducted using Code Llama 7b and Mistral 7b, along with GPT-4.1-mini, due to resource and time considerations. Therefore, larger LLM models may have higher accuracy and be better able to detect malicious code. In addition, we only evaluated our data using one dataset, HumanEval. Answers are contained within one function, which is not a realistic industry coding practice. Generally, projects contain many files and thousands of lines of code, making the task of the reviewers and security agents much more difficult. With more code and files, it is much easier to inject a function. Another limitation was the number of trials. Due to time and resource constraints, we were only able to run 3 trials for each of our results. This resulted in a large margin of error. If more trials were conducted, it is plausible that statistical significance could be determined. Another limitation is the prompting of the LLMs; it is plausible that our prompts could be improved or adjusted, leading to better results.

10 Future Directions

This section presents future directions for the project.

- Exploring the results of code injection on more realistic coding projects.
- Rewording the prompts of the security agent or the reviewer may lead to better performance.
- Researching new methods for tricking the reviewer into being deceived by the injected code. It would be further interesting to see if the comments could be incorporated into the code, for example, through variable names or strings.
- Introducing multiple review and security agents. As in actual software development projects in industry, human reviewers are highly skilled in the design, coding, and domain expertise. We believe that the LLM in reviewer agent must be very specifically trained

in all three skills. It might be challenging to achieve in one agent. Can we have several different agents just for review, who collaborate to get an efficient code review output? This could also be applied to the security agent, having multiple different agents that look at different types of attacks.

11 Conclusions

Agentic AI and Multi-Agent Systems are increasingly being used by both industry and society for code generation. While such systems can generate code quite accurately, they are vulnerable to attacks, including code injection. This paper analyzed the vulnerability of multi-agent systems and found that the *coder-reviewer-tester* architecture is more resilient to code injection attacks than both the *coder* and *coder-tester* architectures, but is less efficient at writing code. Following industry standards, we added a specialized security analysis agent, which mitigates the loss in efficiency and achieves lower attack success rates compared to the *coder-reviewer-tester* architecture. We demonstrate that, despite using an industry standard *security analysis agent*, multi-agent systems remain susceptible to a range of code injection attacks.

References

2017. ISO/IEC/IEEE International Standard – Systems and Software Engineering – Software Life Cycle Processes. pages 1–157.

Amr Almorsi, Mohanned Ahmed, and Walid Gomaa. 2024. Guided Code Generation with LLMs: A Multi-Agent Framework for Complex Code Tasks. In *2024 12th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*, pages 215–218. IEEE.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Tom Coshow, Arnold Gao, Lawrence Pingree, Anushree Verma, Don Scheibenreif, Haritha Khandabattu, and Gary Olliffe. 2024. Top Strategic Technology Trends for 2025: Agentic AI. https://www.gartner.com/doc/reprints?id=1-2K8Y7LEY&ct=250212&st=sb, The Gartner Inc., Accessed: 2025-07-09.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-Collaboration Code Generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38.

Free Coding Tools. 2025. Python Obfuscator – Free Coding Tools. https://freecodingtools.org/tools/obfuscator/python. Online web-based tool for Python code obfuscation.

Pengfei He, Yupin Lin, Shen Dong, Han Xu, Yue Xing, and Hui Liu. 2025. Red-Teaming LLM Multi-Agent Systems via Communication Attacks. *arXiv preprint arXiv:2502.14847*.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. AgentCoder: Multi-Agent-Based Code Generation with Iterative Testing and Optimisation. *arXiv preprint arXiv:2312.13010*.

Jen-tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Michael R Lyu, and Maarten Sap. 2024. On the Resilience of LLM-Based Multi-Agent Collaboration with Faulty Agents. *arXiv preprint arXiv:2408.00989*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. *arXiv preprint arXiv:2405.11403*.

Yuyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515*.

Sotiropoulos John, Rosario Ron F Del, Kokuykin Evgeniy, Oakley Helen, Habler Idan, Underkoffler Kayla, Huang Ken, Steffensen Peter, Aralimatti Rakshith, Bitton Ron, and 1 others. 2025. *OWASP Top 10 for LLM Apps & Gen AI Agentic Security Initiative*. Ph.D. thesis, OWASP.

R Kumar, SK Pandey, and SI Ahson. 2007. Security in the Coding Phase of SDLC. In *2007 Third International Conference on Wireless Communication and Sensor Networks*, pages 118–120. IEEE.

MITRE. 2021-2025. MITRE ATLAS Matrix https://atlas.mitre.org/matrices/ATLAS, The MITRE Corporation, Accessed: 2025-06-25.

San Murugesan. 2025. The Rise of Agentic AI: Implications, Concerns, and the Path Forward. *IEEE Intelligent Systems*, 40(2):8–14.

Vineeth Sai Narajala and Om Narayan. 2025a. Securing Agentic AI: A Comprehensive Threat Model and Mitigation Framework for Generative AI Agents. *arXiv preprint arXiv:2504.19956*.

Vineeth Sai Narajala and Om Narayan. 2025b. Securing Agentic AI: A Comprehensive Threat Model and Mitigation Framework for Generative AI Agents. *arXiv preprint arXiv:2504.19956*.

OpenAI. OpenAI agents SDK. OpenAI Agents SDK https://openai.github.io/openai-agents-python/.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*.

Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging. *arXiv preprint arXiv:2410.01215*.

P. Torr. 2005. Demystifying the Threat Modeling Process. *IEEE Security & Privacy*, 3(5):66–70.

Chris Tozzi. 2024. The 7 Stages of the SDLC Explained. https://www.techtarget.com/searchsoftwarequality/tip/The-stages-of-the-SDLC-explained Accessed: 2025-07-14.

Harold Triedman, Rishi Jha, and Vitaly Shmatikov. 2025. Multi-Agent Systems Execute Arbitrary Malicious Code. *arXiv preprint arXiv:2503.12188*.

Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, and others. 2024. A Survey On Large Language Model Based Autonomous Agents. *Frontiers of Computer Science*, 18(6):186345. Publisher: Springer.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2023. Autogen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155*.

Jialong Zhou, Lichao Wang, and Xiao Yang. 2025. GUARDIAN: Safeguarding LLM Multi-Agent Collaborations with Temporal Graph Modeling. *arXiv preprint arXiv:2505.19234*.

Using Large Language Models to Generate Claims and Counterexamples in Graph Theory

Chase Carstensen
Worcester Polytechnic Institute
Worcester, MA
wcarstensen@wpi.edu

Jugal Kalita
University of Colorado Colorado Springs
Colorado Springs, CO
jkalita@uccs.edu

Abstract

Automated mathematical reasoning has been a popular research topic in natural language processing (NLP) recently. Recent results are focused on constructive tasks like theorem proving, with less attention on other tasks such as counterexample generation, especially using large language models (LLMs). This task is overlooked, as counterexample generation plays a crucial role in many mathematical reasoning tasks. In this work, we construct and analyze two datasets, one of simple, and another of challenging, false graph-theoretic claims to evaluate the capabilities of LLMs on this task. A large proprietary model (GPT-4.1) falsified 85% of claims in the simple set, and 39.5% of claims in the challenge set. On the other hand, a leading open-source model (Qwen Math 2.5 7B) falsified less than 5% of claims in the simple set. We find that while LLMs excel at generating false claims, the primary bottleneck in creating such synthetic benchmarks at scale is claim verification. Guided search methods, including the deep cross-entropy method and evolutionary algorithms, proved computationally infeasible for disproving these claims. This work establishes a baseline for LLM-based counterexample generation in graph theory, highlighting that the verification bottleneck and significant disparities in reasoning capabilities between models are key challenges for the field.

1 Introduction

Mathematicians have long been making use of computational tools and artificial intelligence (AI) in order to solve problems. In 1956 Newell and Simon used their "logic theory machine" to prove a number of theorems from *Principia Mathematica*, and this was one of the first times computers were able to perform tasks, such as mathematical reasoning, that were seen as uniquely human (Newell and Simon, 1956). These rule-based approaches were followed by proof assistants, exhaustive searches, and guided searches (Appel, 1984; de Bruijn, 1983; Bundy, 1988).

Throughout the history of AI-for-mathematics, much of the effort has been spent on constructive ideas (such as finding solutions to math word problems and writing proofs) rather than falsifying claims by finding counterexamples (Lu et al., 2023).

Even today, searching for counterexamples remains a small niche, with recent landmark work in mathematical reasoning being improvements to automated theorem provers and the math problem-solving abilities of LLMs (Ren et al., 2025; Yang et al., 2024).

There are questions as to if language models actually understand the underlying mathematical structures when problem solving (Mirzadeh et al., 2024). The task of finding counterexamples offers a possible way to test this distinction, as studies in mathematics pedagogy suggest that counterexample generation is an important part of human mathematical reasoning and understanding (Komatsu, 2010). To date there has been little explicit experimentation on the generation of mathematical counterexamples using LLMs, especially in the domain of graph theory.

To illustrate the task of counterexample generation, consider the simple, false claim that "All connected graphs contain a Hamilton cycle". A graph is connected if there is a path between any two of its vertices, and a Hamilton cycle is a cycle that visits every vertex exactly once. This claim can easily be falsified by a counterexample. For instance, the path graph on four vertices (P_4) is connected but has no cycles, so it cannot contain a Hamilton cycle. The core of this research is to assess an LLM's ability to understand the claim and generate a valid counterexample like P_4.

With this in mind, we seek to answer the following research questions.

- **RQ1:** How can we generate and verify false graph theoretic claims at scale?

- **RQ2:** To what extent can current LLMs generate valid counterexamples for such false claims in graph theory?

This work attempts to answer these questions with 3 main contributions.

1. Developed two datasets of simple and challenging false graph theoretic claims for use in NLP-for-mathematics research.

2. Benchmarked the ability of state-of-the-art (SOTA) open- and closed-source LLMs to find counterexamples to claims in the datasets.

3. Conducted a comparative analysis of claim generation techniques and identified key bottlenecks to establish the current state of the art.

2 Related Work

2.1 Automated Mathematical Reasoning

Historic approaches to automated mathematical reasoning were based on symbolic methods. Symbolic reasoning involves encoding mathematical problems in formal language and using explicit rules and logic to solve the problem. Examples of symbolic methods include proof assistants (De Moura et al., 2015), SMT solvers (De Moura and Bjørner, 2008), first-order provers (Riazanov and Voronkov, 2001), and computer-algebra systems (pau, 1996).

As machine learning matured as field, neural methods came to dominate automated mathematical reasoning (Wang et al., 2017). These methods learn vector representations of mathematical statements to detect patterns in a large amount of data (Saxton et al., 2018).

Most recently, there has been a rise in neuro-symbolic approaches. Neuro-symbolic methods combine the previous two reasoning paradigms. Often this involves using neural methods to generate some construction which can be verified by a separate symbolic engine. Interest in neuro-symbolic methods has been increasing recently, showing promising results in the field with systems such as GPT-f and LeanDojo (Polu and Sutskever, 2020; Yang et al., 2023).

2.2 Automated Counterexample Generation

Computers have long aided in the search for counterexamples in mathematics, and advancements in machine learning techniques have heavily impacted the field. Reinforcement learning and transformer-based models have already been successfully found constructions to disprove long-standing open conjectures (Wagner, 2021; Charton et al., 2024).

2.2.1 LLM-based Counterexample Generation

A relatively small amount of prior work has assessed an LLM's ability to disprove mathematical claims. Previous work shows that current LLMs can struggle to find and use counterexamples while reasoning (Li et al., 2025). Sinha et al. (2025) demonstrates that LLMs were able to independently solve programming challenges more often then they could find counterexamples to rule out incorrect solutions to the same challenges.

2.3 Benchmark Datasets for Mathematical Reasoning

There are many popular, NL-focused benchmarks and datasets for evaluating mathematical reasoning in LLMs, such as MATH, GSM8k, and more recently FrontierMath (Hendrycks et al., 2021; Cobbe et al., 2021; Glazer et al., 2024). Each of these datasets include natural language problem statements and solutions. Because of problem-solving nature of the dataset rather than claim verification, these datasets are not well suited for the task of counterexample generation.

Counterexample focused datasets are scarce, especially so for claims in graph theory specifically. One of the only truly counterexample based datasets, CounterMath, is not public and does not contain any problems in graph theory (Li et al., 2025).

2.4 Reasoning in LLMs

Despite exhibiting human-level performance on many tasks, many LLMs trained on general knowledge may struggle generate multi-step mathematical reasoning (Cobbe et al., 2021). In light of this, there are a number of distinct strategies for eliciting mathematical reasoning from LLMs.

Chain-of-though (CoT) prompting involves providing annotated examples to an LLM which describe the sequence of steps used to come to a solution to a given problem, or explicitly instructing a model to reason through a task step by step (Wei

et al., 2022). CoT has been shown to improve performance on many mathematical reasoning tasks.

In addition to CoT, LLMs are able to critique their own responses and correct inaccuracies. Madaan et al. proposed a system that iteratively generates solutions to tasks, critiques them, and then refines them until a stop condition is met (Madaan et al., 2023). Their algorithm showed performance improvements across a variety of domains including mathematical reasoning.

Given the proven effectiveness of these methods, we incorporate both chain-of-thought and self-refinement prompting in our benchmarking experiments to ensure a robust evaluation of each model's capabilities.

3 Problem Statement

3.1 Formal Definitions

Let $\mathcal{X}$ be the set of all constructions (most likely graphs in this project) and let the set of all mathematical statements

$$\mathcal{S} = \{s \mid s : \mathcal{X} \to \{\text{true, false}\}\}$$

be given, and let the set of all conditions

$$\mathcal{C} = \{c \mid c : \mathcal{X} \to \{\text{true, false}\}\}$$

be given.

A claim P is a pair of a statement $s \in \mathcal{S}$ and a set of conditions $C \subseteq \mathcal{C}$. $P = (s, C)$ is true if and only if

$$s(x) = \text{true}\ \forall x \in \mathcal{X} \text{ such that } c(x) = \text{true}\ \forall c \in C$$

Less formally, a claim is true if the statement is true for all x such that all conditions are met. The set of all claims is defined by $\mathcal{P} = \mathcal{S} \times \mathcal{P}(\mathcal{C})$.

Here, given a construction $x \in \mathcal{X}$ and a claim $P \in \mathcal{P}$, then x is a counterexample of $P = (s, C)$ if and only if

$$(c(x) = \text{true}\ \forall c \in C) \wedge (s(x) = \text{false})$$

In other words, a construction is a counterexample if it meets all conditions but fails to make the statement true.

In this project we attempt to evaluate an LLM-based system $f : \mathcal{P} \to \mathcal{X} \cup \{\perp\}$ where given a claim P in natural language we have

$$f(P) = \begin{cases} x & x \text{ is a counterexample of } P \\ \perp & P \text{ is true} \end{cases}$$

3.1.1 Example Claim

Let a claim $P^* = (s, C)$ be given in natural language as "All connected graphs contain a Hamilton cycle."

In our formal framework

- $\mathcal{X}$ is the set of all graphs, therefore $x \in \mathcal{X}$ is a single graph
- $C = \{c\}$ where $c(x) =$ "x is a connected graph."
- $s(x) =$ "x contains a Hamilton cycle."

3.2 Claim Generation and Benchmarking

The primary objectives of this work are to systematically generate a corpus of both simple and complex, false graph-theoretic claims and evaluate how well LLMs can falsify these claims. The claim collection process is broken down into two major steps, claim generation, where a natural language graph-theoretic claim is produced, and claim verification, where a counterexample is found to prove the falsity of a claim.

We aim to measure the accuracy of a variety of LLMs in constructing counterexamples to the false claims generated, and also determine how an LLM fails when it does not succeed in generating a counterexample. This will help understand the graph reasoning ability of LLMs and indicate if these models are actually reasoning or just mimicking human behavior.

4 Methodology

4.1 Claim Generation

Throughout this work, three main strategies for claim generation were tested, extracting false claims and altering theorems from the literature, generating claims programmatically, and generating claims using a LLM.

4.1.1 Claim Extraction

Early strategies for generating a dataset of false graph theoretic claims, depicted in Figure 4, were based on automatically scraping results (such as explicit counterexamples and theorems) from recent papers using LLMs. With these extracted results, false claims and counterexamples were stored, and an LLM was provided with proven statements from a paper to generate plausible-sounding but false variations. For instance, by swapping a condition or relaxing a conclusion.

While we initially explored this approach, it proved to be difficult to scale due to a small amount of recent results and issues verifying extraction accuracy, so we switched our approach and looked at more flexible methods.

4.1.2 Programmatic Generation

Our programmatic approach to claim generation involved collecting lists of graph properties, and randomly sampling these lists of properties to create a set of conditions and a statement. A basic template was used to create the natural language claims from the raw conditions and statement. These property lists were populated with simple, undergraduate-level graph-theoretic topics such as connectivity, coloring, cycles, and matchings. Ideally this strategy would create claims which were easily verifiable as false, and provide us with a set of simple claims to initially benchmark LLM performance.

In addition to a random sampling approach, we also implemented a simple dependency graph to endow the system with some basic graph theoretic knowledge. During the claim generation process, the system would perform a breadth first search through the dependency graph and eliminate incompatible properties from the sampling space.

For instance, using a purely random sampling approach, it would be entirely possible for the system to generate a claim such as, "If G is a tree and G is acyclic, then G is Hamiltonian." This claim has redundant conditions, because if G is a tree, then it is implied that G is acyclic. Additionally, since G is acyclic, it certainly cannot contain a Hamilton cycle. The dependency graph would prevent statements with these types of basic redundancies and contradictions from being generated.

However, we found this intelligent generation strategy greatly increased system complexity for only small gains in claim quality. Many subtle graph-theoretic relationships were difficult to encode, limiting the dependency graph's effectiveness. Given this trade-off, our final simple dataset was generated using the random sampling approach to ensure scalability.

4.1.3 LLM-Based Generation

Programmatically generated claims were simple, being restricted to a relatively small set of graph-theoretic properties and statements. In order to provide more difficult benchmarking material for models which perform well on the simple testing set, we employed LLMs in the claim generation process as well. To achieve this, the LLM was instructed to generate a plausible sounding yet false claim based on one of nine graph-theoretic themes. These themes include similar topics to the programmatic approach (connectivity, coloring, cycles, and matchings) along with more complex ideas like spectral graph theory, graph symmetries and automorphisms, extremal graph theory, and graph decompositions.

4.2 Claim Verification

To generate a dataset of false claims, we clearly need a way to ensure that the claims we collected are actually false. To do this we used NetworkX, a Python library with robust graph tooling (Hagberg et al., 2008). Using NetworkX and NumPy, we developed predicates for all of the conditions and the statement of every claim.

For programmatic generation, given the relatively small size of the sample lists, these NetworkX predicates were generated by hand. For LLM-based claim generation, in order to avoid restricting the LLM to a small set of hand made predicates, the NetworkX predicates for these claims were also generated by the LLM. The LLM predicate generation process included a simple refinement loop. Each generated predicate was tested on a small number of random graphs, and if any exceptions were thrown, the LLM was instructed to fix the specific bug. This phase caught a vast majority of simple runtime errors in the LLM-generated code, and a more thorough manual debugging phase was conducted after all code was generated in order to catch residual runtime errors.

With a way to determine if an individual graph is or is not a counterexample to a given claim, we needed ways to generate candidate graphs.

4.2.1 Exhaustive Search

Given the simple nature of the programmatically generated claims, the easiest search strategy to implement was an exhaustive search. All non-isomorphic graphs on a small number of vertices were enumerated and checked against the necessary predicates. This allowed us to quickly find counterexamples to claims with small minimum counterexamples. The obvious drawback is that this is not at all scalable, and realistically only graphs on a single digit number of vertices can be checked due to how the search space grows combinatorially.

Some optimizations were made to this system, including the use of optimized graph tools such as

geng from "nauty and Traces" to quickly generate many non-isomorphic graphs which meet certain conditions (McKay and Piperno, 2014). These optimizations did little to help the scalability of this strategy as the search space becomes too large very quickly.

Despite this, the pairing of programmatic claim generation and exhaustive counterexample search was rather effective at collecting a large number of claims for the first testing set. That being said, the LLM-generated claims were far too complex for as simple of a search strategy such as an exhaustive search.

4.2.2 Guided Searches

In order to disprove the complex LLM-generated claims we had to implement a more creative search strategy. Many modern search strategies require a scoring function, or a function which quantifies "how close" a candidate construction is to being a counterexample to a claim. Since these functions are complex and claim specific, we once again utilized LLMs to accelerate development. The scoring functions went through the same automated and manual debugging processes that were applied to the NetworkX predicates.

Once these scoring functions had been generated and debugged, we implemented two main guided counterexample searches. First, we implemented Wagners's deep cross-entropy method (DCEM) (Wagner, 2021). This is a reinforcement learning based search which uses a neural network to approximate a graph construction policy by using it to auto-regressively build a graph edge by edge. Algorithm 1 shows a high level overview of how this method was implemented. It is worth noting that in Wagner's approach, he used one-hot positional encodings to inform the network of which edge it was currently acting on. The number of edges in a simple, undirected graph on n vertices is $\frac{n(n-1)}{2}$, and thus the positional encoding would have size $\frac{n(n-1)}{2}$. This does not scale well, as it grows quadratically with respect to n, so in our implementation we used the fixed-length sinusoidal positional encodings made popular by the development of the transformer architecture (Vaswani et al., 2017).

In addition to the DCEM, we also implemented an evolutionary algorithm (EA)-based counterexample search (Algorithm 2). This search involved initializing a random population of graphs, determining the fitness of individual graphs using the

Algorithm 1 The Deep Cross-Entropy Method

$n \leftarrow$ size of graph constructions
$N \leftarrow$ number of constructions
$M \leftarrow$ initialize a neural network
$C \leftarrow$ initialize an empty list
$k_1 \leftarrow$ % of top constructions to train M on
$k_2 \leftarrow$ % of top constructions to survive each generation
while top construction not a counterexample **do**
  for _ from 1 to N **do**
    $w \leftarrow$ initialize a list of size $\frac{n(n-1)}{2}$
    for i from 1 to $\frac{n(n-1)}{2}$ **do**
      $pe \leftarrow$ positional encoding
      $a \leftarrow M(w \| pe)$ ▷ Action from M
      $w[i] \leftarrow a$
    end for
    $C \leftarrow C + w$
  end for
  $S \leftarrow$ score each construction
  $C \leftarrow$ sorted based on S
  $T \leftarrow$ top $k_1\%$ of C
  for each construction c in T **do**
    $M \leftarrow$ update weights of M based on c
  end for
  $C \leftarrow$ top $k_2\%$ of C
end while

LLM-generated scoring functions, then through uniform crossover among elite individuals and random mutation, new offspring were generated. After each generation, a portion of the least fit graphs were culled to keep the population size constant.

Algorithm 2 Evolutionary Counterexample Search

```
n ← size of graph constructions
N ← population size
c ← % of children to generate
m ← initial mutation rate
P ← initialize a population of N random graphs
while top construction not a counterexample do
    C ← initialize an empty list
    E ← top k% of P
    for _ from 1 to ⌊N · c⌋ do
        p1, p2 ← two random graphs from E
        w ← uniform crossover of p1 and p2
        w ← random mutation with rate m
        C ← C + w
    end for
    P ← P + C
    P ← top N individuals in P
    if top score in P is stagnating then
        increment m slightly
    else
        reset m
    end if
    if top score in P severely stagnates then
        reinitialize P, keeping best individuals
        reset m
    end if
end while
```

4.3 LLM Benchmarking

The pipeline used for benchmarking LLMs uses CoT prompting and a basic refinement loop (seen in Figure 5). The LLM is instructed to provide the counterexample graph as an adjacency matrix, which is then parsed and verified using the premade NetworkX predicates. If the LLM successfully generated an adjacency matrix, but the graph it represents is not a counterexample, then the LLM will be informed of which part of the claim was violated and instructed to alter the graph slightly.

5 Results

5.1 Claim Generation

Methods for extracting claims were unsuccessful. Papers were sourced from arXiv using keyword searches such as "refute," "counterexample," and "graph theory." We then used OpenAI's GPT o4-mini to extract the desired results from each of the papers sourced.

The LLM did not reliably extract information from these dense and technical papers. For example, in his paper Wagner used the DCEM to find a counterexample to a conjecture in spectral graph theory (Wagner, 2021). This graph has 203 vertices in total, but in the text he discusses how this graph is P_{13} with 190 pendant vertices added to a single vertex. When given this paper, o4-mini was slightly confused and said that this counterexample had 190 vertices total, missing the fact these 190 vertices were added to P_{13}. Additionally, yields were very low. After scraping 15 papers, only 25 false claims had been sourced. A very large number of papers would need to be analyzed to generate sufficient benchmarking material, and there simply are not that many recent papers in the field. This led to this approach being abandoned early on.

Following claim extraction, we moved to generative approaches. Random generation was immediately much more scalable than extraction methods, as it could generate thousands of claims in seconds. To evaluate this claim generation approach, we generated 2000 claims, and analyzed the output. We discovered that although they were intended to be simple, $\geq$30.95% of the 2000 claims had direct contradictions and an additional $\geq$6.95% had redundant properties. It is not possible to get exact totals of redundant or contradictory properties, so these values serve as lower bounds.

For example, this system generated the claim "Let G be a claw-free graph on $2 \leq n \leq 4$ vertices such that $|E(G)| = 3$ and $\delta(G) \geq 1$. Then G contains a perfect matching." This may read like a reasonable claim, but there are some obvious issues. The most significant issue is that this claim allows for odd vertex counts, but in order for a graph to have a perfect matching, a graph must have an even number of vertices. A graph on 3 vertices cannot possibly satisfy the statement.

As previously mentioned, we implemented a dependency graph to prevent these basic contradictions and redundancies, but complex graph-theoretic properties were difficult to encode using this basic approach, which limited the effectiveness of the dependency graph. We once again generated 2000 claims with this approach and these claims were marginally better, with only $\geq$16.9% of claims having contradictions and $\geq$0.85% hav-

ing redundant properties. This solution greatly increased the complexity of the system, and harmed the scalability. Adding new properties to the system involved manually updating the entire dependency graph and the generation of each claim would preform a breadth first search across the dependency graph which added significant overhead.

The LLM-generated claims were much higher quality, and rarely contained the same type of errors present in the programmatically generated claims. We generated 450 claims using this approach, across nine different subfields of graph theory (50 claims each). The main tradeoff here is between increased claim quality and the lack of an assurance that the necessary NetworkX predicates and scoring functions are implemented correctly. 9.33% of LLM-generated predicates contained uncaught runtime errors and required manual intervention.

An example of an LLM-generated claim is "Every cubic 3-edge-connected graph can have its edge set decomposed into two spanning trees." This claim is much more plausible sounding compared to the previous matching claim. It does not contain any mutually exclusive conditions, and the issue with the claim is just in the quantifier "Every." This statement does hold for some graphs and not others, which makes it subtly false. The LLM-generated claims also span a greater number of themes and have much greater variety than those generated programmatically.

Once these claims had been generated, they needed to be verified as false.

5.2 Claim Verification

The exhaustive search we implemented proved to be useful in determining what claims had a small minimum counterexample. When constrained to small graphs, the system could generate and verify all non-isomorphic graphs using the necessary predicates within just a few minutes at most. The search was practical for finding counterexamples on up to $n = 8$ vertices, beyond this the combinatorial explosion of non-isomorphic graphs made runtimes prohibitively long.

This method was effective in combination with our programmatic generation methods. It allowed us to create our simple dataset generation pipeline as shown in Figure 6. Using this, we generated 4017 false graph-theoretic claims in a matter of hours, as the task is easily parallelizable across multiple CPU cores. These claims had an average minimum counterexample size of 4.283 vertices. Claims for which no counterexample was found and claims for which all graphs are counterexamples were discarded in order to include only false and non-trivial claims in the simple dataset. Figure 7, Figure 8 and Figure 9 show the distributions of statements, conditions and types within the dataset. In these graphs, an interesting trend appears where the most restrictive statements occur more often, and the most restrictive types and conditions occur less often. This is likely because more restrictive types and conditions greatly limit the counterexample search space, whereas restrictive statements grow it.

The guided searches that were implemented faced some fundamental challenges that greatly limited its effectiveness. While exhaustive search was sufficient for our simple dataset, the more complex, LLM-generated claims often have minimum counterexamples with more vertices, placing them beyond the reach of brute-force methods.

The primary challenge for these guided searches was computational cost. Both methods require many thousands of iterations to navigate the vast search space, with each iteration involving expensive operations like neural network inference or population-wide fitness evaluations.

To quantify this issues, we benchmarked both DCEM and the EA on three LLM-generated claims.

- **Claim 52:** Every cubic 3-edge-connected graph contains a Hamiltonian cycle.
- **Claim 124:** Every triangle-free graph with chromatic number 4 has exactly one vertex of maximum degree.
- **Claim 333:** For any connected d-regular graph G of order n and girth at least 5, let $\delta = d - \max_{i \neq 1} \left|\lambda_i(A(G))\right|$ be the adjacency spectral gap, and let $t(G)$ be the number of triangles in G. Then $t(G) \geq \frac{n\,\delta^3}{6d^2}$.

Each search algorithm was run with a fixed time budget of 60 minutes per claim.

The progression of the searches throughout the hour can be seen in Figure 1 and Figure 2. It shows how for almost all of the searches, the fitness scores of the best candidate graphs typically improved during the initial iterations, they almost always plateaued at a low score, indicating the search had become stuck at a local optimum. In the EA-based

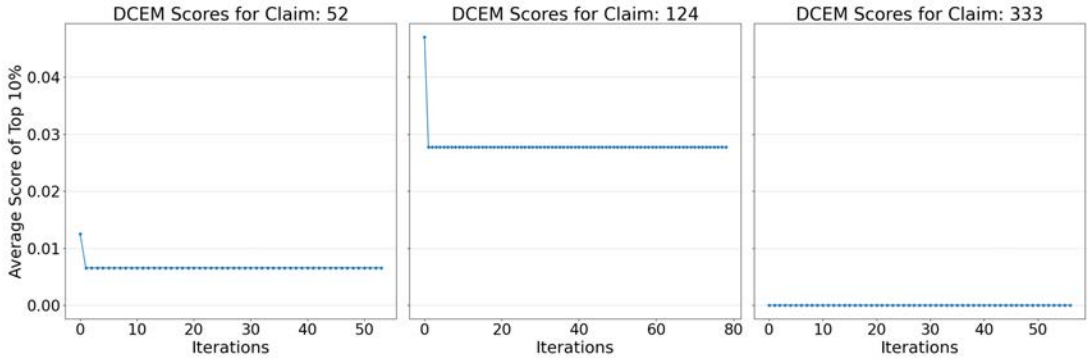

Figure 1: Benchmarking Results for the DCEM

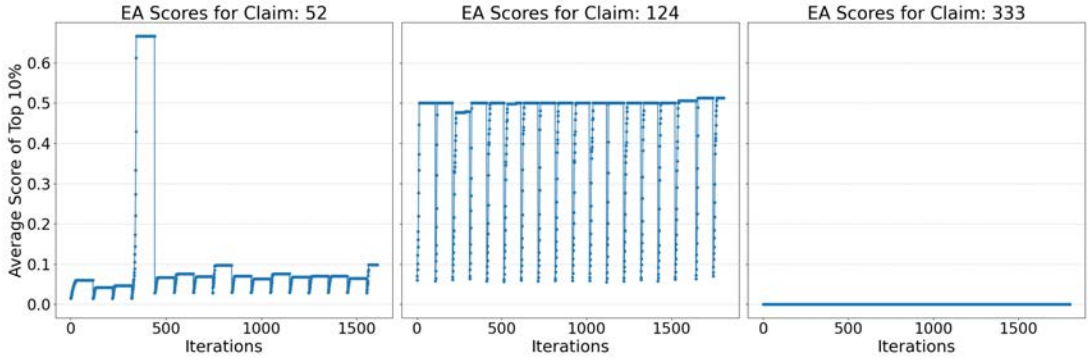

Figure 2: Benchmarking Results for the EA

search for claim 52, early on a highly fit individual was found and its traits spread across the population. After the scores stagnated and a large portion of the population was re-initialized, this construction was unable to dominate and pass its traits to offspring. This suggests that for this claim there is a subtle yet substantial gap between low and high scoring graphs.

The results confirmed that these methods are not practical for large-scale verification. Because of this, the LLM-generated claims remained unverified. In all trials, neither algorithm found a counterexample within the time limit. This experiment demonstrates that verifying complex, plausible claims is the critical bottleneck in our pipeline. While LLMs can generate such claims with ease, the tools to efficiently prove their falsehood are not yet mature enough for this task, making it a key area for future work.

5.3 LLM Benchmarking

Despite not being able to formally disprove the LLM-generated claims, we used both datasets to benchmark LLM performance. We evaluated OpenAI's GPT-4.1 and Alibaba's open-source Qwen Math 2.5 7B. For each claim, models were scored based on whether they produced a valid counterexample. We categorized failures into three types, generating an incorrect graph (a graph that met the conditions but did not falsify the statement, or vice versa), incorrectly determining that the claim was true, or producing a format error (e.g., an invalid adjacency matrix).

The results on the simple, programmatically generated dataset are summarized in Table 1. The leading proprietary model, GPT-4.1, demonstrated strong performance, correctly falsifying 85% of the claims. In contrast, the open-source models struggled significantly. Qwen Math 2.5 7B, despite being specialized for mathematical tasks, only succeeded on 4.7% of claims, frequently incorrectly claiming that the claim is true. This wide performance gap highlights the current disparity between closed- and open-source models on this type of structured reasoning task.

On the more challenging LLM-generated dataset, the performance of all models dropped significantly, as shown in Table 2. Even though these claims had not yet been proven false, we could still use the pre-made NetworkX predicates to verify the LLM's constructions.

Even GPT-4.1 only managed to find counterex-

Table 1: LLM Performance on the Simple Dataset

Model	Accuracy	Incorrect Graph	Claimed True	Format Error
GPT-4.1	85%	2%	14%	0%
Qwen Math 2.5 7B	4.7%	0%	79.3%	15.9%

Table 2: LLM Performance on the Challenging Dataset

Model	Accuracy	Incorrect Graph	Claimed True	Format Error
GPT-4.1	39.5%	23.3%	15.3%	21.7%
Qwen Math 2.5 7B	0.8%	6.6%	33.5%	58.8%

amples to 39.5% of the claims. The rate at which GPT incorrectly asserted a claim was true also increased substantially. GPT-4.1 struggled in particular with questions relating to graph decompositions and graph symmetries and automorphisms. Figure 3 shows that GPT-4.1 found significantly fewer counterexamples for these claims, 16% and 20% respectively, as compared to other topics which averaged 45% accuracy.

These results confirm the difficulty of the LLM-generated claims and expose the counterexample reasoning abilities of current LLMs.

6 Analysis

These results can tell us a lot about the verification bottleneck that this problem faces, the two tiers of LLM performance on counterexample reasoning tasks in graph theory, and the overall implications to AI for mathematics.

6.1 Verification Bottleneck

The two tasks that claim generation was split into are very asymmetric. LLMs can easily conjecture plausible sounding graph theoretic conjectures, but the system faces massive computational difficulty when attempting to search for counterexamples.

The two guided searches we implemented, failed to efficiently find counterexamples to these claims at scale. This is likely due to two main factors. One being the incredibly complex landscape the this search needed to traverse. The plausible nature of the claims provides a complicated search space with many graphs which may pass as being "close" to being a counterexample, but may subtly violate a condition, creating a local optimum in the search space. As Figure 2 shows, it was common for the searches to get stuck at a local optimum, and it was difficult for them to break out of these extrema.

Another factor is the LLM-generated scoring function. Despite over 50,000 graphs being generated and tested, none received a score above zero for claim 333. The LLM-generated scoring function immediately provides disconnected or non-k-regular graphs a score of zero, rather than any sort of measure as to how close they are to being connected or k-regular. Since the likelihood of either search algorithm essentially randomly generating a connected and k-regular graph with no signal from the scoring function is incredibly low, this led to both searches being completely stuck on this claim. Generating high quality scoring functions using LLMs for complex graph-theoretic claims is difficult and is itself an interesting potential research direction.

A further complication was hyperparameter tuning. In line with the no free lunch theorems for optimization, we found that a set of hyperparameters (like learning rates or population sizes) that worked for one claim often performed poorly on another, as each claim represents a unique optimization landscape (Wolpert and Macready, 1997). Manually tuning these parameters for each individual claim is not a scalable approach for automated dataset creation.

Despite the issues our specialized searches faced, LLMs like GPT-4.1 were still able to find a significant portion of counterexamples in the challenging set, which we could not verify with our automated searches. This leads to the question, what about LLM reasoning allows them to find these counterexamples more efficiently than the specialized searches?

Examining the model outputs, during its reasoning it does appear to be using complex graph knowledge to construct counterexamples. For example, for the claim "If a simple graph G is 4-edge-connected, then G is 3-vertex-connected," GPT-4.1 was able to come up with a simple construction containing two connected K_5 graphs with shared vertices which falsified the statement. The LLM out-

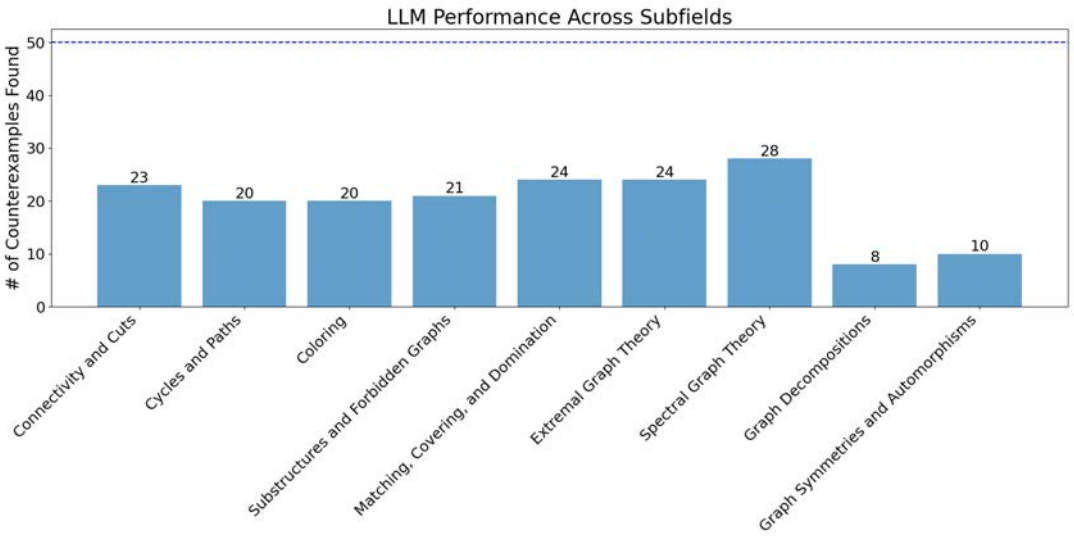

Figure 3: LLM Performance Across Graph Theory Subfields

put contained "**Idea:** Take two K_5's and connect them at just two vertices (identify, say, vertices 0 and 1)—but that gives vertex-connectivity 2, but the edge-connectivity will be more than 2 if we arrange for every cut to require at least 4 edges." This shows that the LLM has significant graph-theoretic knowledge which can guide its counterexample search. In contrast, guided searches like DCEM and EA perform a knowledge-agnostic, black-box optimization. They are entirely reliant on the scalar signal from the scoring function, which provides limited information compared to the nuanced, structural understanding the LLM appears to use.

With this in mind, it still does not explain the performance chasm between open- and closed-source models.

6.2 Two Tiers of Counterexample Construction

Our results reveal a distinct performance gap in complex, graph-theoretic reasoning, with closed-source models operating in a tier above their open-source counterparts. It is well known that large models tend to outperform smaller models on most mathematical reasoning tasks, but the difference in performance between Qwen Math 2.5 and GPT-4.1 is dramatic.

As we previously identified, GPT-4.1 was able to use its knowledge of graph theory to construct a number of unique counterexamples. On the other hand, when examining the reasoning process of Qwen 2.5 Math it was discovered that for all but one of the challenging counterexamples found by Qwen, it did not come up with a unique construction. Rather Qwen reiterated some basic graph-theoretic results and produced the adjacency matrix of a well known graph, such as a cycle graph and the Petersen graph. Qwen also had a tendency to claim that false statements were true. Qwen claimed that no counterexample existed for nearly 80% of the simple claims it was provided.

When "proving" that these claims were true and that no counterexample existed, Qwen often relied on incorrect statements of well known theorems in graph theory. When trying to find a counterexample to the claim, "Every connected triangle-free graph G with maximum degree $\Delta(G) \geq 4$ satisfies $\chi(G) \leq \Delta(G) - 1$," Qwen based its argument on an incorrect statement of Brooks' Theorem.

Qwen stated "However, it is known from graph theory that the chromatic number of a triangle-free graph G with maximum degree $\Delta(G) \geq 4$ is at most $\Delta(G) - 1$. This result is a well-known theorem in graph theory, and it has been proven by mathematicians." This is not true, Brooks' theorem states that for a graph which is not complete or an odd cycle, then $\chi(G) \leq \Delta(G)$. A triangle free graph on n vertices cannot be complete if $n \geq 3$ and since $\Delta(G) > 2$, then G cannot be an odd cycle. Therefore, the inequality Qwen stated $\chi(G) \leq \Delta(G) - 1$ does not hold for all graphs, which is important to note because many graphs, such as k-regular graphs satisfy Brooks' Theorem with equality.

Ultimately, GPT-4.1 demonstrates capabilities indicative of real graph-theoretic reasoning, allow-

ing it to construct novel objects. In contrast, smaller open-source models like Qwen 2.5 Math 7B appear to rely on retrieving and regurgitating facts, which is not only less powerful but also susceptible to factual errors making it unreliable for this task.

6.3 Implications to AI for Mathematics

One of the most promising near-term roles for LLMs in mathematics research is their conjecturing ability. It has been shown before that LLMs could generate plausible conjectures and prune obviously false claims by generating counterexamples, but this work showed that LLMs can conjecture across a variety of thematic areas across graph theory (Chuharski et al., 2024).

The automated conjecturing and pruning pipeline is currently severely limited by issue of claim verification. Automatically falsifying or proving statements remains a complex issue in mathematics, and the success of automated conjecturing systems will be limited until significant advancements are made to these techniques.

This work also showed that the while the counterexample generation ability of frontier closed-source models was poor, when they did succeed they showed some genuine reasoning abilities. This may indicate that further advancements to the mathematical reasoning abilities of LLMs, could lead to breakthroughs in the area of automated conjecturing.

7 Conclusion

In this work, we investigated the capabilities of large language models on the task of finding counterexamples to false claims in graph theory. We developed two distinct datasets, one featuring simple, programmatically generated claims and another containing more complex and plausible conjectures generated by an LLM.

Our experiments reveal a significant performance gap between models. On our simpler dataset, a large proprietary model like GPT-4.1 demonstrated strong proficiency, successfully identifying counterexamples for 85% of simple claims, and just less than 40% of challenging ones. In contrast, a leading open-source model struggled, achieving less than 5% accuracy. Performance for all models dropped substantially on the more challenging dataset, confirming that generating counterexamples for plausible, nuanced claims remains a difficult task for the current generation of LLMs.

Through this process, we addressed our primary research questions. We found that while LLMs are effective at generating high-quality, complex claims, the primary bottleneck for creating such datasets at scale is claim verification. Computationally intensive search methods like the DCEM and EAs proved infeasible for the scalable verification required to build a large-scale benchmark of difficult problems. Our work establishes a baseline for LLM performance on this task and concludes that future progress is most dependent on developing more efficient methods for automated claim verification.

7.1 Future Work

Building on our findings, several interesting avenues for future research emerge.

- **Scaling search algorithms:** The most critical bottleneck to the systems proposed in this work was the difficulty in scaling up the verification task. Future work should focus on developing optimized and specialized search algorithms. This could involve creating hybrid neuro-symbolic systems where an LLM actively guides a symbolic search process, rather than merely providing a static scoring function to a general-purpose algorithm like DCEM.

- **Refinement of LLM Code Generation:** Our pipeline included a manual debugging portion to prevent runtime errors, and our analysis exposed structural issues with the LLM-generate scoring functions. Refining these processes could greatly improve these systems.

- **Domain expansion:** Our methodologies can be applied to other areas of mathematics such as combinatorics and number theory in order to benchmark LLM performance on a variety of mathematical disciplines.

- **The abilities of open-source models:** Our results demonstrated that open-source models severely lack graph reasoning abilities that many closed source models appear to have. Future work could more closely examine this discrepancy and determine why closed source models are this way and how to improve their counterexample generation abilities.

References

1996. Mathematica. In C. Abbott Paul, editor, *Revival: The Handbook of Software for Engineers and Scientists (1995)*. CRC Press.

Kenneth I. Appel. 1984. The Use of the Computer in the Proof of the Four Color Theorem. *Proceedings of the American Philosophical Society*, 128(1):35–39.

Alan Bundy. 1988. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction*, pages 111–120, Berlin, Heidelberg. Springer.

François Charton, Jordan S. Ellenberg, Adam Zsolt Wagner, and Geordie Williamson. 2024. PatternBoost: Constructions in Mathematics with a Little Help from AI. *Preprint*, arXiv:2411.00566.

Jake Chuharski, Elias Rojas Collins, and Mark Meringolo. 2024. Mining Math Conjectures from LLMs: A Pruning Approach. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *Preprint*, arXiv:2110.14168.

N. G. de Bruijn. 1983. AUTOMATH, a Language for Mathematics. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 159–200. Springer, Berlin, Heidelberg.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg. Springer-Verlag.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob Von Raumer. 2015. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 378–388. Springer International Publishing, Cham.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, Olli Järviniemi, Matthew Barnett, Robert Sandler, Matej Vrzala, Jaime Sevilla, Qiuyu Ren, Elizabeth Pratt, Lionel Levine, Grant Barkley, and 5 others. 2024. FrontierMath: A Benchmark for Evaluating Advanced Mathematical Reasoning in AI. *Preprint*, arXiv:2411.04872.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. In *Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Kotaro Komatsu. 2010. Counter-examples for refinement of conjectures and proofs in primary school mathematics. *The Journal of Mathematical Behavior*, 29(1):1–10.

Yinghui Li, Jiayi Kuang, Haojing Huang, Zhikun Xu, Xinnian Liang, Yi Yu, Wenlian Lu, Yangning Li, Xiaoyu Tan, Chao Qu, Ying Shen, Hai-Tao Zheng, and Philip S. Yu. 2025. One Example Shown, Many Concepts Known! Counterexample-Driven Conceptual Reasoning in Mathematical LLMs. *Preprint*, arXiv:2502.10454.

Pan Lu, Liang Qiu, Wenhao Yu, Sean Welleck, and Kai-Wei Chang. 2023. A Survey of Deep Learning for Mathematical Reasoning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14605–14631, Toronto, Canada. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Brendan D. McKay and Adolfo Piperno. 2014. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112.

Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. In *The Thirteenth International Conference on Learning Representations*.

A. Newell and H. Simon. 1956. The logic theory machine–A complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79.

Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. *Preprint*, arXiv:2009.03393.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *Preprint*, arXiv:2504.21801.

Alexandre Riazanov and Andrei Voronkov. 2001. Vampire 1.1. In *Automated Reasoning*, pages 376–380, Berlin, Heidelberg. Springer.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2018. Analysing Mathematical Reasoning Abilities of Neural Models. In *International Conference on Learning Representations*.

Shiven Sinha, Shashwat Goel, Ponnurangam Kumaraguru, Jonas Geiping, Matthias Bethge, and Ameya Prabhu. 2025. Can Language Models Falsify? Evaluating Algorithmic Reasoning with Counterexample Creation. *Preprint*, arXiv:2502.19414.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Adam Zsolt Wagner. 2021. Constructions in combinatorics via neural networks. *Preprint*, arXiv:2104.14516.

Yan Wang, Xiaojiang Liu, and Shuming Shi. 2017. Deep Neural Solver for Math Word Problems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 845–854, Copenhagen, Denmark. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

D.H. Wolpert and W.G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024. Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement. *Preprint*, arXiv:2409.12122.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *Advances in Neural Information Processing Systems*, 36:21573–21612.

A Figures

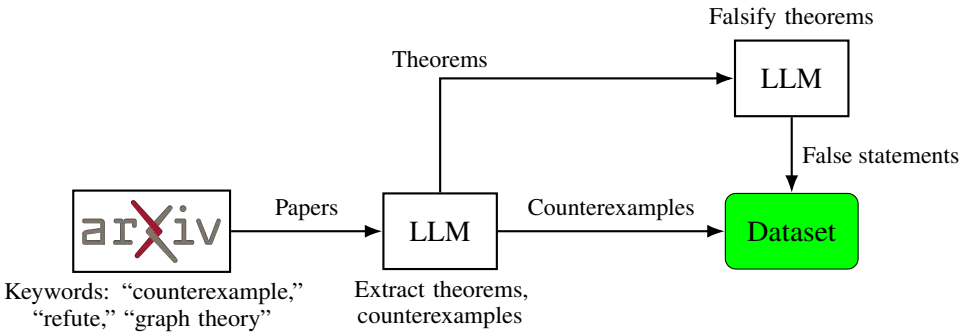

Figure 4: Attempted to scrape counterexamples and theorems from recent papers

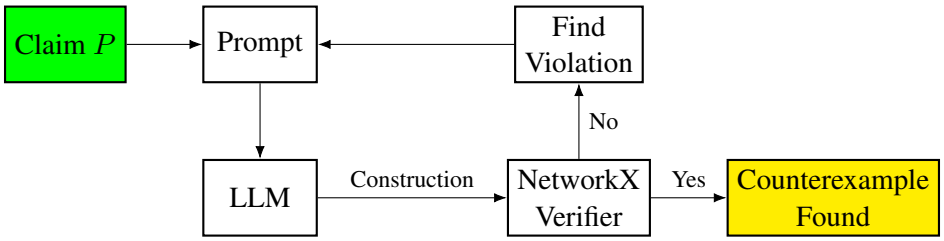

Figure 5: LLM Prompting Pipeline with Refinement Loop

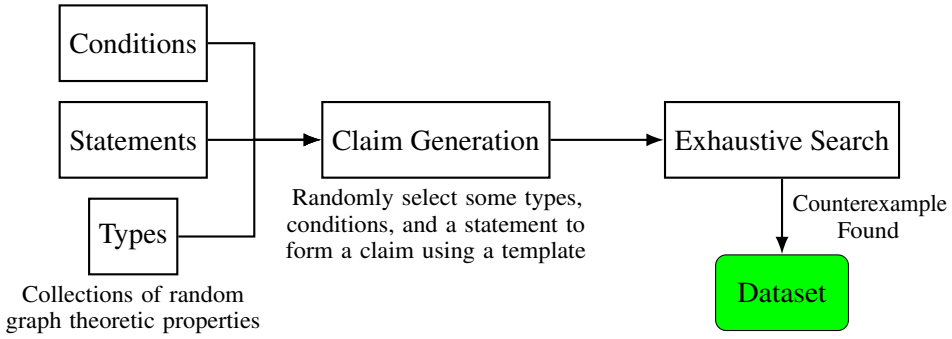

Figure 6: Generating random statements and verifying that they are false with an exhaustive search.

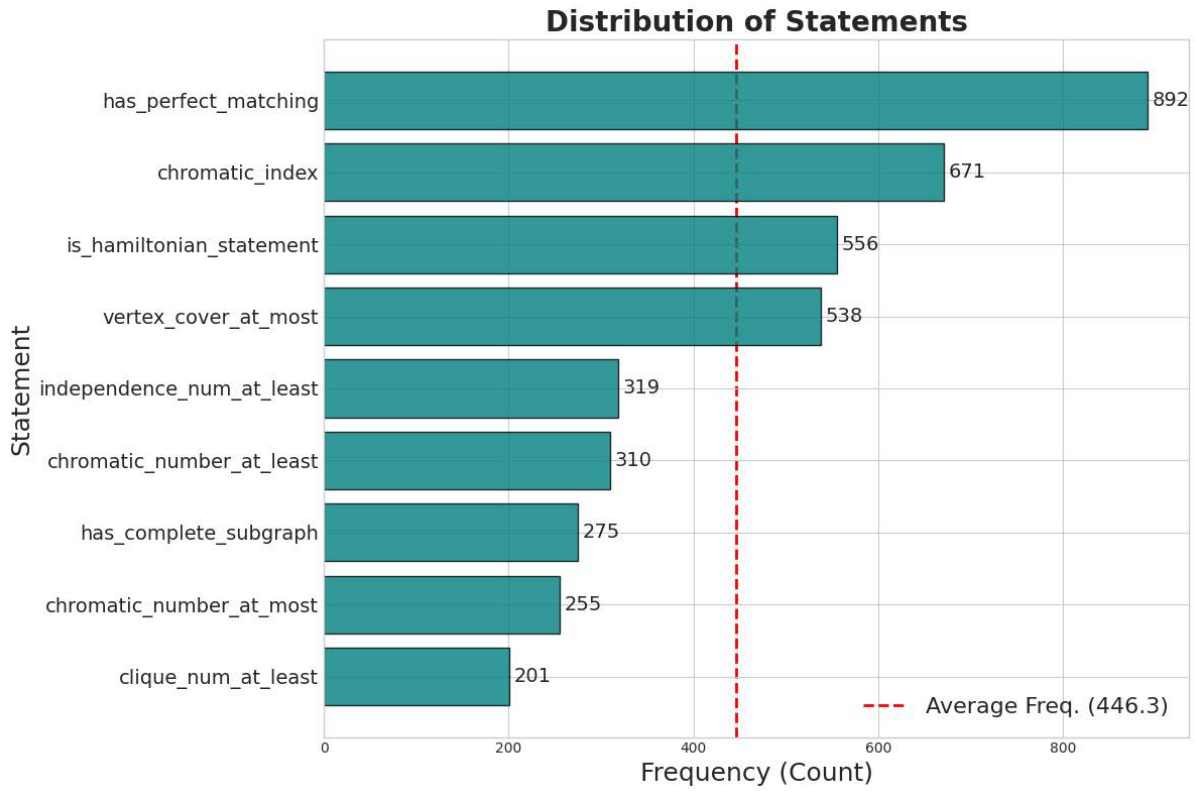

Figure 7: Distribution of Statements in the Current Iteration of the Dataset

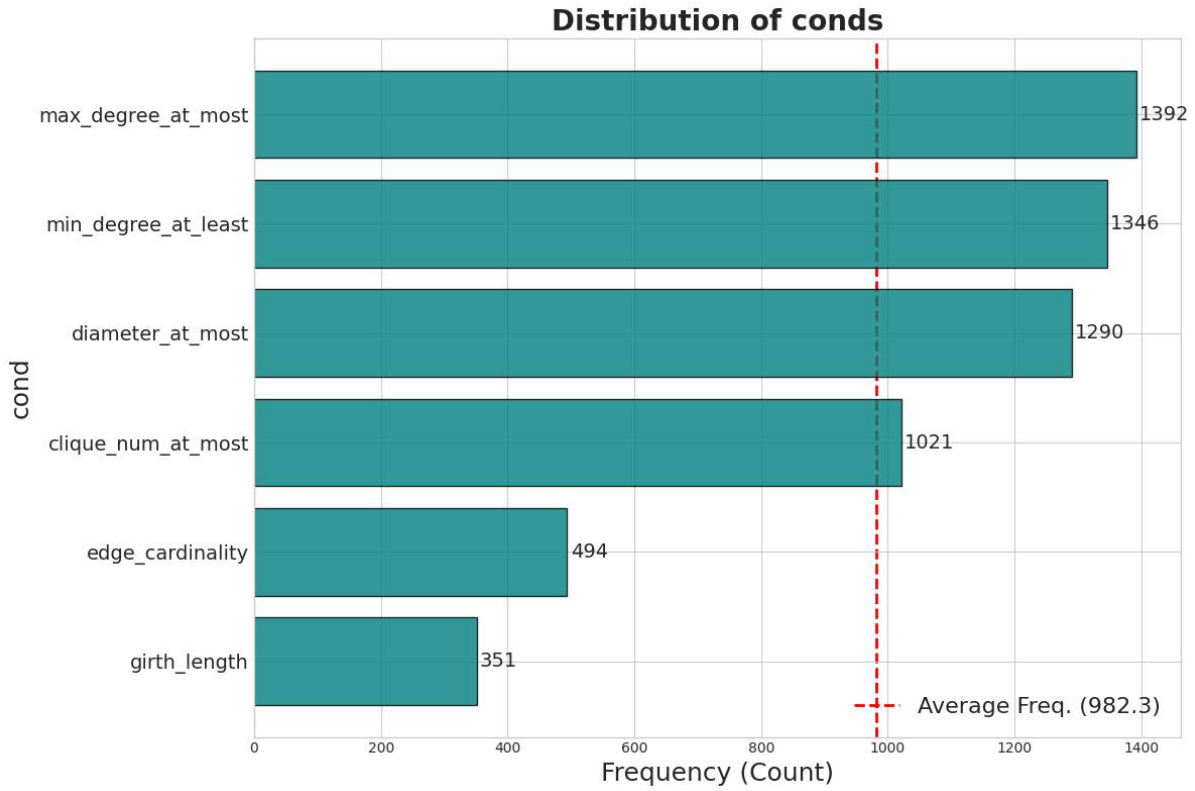

Figure 8: Distribution of Conditions in the Current Iteration of the Dataset

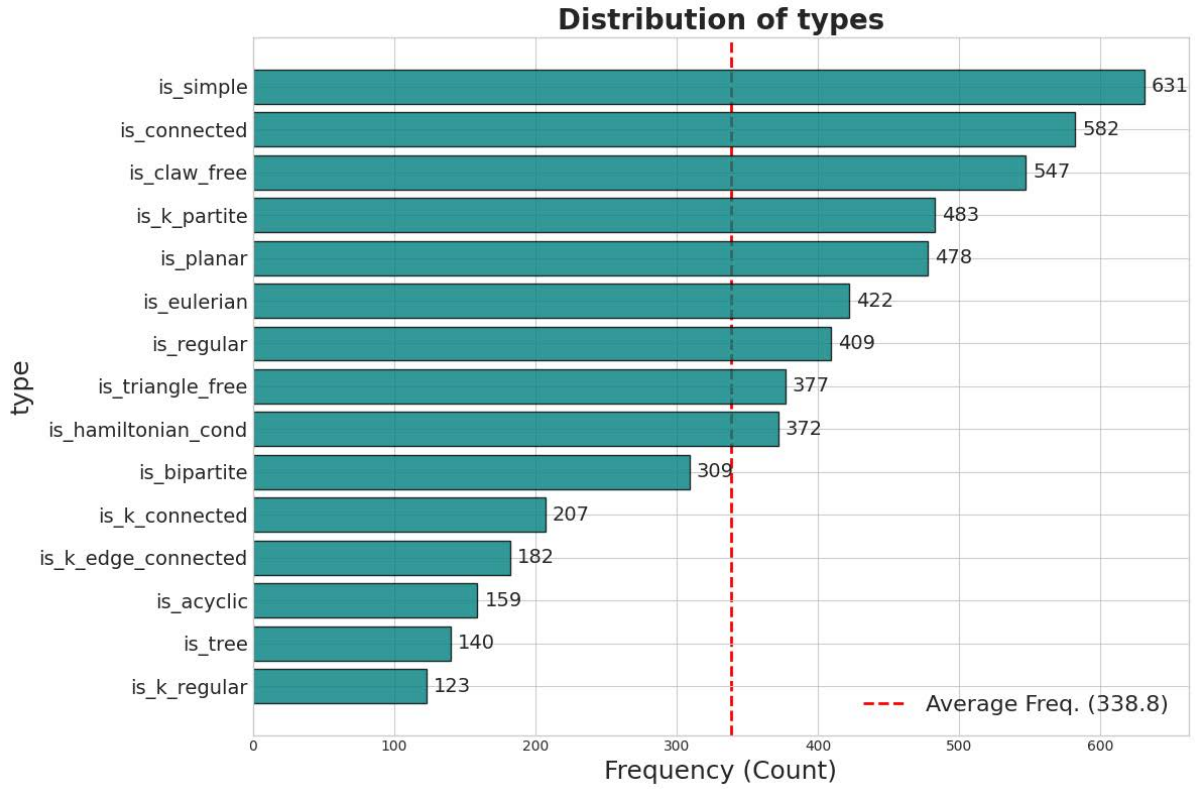

Figure 9: Distribution of Types in the Current Iteration of the Dataset

B LLM Claim Generation Prompt

```
## Role
You are an expert mathematician and creative graph theorist. Your task is to generate a plausible-
    sounding but **false** mathematical claim (a conjecture) in graph theory based on a given theme.
     The claim should be specific, non-trivial, and sound like a real mathematical conjecture.
    Claims should have a relatively large minimum counterexample if possible (10-15 vertices). Avoid
     claims with very small counterexamples (<6-7 vertices). Additionally, each part of the claim (
    each condition and the statement) should be relatively easily verified so an automated
    counterexample search can take place. Ensure all mathematical notation uses LaTeX commands, not
    unicode.

## Framework:
Your response must be structured according to the following formal framework:

A claim, denoted as P, is a pair (s, C).

C is a set of conditions. A condition is a property that a graph x must satisfy.

s is a statement. The statement is a property that is asserted to be true for any graph x that meets
    all the conditions in C.

### Example:
If the input theme is "Hamiltonian Paths," a valid claim is "All connected graphs contain a Hamilton
    cycle." Here is how you would break it down:

Claim: All connected graphs contain a Hamilton cycle.

Conditions (C):

The graph is connected.

Statement (s):

The graph contains a Hamilton cycle.
```

```
## Your Task:
Generate a false, non-trivial graph-theoretic claim on undirected graphs related to the user provided
     theme. Reason through the claim generation process step by step in order to generate a
    realistic claim. Provide the full claim in natural language, and then explicitly list the
    conditions (C) and the statement (s).

Additionally, there are two tasks you must complete once you have created your claim.
1. You must judge the difficulty of finding a counterexample to the claim on a scale of 1 (very easy)
     to 5 (very difficult). Once again, reason through the grading process in order to generate an
    accurate difficulty score.
2. You must determine a reasonable range for the number of vertices on which a counterexample may
    exist. Use your knowledge of graph theory to reason through possible sizes and output 5 possible
     values.

## Output:
Your final response should be in the following JSON schema.

‘‘‘json
{
   "claim": str,
   "conditions": list[str],
   "statement": str,
   "score": int,
   "counterexample_vertices": list[int]
}
‘‘‘
```

C LLM Predicate Generation Prompt

```
## Role
You are an expert in graph theory and a skilled Python programmer with deep knowledge of the ‘
    networkx‘ library. Your task is to translate a given graph-theoretic property into a Python
    predicate function that evaluates that property on a given graph.

## Framework
You will be provided with a single property of a graph, potentially containing LaTeX notation. Your
    goal is to create a Python function that takes a ‘networkx‘ graph object, ‘G‘, as input and
    returns ‘True‘ **only** if ‘G‘ satisfies the property, and ‘False‘ otherwise. There must be no
    false positives or false negatives.

### Instructions:
1.  **Function Definition:** Give the function a descriptive name. The name of all helper functions
    must be ‘helper_...‘. It must accept **only** a single argument, ‘G‘, which will be a ‘networkx‘
     graph object.
2. **Planning:** Before writing any Python code, use your knowledge of graph theory to plan out the
    structure of the function.
3.  **Implementation:** The function’s logic must accurately check for the given graph property. You
    should prioritize using the ‘networkx‘ library’s built-in functionalities for efficiency and
    correctness (Note: ‘networkx‘ does **not** contain a ‘chromatic_number‘, ‘is_k_connected‘ or ‘
    is_hamiltonian‘ function, but does contain helpful functions such as ‘girth‘, ‘is_regular‘, ‘
    adjacency_spectrum‘, ‘node_connectivity‘ etc.). You also have access to the ‘numpy‘ and ‘scipy‘
    library. The code should be self-contained and ready to execute, including any necessary imports
     from ‘networkx‘, ‘numpy‘ or ‘scipy‘. Ensure that all functions needed are imported and used
    correctly (Note: ‘networkx‘ has functions which are only accessible from submodules such as ‘nx.
    approximation.max_clique‘).

## Input Example:
You will receive the property as a single natural language string. Ex.

‘The graph is connected.‘

## Output Structure:
Your final output must adhere strictly to the following structure. Include very few comments other
    than a docstring.

**Python Predicate Function:**
‘‘‘python
[Your complete Python predicate function here.]
‘‘‘
```

D LLM Scoring Function Generation Prompt

```
## Role
You are an expert in graph theory and a skilled Python programmer with deep knowledge of the ‘
    networkx‘ library. Your task is to translate a given graph-theoretic claim into a **Python
    scoring function** for the purpose of *Counterexample Search*. This function will evaluate how
    close a given graph is to being a counterexample to the claim.

## Framework
You will be provided with a single natural language graph-theoretic claim, potentially containing
    LaTeX notation. Your goal is to create a Python function that takes a ‘networkx‘ graph object, ‘
    G‘, as input and returns a **fine-grained score as a float** based on closeness to being a
    counterexample. A graph is a counterexample of a claim if it satisfies all conditions, but
    violates the statement. The score should be in the interval [0, infinity).

A score **>= 1.0** must mean the graph is very likely a true counterexample to the claim. A score
    less than 1.0 should quantify how "close" the graph is to being a counterexample, with values
    close to and above 1.0 should indicate greater closeness, and scores closer to 0 should indicate
     that the graph is very far from being a counterexample to the claim.

### Instructions:
1.  **Function Definition:** You must write a single function, helper functions are allowed. Give the
     functions descriptive names. The names of all helper functions must start with ‘helper_‘. The
    main scoring function must accept **only** a single argument, ‘G‘, which will be a ‘networkx‘
    graph object, and it must return a ‘float‘.
2.  **Devise a Metric:** Before writing any Python code, use your knowledge of graph theory to plan
    the scoring logic. Think critically about how to measure "closeness" for the given claim. For
    example, if the claim states the graph is connected, closeness could be inversely related to the
     number of connected components. During this step, also consider efficiency. This function will
    be called many times, so avoid exponential time algorithms.
3.  **Implementation:** The function’s logic must accurately implement the scoring metric you devised.
     You should prioritize using the ‘networkx‘ library’s built-in functionalities for efficiency
    and correctness (Note: ‘networkx‘ does **not** contain a ‘chromatic_number‘, ‘is_k_connected‘ or
     ‘is_hamiltonian‘ function, but does contain helpful functions such as ‘girth‘, ‘is_regular‘, ‘
    adjacency_spectrum‘, ‘node_connectivity‘, etc.). The code should be self-contained and ready to
    execute, including any necessary imports from ‘networkx‘, ‘scipy‘ or ‘numpy‘. Ensure that all
    functions needed are imported and used correctly (Note: ‘networkx‘ has functions which are only
    accessible from submodules such as ‘approximation.max_clique‘).

## Input Example:
You will receive the claim as a single natural language string. Ex.

‘All connected graphs contain a Hamilton cycle.‘

## Output Structure:
Your final output must adhere strictly to the following structure. Include very few comments other
    than a docstring.

**Python Scoring Function:**
‘‘‘python
[Your complete Python scoring function here.]
‘‘‘
```

E LLM Counterexample Generation Prompt

```
You are an LLM assistant which specializes in graph theory.
Your task is to construct **explicit counterexamples** to user-supplied claims about finite
    undirected graphs and to return each counterexample solely as an adjacency-matrix representation.


---

1. **Input you will receive**

   * Input will be given in the following format:
     * "Let $G$ be a [CONDITIONS] graph on $x\\leq n \\leq X$ vertices such that [CONDITIONS]. Then
         [STATEMENT]."
   * A claim is a set of conditions and a statement. A counterexample to a claim satisfies all
       conditions but violates the statement.
```

```
2. **Parse the claim**

   * Identify all conditions asserted (order, connectivity, planarity, Hamiltonicity, chromatic
       number, etc.).
   * Identify the meaning of all notation used.
   * Identify the statement and negate it to determine what a counterexample looks like.

3. **Search for a Counterexample**

   * **Leverage Known Graphs:** Consider well-known graphs that are famous for being counterexamples
       or having specific properties.
     * **Complete Graphs ($K_n$):** Useful for claims involving cliques, colorings, and edge density.

     * **Complete Bipartite Graphs ($K_{m,n}$):** Especially the "star graphs" ($K_{1,n}$) and the
         utility graph ($K_{3,3}$). These are critical for testing claims about bipartiteness,
         planarity, and cycles.
     * **Cycle Graphs ($C_n$):** Testbeds for claims about connectivity, Hamiltonicity, and
         chromatic number. Odd cycles are not bipartite.
     * **Path Graphs ($P_n$) and Wheel Graphs ($W_n$):** Simple structures for testing basic
         properties.
     * **Platonic Solid Graphs:** The graphs of the tetrahedron, cube, octahedron, dodecahedron, and
          icosahedron are highly symmetric and have distinct properties related to regularity and
         connectivity.

   * **Constructive Approach:** If the claim involves multiple conditions (e.g., "bipartite and 3-
       regular"), try to build a graph that meets them. Start with a graph that satisfies the most
       restrictive condition and incrementally modify it to satisfy the others. During this process,
       constantly check if the statement is being violated.

   * **Heuristic Checks:**
     * **Connectivity:** Does removing a vertex or edge disconnect the graph? Check for cut vertices
          and bridges.
     * **Degree Sequence:** What do the degrees of the vertices tell you? For instance, Dirac's
         theorem gives a sufficient condition for a Hamiltonian cycle based on minimum degree. If
         the claim is about Hamiltonicity, look for a graph that *just* fails to meet this condition.

     * **Girth:** What is the length of the shortest cycle? This is crucial for claims about
         bipartiteness (a graph is bipartite if and only if it has no odd cycles).
     * **Planarity:** Can the graph be drawn without edge crossings? Test for minors of $K_5$ and
         $K_{3,3}$ (Kuratowski's theorem).

   * By systematically applying these search strategies, you will increase the likelihood of
       efficiently identifying a valid counterexample. Always prioritize the smallest valid
       counterexample you can find.

4. **Verify correctness**

   * Internally prove to yourself (e.g., by reasoning or quick algorithmic checks) that
     * All stated conditions are met, and
     * The statement fails for this graph.
   * If every finite graph satisfying the constraints fulfills the claim, output "No counterexample
       exists."

5. **Prepare output**

   * Convert the counterexample graph found to an adjacency matrix.
   * An adjacency matrix is a matrix with dimensions $n \times n$ where $n$ is the number of vertices
        in the graph, and the $i,j$-th entry in the matrix is a 1 if there is an edge between vertex
       $i$ and vertex $j$ and is 0 otherwise.

---

Use these rules exactly to generate adjacency-matrix counterexamples for any graph-theory claim the
    user supplies. Please reason step by step, and put your final answer within \\boxed{}.
```

Generating Hazard Reports from Video Using Vision-Language Models (VLMs)

Willem S. De Veirman
California Polytechnic State University,
San Luis Obispo, CA, USA
willemdeveirman@me.com

Ali K. AlShami
University of Colorado,
Colorado Springs, CO, USA
aalshami@uccs.edu

Jugal K. Kalita[†]
University of Colorado,
Colorado Springs, CO, USA
jkalita@uccs.edu

Abstract

Understanding and responding to unusual road behavior is critical for ensuring safety in autonomous driving. With advancements in artificial intelligence (AI) and Vision-Language Models (VLMs), our work focuses on generating detailed reports that describe what occurred before and after an unexpected or hazardous event, as well as identifying the potential cause. Our pipeline first detects unusual behavior on the road, then localizes the hazard, identifies it in a zero-shot manner, and finally feeds this contextual information into VLMs to generate comprehensive and informative reports. These reports offer valuable insights for law enforcement, insurance companies, and autonomous vehicle developers, providing answers to key questions and offering context that supports informed decision-making and accountability. Different VLMs have been used, including Qwen and LLaVA, to generate reports with the CHALLENGE OF OUT-OF-LABEL IN AUTONOMOUS DRIVING (COOOL) benchmark. Quantitative evaluation shows that the generated reports achieve a mean CLIPScore of 0.3015 and an overall accuracy of 0.66423 on the public COOOL leaderboard, surpassing the official baseline and demonstrating high factual alignment and fluency.

1 Introduction

Road crashes claim *1.19 million lives every year* and injure many millions more, making traffic injury one of the leading causes of death world-wide (World Health Organization, 2023). Autonomous-driving technology promises to reduce that toll, yet even the most advanced perception stacks still struggle with unforeseen road hazards—an elk darting from the treeline, a mattress sliding off a pickup, or a cyclist swerving to avoid debris. When such anomalies occur, society needs clear, auditable accounts: (i) *how* the incident unfolded on the road, (ii) *who* reacted (or failed to)—critical for insurers and investigators, and (iii) *why* the accident occurred—essential for regulators shaping safety policy. Today, compiling that narrative still demands hours of manual video review by human analysts.

We tackle the challenge of automatic hazard-event reporting: given a short dash-cam clip, output a summary that states what the driver saw, what happened, and how the situation resolved. The recently-released COOOL benchmark (AlShami et al., 2024) formalizes the task as three computer-vision (CV) subtasks—detecting the driver-reaction frame, localizing the hazardous object, and producing a caption—but stops short of assembling the full report. Existing solutions rely on large supervised detectors or additional sensors (speed, LiDAR, audio) that are expensive to annotate or unavailable in legacy footage.

We ask: *How far can we go with no task-specific training?* Inspired by the surge of open-weight VLMs that can reason about arbitrary images, we design an interpretable pipeline that combines classical vision cues with modern VLMs. Unlike task-specific pipelines, our zero-shot approach does not rely on domain-specific training data and thus *generalizes better to previously unseen hazards*, which are common in real-world driving. Every step can be inspected and no parameter is fitted to COOOL.

Outline and Contributions. Figure 1 previews the four-stage architecture.

1. **Reaction Detection** (Section 3.1). Dense optical flow and object-size dynamics are analyzed with kernel change-point detection to mark the first driver response frame—no thresholds, no learning.

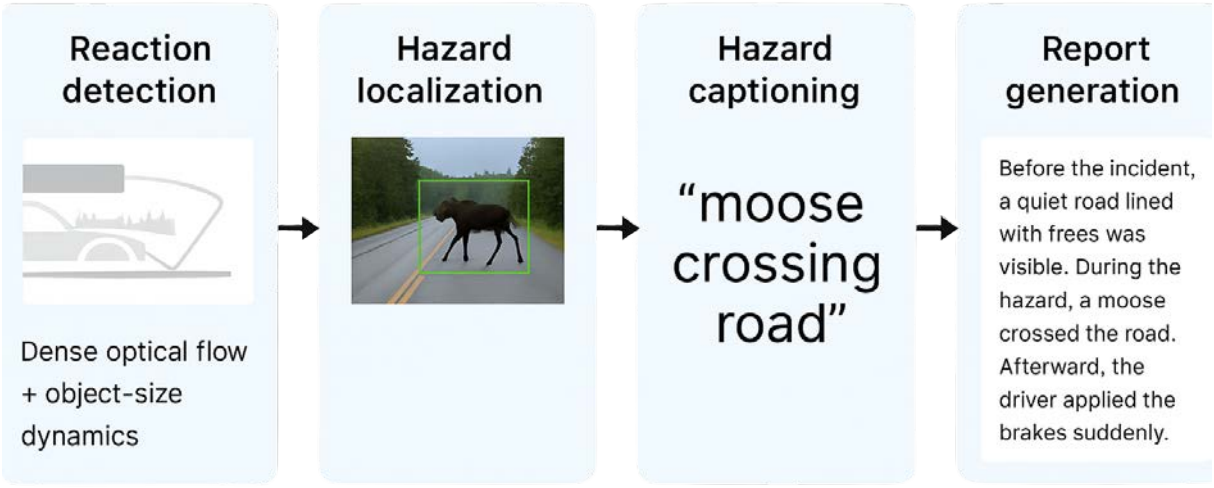

Figure 1: Four-stage zero-shot pipeline for hazard-event reporting. Dense optical flow and object-size dynamics detect the driver's reaction; weak-cue voting localizes the hazard (green box); a VLM (MOLMO) generates a concise caption (*"moose crossing road"*); and another VLM composes the final report.

2. **Hazard Localization** (Section 3.2). We fuse weak heuristics (proximity, motion deviation, looming, persistence, semantic novelty) via a differentially-private noisy-voting scheme that yields a stable, interpretable hazard box.

3. **Hazard Captioning** (Section 3.3). A zero-shot VLM receives the hazard crop and returns a short description, steered by prompt engineering to favor precise nouns (*"moose"*) over vague ones (*"animal"*).

4. **Report Generation** (Section 3.4). Three key frames (*before*, *during*, and *after*), the caption, and the reaction timestamp are passed to a multimodal VLM which composes a fluent accident report.

Evaluated on the public COOOL leaderboard, our training-free pipeline is on par with the best published systems, reaching 0.66423 overall accuracy and a 0.3015 mean report CLIPScore. Beyond metrics, the generated narratives offer immediate value to engineers, insurers, and law-enforcement professionals who need rapid, reliable insight into hazardous events.

The remainder of the paper details the methodology (Section 3), presents quantitative and qualitative results (Section 4.1), and analyzes strengths, limitations, and future directions (Section 5).

2 Related Work

Open-World Hazard Perception. Standard driving datasets (KITTI, nuScenes) assume a closed set of object classes; models trained on them struggle with unseen hazards. *Open-set recognition* (OSR) extends closed-set classifiers by calibrating confidence or modelling distribution tails, e.g. OpenMax (Bendale and Boult, 2016). *Out-of-distribution* (OOD) detectors instead flag samples that fall outside the training manifold. The COOOL benchmark (AlShami et al., 2024) combines OSR and OOD challenges by introducing rare hazards—kangaroos, tire debris, plastic bags—to stress-test perception systems.

Driver-Reaction Detection and Accident Anticipation. Anticipating collisions from video has been addressed with supervised deep networks that regress time-to-accident from optical flow and attention maps (Chan et al., 2016; Karim et al., 2020). Unsupervised alternatives locate reaction onsets by peaks in speed, sound, or motion: Duong and Gómez-Krämer (2025) fuse vehicle deceleration with auditory cues, while Picek et al. (2025) average optical flow and object-size change points. Gaze datasets such as DADA-2000 (Fang et al., 2022) confirm that abrupt motion spikes correlate with human attention during accidents.

Hazard Localization. Rule-based ensembles have proved effective without supervision. Duong and Gómez-Krämer (2025) rank object tracks by heuristics (distance, motion, sudden appearance)

and use differentially-private (DP) noisy weights to vote for the hazard. Picek et al. (2025) apply a zero-shot semantic filter—any object classified by a Vision Transformer (ViT) as a common vehicle or sign is discarded—to focus on novel threats. Subsequent works combine spatial, kinematic and semantic cues, demonstrating that no single heuristic suffices in open-world scenes.

Hazard Captioning with VLMs. The COOOL baseline employs CLIP-Interrogator (CLIP (Radford et al., 2021) + BLIP-2 (Li et al., 2023)) to caption cropped hazard images. Hatami et al. (2025) refine BLIP-2 captions with depth and road masks, whereas Picek et al. (2025) show that prompting the multimodal model MOLMO-7B (Deitke et al., 2025) for a list of noun labels improves accuracy because the metric rewards correct keywords. Recent captioners therefore favor short, noun-focused outputs (e.g. "*moose on road*") over florid sentences.

VLMs for Accident Reporting. VLMs can turn multimodal evidence into structured incident narratives. GPT-4 compiled sensor-augmented accident reports in Ahmadi et al. (2024). Zhang et al. (2025) proposed SEEUNSAFE, where a multimodal VLM agent reasons over traffic-camera video to output event type, context and explanations. These studies highlight the potential of VLMs to fuse visual cues and domain knowledge into actionable prose.

Our work builds on these strands—OSR/OOD perception, motion-based reaction detection, heuristic hazard voting, VLM captioning, and VLM report writing—combining them into a single zero-shot pipeline for COOOL.

3 Methodology

Figure 1 illustrates our overall pipeline, which operates in four stages to convert a video into a hazard report. First, we detect when the driver reacts to an anomaly. Second, we determine which object caused that reaction. Third, we generate a short caption describing the hazard. Finally, we compile a three-sentence report of the event using VLMs. All components are zero-shot (training-free) and designed to be interpretable. We describe each stage in detail below.

3.1 Driver-Reaction Frame Detection

Optical Flow and Object-Size Dynamic Signals. We downsample each video to 5 fps to reduce noise and computation. Between consecutive frames, we compute dense optical flow using Farnebäck's algorithm (Farnebäck, 2003) as implemented in OpenCV. The mean magnitude of the flow field M_t at frame t provides a coarse measure of camera or scene motion:

$$M_t = \frac{1}{|\Omega|} \sum_{(x,y)\in\Omega} \sqrt{u_{x,y}^2 + v_{x,y}^2},$$

for optical flow vectors $(u_{x,y}, v_{x,y})$ over all pixel locations Ω. Sharp peaks in M_t indicate sudden acceleration, braking, or a rapid movement in the scene (e.g. a hazard appearing). However, optical flow alone can be sensitive to ego-vehicle jolts or camera shake, causing false positives. We therefore introduce a complementary cue: the Object-Size Dynamic (OSD). Using the provided object tracks, we calculate $A_t = \sum_i \text{area}(b_{i,t})$ as the total pixel area occupied by all annotated challenge objects at frame t. This approximates the inverse distance of important objects—when a hazard object gets closer (looms larger), A_t increases. OSD is immune to camera motion that does not bring objects closer, so it stays flat on events like rough road bumps, but responds strongly to true looming threats outside the vehicle.

Change-Point Detection. We independently normalize M_t and A_t to $[0, 1]$ across the video and then apply an offline change-point detector to each time series. We use the Kernel Change Point Detection (KernelCPD) algorithm (Truong et al., 2020) with a Gaussian Radial Basis Function (RBF) kernel. KernelCPD finds the earliest frame where the distribution of the signal changes significantly. Intuitively, it identifies the onset of a sustained change in motion or object growth. Let b_{flow} be the first change-point in the optical flow series and b_{OSD} that in the OSD series. We take the driver reaction frame prediction as the average of the two:

$$t^* = \left\lfloor \frac{b_{\text{flow}} + b_{\text{OSD}}}{2} \right\rfloor,$$

which is similar to the mean ensemble proposed by Picek et al. (2025). By fusing flow and OSD, we mitigate their individual failure modes—e.g. in a shaky scene, b_{flow} might trigger early but b_{OSD} will not, and vice versa for a lateral-moving hazard with

little size change. The dotted black line in Figure 3 illustrates how this ensemble aligns well with the true reaction moment across diverse scenarios. Our approach requires no threshold tuning, unlike earlier methods that hard-coded motion variance limits or relied on learned confidence scores.

3.2 Hazard Object Localization

Once the reaction frame t is known, we restrict our attention to objects present near that time. The goal is to determine which object(s) caused the reaction. We generate an initial set of candidate hazards consisting of all annotated objects that satisfy either of two criteria: *(a)* the object is spatially central (its bounding box centroid within 20% of the frame center) at frame t, or *(b)* the object exhibits a sudden increase in size or velocity in the 1–2 seconds preceding t^*. Criterion *(a)* captures the intuition that an immediate threat is likely in front of the vehicle, while *(b)* captures objects that might not be centered but whose unusual motion could draw attention (for example, a bike swerving in from the side). For each candidate track j, we compute a feature vector $\mathbf{f}_j \in \mathbb{R}^8$ inspired by Duong and Gómez-Krämer (2025):

(i) Horizontal position span (normalized width of the track's trajectory in the frame),
(ii) Inverse distance from image center (average $\frac{1}{d}$ of centroid to image center, emphasizing near-center objects),
(iii) Road adherence (fraction of the track's frames where the object lies within a predefined road area in the image, e.g. below horizon and near lane center),
(iv) Visibility ratio (fraction of video frames in which the object is present, smaller for objects that appear suddenly),
(v) Size trend (difference between the object's bounding box area at t^* and 1 second earlier, capturing looming),
(vi) Motion deviation (whether the object's direction of travel is against the ego-vehicle's direction, a binary feature),
(vii) Classification penalty (whether the object is classified as a known safe class like *car*, *truck*, etc.; this is 0 for safe classes, 1 for others),
(viii) Constant bias (set to 1 as a baseline vote).

Features (i)–(vi) are similar to the rules in prior work, and (vii) introduces semantic knowledge as in Picek et al. (2025). The classification for (vii) comes from a ViT-based image classifier: we feed the largest cropped instance of each object into a ViT (pre-trained on ImageNet, fine-tuned on CIFAR-100), and take the top predicted label. If this label is one of *car*, *bus*, *truck*, *motorcycle*, *bicycle*, *traffic light* (common non-hazardous objects), the penalty is 0; otherwise 1. This step filters out objects that are likely part of normal traffic.

DP Voting Ensemble. Given feature matrix $F = [\mathbf{f}_1, \ldots, \mathbf{f}_m]^T$ for m candidate objects, we adopt the differential privacy (DP) weight sampling approach of Duong and Gómez-Krämer (2025) to combine these features robustly. We define a nominal weight vector $\mathbf{w}$ (8-dim) that reflects our initial heuristic weighting of each feature's importance. Rather than using $\mathbf{w}$ directly, we sample $K = 100$ noisy weight vectors:

$$\tilde{\mathbf{w}}^{(i)} \sim \mathcal{N}\big(\mathbf{w},\, \sigma^2 I\big) \quad \text{for } i = 1, \ldots, K,$$

with σ tuned such that the sampling obeys an ε-DP budget (we use $\varepsilon = 1.0$) following Duong and Gómez-Krämer 2025). For each sample i, we compute a hazard score for each object: $\mathbf{s}^{(i)} = F, \tilde{\mathbf{w}}^{(i)}$, and select the top-scoring object. Every object that either scores above 0 or is the top object in that sample is considered a "vote" for being hazardous in that iteration. Finally, we count votes across the 100 trials and declare any object with ≥ 50 votes as a hazard. In practice, usually one object (occasionally two) will surpass this majority threshold, aligning with the expectation of a single primary hazard. The DP noise ensures that no single feature dominates the decision: e.g., even if the center distance strongly favors an object, if other cues disagree (it was present throughout, moving normally, and classified as *car*), many sampled $\tilde{\mathbf{w}}$ vectors will downplay the center feature and cause another object to win in those trials. By requiring a stable majority, we suppress spurious detections due to any one unreliable heuristic. This also makes the system more resilient to the lack of ground-truth tuning—our $\mathbf{w}$ is set manually (giving slightly higher weight to features (ii), (iv), (v) based on intuition), but exact values are not critical under the noise regime.

In summary, we output a set of hazardous object IDs per frame (often just one). Figure 3 provides a visualization, where green boxes highlight the predicted hazard and red boxes denote other objects.

3.3 Hazard Caption Generation

For each identified hazard track, we generate a brief caption describing it. Rather than training a captioner, we use the pretrained MOLMO-7B VLM (Deitke et al., 2025) in a zero-shot manner. We extract up to 5 images of the object: specifically, the frames where the object's bounding box is largest (typically near t^*). These cropped images (padded to square) are fed into MOLMO with two types of prompts:

- **Class list prompt:** "List five likely class labels for the unusual object in this traffic scene."
- **Sentence prompt:** "In a traffic scenario, write one short sentence (max 6 words) describing the hazard."

The class list prompt makes MOLMO act like a classifier and output keywords (e.g. *"dog deer cow horse elk"*), whereas the sentence prompt produces a more natural phrase (e.g. *"deer running across road"*). We found that the class list often covers relevant nouns, and the sentence ensures context (like *"on road"*, *"crossing"*, etc.). We post-process MOLMO's outputs by taking the union of the top 5 class words and the sentence, then selecting a concise phrase up to 35 characters that includes the most relevant terms. For the example in Figure 3, we would form the caption *"dog crossing road."* This approach of mixing categorical and descriptive prompts is influenced by Picek et al. (2025), who noted that focusing on nouns can improve the official metric. The final captions are lowercased and stripped of punctuation to match the evaluation format, which checks for substring matches against ground truth phrases. Despite not being fine-tuned on the COOOL data, MOLMO reliably generated appropriate descriptions for even obscure hazards (e.g. *"kangaroo on road"*, *"flying tire debris"*). In a few cases of very tiny distant objects, it produced generic terms like *"animal"* or *"object"*, which, while not specific, were usually sufficient to get partial credit under the evaluation.

3.4 Hazard-Event Report Generation

The final stage turns the structured outputs from the previous steps into a fluent, human-readable report. We exploit multimodal VLMs that can attend to a set of images plus a textual prompt. In all experiments we use open-weight models runnable on a single GPU:

- **Qwen-VL-Chat-7B** (Bai et al., 2023), and

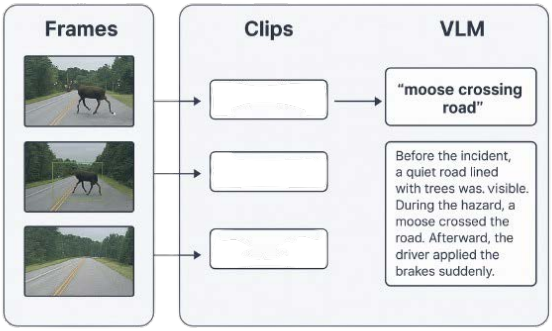

Figure 2: Three–Clip visual context used by our pipeline. From each video we extract a *before*, *during*, and *after* frame (left). Short clips around those key frames are fed to a VLM in two passes: first for hazard captioning (yielding *"moose crossing road"*), then—together with the reaction timestamp—for report generation, producing a three-sentence summary.

- **LLaVA-v1.5-7B** (Liu et al., 2023). [1]

Three-Clip Visual Context. Figure 2 illustrates the pipeline that turns a raw dash-cam video into a hazard report. For every sequence we cut three key frames identified by Section 3.1—the midpoint of the *before* window, the peak (*during*) frame, and the midpoint of the *after* window—and wrap a ± 0.5 second clip around each. This fixed three-clip structure keeps the prompt compact while still showing the VLM the full temporal arc of the incident. The first VLM (MOLMO as described in Section 3.3) call produces a concise hazard caption from the *during* clip; the second call, conditioned on all three clips plus the caption and reaction time, composes the final three-sentence accident report.

Prompt Engineering and Rationale. Although the official caption score $\mathbf{Acc_{cls}}$ (defined in Section 4.1) ignores fluency, we cast the system role as a *traffic-accident analyst* to keep the model's vocabulary on-topic and to enforce the three-sentence template. This analyst framing trims off-topic generations, boosts regular expression compliance, and biases the VLM toward domain terms (*brake*, *swerve*, *pedestrian*) rather than generic captions.

We tested two prompt styles:

(a) Minimal prompt

```
System: "You are a concise
traffic-accident analyst. Produce
```

[1] Both checkpoints are publicly available; no fine-tuning was performed.

```
exactly three sentences. Sentence 1
starts with 'Before the incident,';
Sentence 2 with 'During the hazard,';
Sentence 3 with 'Afterward,'."
User: [Clip_before] [Clip_during]
[Clip_after] "Describe what happens."
```

(b) Guided prompt extends the minimal version with style rules, a 35-token limit, and the detected reaction time (e.g., *"The driver begins reacting at t^* s."*).

In practice, all VLMs tested follow the guided prompt more reliably, so we use it for all reported results.

Output Post-Processing. A $3\times$ regular expression filter enforces:

1. exactly three sentences;
2. required starters (*Before/During/Afterward*); and
3. maximum 35 tokens total.

Invalid generations are re-prompted once with a shorter temperature-0.2 request (fallback decoding). The accepted reports are written to a JSON dictionary keyed by video ID, e.g.:

```
"video_0002": "Before the incident, a quiet road shows
damp pavement beneath overhanging trees. During the hazard,
a moose suddenly crosses the lane, forcing immediate braking.
Afterward, drivers slow and swerve to avoid collision."
```

Runtime. On an RTX 4090, Qwen-VL-Chat needs ≈ 3.4 s per video and LLaVA-v1.5 ≈ 2.8 s (batch size 1, FP16), so the full 200-video test set is processed in ≈ 12 minutes.

4 Experiments

4.1 Datasets and Metrics

We evaluate on the COOOL benchmark, which consists of 200 dashcam videos (roughly 5–15 seconds each) with frame-by-frame annotations of all "challenge" objects and important events. The organizers partition the 200-video test set into a *public slice* ($\approx 8\%$) whose metrics are returned for sanity-checking, and a *private slice* ($\approx 92\%$) whose annotations remain hidden. We perform all ablations and prompt tuning on the public slice only, then submit our predictions to the official evaluation server; the server computes the metrics with the withheld labels and reports the final leaderboard scores for the private slice.

The evaluation metrics are:

- **$\text{Acc}_{\text{react}}$:** the fraction of frames where the predicted driver reaction state matches the ground truth (essentially accuracy of a binary sequence where 0 means no reaction yet and 1 means reaction started).
- **Acc_{det}:** hazard detection accuracy per frame, defined as $\frac{|H_i \cap \hat{H}_i|}{\max(|H_i|,|\hat{H}_i|)}$ averaged over frames, where H_i is the set of ground-truth hazard objects (the ones labeled as causing the event) at frame i and $\hat{H}_i$ is the set of predicted hazard objects.
- **Acc_{cls}:** hazard caption accuracy per frame, defined similarly as overlap of ground-truth vs. predicted caption words (after some normalization). In practice, credit is given if the predicted caption contains the correct hazard noun.

The final score **Acc** is the average of these three metrics. For our additional task (report generation), the COOOL benchmark does not provide ground-truth sentences, so we perform a separate evaluation. We use the CLIPScore metric (Hessel et al., 2021) as a reference-free measure of image-text alignment: it computes the cosine similarity between a CLIP visual embedding (of the video frame or hazard image) and CLIP textual embedding (of the generated report). This indicates how well the report describes the visual scene. We also perform a qualitative assessment of report correctness.

4.2 Implementation Details

Our optical flow and OSD computation is implemented in Python using OpenCV for flow and runs in real-time (5 fps) on a CPU, producing a pickle of motion scores and area time series (12 MB for all videos). For change-point detection, we use the `ruptures` library with default RBF kernel and a minimum segment length of 5 frames. The DP voting ensemble uses $\sigma = 0.3$ Gaussian noise on weights and is run 100 times per video; this is also fast (< 1 second per video) given the small number of tracks. For captioning, we use the official MOLMO-7B model implementation; generating outputs for all hazard crops in a video takes about 20 seconds on an NVIDIA RTX 4090 GPU (for fair comparison, BLIP2-based methods have similar inference time). For report generation, we use Qwen-VL-Chat-7B and LLaVA-v1.5-7B; each model generated three-sentences per video, and we logged the outputs for analysis. We

emphasize that none of our components require training or fine-tuning on COOOL. The only learned elements are the pre-trained models we employ (optical flow's algorithm, ViT classifier, MOLMO, and report VLMs). This makes our approach easily transferable to new domains.

4.3 Results on COOOL

Table 1 compares our method to the challenge baseline and two recent approaches. The baseline (provided by organizers) had a simplistic reaction detector based on object motion median and a center-box hazard selector; it achieved only 0.256 overall accuracy on public data. Our approach improves each component: $\mathbf{Acc_{react}}$ rises to 0.83446, confirming that the flow + OSD mean ensemble is highly effective. This aligns with Picek et al. (2025), who also reported ~ 0.83 on private for a similar ensemble. Notably, using only optical flow or only OSD was worse, but together they neutralize each other's failure cases (see Figure 3 and Section 5). Duong and Gómez-Krämer (2025) scored ~ 0.79 on $\mathbf{Acc_{react}}$, and our method does not rely on vehicle speed or audio, which may not always be available.

For hazard identification, we obtain $\mathbf{Acc_{det}} = 0.52755$, a substantial jump from the baseline's 0.22. This is above par with the top entry (0.572 by Duong and Gómez-Krämer 2025). We attribute this success to the combination of semantic filtering (removing known vehicles) and DP robust fusion: ablation showed that without the classification filter, our $\mathbf{Acc_{det}}$ dropped to 0.49322, and without DP noise (using fixed $\mathbf{w}$) it dropped to 0.48248, indicating each component's benefit. Interestingly, when we replaced our ViT classifier with MOLMO's class predictions to filter hazards (as an alternative zero-shot approach), performance was similar (0.52633). This suggests that large VLMs can potentially replace traditional classifiers for identifying novel objects.

Regarding hazard captioning, our method yields $\mathbf{Acc_{cls}} = 0.1276$. Modest in absolute terms, this far outperforms the baseline (0.016) and is in range with others (the winning entry Duong and Gómez-Krämer 2025 yielded ~ 0.17, and Picek et al. 2025 scored ~ 0.16). Our captions nearly always include the correct hazard noun, due to the MOLMO prompt strategy, whereas the baseline often missed it. The captioning metric is stringent: omitting a key modifier or using a generic term such as *"animal"* instead of the correct species reduces the score. Our hybrid prompt—requesting both class keywords and a short phrase from MOLMO—consistently retrieved the correct noun (e.g., *"bird"* rather than *"animal"*), improving caption precision. With this strategy, our system attains a final overall accuracy of $\mathbf{Acc} = 0.66423$, placing it among the top leaderboard submissions.

Method	$\mathbf{Acc_{react}}$	$\mathbf{Acc_{det}}$	$\mathbf{Acc_{cls}}$
Baseline (organizers)	0.584	0.221	0.009
Duong & Gomez-Krämer (2025)	0.785	0.572	0.173
Picek *et al.* (2025)	0.829	0.535	0.162
Our Pipeline	**0.83446**	0.52755	0.1276

Table 1: COOOL Challenge results on the test set. Our zero-shot fusion pipeline is competitive with top supervised or multimodal approaches on all three tasks.

VLM	Avg. CLIPScore
Qwen-VL-Chat-7B	0.298
LLaVA-v1.5-7B	0.305

Table 2: Comparison of VLMs used for report generation (200 videos). LLaVA yields higher image–text alignment (CLIPScore) and generally produces slightly more detailed sentences on average.

In addition to the official metrics, we evaluated the quality of our hazard reports. Using CLIPScore (higher indicates better image-text alignment), we found that LLaVA outperformed Qwen in generating accurate reports. Specifically, Qwen achieved an average CLIPScore of 0.298, while LLaVA achieved 0.305. This suggests that LLaVA descriptions aligned more closely with the actual video content. We also observed that LLaVA was more likely to include key details like the specific hazard and the driver's action. Table 2 summarizes these results. Based on this, our final pipeline uses LLaVA for report generation.

5 Analysis and Discussion

Case Study. Figure 3 illustrates a typical scenario (`video_0001` from COOOL). Early in the video, a camera shake (going over a bump) causes a surge in optical flow (blue curve) but OSD (orange curve) remains flat because no object size changed. Our ensemble correctly ignores this as a false reaction. Later, a pedestrian suddenly runs into the street: both flow and OSD spike sharply, indicating genuine hazard onset. The flow change-point b_{flow} is a few frames earlier than the OSD change b_{OSD}, but their mean falls exactly

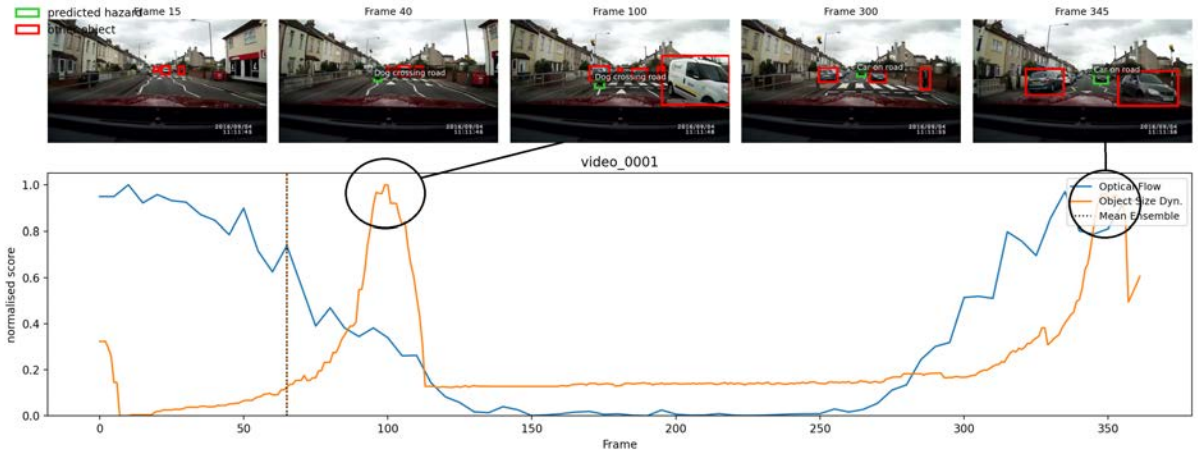

Figure 3: Representative scene from the COOOL dataset (`video_0001`).

on the true reaction frame (dotted line). In this moment, our hazard vote ensemble considered three objects: the pedestrian, a car ahead, and a sign. The pedestrian had high inverse-distance and motion-deviation scores and was not a known car, so it accumulated 100/100 votes and was tagged as the hazard (green box). The car and sign (red boxes) had some features triggered (the car was centered but moving predictably; the sign was novel but off-road), yet neither reached 50 votes. MOLMO was given the pedestrian crop and output labels including *"person, pedestrian"* and the phrase *"person crossing road"*, yielding the final caption *"pedestrian crossing road"*, which matched the ground truth description. Finally, LLaVA takes the reaction time and the caption, and produces the report: *"Before the incident, a quiet urban street was clear and vehicles moved steadily under the morning light. During the hazard ($\approx$ 6 s), a pedestrian suddenly ran into the roadway in front of the car, forcing the driver to brake sharply. Afterward, the pedestrian reached the opposite curb unharmed and traffic resumed its normal flow."* This three-sentence summary clearly communicates the hazard and the driver's response.

This case study highlights how each module contributes to the final result. The reaction detector provides the critical timing of the event, the localization zeroes in on the correct object, and the caption supplies the precise object identity. The VLM then combines these pieces into a fluent description. If any upstream module errs (e.g., picking the wrong object or timing), the final report would reflect that error (e.g., mentioning a wrong hazard or an incorrect time). In practice, we found that our reaction detector and localization were usually accurate enough to guide the VLM correctly. In a few failure cases, the pipeline identified a secondary object as the hazard (e.g., a swerving car instead of the debris it swerved for), leading the VLM to report a plausible but incorrect event. Addressing such cases may require further integration between the vision modules and the language model, possibly by having the VLM reason over multiple candidate hazards or verify consistency with the visual evidence. Overall, the modular nature of our system offers interpretability at each step. We can inspect the optical flow plots or feature weights to understand why an event was detected, or examine the VLM prompt and output to ensure no hallucinated details are introduced. This transparency is valuable for safety-critical applications.

Additional Observations. LLaVA superiority in report generation suggests that larger VLMs with better reasoning and adherence to instructions make a difference in generating accurate reports. Qwen occasionally omitted the timestamp or used vaguer language (e.g., *"the driver stops for a person on the road"*), which still conveys the gist but lacks specificity. In contrast, LLaVA was more likely to include precise details (e.g., *"brakes hard to avoid a dog"*). We also note that our use of a structured prompt (specifying the desired format) was crucial—without it, the VLM might produce overly verbose outputs or miss mentioning the reaction. Enforcing style consistency via the prompt and regular expression checks helped ensure all 200 reports followed a uniform pattern, which could be important if these

reports are to be parsed or compared automatically.

Ablations. We already discussed the importance of combining flow + OSD and using DP + classification in the ensemble (Section 4.3). Here we briefly note two other findings: (1) If we remove the OSD cue entirely and rely on just optical flow with KernelCPD, $\mathbf{Acc_{react}}$ drops by about 0.03–0.04. Conversely, using just OSD drops about 0.02. Thus both are useful, and optical flow is slightly more critical since it often triggers a bit earlier (OSD reacts only after an object has grown sufficiently). (2) If we use only the class-list prompt for MOLMO (five nouns) and not the sentence, $\mathbf{Acc_{cls}}$ actually improves to 0.1573. However, the output is then not a fluent caption (it is a list of nouns). Since the challenge metric does not require a well-formed sentence, one could do this to maximize score. Our approach balanced human-readability with score; we consider 0.1276 acceptable given we produce intelligible phrases.

Failure Modes. Some failure cases include: very dark scenes or glare where optical flow fails to pick up motion (leading to a missed reaction, though OSD might still work if the hazard had lights or reflective surfaces). Also, if the driver reacts preemptively to something not visible (e.g. hears something, or anticipates a hazard beyond the camera view), our vision-based detector will not catch it. In hazard localization, our method can confuse multiple hazards if they occur together (e.g. a falling tree hits a car – two objects). The DP voting may mark both as hazards, and while this still yields partial credit, it is not perfect. Finally, the captioning struggles with extremely small or blurry hazards, sometimes defaulting to *"object on road"* which is not specific enough.

6 Conclusion

We present a multi-cue fusion pipeline for open-world hazard detection and report generation, which achieved respectable results on the COOOL challenge without any task-specific training. By pairing simple vision signals (optical flow, object-size dyanamics) with an ensemble of human-interpretable heuristics and powerful VLMs for captioning and reporting, our system can robustly detect when a driver reacts, identify the cause, and articulate it in a human-readable report. Unlike task-specific pipelines, our zero-shot approach does not rely on domain-specific training data and therefore generalizes more gracefully to the novel hazards that dominate real-world driving.

Test domains for road safety evolve faster than labeled data can be curated: new geographies, weather, road furniture, and rare hazards (animals, debris types) continuously appear. The approach is computationally lightweight and inherently explainable, as each hazard prediction comes with a rationale (motion spike, object proximity, etc.) and the final description is grounded in detected facts. We also demonstrate the integration of VLMs for generating accident reports, showing that a few geberated sentences can capture the essence of a hazard scenario.

Limitations and Future Work. One limitation is the reliance on provided object detections/tracks. In a real deployment, one would need a reliable upstream detector; missing a hazard in the perception stage means our pipeline cannot reason about it. However, the zero-shot classification filter could itself help a detector reject known objects and highlight unknown ones in a semi-supervised loop.

Another limitation is the lack of cues like audio or speed input, which could signal certain reactions (horns, ABS braking noise); integrating multi-modal anomalies is a promising direction (as Duong and Gómez-Krämer 2025 implemented). In terms of report generation, our current VLM output is a three-sentence output focusing on the immediate event.

In the future, we plan to extend this to richer multi-sentence reports that include more context (weather, location, outcome), akin to a short accident narrative. This could involve using more advanced multimodal VLMs or fine-tuning on traffic incident data. Recent frameworks like SEEUNSAFE (Zhang et al., 2025) demonstrate the potential of interactive, structured prompts to extract detailed accident information. Building on these ideas, we envision an AI assistant that not only detects hazards in real time but also provides a coherent summary for drivers, engineers, or insurance analysts. We hope our work serves as a step toward that goal, bridging low-level perception and high-level reasoning in autonomous driving in the open world.

References

Hossein Ahmadi, Alice Smith, and Jugal Kalita. 2024. Llm-based automatic accident report generation. *IEEE Transactions on Intelligent Transportation Systems*.

Ali K. AlShami, Ananya Kalita, Ryan Rabinowitz, Khang Lam, Rishabh Bezbarua, Terrance Boult, and Jugal Kalita. 2024. COOOL: Challenge of out-of-label—a benchmark for open-world autonomous driving. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*.

Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen–VL: A versatile vision–language model for understanding, localization, text reading, and beyond. https://arxiv.org/abs/2308.12966. ArXiv:2308.12966.

Abhijit Bendale and Terrance E. Boult. 2016. Towards open set deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Fu-Hsiang Chan, Yu-Ting Chen, Yu Xiang, and Min Sun. 2016. Anticipating accidents in dashcam videos. In *Proceedings of the Asian Conference on Computer Vision (ACCV)*.

Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, Jiasen Lu, Taira Anderson, Erin Bransom, Kiana Ehsani, Huong Ngo, YenSung Chen, Ajay Patel, Mark Yatskar, Chris Callison-Burch, and 31 others. 2025. Molmo and pixmo: Open weights and open data for state-of-the-art vision–language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Anh-Kiet Duong and Petra Gómez-Krämer. 2025. Addressing out-of-label hazard detection in dashcam videos: Insights from the COOOL challenge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

Juanfang Fang and 1 others. 2022. DADA-2000: Driver attention prediction and accident dataset. *IEEE Transactions on Intelligent Transportation Systems*.

Gunnar Farnebäck. 2003. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the Scandinavian Conference on Image Analysis (SCIA)*.

Parisa Hatami, Maged Shoman, and Mina Sartipi. 2025. Open-world hazard detection and captioning for autonomous driving with a unified multimodal pipeline. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

Jack Hessel, Ari Holtzman, Jack Bandy, and Yejin Choi. 2021. CLIPScore: A reference-free evaluation metric for image captioning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Muhammad Monjurul Karim, Yu Li, Ruwen Qin, and Zhaozheng Yin. 2020. A dynamic spatial–temporal attention network for early anticipation of traffic accidents. In *Proceedings of the IEEE Intelligent Transportation Systems Conference (ITSC)*.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven C. Hoi. 2023. BLIP-2: Bootstrapping language–image pre-training with frozen image encoders and large language models. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Haotian Liu, Chunyuan Li, Pengchuan Zhang, Dongdong Zhang, Jianwei Yang, Lu Yuan, and Lei Zhang. 2023. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*. The LLaVA (Large Language and Vision Assistant) paper.

Lukas Picek, Vojtěch Čermák, and Marek Hanzl. 2025. Zero-shot hazard identification in autonomous driving: A case study on the COOOL benchmark. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, and 1 others. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change-point detection methods. https://arxiv.org/abs/1801.00718. ArXiv:1801.00718.

World Health Organization. 2023. Global status report on road safety 2023. https://www.who.int/teams/social-determinants-of-health/safety-and-mobility/global-status-report-on-road-safety-2023.

Ruixuan Zhang, Beichen Wang, Juexiao Zhang, Zilin Bian, Chen Feng, and Kaan Ozbay. 2025. SEEUNSAFE: When language and vision meet road safety— leveraging multimodal large language models for video-based traffic accident analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

Adapting Feature Attenuation to NLP

Tianshuo Yang
University of Michigan
yangts@umich.edu

Ryan Rabinowitz
University of Colorado
Colorado Springs
rrabinow@uccs.edu

Terrance E. Boult, Jugal Kalita
University of Colorado
Colorado Springs
tboult@vast.uccs.edu,
jkalita@uccs.edu

Abstract

Transformer classifiers such as BERT deliver impressive closed-set accuracy, yet they remain brittle when confronted with inputs from unseen categories—a common scenario for deployed NLP systems. We investigate Open-Set Recognition (OSR) for text by porting the *feature attenuation hypothesis* from computer vision to transformers and by benchmarking it against state-of-the-art baselines. Concretely, we adapt the COSTARR framework—originally designed for classification in computer vision—to two modest language models (BERT (base) and GPT-2) trained to label 176 arXiv subject areas. Alongside COSTARR, we evaluate Maximum Softmax Probability (MSP), MaxLogit, and the temperature-scaled free-energy score under the OOSA and AUOSCR metrics. Our results show (i) COSTARR extends to NLP without retraining but yields no statistically significant gain over MaxLogit or MSP, and (ii) free-energy lags behind all other scores in this high-class-count setting. The study highlights both the promise and the current limitations of transplanting vision-centric OSR ideas to language models, and points toward the need for larger backbones and task-tailored attenuation strategies.

1 Introduction

Transformer models, exemplified by BERT and GPT-2, are widely used for text classification and many other NLP tasks (González-Carvajal and Garrido-Merchán, 2020). Most of these models are trained under a *closed-set* assumption: every test document is expected to fall into one of the categories seen during training. In practical settings this assumption often breaks. Search engines must tag papers from brand-new research areas, legal-tech systems face unfamiliar case types, and content filters encounter fresh slang or memes. When a model meets such unseen classes, it can produce confident yet incorrect predictions, which is especially risky in sensitive domains like healthcare or law (Leo and Kalita, 2020; Mundt et al., 2023). Open-Set Recognition (OSR) seeks to address this issue by asking a classifier to (i) label known inputs and (ii) flag inputs that come from unknown classes (Bendale and Boult, 2016). Two obstacles make OSR difficult for transformers: soft-max scores tend toward over-confidence, and the final linear head may suppress latent dimensions that still hold useful signals for novelty detection.

In computer vision, the *feature attenuation hypothesis* suggests that combining features *before* and *after* the classification layer can improve detection of unknowns. COSTARR implements this idea and has produced strong open-set results for image models (Anonymous, 2025). Whether the same concept helps in NLP is an open question. In this work we adapt COSTARR to two medium-size language models—BERT (base) and GPT-2—fine-tuned to predict 176 arXiv subject areas. We compare the adapted score against three post-hoc baselines: Maximum Softmax Probability (MSP), MaxLogit, and the temperature-scaled free-energy score originally proposed for vision models (Liu et al., 2020). Evaluation uses two deployment-oriented metrics, Operational Open-Set Accuracy (OOSA) and AUOSCR.

Our contributions are:

1. **Adapting feature attenuation to NLP.** We present the first systematic transfer of COSTARR and the feature attenuation hypothesis to transformer-based text classifiers.

2. **Baseline study at scale.** We benchmark attenuation-, logit-, and energy-based OSR scores on a 176-class arXiv abstract task using BERT (base) and GPT-2.

3. **Empirical findings.** We observe that

COSTARR works in NLP without extra training but offers no clear advantage over MaxLogit or MSP, while the free-energy score lags behind—suggesting model capacity rather than scoring choice may be the main limiting factor.

2 Related Works

2.1 Softmax and Logit-based OSR

Maximum Softmax Probability (MSP) is one of the most widely used baseline for OSR (Vaze et al., 2022). It works by thresholding the highest probability output by a standard softmax classifier. While intuitive and easy to implement, its effectiveness is limited by the tendency of neural networks to be overconfident on unfamiliar inputs, making it less reliable than more advanced OSR methods specifically designed to model uncertainty or the space of unknowns.

A simple way to mitigate some of the overconfidence issues of MSP is by using the maximum value among the raw logits (pre-softmax activations) output by the model for classification purposes. This method, called *MaxLogit*, bypasses the need for a linear classification layer.

2.2 Feature Attenuation in OSR

Recent breakthroughs in computer vision reveal that classification layers suppress information critical for novelty detection—a phenomenon termed *feature attenuation*. The COSTARR framework (Anonymous, 2025) formalizes this through the *attenuation hypothesis*, which suggests that both pre-attenuation features (deep embeddings before the final layer) and post-attenuation features (after interaction with class weights) provide complementary signals for Open-Set Recognition (OSR). COSTARR leverages this by combining both signals into a unified scoring function that significantly outperforms prior OSR methods across image classification tasks using ViTs, ResNets, and ConvNeXts. While effective in vision, its application to NLP has not been explored until now.

2.3 Energy-based OSR

An alternative line of work swaps the usual soft-max confidence for a *free-energy* score computed directly from the logits. For an input x with logit vector $f(x) = \big[f_1(x), \dots, f_K(x)\big]$ and a fixed temperature T, Liu *et al.* (Liu et al., 2020) define

$$E(x; f) \;=\; -T\,\log\!\Big(\textstyle\sum_{k=1}^{K} e^{f_k(x)/T}\Big).$$

Because $-E(x; f)$ is a monotone transform of the soft-max log-partition function, higher values indicate more in-distribution–like inputs. At test time we simply threshold the *negative* energy:

$$\text{score}(x) = -E(x; f), \qquad \text{reject if score}(x) < \tau,$$

with τ tuned on a small validation split containing known and unknown examples. The method is completely post-hoc and model-agnostic, so we include it as a strong baseline in our transformer OSR experiments.

2.4 Open-Set Techniques for Text

OSR for NLP remains underdeveloped relative to vision. Leo and Kalita (2020) introduced one of the earliest open-set approaches for text, focusing on incremental learning in authorship attribution. Their CNN-based pipeline emphasizes clustering and novelty detection but differs from our closed-class training scenario. Still, their work highlights the utility of outlier analysis and motivates evaluation strategies based on cluster separation and label discovery.

2.5 OSR Evaluation Standards

We adopt standardized OSR metrics to benchmark our method. In particular, Operational Open-Set Accuracy (OOSA) (Anonymous, 2025) simulates deployment settings by determining decision thresholds on validation data that include both known and unknown samples. Additionally, we report Area Under the Open Set Classification Rate Curve (AUOSCR) (Dhamija et al., 2018), which captures the trade-off between recognizing known classes and rejecting unknowns across varying thresholds. These metrics provide a rigorous framework to compare COSTARR against baseline methods such as softmax logit (MaxLogit), Maximum Softmax Probability (MSP), and the free-energy score.

3 Proposed Approach

This section presents our adaptation of the COSTARR framework to transformer-based NLP models. We combine formal notation from the attenuation hypothesis with practical steps for implementation, training, and evaluation on the arXiv dataset.

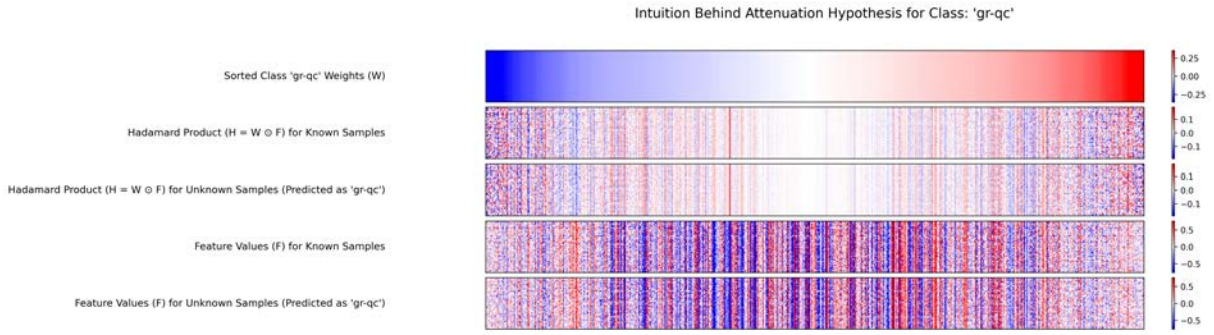

Figure 1: Logit for class j is formed from a dot product of its weights W_j , and the input features F, but the low weighted features are attenuated and hence ignored by this class's projection. The top color bar shows the weights W for category 'gr-qc' from BERT (base)'s custom classification layer, sorted from low to high, left to right; the white zone denotes where weights approach zero. This sorting index orders all other color bars. The second bar shows the Hadamard Product H for a known input x_k, and the 3rd row is for an unknown x_u, for which 'gr-qc' is the max logit. Each column represents a single feature dimension. The 4th and 5th bars show the feature vectors from the same test run and the most confident unknown samples from for that class, respectively. The consistency in low-saturation bars in the weights and Hadamard Product shows that regardless of what features are present, many dimensions are attenuated before classification

3.1 Initial Implementation Overview

Our approach consists of four core steps:

(1) **Dataset Selection**: Identify a labeled corpus of textual documents.

(2) **Model Training**: Fine-tune a transformer model with a custom linear classification head.

(3) **COSTARR Integration**: Implement COSTARR scoring using both pre- and post-attenuation features.

(4) **Evaluation**: Compare COSTARR to state-of-the-art OSR baselines such as MaxLogit, MSP, and free-energy score.

3.2 Dataset

We use the Hugging Face `arxiv-abstracts-2021` dataset, which includes metadata for over 2 million preprints. These are categorized into 176 scientific domains called subcategories (ex: 'cs.AI' for artificial intelligence). Out of these, we randomly select 132 of them as known categories and treat the remaining 44 as unknown categories (a 0.75:0.25 ratio). Preprints belonging to no known category are considered out-of-distribution data for the purposes of OSR.

We randomly reserve 10% of the documents across all categories for testing. The remaining documents from known categories are used for training.

3.3 Model Selection and Feature Extraction

We select two relatively lightweight pretrained models from Huggingface, BERT (base) and GPT-2, and added a custom classification head to enable COSTARR computation.

We extract deep features $F(x)$ from the penultimate layer ([CLS] embedding), and define post-attenuation features $H_j = F(x) \odot W_j$, where W_j is the classification weight for class j. These features are used in the remaining steps to calculate the COSTARR score and perform evaluation.

3.4 COSTARR Score for Transformers

Given a test input x and predicted class $m = \arg\max_j l_j(x)$:

$$C_j(x) := \text{Concat}(F(x), H_j) \tag{1}$$

$$\text{Sim}_j(x) := 0.5 \left(1 + \frac{C_j(x) \cdot \mu_{C_j}}{\|C_j(x)\| \cdot \|\mu_{C_j}\|}\right) \tag{2}$$

$$\lambda_m(x) = \max_j \text{GNL}(l_j(x)) \tag{3}$$

$$\mathcal{S}(x) = \lambda_m(x) \cdot \text{Sim}_m(x) \tag{4}$$

Here, μ_{C_j} is the class-wise mean of concatenated features computed from training data, and GNL is a global min-max normalization of logits.

3.5 Evaluation Metrics

We measure COSTARR's performance against commonly used methods for OSR such as MSP

and free-energy score. MaxLogit is also included as a baseline for comparison, implemented through taking the maximum logit output by the LLM before the custom classification layer.

All methods are evaluated on the following metrics:

- **OOSA (Operational Open-Set Accuracy)**: Threshold derived from a validation set (with held-out unknowns) to simulate deployment accuracy. We calculate OOSA at thresholds of 0, 0.1, 0.2, 0.3, . . ., 1.0.
- **AUOSCR (Area Under OSCR Curve)**: Plots trade-off between known classification and unknown rejection over all thresholds.

4 Results and Analysis

For our experiments, we derive the optimal threshold for OOSA from a validation set of 10% of the data for each model/score combination. From the test dataset, MaxLogit achieves an OOSA score of 0.319 for BERT (base) and 0.311 for GPT-2. On the other hand, COSTARR achieves an OOSA score of 0.319 for BERT (base) and 0.303 for GPT-2, while MSP achieves 0.320 for BERT (base) and 0.320 for GPT-2. Finally, free-energy achieves 0.172 for BERT (base) and 0.174 for GPT-2. Thus, there is no significant difference in OOSA scores between MaxLogit, COSTARR, and MSP, while free-energy lags substantially behind.

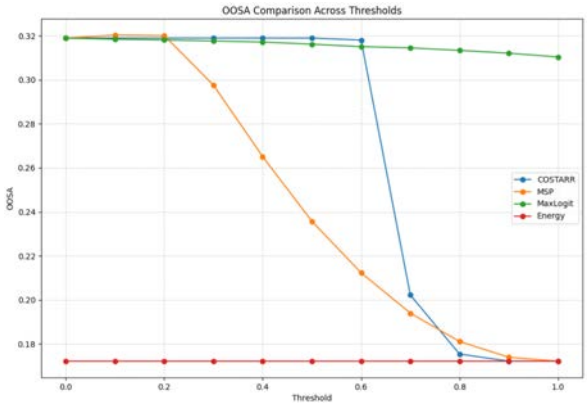

Figure 2: OOSA at various thresholds for BERT (base).

However, interesting differences can be observed when we study the OOSA score at each threshold level (see figures 2 and 3). In both BERT (base) and GPT-2, we observe a substantial dropoff in MSP scores when the threshold exceeds 0.2, while COSTARR only substantially declines at thresholds above 0.5 or 0.6. This indicates that

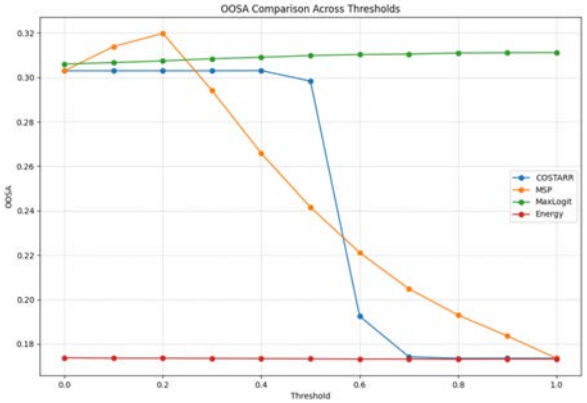

Figure 3: OOSA at various thresholds for GPT-2.

COSTARR is a more discriminatory approach that works for a larger range of thresholds compared to MSP. However, its performance is not significantly better than MaxLogit, whose OOSA score remains consistent across all thresholds.

We draw a similar conclusion when we look at AUOSCR scores. The OSCR curves for BERT (base) and GPT-2 are shown in figures 4 and 5 respectively. COSTARR and MaxLogit achieved very similar AUOSCR scores. COSTAR has an AUOSCR of 0.236 for BERT (base) and 0.223 for GPT-2, while MaxLogit achieves 0.233 for BERT (base) and 0.236 for AUOSCR. On the other hand, MSP slightly outperformed the competition with an AUOSCR of 0.251 for BERT (base) and 0.250 for GPT-2. Finally, free-energy again performed poorly, netting an AUOSCR of only 0.167 for BERT (base) and 0.150 for GPT-2.

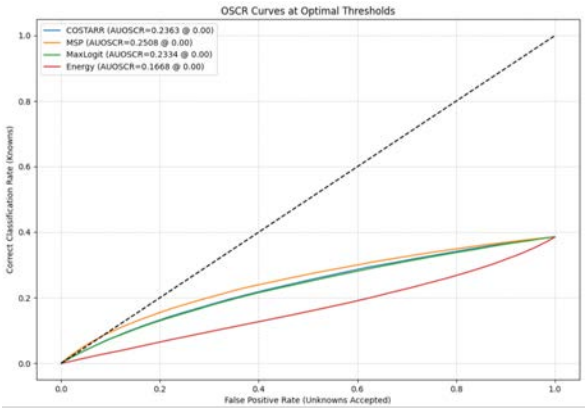

Figure 4: OSCR curves for BERT (base).

Despite these minor differences, it is worth noting that all of the various OSR approaches struggled with the task of arXiv category classification. We will address this observation in the next section. In

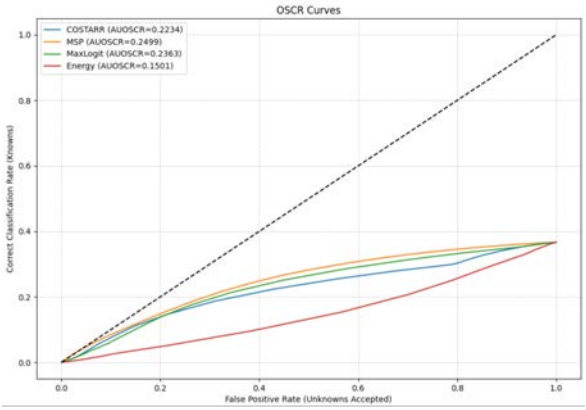

Figure 5: OSCR curves for GPT-2.

the meantime, our results point towards the finding that none of the state-of-the-art OSR frameworks in classification settings generalize well to the NLP setting. At least when it comes to small, weak models like BERT and GPT-2 in complex settings with a large number of classes, they do not offer meaningful improvements over simply taking the MaxLogit before the classification layer.

5 Limitations and Challenges

It is important to recognize the limitations of our experiment. One of the challenges exposed by our findings is the poor performance of all the tested methodologies, including COSTARR. None of them achieve a OOSA or AUOSCR score above 0.4. Since all of the approaches struggled with the task, the issue likely lies not in the OSR methods, but in the LLM models themselves. BERT (base) and GPT-2 are both small and outdated models that may struggle to perform complex tasks such as arXiv category classification based on abstract. Certain abstracts from unknown categories like statistics may look very similar to abstracts from a known category like mathematics, and vice versa. This problem is exacerbated by the presence of 176 classes. It is worth noting that in preliminary experiments involving only 8 categories, we finetuned BERT (base) on a small subset of 200,000 samples and achieved an OOSA of 0.752 and an AUOSCR of 0.682. This demonstrates that the difficulty of the task drastically increases with larger numbers of classes. We would like to test our framework on larger and more powerful models such as GPT-4 and Deepseek-V3 to see if the performance improves.

Traditionally, OSR is a direction mostly focused in computer vision. One of our project's primary contributions is to bring this framework into the NLP setting. However, computer vision features may not directly translate to linguistic representations. The layer normalization and attention mechanisms of LLMs may suppress features differently. Attenuation effects in transformers could differ fundamentally from CNNs.

Moreover, arXiv categories are unevenly distributed, and abstracts vary in length and structure. We may find it prudent to implement stratified sampling or abstract length thresholds. Furthermore, we would also like to test our findings on alternative datasets to validate whether our results are generalizable. Finally, NLP tasks are computationally intensive, and resource constraints can present great challenges especially as we scale up our project to include more models and more comprehensive ablation studies.

6 Conclusion

Our paper addresses a critical vulnerability in transformer-based text classifiers: their tendency toward overconfident misclassification of inputs from unknown categories under real-world open-set conditions. Deployed NLP systems (e.g., content moderation, academic search engines) frequently encounter novel categories. Our approach can help reduce hazardous overconfidence in such scenarios. Moreover, by revealing how classification layers suppress novelty-detection signals, we offer a pathway to design more transparent and adaptable transformers.

Through experimentation over classifying arXiv categories based on abstract, we observe no substantial improvement in the OOSA and AUOSCR metrics when using COSTARR and other state-of-the-art OSR methods. There appears to be no evidence that these powerful approaches in classification settings generalize well to NLP. For future research, we would like to test our framework on larger and more powerful LLMs such as GPT4 and Deepseek-V3 to see if these observations hold for high performing models. We also plan to conduct ablation studies to determine whether the feature attenuation hypothesis holds empirically in LLMs.

References

Anonymous. 2025. COSTARR: Consolidated open set technique with attenuation for robust recognition. Manuscript under review at International Conference on Computer Vision (ICCV).

Abhijit Bendale and Terrance E. Boult. 2016. Towards Open Set Deep Networks . In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1563–1572, Los Alamitos, CA, USA. IEEE Computer Society.

Akshay Raj Dhamija, Manuel Günther, and Terrance Boult. 2018. Reducing network agnostophobia. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Santiago González-Carvajal and Eduardo C. Garrido-Merchán. 2020. Comparing BERT against traditional machine learning text classification. *CoRR*, abs/2005.13012.

Justin Leo and Jugal Kalita. 2020. Moving towards open set incremental learning: Readily discovering new authors. In *Advances in Information and Communication*, pages 739–751, Cham. Springer International Publishing.

Weitang Liu, Xiaoyun Wang, John Owens, and Yixuan Li. 2020. Energy-based out-of-distribution detection. In *Advances in Neural Information Processing Systems*.

Martin Mundt, Yongwon Hong, Iuliia Pliushch, and Visvanathan Ramesh. 2023. A wholistic view of continual learning with deep neural networks: Forgotten lessons and the bridge to active and open world learning. *Neural Networks*, 160:306–336.

Sagar Vaze, Kai Han, Andrea Vedaldi, and Andrew Zisserman. 2022. Open-set recognition: a good closed-set classifier is all you need?

Introducing Semantic Feature Dependencies in NLP XAI Systems with SUPLIME

Daniel Kamen
University of Colorado Colorado Springs
dkamen@uccs.edu

Melkamu Abay Mersha
University of Colorado Colorado Springs
mmersha@uccs.edu

Jugal Kalita
University of Colorado Colorado Springs
jkalita@uccs.edu

Abstract

Explainable AI (XAI) systems have broad applications, however there are many obstacles they must overcome before they can be broadly implemented. We focus on the issue of the absence of higher-level feature explanations, specifically on text classifiers. Traditional XAI systems used on text classifiers explain a given model's prediction in terms of low-level features such as individual words; however, should the model recognize higher-level features, the explainer cannot reveal them to the user. We implement a novel XAI technique, based on LIME, which uses linguistic parse trees to heighten the feature-level of the explanation and provide users with insight into the model's recognition of intra-sentence dependencies. We name this system Universal Syntactic Parsing with LIME, or SUPLIME.

1 Introduction

With the recent adoption of AI systems into increasingly numerous fields, and being used for tasks such as driving autonomous vehicles, diagnosing medical diseases, and protecting national security, these complex black-box models are taking on larger and larger responsibilities. Naturally, alongside the rise of these black-box responsibilities, further insight into the processes behind these high-stakes and potentially life-saving decisions becomes increasingly important to establish both trust and transparency in the output. This establishment of further insight would allow for the ability to debug dangerous outputs, reveal biases, or simply ensure that a model is learning the desired patterns rather than some unrecognized exploit within the training dataset. This is the study of the topic of eXplainable AI (XAI).

When applied to text, the common perturbation-based explainer *LIME* (Ribeiro et al., 2016) involves removing singular tokens, observing the change in the prediction, and then establishing a simple approximate model for the local inputs immediately surrounding the example of interest in the input space. This allows for the development of a word relevance, or 'saliency', map which indicates the magnitude and direction of contribution of each token, or word, towards the final prediction. But this method does not take into account the structure of the sentence, ignoring crucial dependencies such as grammatical and semantic relations within the text, which we will hereafter refer to as *higher-level* features. Development of insight into how a model handles these word dependencies could allow for deeper analysis into the workings of complex text classifiers.

The contribution of our research is our attempt to resolve this oversight. We establish a system to perturb dependent groups of words at a time, and establish the effects of larger related chunks of input text on the model's final prediction. Our objective is to create the ability to evaluate and examine how the model recognizes dependencies, and to improve on the current output of individual word relevancy. We then use our developed system to evaluate models of varying sizes and architectures to determine the effect of model complexity on higher-level feature recognition in text.

2 Related Works

Although many types of techniques are used in XAI, our approach will focus on improving the *perturbation* class, and hence, we limit the provided background to the common perturbation techniques.

2.1 LIME

LIME (Ribeiro et al., 2016) is a perturbation technique, meaning it makes small changes around a

given input, sampling neighboring points in the input space, examining the change in output, and assigning a locally faithful model approximation to that instance. These local explanations are given as linear models using Support Vector Machines (SVMs) to separate the surrounding classifications. A visual of the sampling and approximation process is shown in Figure 1. Because LIME only deals with the input and output, it is *post-hoc* and *model-agnostic*, meaning it is implemented after the training process, and that it can be applied to any model architecture, respectively. Though LIME is almost explicitly used to approximate models using linear interpretations, the formulation given is not restricted to this class of functions.

LIME converts the input into an interpretable representation. This means, in tasks involving text input, even though the model may use vector embeddings as representations, LIME will represent the input as binary vectors indicating the presence or absence of each word to allow for better human interpretability when aligning to an approximate local model. When used for models trained for text classification tasks, what is established as the output is the importance or relevance of each individual binary word presence, while ignoring larger semantic structures.

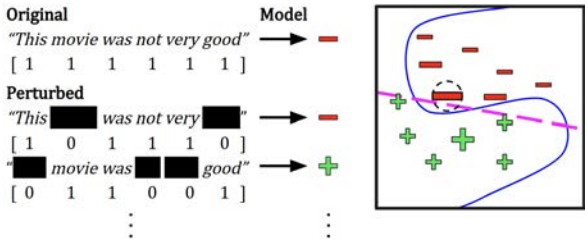

Figure 1: Demonstrating the intuition of LIME. The circled red minus sign is the instance being explained. A binary input space is established, representing the presence of each word in the input, and a random sample is classified with the original model where the distance from the input being examined weighs each sample's influence on the final linear approximation. The explanation (pink, dashed) is an approximation of the classifier function (blue, solid).

2.2 SHAP

SHAP (Lundberg and Lee, 2017) works similarly to LIME, building on top of LIMEs theory while incorporating approximate Shapley values from game theory. SHAP is another model-agnostic perturbation method which explains local predictions, evaluating each features contribution over a sample of subsets of the entire input. Though SHAP is similar to LIME, it has been shown to produce worse explanations when dealing with text classification tasks (Mersha et al., 2025).

2.3 Areas of Improvement

These techniques have been implemented widely in the field of NLP, however results mostly fall short. (Lyu et al., 2024) states that interpretable AI still has many major challenges to overcome. One of these obstacles is the difficulty to expand these interpretations to broader, more complicated tasks not limited to classification and regression. Another obstacle is the lack of consensus on qualitative metrics, making it difficult to determine whether a given interpretation is better than another. The obstacle we will focus on is the porting of textual XAI systems, currently aimed at explaining from the perspective of individual surface-level features, to systems that include a higher-level feature perspective. Often, the meaning of words is determined by other words in a sentence, leading to dependencies of their semantics. For example, take *"This movie was not good"*. While *good* would be assigned a positive score towards a positive review, the phrase *not good* is clearly a negative comment. Models may intrinsically develop systems for handling these dependencies, however these dependencies are not able to be visualized by XAI systems while they only examine one word at a time.

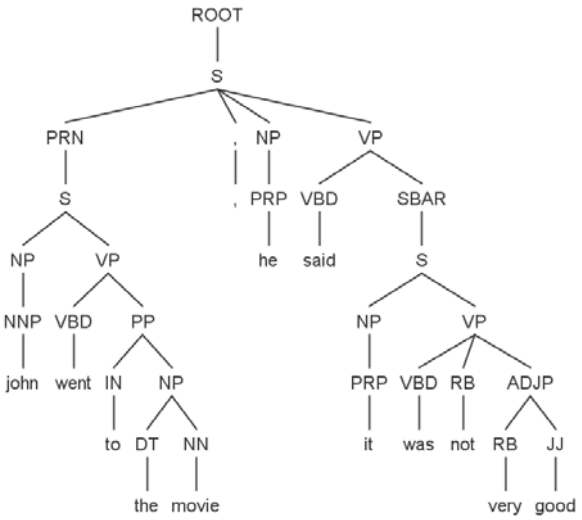

Figure 2: Example of Constituency Parsing

3 Problem Statement

XAI techniques on NLP systems do not take into account high level features, leaving interpretations difficult to understand. We aim to modify current XAI techniques such that they produce the model's perspective on semantic dependencies and provide higher-level explanations of models' performances on text. This problem has been researched on other modalities such as image classification, but our extension to NLP is novel. This problem can be extended to other modalities where high-level features exist; however, we will only focus on the text classification task.

4 Approach

For our approach, we build on top of LIME (Ribeiro et al., 2016), adding the ability to perturb sets of dependent words at a time, potentially yielding deeper insight into the model's consideration of related words with dependent meanings. We name this system Universal Syntactic Parsing with LIME, or SUPLIME. We use the *Stanza* parser (Qi et al., 2020) to tokenize text as well as to generate constituency and dependency parsing trees to perturb. Stanza produces dependency and constituency parsing of text. This breaks down sentences into their grammatical, syntactic, and semantic pieces. An example of constituency parsing can be seen in Figure 2. Our LIME modifications add the ability to mask entire branches of these parse trees as well as single words, yielding coefficients of influence for the presence of dependent structures. This follows the process of LIME, but samples points in an input space that represents the presence of phrases and parts of speech, allowing for the determination of the relevance of higher-level features. In addition to these linguistically motivated parse trees, we generate random parse trees to be able to compare both linguistic dependencies and other couplings that may have been established by the models. Figure 3 illustrates how we plan to perturb the input, and Figures 1 and 4 highlight the similarities of the original LIME sampling procedure and our modified version, only changing the binary space being sampled.

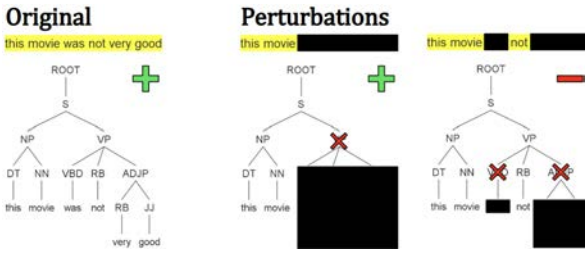

Figure 3: Parse Tree Perturbations

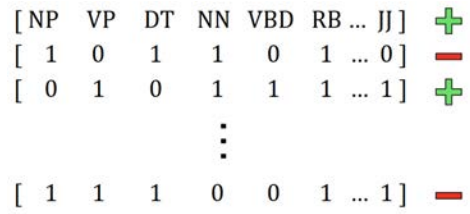

Figure 4: Parse Tree Samples

5 Metrics

Metrics in XAI are difficult to define, and often, a new XAI method will introduce an improvement on an aspect that is not yet measured in existing metrics; this leads to many metrics being proposed without wide adoption. (Canha et al., 2025) examines 78 existing proposed explanation metrics and condenses them into just 24 metrics, which then measure 11 properties. We will use only two metrics for now, with plans to expand later. These metrics were chosen from the Fidelity and Stability properties (Canha et al., 2025) and are named *Surrogate Agreement* and *Identity*. *Surrogate Agreement* measures the average difference between the prediction from the explainer and the output from the black-box model over a sample. *Identity* measures the variation of the output when an explainer is given the same instance to explain many times.

For sample size N, black-box prediction $b(x)$, explaniner prediction, or *surrogate*, $s(x)$, Surrogate Agreement SA is defined as

$$SA = 1 - \frac{\sum_{i=1}^{N} |b(x_i) - s(x_i)|}{N \cdot \max(b(x))}$$

With euclidian distance $d(x_1, x_2)$, number of instances examined r, and ith explanation of instance j x_{ji}, Identity I is

$$I = \frac{1}{r} \sum_{j=1}^{r} \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1 + d(x_{1j}, x_{ij})}$$

Explainer	SA	Identity
LIME	0.70	0.40
SUPLIME w/ Const. Par.	0.70	0.40
SUPLIME w/ Depend. Par.	0.70	0.40
SUPLIME w/ Random Par.	0.65	0.40

Table 1: Surrogate Agreement (SA) and Identity Metrics Comparison by Parsing Types: None, Constituency, Dependency, and Random

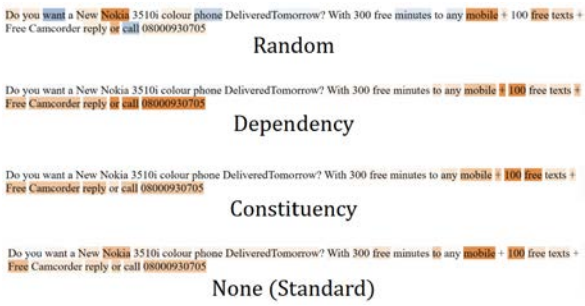

Figure 5: All LIME, Modified and Standard, Outputs

6 Results

Our limited results, shown in Table 1, indicate that adding higher-level feature dependencies do not affect the quality of explanation, based on surrogate agreement and identity. However, the explanations produced are visually very different, as seen in Figure 5.

7 Future Work

With the working system we had set out to develop, our next steps are gathering more evidence for further analysis. Further work includes more objective metrics as well as human evaluation. Objective metrics include stability and average recognized dependency size, and human evaluation involves clarity and preference. We seek to develop new measurements, as well as test on more existing metrics, to highlight the differences between SUPLIME, SHAP, and the standard individual token masking LIME system.

8 Conclusion

We aim to tackle the current oversight of XAI systems which ignore dependencies on high-level features. We have completed our proposed system, which is able to mask dependent structures, allowing for higher-level features to be assigned blame for the prediction more explicitly. We have gathered minimal preliminary results which show that our system yields very similar performance on explaining text classification as the standard LIME system. And, in the future, we hope to show with further evidence more insights into text classification models, recognition of higher-level features in NLP.

References

Dulce Canha, Sylvain Kubler, Kary Främling, and Guy Fagherazzi. 2025. A functionally-grounded benchmark framework for xai methods: Insights and foundations from a systematic literature review. *ACM Comput. Surv.*, May. Just Accepted.

Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Qing Lyu, Marianna Apidianaki, and Chris Callison-Burch. 2024. Towards faithful model explanation in nlp: A survey. *Computational Linguistics*, 50(2):657–723.

Melkamu Abay Mersha, Mesay Gemeda Yigezu, and Jugal Kalita. 2025. Evaluating the effectiveness of xai techniques for encoder-based language models. *Knowledge-Based Systems*, 310:113042.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A python natural language processing toolkit for many human languages.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA. Association for Computing Machinery.

Hybrid GNN-LLM Pipeline for Multi-Vulnerability Attack Chain Prediction in Software Supply Chains

Laura Baird
University of Colorado
Colorado Springs (UCCS)
Colorado, USA
lbaird@uccs.edu

Armin Moin
University of Colorado
Colorado Springs (UCCS)
Colorado, USA
amoin@uccs.edu

Abstract

Software supply chain vulnerabilities are typically analyzed in isolation, missing attack vectors that can arise from multi-component interactions. Traditional vulnerability scanners evaluate individual CVEs independently, overlooking how multiple low-severity vulnerabilities can combine to create high-risk attack chains. We present a hybrid approach that combines Heterogeneous Graph Attention Networks (HGAT) for vulnerability chain prediction with Large Language Models (LLMs) for explanation and validation. Our HGAT model uses type-specific attention mechanisms to learn complex interactions between different vulnerability entities, automatically identifying which multi-type vulnerability patterns are most predictive of chained attack vectors. The GNN predictions are then embedded as nodes in enriched knowledge graphs, enabling LLMs to provide grounded explanations and validate or challenge predictions through Model Context Protocol (MCP) function calling. To our knowledge, no existing tool performs multi-vulnerability chain analysis and attack prediction. Our evaluation demonstrates significant improvements in LLM accuracy for vulnerability analysis tasks, with our MCP-enhanced system achieving 87.5% F1.5 score compared to 74.9% for traditional RAG approaches and 16.0% for standalone LLM systems. We address the fundamental challenge of creating comprehensive datasets for multi-vulnerability attack chain analysis through systematic methodology development and validation framework construction.

1 Introduction

Software supply chains are fragile, and current security analysis approaches fundamentally misunderstand how attackers operate. While Software Bills of Materials (SBOMs) can expose isolated vulnerabilities, attackers don't think in isolation. Multiple vulnerabilities in separate components can combine to create multi-vulnerability interactions that represent a new class of complex attack vectors. This kind of combinatorial threat isn't well-modeled by existing scanners and scoring systems, which treat Common Vulnerabilities and Exposures (CVEs) as independent, identically distributed (i.i.d) samples (Zhao et al., 2024).

Software supply chain security has become a critical concern following sophisticated cyberattacks such as Log4j (Review, 2021) and SolarWinds (Fortinet, 2022; Target, 2023). These attacks demonstrate how vulnerabilities can propagate through interconnected dependencies, but more importantly, they reveal how attackers chain multiple weaknesses across component boundaries to achieve their objectives. Current vulnerability assessment pipelines score, rank, and patch each flaw in isolation, missing the critical insight that seemingly low-impact CVEs in one library can combine with another issue upstream to form an end-to-end compromise.

To address this fundamental gap, we present a hybrid approach that combines the pattern recognition capabilities of Graph Neural Networks with the reasoning and explanation capabilities of Large Language Models. Our system uses Heterogeneous Graph Attention Networks (HGAT) to identify potential vulnerability chains and complex attack patterns, then integrates these predictions directly into knowledge graphs where LLMs can provide grounded explanations, validate predictions, and generate actionable insights for security analysts.

This work addresses an important research direction that faces significant methodological challenges. To our knowledge, no existing tool performs multi-vulnerability chain analysis and attack prediction, and more critically, comprehensive evaluation datasets for such systems do not exist. Current vulnerability databases document individual CVEs but rarely capture the complex attack chains that span multiple components. We address this

fundamental evaluation challenge through our systematic corpus construction methodology and comprehensive evaluation framework.

The rest of this paper is structured as follows. Section 2 provides background information on multi-vulnerability attacks and the evaluation challenges in this domain. Section 3 reviews related work and identifies gaps in current approaches. We present our hybrid methodology in Section 4, demonstrate our evaluation framework and validation approach in Section 5, elaborate on threats to validity in Section 6, and conclude with future work directions in Section 7.

2 Background & Motivation

2.1 The Multi-Vulnerability Attack Problem

SBOMs capture information about component-level packages, transitive dependencies, CVEs, and more. However, critical information about the exploitability of *component chains* is often missing from the generated SBOM. Existing frameworks like the Vulnerability Exploitability eXchange (VEX) (NTIA, 2021) and the Exploit Prediction Scoring System (EPSS) (FIRST, 2021) can only provide predictions for individual vulnerabilities, missing the complex risks that arise when multiple components interact.

Static scanners like Snyk (Snyk, 2025) and Trivy (Aquasecurity, 2025) examine CVEs in isolation, ignoring how shared libraries or linked vulnerabilities might cascade. *Chained attacks* represent interactions where multiple low-severity CVEs that would normally be de-prioritized by static scanners or frameworks like VEX and EPSS could create a high-severity risk when combined. These attack patterns require reasoning over dependency relationships, vulnerability inheritance, and the complex ways that component interactions can create unexpected attack surfaces.

2.2 Graph Neural Networks for Structured Vulnerability Analysis

Graph neural networks excel at modeling complex relationships in graph-structured data, making them naturally suited for reasoning over component interactions and vulnerability propagation patterns. As software supply chains are inherently directed structures with complex dependency relationships, representing them as graphs enables GNNs to capture the structural patterns that indicate potential attack chains. The attention mechanisms in modern GNN architectures can automatically identify which relationship patterns are most predictive of exploitability, potentially discovering attack vectors that would be missed by rule-based approaches.

2.3 Large Language Models for Explanation and Validation

While GNNs can identify patterns in vulnerability data, they lack the ability to provide human-interpretable explanations for their predictions. Large Language Models offer complementary capabilities: they can generate natural language explanations, reason about complex technical relationships, and validate predictions against domain knowledge. However, LLMs suffer from hallucination problems when reasoning about technical data without proper grounding (Zhang et al., 2023).

Model Context Protocol (MCP) addresses these limitations by providing LLMs with structured function calling interfaces to external systems (Anthropic, 2025). Instead of relying on potentially incomplete or outdated training data, MCP enables LLMs to make dynamic queries for specific, verified information, reducing hallucinations and improving the accuracy of technical reasoning.

2.4 The Evaluation Challenge

A fundamental challenge in this research direction is the lack of comprehensive evaluation datasets. Unlike traditional vulnerability detection, which can be evaluated against known CVE databases, multi-vulnerability attack prediction requires datasets that document how multiple vulnerabilities were combined in real-world attacks. Such datasets are rare because:

- Attack documentation typically focuses on the primary vulnerability rather than complete attack chains
- Multi-vulnerability attacks often involve proprietary systems where full technical details are not disclosed
- The temporal aspect of attacks makes it difficult to reconstruct complete vulnerability interaction patterns
- Current threat intelligence focuses on individual indicators of compromise rather than vulnerability interaction patterns

This evaluation gap represents both a significant research challenge and an opportunity to contribute foundational datasets to the cybersecurity research community.

3 Related Work

3.1 Graph-Based Vulnerability Analysis

The application of graph neural networks to cybersecurity has gained significant attention in recent years. Yin et al. (2023b) pioneered the use of vulnerability knowledge graphs for co-exploitation prediction, framing the problem as link prediction between vulnerability pairs. Their Modality-Aware GCN (MAGCN) approach demonstrated that graph-based methods could outperform text-only baselines for predicting whether two vulnerabilities might be exploited together, introducing the foundational concept of modeling vulnerability interactions as graph learning problems.

Building on this foundation, Yin et al. (2023a) extended the approach to heterogeneous graphs, incorporating product, weakness, and vendor nodes to capture richer relationships. Their work showed that heterogeneous GNNs could improve exploitability prediction by 5.44% compared to homogeneous approaches. However, both works focused on pairwise vulnerability relationships rather than the complex multi-component chains that characterize sophisticated supply chain attacks.

The release of VulKG by Yin et al. (2024) provided a large-scale vulnerability knowledge graph with 276,000 nodes and 1 million edges, unifying CVEs, exploits, products, vendors, and domains. While this resource enables large-scale vulnerability analysis, it still treats vulnerabilities as discrete entities rather than modeling the dependency chains and component interactions that create complex attack vectors in software supply chains.

Our work extends these graph-based approaches by modeling vulnerability chains that span multiple components and dependency relationships, moving beyond pairwise co-exploitation to capture the complex interaction patterns that characterize supply chain attacks.

3.2 LLM-Enhanced Security Analysis and Function Calling

The integration of Large Language Models into cybersecurity applications has shown promise for tasks requiring natural language reasoning and explanation. Kurniawan et al. (2024) developed CyKG-RAG, which combines knowledge graphs with retrieval-augmented generation for forensic analysis tasks. Their system integrates CVEs, CWEs, CAPECs, and MITRE ATT&CK data into a Neo4j graph, enabling both symbolic queries and semantic retrieval for log attribution and attacker behavior reconstruction.

However, CyKG-RAG focuses on post-incident forensic analysis rather than predictive vulnerability assessment, and relies on traditional document retrieval rather than the structured function calling interfaces that can provide more precise access to graph relationships. Our approach builds on their knowledge graph integration concepts while applying them to predictive analysis and implementing direct graph querying through Model Context Protocol.

The emergence of function calling capabilities in LLMs has created new possibilities for structured reasoning. Schick et al. (2023) demonstrated that LLMs can learn to use external tools effectively when provided with appropriate interfaces, while Yao et al. (2023) showed that reasoning-action loops enable more sophisticated problem solving than static context approaches. These works establish the foundation for using LLMs as intelligent interfaces to structured data systems, which we leverage for vulnerability analysis and explanation.

3.3 Supply Chain Security and Multi-Component Analysis

Outside of academia, supply-chain tools such as Dependency-Track (OWASP, 2025b) and OSS Review Toolkit (ORT, 2025) visualize CVE propagation across software dependencies. However, these systems rely on static heuristics rather than learned models of vulnerability interactions. Recent work with code-graph GNNs focuses on vulnerability *detection* or explainability within individual components, not complex exploit paths that cross component boundaries.

The gap between academic research and practical supply chain security tools highlights the need for approaches that can model real-world attack patterns while providing actionable insights for security practitioners. Our hybrid approach addresses this need by combining the predictive capabilities of modern machine learning with the interpretability and validation capabilities that are essential for practical security applications.

3.4 Research Gap: Multi-Vulnerability Chain Prediction

While existing work has made significant progress in individual vulnerability analysis and pairwise vulnerability relationships, a critical gap remains in modeling and predicting multi-vulnerability attack chains that span component boundaries. Current approaches typically focus on:

- Individual vulnerability exploitability prediction
- Pairwise vulnerability co-exploitation
- Static dependency analysis without learned interaction patterns
- Post-incident forensic analysis rather than predictive modeling

Our work addresses this gap by introducing the first hybrid approach specifically designed for multi-vulnerability chain prediction in software supply chains, combining the structured reasoning capabilities of GNNs with the explanation and validation capabilities of LLMs through integrated knowledge graph access.

4 Hybrid Architecture and Implementation

Our hybrid approach operates through a three-stage pipeline that combines Graph Neural Network prediction with Large Language Model explanation and validation. The key innovation lies in the integration mechanism: GNN predictions are embedded directly into knowledge graphs as first-class entities, enabling LLMs to reason over both factual vulnerability data and predictive insights simultaneously.

4.1 Stage 1: Heterogeneous Graph Neural Network Architecture

We employ a Heterogeneous Graph Attention Network (HGAT) (Yang et al., 2021) architecture to model complex multi-type relationships in vulnerability knowledge graphs. Unlike homogeneous GNNs that treat all nodes and edges identically, HGAT explicitly handles different node types (Components, CVEs, CWEs, CAPECs) and relationship types (dependency, vulnerability, weakness, attack pattern relationships).

Our heterogeneous approach requires type-specific feature representations:

Component Nodes: Feature vectors combining vulnerability characteristics (CVE count, max/avg CVSS scores, critical vulnerability presence), structural properties (dependency centrality, direct/transitive status), and metadata (license information, component type).

CVE Nodes: Features include CVSS scores, severity levels, publication dates, and affected component counts.

CWE Nodes: Weakness categorization features, severity rankings, and exploitation difficulty indicators.

CAPEC Nodes: Attack pattern complexity, required skills, and target vulnerability types.

The HGAT model learns to identify vulnerability chain patterns by predicting edge probabilities between components that could form attack paths. During training, the model learns which combinations of vulnerability types, component relationships, and structural patterns are most indicative of exploitable attack chains.

4.2 Stage 2: GNN Prediction Integration into Knowledge Graphs

A critical innovation of our approach is the integration of GNN predictions directly into the knowledge graph structure as a new node type. When the HGAT model identifies potential vulnerability chains or complex attack vectors, these predictions are materialized as **AttackPrediction** nodes connected to relevant components and vulnerabilities through typed relationships.

The integration process works as follows:

1. **Prediction Generation**: The trained HGAT model processes an SBOM-derived knowledge graph and generates predictions for potential attack chains, outputting confidence scores and attention weights for each predicted relationship.
2. **Prediction Materialization**: High-confidence predictions (above a learned threshold) are converted into AttackPrediction nodes containing:
    - Confidence score from the HGAT model
    - Component sequence forming the predicted attack chain
    - Attention weight distribution showing which relationships were most influential

- Predicted attack vector classification (e.g., privilege escalation, data exfiltration, lateral movement)
- Temporal metadata about prediction generation

3. **Graph Enrichment**: AttackPrediction nodes are connected to the knowledge graph through typed relationships:
   - `PREDICTS_VULNERABILITY_CHAIN` edges connecting predictions to component sequences
   - `EXPLOITS_WEAKNESS_PATTERN` edges linking predictions to relevant CWE nodes
   - `IMPLEMENTS_ATTACK_PATTERN` edges connecting predictions to applicable CAPEC nodes
   - `BASED_ON_EVIDENCE` edges referencing the vulnerability relationships that support the prediction

This integration creates enriched knowledge graphs that contain both factual vulnerability information and predictive insights, enabling downstream LLM analysis to reason over both observed data and model predictions simultaneously.

Figure 1 illustrates this hybrid knowledge graph structure, showing how traditional vulnerability entities (Components, CVEs, CWEs, CAPECs) are enhanced with GNN-generated AttackPrediction nodes that connect to all relevant entity types through typed relationships, enabling comprehensive reasoning over both factual data and predictive insights.

4.3 Stage 3: LLM Integration via Model Context Protocol

We implement Model Context Protocol (MCP) to provide LLMs with structured function calling access to the enriched knowledge graphs containing both traditional vulnerability data and GNN predictions. Our Knowledge Query Protocol (KQP) implementation enables dynamic querying of prediction data and supporting evidence.

Core Prediction Analysis Functions:

- `get_attack_predictions(id)`: Retrieve all GNN predictions involving specific components, including confidence scores and supporting evidence

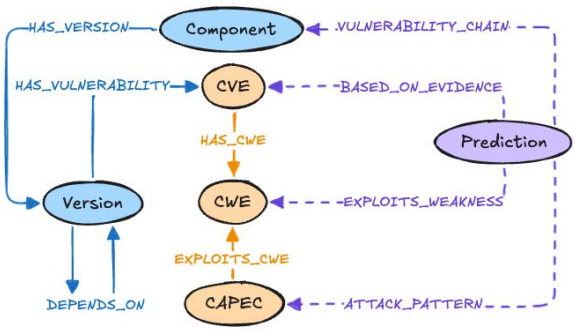

Figure 1: Hybrid knowledge graph structure showing traditional vulnerability entities (Components, CVEs, CWEs, CAPECs) enhanced with GNN-generated AttackPrediction nodes that capture complex vulnerability chains spanning multiple components and connect to supporting evidence and attack patterns.

- `analyze_prediction_chain(id)`: Examine the complete vulnerability chain for a specific prediction, including component dependencies and attack vector details
- `validate_prediction_evidence(id)`: Cross-reference GNN predictions against known vulnerability patterns and historical attack data
- `explain_attention_weights(id)`: Generate natural language explanations for why the GNN model identified specific relationships as important

Integrated Reasoning Functions:

- `compare_predictions_to_knowledge()`: Validate GNN predictions against established vulnerability databases and threat intelligence
- `generate_alternative_hypotheses()`: Explore alternative attack vectors that might achieve similar objectives
- `assess_prediction_reliability()`: Evaluate confidence factors and potential sources of prediction error
- `recommend_mitigations()`: Generate actionable recommendations for preventing predicted attack vectors

This comprehensive interface enables LLMs to serve as intelligent validation and explanation layers for GNN predictions, creating a system where machine learning insights are continuously refined through natural language reasoning and domain knowledge application.

4.4 Data Pipeline and Training Framework

Our approach processes Software Bills of Materials through a comprehensive pipeline designed to create training data for multi-vulnerability interaction learning:

1. **SBOM Collection and Processing**: Starting with the Wild SBOMs dataset (Soeiro et al., 2025), we select Python projects and generate consistent SBOMs using Syft (Anchore, 2025b) in CycloneDX (OWASP, 2025a) format.

2. **Vulnerability Enrichment**: SBOMs are enriched with vulnerability data using Grype (Anchore, 2025a), which queries the Open Source Vulnerabilities (Google, 2025) database to identify known CVEs associated with each component.

3. **Knowledge Graph Construction**: Enhanced SBOMs are converted into heterogeneous knowledge graphs with selective enrichment from MITRE's CVE, CWE, and CAPEC databases, fetching only data relevant to vulnerabilities actually present in the SBOM.

4. **Training Data Generation**: We developed a systematic approach combining synthetic labeling based on known attack patterns with semi-supervised learning techniques to generate training signals for the HGAT model, addressing the scarcity of ground truth multi-vulnerability chain data.

5. **Model Training Framework**: The HGAT model is designed to use a 70/30 split of the Wild SBOMs dataset, enabling learning of potential vulnerability chains and complex attack patterns from component and vulnerability features.

This comprehensive pipeline is illustrated in Figure 2, which shows the complete data flow from GitHub repositories through SBOM generation, vulnerability enrichment, knowledge graph construction, GNN prediction, and LLM validation with iterative feedback loops.

5 Evaluation Framework and Validation Results

The evaluation of multi-vulnerability attack chain prediction systems faces fundamental methodological challenges that represent a significant barrier

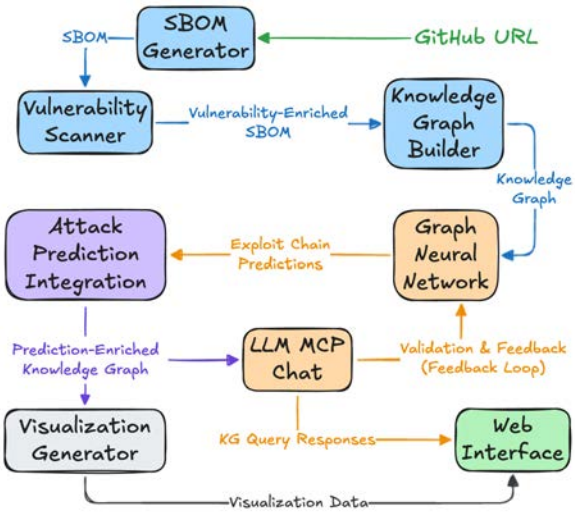

Figure 2: Hybrid GNN-LLM processing pipeline showing SBOM enrichment, GNN prediction generation, AttackPrediction integration into knowledge graphs, and LLM-based validation and explanation through MCP function calling with iterative feedback loops for continuous prediction refinement.

to progress in this research area. Unlike traditional vulnerability detection, which can be validated against established CVE databases, our approach requires evaluation datasets that document how multiple vulnerabilities were combined in real-world attacks. These datasets largely do not exist in comprehensive, machine-readable formats. We address this challenge through a systematic evaluation framework and demonstrate validation results that establish the effectiveness of our proposed methodology.

5.1 Evaluation Corpus Construction Methodology

Current cybersecurity datasets focus primarily on individual vulnerabilities rather than attack chains. To address this limitation, we developed a comprehensive evaluation corpus construction methodology that combines multiple data sources:

Historical Attack Analysis: We systematically analyzed documented supply chain attacks from sources including the Atlantic Council's Software Supply Chain dataset (Messieh, 2023), CISA advisories, and detailed incident response reports. For each attack, we reconstructed complete vulnerability chains by:

- Identifying all software components involved in the attack
- Mapping specific CVEs and weakness pat-

terns that enabled each attack stage

- Documenting the dependency relationships that allowed vulnerability chaining
- Creating graph representations of complete attack paths from initial compromise to objective achievement

Synthetic Attack Chain Generation: Given the scarcity of documented multi-vulnerability chains, we developed principled approaches for generating synthetic attack scenarios based on:

- Known vulnerability interaction patterns from security research
- Dependency analysis of real-world software systems
- Validation against available incident data to ensure realistic attack progression patterns

5.2 Validation Results and Performance Analysis

To validate the effectiveness of our MCP-based grounding approach, we conducted comprehensive testing comparing three system configurations: our proposed MCP-Enhanced system, a Legacy RAG system using traditional document retrieval, and a Standalone LLM baseline. Testing was performed on a repository containing 110 vulnerabilities, with evaluation focused on citation accuracy using precision, recall, and F1.5 scores.

Experimental Setup: Each system was evaluated on vulnerability analysis tasks requiring accurate citation of CVE, CWE, and CAPEC identifiers across seven vulnerable application repositories from the vulnerable-apps organization[1]. Testing was conducted on 193 total vulnerabilities spanning diverse Python applications including REST APIs, web frameworks, and security testing tools.

Results: Tables 1 and 2 show the validation results and comparative improvements across all three systems:

The MCP-Enhanced system achieved superior performance across all accuracy metrics, with a 71.5 percentage point improvement in F1.5 score compared to the standalone LLM baseline (Table 2). The system achieved near-perfect precision (98.7%), demonstrating that structured knowledge graph access effectively reduces hallucinations in vulnerability analysis tasks. The 12.6 percentage point improvement over traditional RAG (87.5% vs 74.9% F1.5 score) establishes the value of dynamic, targeted information retrieval versus static document context.

Table 1: Validation Results: Accuracy Metrics (7 Repositories, 193 Vulnerabilities)

System	Precision	Recall	F1.5 Score
MCP	98.7%	83.3%	**87.5%**
RAG	83.3%	71.7%	74.9%
Standalone	35.0%	12.9%	16.0%

Table 2: F1.5 Score Improvements Between Systems and Average Response Time

System	vs Standalone	Avg. Time
MCP	+71.5 pp	24.01s
RAG	+58.9 pp	17.08s
Standalone	Baseline	9.81s

These results validate our hypothesis that structured function calling significantly improves LLM accuracy in knowledge-intensive domains across diverse vulnerability analysis scenarios. The testing across seven repositories and 193 vulnerabilities demonstrates robust performance improvements that generalize beyond individual applications.

The improved accuracy comes with increased computational cost, as shown in Table 2, with the MCP system requiring 24.01s average response time compared to 17.08s for RAG and 9.81s for standalone systems.

5.3 Comprehensive Evaluation Framework

Building on these preliminary findings, our comprehensive evaluation framework addresses the core challenges in multi-vulnerability attack prediction through systematic assessment across multiple dimensions:

Multi-Repository Analysis: The framework supports extension of evaluation across diverse repositories from the vulnerable-apps GitHub organization, covering multiple programming languages and vulnerability types to establish broader generalizability.

GNN Component Evaluation: Full HGAT model training and evaluation on the Wild SBOMs dataset will assess the model's ability to identify

[1]Repositories included Tiredful-API-py3-beta, VAmPI, vulnapi, django-rev-shell, rest-api-goat, vfapi, and log4j-honeypot-flask.

potential vulnerability chains across heterogeneous software systems.

Integrated System Assessment: End-to-end evaluation of the complete GNN-LLM pipeline will measure the system's ability to predict and explain multi-vulnerability attack chains, comparing against existing vulnerability assessment tools.

Attack Pattern Correspondence: The framework evaluates whether predictions correspond to documented attack patterns, measuring both recall (coverage of known attacks) and precision (avoiding false positive predictions) against our constructed corpus of historical attack chains.

Explainability and Actionability: We assess the quality of natural language explanations generated by the LLM component and whether security practitioners can use these explanations for informed decision-making about vulnerability prioritization and mitigation strategies.

5.4 Contribution to Evaluation Infrastructure

Our systematic approach to evaluation corpus construction addresses a critical need in the cybersecurity research community. The framework produces:

- Structured representations of documented supply chain attacks with complete vulnerability chains
- Validated synthetic attack scenarios that reflect realistic attack progression patterns
- Standardized evaluation metrics for multi-vulnerability prediction systems
- Benchmark datasets for comparing different approaches to vulnerability chain analysis

This evaluation infrastructure enables more rigorous assessment of future work in multi-vulnerability attack prediction and provides a foundation for advancing research in this area.

6 Threats to Validity

6.1 Training Data Scope

The HGAT model architecture is designed for Python CycloneDX SBOMs from the Wild SBOMs dataset. This creates several potential validity considerations: the approach may exhibit different performance characteristics across programming languages or ecosystems with different dependency management patterns, and the focus on open-source Python projects may not fully represent the complexity and vulnerability patterns found in enterprise or polyglot software systems.

Additionally, the scarcity of ground truth data for multi-vulnerability interactions necessitates synthetic labeling and semi-supervised learning approaches. The synthetic nature of training data may introduce biases that don't reflect real-world attack patterns.

6.2 Graph Construction and Feature Engineering Dependencies

The effectiveness of our approach depends on how software dependencies and vulnerabilities are represented as graph structures. Our choice of node types, edge relationships, and feature representations may bias the model toward certain types of attack patterns while missing others. The sensitivity of GNN performance to graph construction choices represents an inherent challenge in graph-based vulnerability analysis.

6.3 Evaluation Methodology Constraints

The evaluation of multi-vulnerability attack prediction systems faces fundamental challenges. Our evaluation corpus construction methodology relies on retrospective analysis of documented attacks, which may underrepresent attack patterns that haven't been publicly documented or may overrepresent attack types that receive more attention in security literature.

The temporal aspect of cybersecurity creates additional constraints: attack techniques evolve rapidly, and evaluation approaches must account for the dynamic nature of threat landscapes when assessing prediction accuracy.

6.4 LLM Integration Dependencies

Our hybrid approach leverages the function calling capabilities of current LLM architectures, which vary across different model families and versions. The effectiveness of the MCP-based integration depends on the underlying model's ability to use structured interfaces effectively, and performance characteristics may differ across LLM architectures with different reasoning capabilities.

The quality of LLM explanations and validation depends on the model's training data and domain knowledge representation, which may require updates as cybersecurity practices and attack techniques evolve.

6.5 Scalability Considerations

The approach demonstrates effectiveness on research datasets, though scalability to enterprise-scale software systems with thousands of components and complex dependency networks presents computational challenges. The requirements for graph construction, GNN inference, and LLM reasoning may create considerations for real-time vulnerability analysis applications.

The integration of GNN predictions into knowledge graphs also introduces complexity scaling factors, as the size and computational requirements of enriched graphs grow with the number of predictions and their supporting evidence.

7 Conclusion

We have presented a novel hybrid approach that combines Heterogeneous Graph Attention Networks with Large Language Models for multi-vulnerability attack chain prediction in software supply chains. This work addresses a critical gap in current cybersecurity practices, which analyze vulnerabilities in isolation and miss the complex attack vectors that arise from multi-component interactions. To our knowledge, this is the first system designed specifically for predicting and explaining vulnerability chains that span multiple software components.

Our contributions include: (1) a hybrid architecture that integrates GNN predictions directly into knowledge graphs, enabling LLMs to reason over both factual vulnerability data and predictive insights, (2) a comprehensive methodology for processing Software Bills of Materials and constructing heterogeneous vulnerability knowledge graphs, (3) an implementation of Model Context Protocol that provides LLMs with structured access to integrated prediction and vulnerability data, and (4) a systematic framework for evaluation corpus construction that addresses the fundamental challenge of evaluating multi-vulnerability attack prediction systems.

Our evaluation demonstrates the effectiveness of the approach through systematic methodology development and validation across seven vulnerable application repositories. The MCP-enhanced system achieved 87.5% F1.5 score, representing a 71.5 percentage point improvement over standalone LLM approaches and 12.6 percentage point improvement over traditional RAG systems. Testing across 193 vulnerabilities demonstrates robust performance improvements that generalize across diverse vulnerability analysis scenarios.

The broader implications of this work suggest that as software supply chains become increasingly complex, security analysis must evolve beyond individual vulnerability assessment to consider the complex risks that arise from component interactions. Our hybrid approach provides this evolution, combining the structured pattern recognition capabilities of graph neural networks with the interpretability and validation capabilities of large language models.

Data and Source Code Availability

Our implementation framework, evaluation methodologies, and constructed datasets support reproducibility and future research in hybrid AI approaches for vulnerability analysis. The evaluation corpus serves as a community resource to enable standardized benchmarking of multi-vulnerability prediction systems.

Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant No. 2349452. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

Anchore. 2025a. Anchore/grype.

Anchore. 2025b. Anchore/syft.

Anthropic. 2025. Model Context Protocol.

Aquasecurity. 2025. Aquasecurity/trivy.

FIRST. 2021. Exploit Prediction Scoring System (EPSS).

Fortinet. 2022. SolarWinds Supply Chain Attack.

Google. 2025. OSV - Open Source Vulnerabilities.

Kabul Kurniawan, Elmar Kiesling, and Andreas Ekelhart. 2024. CyKG-RAG: Towards knowledge-graph enhanced retrieval augmented generation for cybersecurity. In *Proceedings of the 1st Workshop on Formal Verification for Secure and Safe Software (FV4S)*, volume 3950 of *CEUR Workshop Proceedings*, pages 1–10, Baltimore. CEUR-WS.org.

Nancy Messieh. 2023. Software supply chain security: The dataset.

NTIA. 2021. Vulnerability Exploitability eXchange (VEX).

ORT. 2025. Oss-review-toolkit/ort: A suite of tools to automate software compliance checks.

OWASP. 2025a. CycloneDX Bill of Materials Standard | CycloneDX.

OWASP. 2025b. Dependency-Track | Software Bill of Materials (SBOM) Analysis.

MIT Technology Review. 2021. The internet runs on free open-source software. Who pays to fix it? | MIT Technology Review.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *Preprint*, arXiv:2302.04761.

Snyk. 2025. SBOM Security Checker.

Luís Soeiro, Thomas Robert, and Stefano Zacchiroli. 2025. Wild SBOMs: A Large-scale Dataset of Software Bills of Materials from Public Code. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 164–168, Ottawa, Canada. IEEE.

Tech Target. 2023. SolarWinds hack explained: Everything you need to know.

Tianchi Yang, Linmei Hu, Chuan Shi, Houye Ji, Xiaoli Li, and Liqiang Nie. 2021. HGAT: Heterogeneous Graph Attention Networks for Semi-supervised Short Text Classification. *ACM Trans. Inf. Syst.*, 39(3):32:1–32:29.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *Preprint*, arXiv:2210.03629.

Jiao Yin, Guihong Chen, Wei Hong, Hua Wang, Jinli Cao, and Yuan Miao. 2023a. Empowering Vulnerability Prioritization: A Heterogeneous Graph-Driven Framework for Exploitability Prediction. In *Web Information Systems Engineering – WISE 2023*, pages 289–299, Singapore. Springer Nature.

Jiao Yin, Wei Hong, Hua Wang, Jinli Cao, Yuan Miao, and Yanchun Zhang. 2024. A Compact Vulnerability Knowledge Graph for Risk Assessment. *ACM Trans. Knowl. Discov. Data*, 18(8):194:1–194:17.

Jiao Yin, MingJian Tang, Jinli Cao, Mingshan You, Hua Wang, and Mamoun Alazab. 2023b. Knowledge-Driven Cybersecurity Intelligence: Software Vulnerability Coexploitation Behavior Discovery. *IEEE Transactions on Industrial Informatics*, 19(4):5593–5601.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. 2023. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *Preprint*, arXiv:2309.01219.

Xiaojuan Zhao, Rong Jiang, Yue Han, Aiping Li, and Zhichao Peng. 2024. A survey on cybersecurity knowledge graph construction. *Computers & Security*, 136:103524.

Large Language Models for Memory-Safe Code Transpilation

Sarah Bedell
Department of Computer Science, University of Colorado Colorado Springs (UCCS) CO, USA & University of Georgia, USA
sarah.bedell@uga.edu

Nazanin Siavash
Department of Computer Science, University of Colorado Colorado Springs (UCCS) CO, USA
nsiavash@uccs.edu

Armin Moin
Department of Computer Science, University of Colorado Colorado Springs (UCCS) CO, USA
amoin@uccs.edu

Abstract

While security issues have always been a discussion with computers and documents, safety measures built within programming languages are a more recent construct, with one of the safest languages, Rust, being one of the youngest programming languages to exist. Therefore the idea of transpiling code from C++, one of oldest, most unsafe languages, to Rust, has become a problem of interest. Unfortunately, the difficult part of transpiling between the two is heavily dependent on adding the safety measures of Rust to the C++ program. Manually transpiling between the two is also largely inefficient and time-consuming, so recent research has been focused on whether LLMs can fix this problem. In this paper, we use Retrieval Augmented Generation (RAG) with LLMs to combat the issue. Our approach focuses on simplicity and connecting previous approaches into one full approach. Using these approaches, we find a continuous increase in memory security, with multiple programs ending with no raw pointer deferences or unsafe type casts.

1 Introduction

Code *transpilation* typically involves transforming the source code of a program written in one programming language to the source code of an equivalent program written in a different programming language. In this work, we conduct transpilation with the goal of producing the memory-safe variant of a program, which will be inherently more secure. To this aim, we transpile a program written in a memory-unsafe language, specifically C/C++, to its equivalent program written in Rust. Unlike C/C++, Rust is memory-safe by design. However, it also has another significant advantage: unlike other memory-safe programming languages, such as Python or Java, Rust is fast and efficient. Therefore, it is a reasonable choice for the target of the transpilation task.

Moreover, we are interested in automated methods and techniques that can carry out the transpilation task and convert code from one programming language to another. This has been a recent focus of many researchers, leading to recent developments in what can be considered the state-of-the-art, as well as multitudinous advancements in the idiomacy and accuracy of transpiled Rust code. It has also been a recent focus of the US Government, as the program TRACTOR, from DARPA shows (Agency). These methods and techniques use Artificial Intelligence (AI), specifically its subdiscipline Machine Learning, and in particular, state-of-the-art pre-trained Machine Learning models, called Large Language Models (LLMs), also known as Frontier AI Models or Foundation AI Models.

The contribution of this paper is to propose a novel approach to increasing memory safety within code transpilation based on a mixture of reliable LLMs and an SLM (gpt-4o, gpt-4-turbo, and o3-mini), all of which can be deployed for code transpilation. These LLMs were chosen specifically for their accuracy and reproducibility in code generation (Crupi et al., 2025; Zeng et al., 2025), as well as their accuracy in adding security measures to code (Bae et al., 2024).

We will use a Retrieval Augmented Generation (RAG) pipeline to further focus the LLM, adding to the novelty of our approach to the transpilation. To validate and evaluate the proposed approach, we use an abrieviated benchmark dataset provided by Nitin et al. (Nitin et al., 2025), which includes 7 programs in C++ and 10 programs translated prior by c2rust (Inc.).

Due to time constraints, we will be abbreviating the 17 program long dataset to the 7 coreutils programs for now. However, these programs are diverse in length and focus, and should still provide us with valuable results.

In particular, we answer the following Research Questions (RQs): RQ1. Can LLMs transpile C/C++

code to Rust code effectively and efficiently? RQ2. Can the common LLM *hallucination* problem be mitigated by using a Retrieval Augmented Generation (RAG) pipeline?

The rest of this paper is structured as follows. Section 2 provides some background information on code transpilation, LLMs, SLMs, RAGs, memory safety, and Rust. Afterward, Section 3 reviews the literature of automated code transpilation from C/C++ Into Rust, as well as literature on the usage of RAGs against LLM hallucination. Further, we propose our novel approach in Section 4. The experimental study in Section 5 validates and evaluates the proposed approach, comparing results to previous papers to verify it's improvement in the field. Additionally, Section 6 elaborates on the potential threats to validity of the research outcomes. Finally, we conclude and suggest future work in Section 7.

2 Background

This section provides a general background of code transpilation, LLMs, RAG, memory safety, and Rust, with a central focus on how these concepts relate to the proposed approach of the paper. The background is separate from the related works, which focuses more on the specifics of the architectures, results, and research of the papers that have contributed to the approach taken in this paper, and the architecture built for this research.

2.1 Code Transpilation

Code transpilation, the process of transferring source code from one programming language to another, was historically performed manually using extensive hand-crafted rules and requiring deep domain expertise (Lachaux et al., 2020). While this approach facilitated some early efforts, it proved to be both time-consuming and resource-intensive, ultimately making it inefficient and ineffective. As a result, manual transpilation methods saw limited adoption in practice. In recent developments of software research, there has been a continuous effort to use LLMs (2.2) to lower the time necessary to transpile code, yet keep the code usable, idiomatic, and efficient (Roziere et al., 2020).

Code transpilation serves various purposes; however, one of the primary focuses of this paper is the improvement of safety by transpiling a program from an unsafe language, such as C++, to a safe language, such as Rust. Memory safety will be given more detail later in (2.5). Transpiling code involves changing every function to match the new syntax, as well as making all of the types type-match.

2.2 Large Language Models (LLMs)

LLMs, in their recent developments, have become a large area of intrigue. While many LLMs are open source and freely available for various applications, it is often the case that user interactions contribute to ongoing training, helping improve the model's accuracy and performance(Hagos et al., 2024). LLMs can be used for myriad purposes - to create a story, to answer a question, or to fix/generate code. Despite being made of code themselves, getting LLMs to generate usable, and idiomatic code in higher-languages has proven a challenge to developers. Recent developments, however, have shown that many LLMs are now more proficient at generating usable code than the average developer, with many programs even offering AI assistance for code completion (Svyatkovskiy et al., 2019), however there are still issues with the code being idiomatic, or legible to human developers (Krebs and Mazumdar, 2025). With the increasing attention brought to LLMs, researchers now attempt to solve code transpilation through the new tool at hand (Bhatia et al., 2024), hoping to minimize computational costs and time of translation.

However, even with the rising abilities that the LLMs have shown, they are still wildly inaccurate, in many areas, and have a tendency to *hallucinate* in order to maximize efficiency. Hallucinations are shown by the LLMs conjuring up false information as a response to a prompt, creating distrust of LLMs, as well as spreading new falsities as truths.

2.3 Small Language Models (SLMs)

While less well-known compared to LLMs, Small Language Models have a few advantages compared to their larger brethren. They are trained on smaller datasets than the LLMs. This can lead to a trade-off of accuracy for efficiency. However, if you give them context, it can have a larger impact to your results, and recent studies have found that SLMs with RAG pipelines can perform better than general LLMs (Liu et al., 2024).

Similarly to LLMs, SLMs can be open or closed-source. Open-source SLMs, such as tinyllama, have become very popular online, as they are easily accessible and possible to run locally. SLMs can be highly effective with added context, and have the

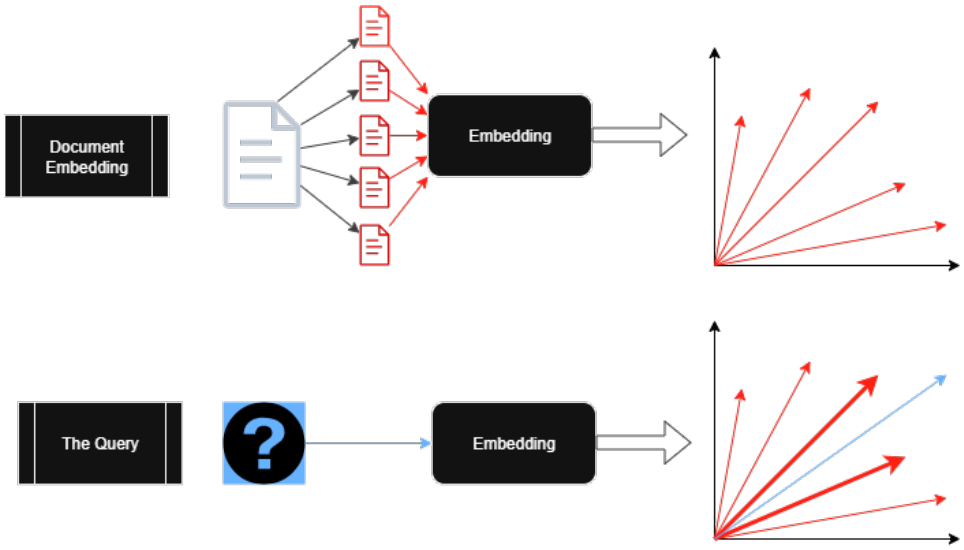

Figure 1: How RAG connects similarly chunked documents to a query.

computational efficiency from their smaller nature, making them useful for highly specified tasks (Lee, 2024).

2.4 Retrieval Augmented Generation (RAG)

RAG is a technique that enhances the quality of responses from an LLM by incorporating external context. When using RAG, the coding architect will create a RAG pipeline that allows the knowledge to be searched for, to find similarity to the query to create context for the LLM to generate an answer form. Figure 1 is a visualization of the way the embedded documents in a RAG work with the similarity search and the query. The two bolded vectors next to the embedded query represent the two most relevant document embeddings. Thus, they are added to the query to make the context for the user's prompt for the LLM.

The external context is able to be chosen and segmented by the architect, alllowing for specificity and focus. By retrieving relevant information, a RAG pipeline enables the LLM to access additional knowledge, resulting in more accurate and context-aware outputs (Church et al., 2024). Part of the enhancement from the additional focus is an ability to minimize the noise, or the unnecessary/unwanted information, that may cloud the LLM's search and lead it down the wrong path. Since noise is a main contributor to hallucinations, there is a revolving question in the field right now about how well RAGs can be used to minimize the issue of LLM hallucination. In other words, the integration of RAG with LLMs has led to a significant improvement in the effectiveness of the generated output (Okutan et al., 2024). Also, with the addition of context via RAG, LLM should be more effective in many areas - hopefully with the hallucination dilemma, as well.

2.5 Memory Safety

Memory safety refers to a program's ability to prevent unauthorized or unintended access to its memory, protecting against issues such as buffer overflows, dangling pointers, heap metadata overwrites, and other forms of memory corruption (Xu et al., 2021; Berger and Zorn, 2006). It becomes of huge importance for developers, as they need to be able to protect their programs, websites, and anything they publish, thereby ensuring safety. Unfortunately, from a safety perspective, a significant portion of older software is written in C or C++ programs, which are inherently unsafe languages. C and C++ predate many safety measures added to newer languages, such as Rust, Python, or Java. We can see this through the lack of type safety or locking. This common issue has led to extreme interest in the ability to translate C/C++ code into a safer language - such as Rust, however manual transpilation is often slow, inefficient, and quite difficult.

So, in order to make memory safety an efficient and plausible addition to older software, there has been growing interest in using LLMs to hasten the process and protect the programs.

2.6 Rust

Comparatively, Rust is a young programming language, as it was not released until 2013. Therefore, despite its multitude of safety measures and ability to ensure memory safety, it is less prominent than other higher-level languages (such as Python or Java). In spite of this, it's popularity is quickly gaining. Nonetheless, Rust's youth is part of its benefit, as it incorporates many new concepts designed to make its usage more security and memory safe and thus more trustworthy, but also has the proficiency and computational abilities on par with C (Yang et al., 2024a). Rust has the ability to be used safely or unsafely - although its automatic assumption is that it will be used for memory safety, so users have to use the key word *unsafe* to turn off all of the safety measures within the compiler, rustc.

3 Related Work

Research and ideas do not exist in a void, instead they are built upon one another, as ideas create pathways for new concepts to take root. Thus, this research is heavily dependent on all of the research that came before it, which was attempting to solve the same, or similar problems. There have been some programs created as a way to automate code translation, regardless of the language, such as UniTrans (Yang et al., 2024b), which increased the efficacy of the LLM attempting to aid in code transpilation.

3.1 Transpiling C++ Into Unsafe Rust

There have been a few attempts to automate the translation of C++ into a version of unsafe Rust, due to C++'s commonality in development, and the idea that unsafe Rust can be changed into Rust with safety measures later. Although there are still difficulties within the process, such as keeping the general idea of the program and matching types that may not have clear matches between them, there is one huge advantage of transpiling C++ into unsafe Rust - it is much easier. Rust can be made unsafe by simply adding unsafe to each function. This is a lot easier than adding all of the specific safety measures within the language to different types and making sure they still match, avoiding errors.

The first research into translating C into unsafe Rust came from (Inc.), which allows for free translation from C into non-idiomatic, unsafe Rust.

Following c2rust, a few papers, such as (Okutan et al., 2024), have focused on transpiling C++ into Rust, but now with a focus on syntax and efficiency, rather than on security and memory safety. This is a huge step forward that used the framework and results of c2rust as the background of their work. It allowed for the transpiled code to now have accuracy, idiomacy, and efficiency, but the code still lacked the necessary memory safety measures that makes Rust so beneficial.

In Figure 2, there's an example of C++ code being transpiled into Rust code. The small segment comes from the beginning of the program uniq in our dataset. This example comes from our first transpilation of the code using o3-mini. It shows understanding by the model the code's usage and intent and idiomatic results.

3.2 Transpiling C++ Into Safe Rust

While there is still some work to be done on the idiomacy and accuracy of the newly transpiled Rust code, the generally compilable Rust code now opens the door for looking into increasing the memory safety and security of the programs, once they've been transpiled into Rust.

Research does not exist in a vacuum, it is borne from the ideas that came before it. Therefore, this paper is building upon the ideas of (Nitin et al., 2025), which successfully transpiled C++ into a safer version of Rust, but only increasing the safety of the lines of code by 24%, and type casts by 8%. With the addition of LLM, but without the usage of RAG, LAC2R managed mixed results, compared to C2SaferRust with larger datasets (Sim et al., 2025). Using source-to-source transpilers has added to the efficacy of programs, as it works with API-safety, rather than just focusing on syntax (Ling et al., 2022).

Hong and Ryu have been working on each individual part of transpiling C++ into safer Rust - through locks (Hong and Ryu, 2023), union tags (Hong and Ryu, 2024a), and type casts (Hong and Ryu, 2024b). This work will be essential background for our proposed approach, as it shows the recent advancements in the attempts to migrate C++ into safe Rust programs. While we will not be focusing on locks and union tags, we will be focusing on type casts, similar to Hong and Ryu.

3.3 Using RAGs Against LLM Hallucination

There have been many advancements on the ability to minimize hallucinations by using RAG pipelines with LLMs. Recent developments have shown that

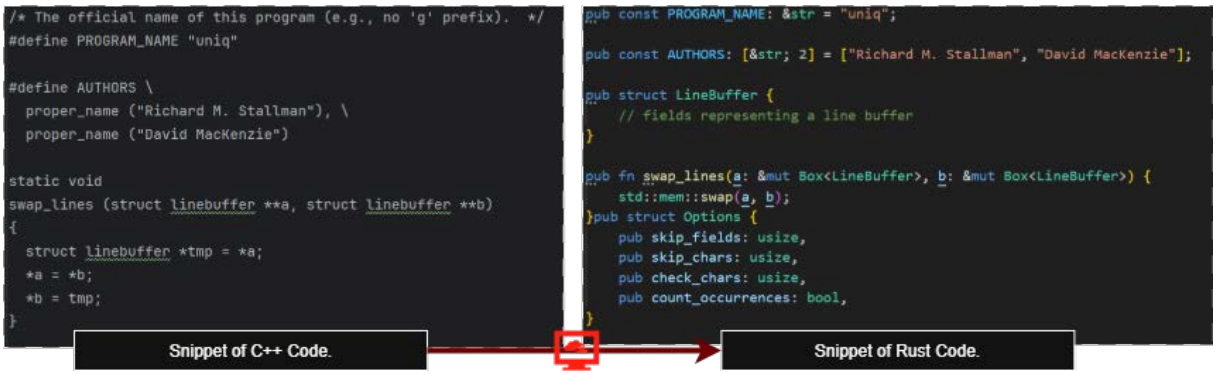

Figure 2: Transpiled code from C++ into Rust (using o3-mini.

using RAGs rather than straightforward parametric sequence-to-sequence models improves the accuracy with knowledge-intensive NLP tasks (Lewis et al., 2020). The sequence-to-sequence models mix parametric and non-parametric memory to maximize utility. However, sequence-to-sequence models are still inaccurate, however, as many have issues with narrowing down correlations between concepts, thus there has been recent research in using the RAG to focus on the intent of the query/prompt (Ko÷ et al., 2024).

The extra information downloaded helps to minimize hallucinations and allow code transpilation to work more effectively, as shown in (Okutan et al., 2024), where RAGs are used with LLMs to translate C++ into Rust. There has also been focus on if generation augmented retrieval can further improve the reliability (Shao et al., 2023).

Another modern focus on RAG implemented research has been on the previous failures experienced by LLMs and RAG-infused LLM pipelines, and how to mitigate that with knowledge graphs (Agrawal et al., 2024).

3.4 OpenAI Models In Code Transpilation

The three models we chose for this paper are

- gpt-4o,
- gp-4 turbo, and
- o3-mini.

These LLMs and SLM have been useful and accurate in code transpilation and code generation tasks in the past.

Gpt-4o has been reliable with memory safety and security, as shown in software security research, such as the research done by (Bae et al., 2024). It is also reliable at generating code that more users find reliable, compared to other LLMs (Huynh et al., 2025). It's high accuracy, reliability, and relatively quick generation time are all beneficial for this research. Gpt-4o was the first model we used due to its low computation costs, for an LLM.

We chose gpt-4 turbo because it has been optimized for accuracy and complex code generation, as shown by (Palla and Slaby, 2025). Gpt 4-turbo is designed to give high quality output, but is computationally expensive compared to other models.

The final model we chose was o3-mini. o3-mini is an SLM, compared to gpt-4o and gpt-4 turbo, which are both LLMs. This means that o3-mini has less of a knowledge base built from training and testing. Despite this, SLMs with a RAG pipeline tend to be as accurate as LLMs, (Liu et al., 2024). The main benefits of SLMs are the cheaper computational costs and quicker output, compared to LLMs.

4 Proposed Approach

The general concepts of the problems being approached here are certainly not novel, but this paper adds novelties in its approach through:

- Transpiling C++ into Rust with a main focus on memory safety and security instead of idiomacy.
- Implementing a RAG pipeline that focuses on specific areas of memory safety.
- Verifying the security and efficacy of LLMs by comparing their results with the results of the Rust compiler.

Figure 3 represents the approach this paper follows for when transpiling C++ code straight into Rust. This approach is tested through a condensed version of the dataset used by (Nitin et al., 2025),

which contained 7 coreutils programs, all in C++ code. The programs were originally chosen for their differing lengths and focuses, so results will not be affected by similarity, or be less accurate due to a lack of variety.

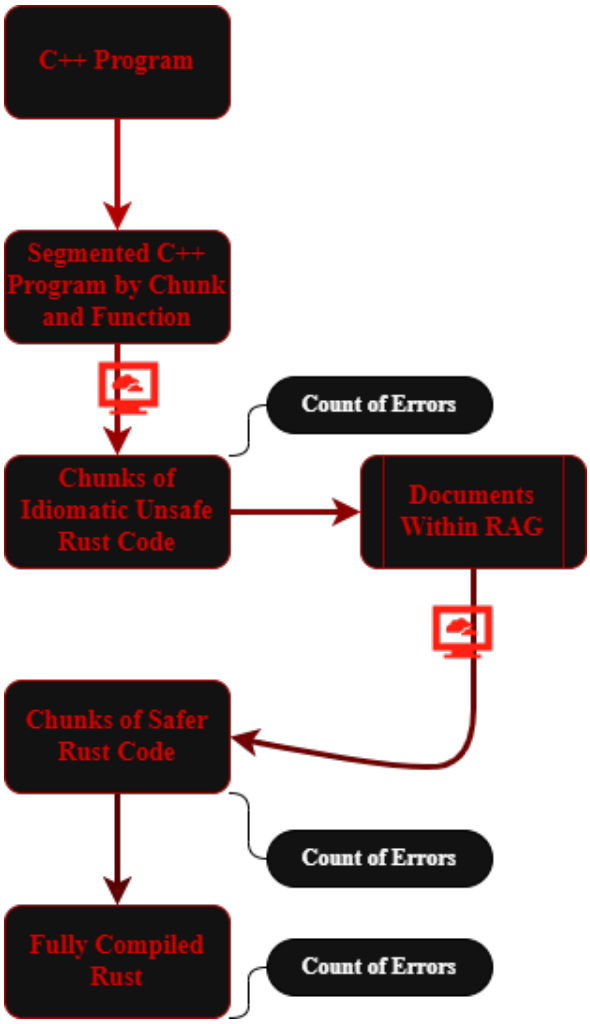

Figure 3: Overall Architecture of Our Approach

Our approach begins with the input C++ code targeted for transpilation. This code is segmented when two conditions are met:

1. The segment has reached a natural stopping point of the end of a function.

2. The function is greater than 500 characters.

Thus, we create snippets of code that are long enough for the LLM to understand the function but short enough to avoid overwhelming the system. It is also important to try and keep the segments as functional on their own, as possible. In smaller, early test cases, we found that stopping in the middle of a function tended to confuse the LLM and raise more errors by the rustcompiler.

Dataset	Program	# of Segments	# of LoC
coreutils	uniq	12	544
	cat	10	693
	pwd	12	333
	truncate	4	343
	head	20	932
	split	22	1494
	tail	42	2205

Table 1: Amount of Segments of Code per codeutils Program

Table 1 shows how dense each program is, and how many segments/chunks of code were created with this separation technique, and the Original amount of C ++ lines of code (LoC). To avoid token limits, we have it set to separate out any original transpilations that get too large, so many programs may end up with a different final amount of segments. The programs sizes range from 333 LoC to 2205 LoC, creating a variety of density and length for the dataset. The segments number is not related to LoC, but rather to character length and function, so the two are put side-by-side to allow for an understanding of the size of each program.

Before transpilation, a brief summary describing the overall goal of the program is generated. This summary is included with each segment to help preserve the functional intent of the original code. Each segment is then submitted to the LLM for transpilation into unsafe Rust. After the segments are transpiled into unsafe Rust, they will each be compiled into a new, safer Rust program, with a specific focus on fixing raw point dereferences and unsafe type casts. Following the initial transpilation, each segment is refined into safe Rust through a two-step process:

1. with the focus of general memory safety and security by the LLM, and

2. with context of the most likely errors to appear for raw point dereferences and unsafe type casts.

The resulting safe Rust segments are compiled into a single file to form a complete Rust program with the same functional goal as the original C++ version.

All of the separated rust programs are then provided to the RAG-assisted LLM. Memory bugs, i.e., unsafe type casts and raw pointer dereferences,

are examined before and after the application of security measures. There is also a check to see how many memory safety bugs were dispelled throughout the process.

Each section of the program is checked for raw point dereferences and unsafe type casts. The LLM counts the amount in the original Rust code and the safer Rust code, whilst the compiled Rust code will be fed to the Rust compiler, which will attempt to build the program - this will output any error messages. These numbers should allow for the testing of

1. the efficacy of our approach, and
2. the hallucination rate of the LLM.

In the Rust Compiler, the Error Codes E0133, E0392, and E0793 reference raw pointer dereferences, and the codes E0604, E0605, E0606, and E0607 reference unsafe type casts. Thus, we search through the output errors for these specific codes to validate that the LLM and RAG pipeline are efficient. We will also use a bash script to check for any unsafe type casting or raw point dereferences that may have occured without raising an error. These can exist within unsafe blocks (UBs) and thus unsafe lines of code (ULoCs).

Finally, to fully verify that there have been general security additions added - outside of our specific focuses, we will be using the bash script to see the change from the original code to the final code in the UBs and ULoCs.

4.1 Process

When creating the architecture for this paper, we originally started with much smaller sections of code: code separated out by a minimum length of 200 character functions. We ended up changing this to the 500 minimum to both

- allow for larger chunks of code so the LLM had more context per area,
- and minimize computational costs.

As we were building the interactions with the code, we realized an important part of this research was maximizing recreatibility, despite the improbability of LLMs to reproduce copies. To counter this, we added "temperature=0" and ".choices[0]" to all of our generating functions, as shown in 4.

Creating the predefined architecture of the approach for this paper has been largely dependent on previously existing RAG architectures. There is a comprehensive list of usable RAG pipelines that were studied for it's creation (FareedKhan-dev, 2025).

This process has carried a few challenges - the majority of which are related to the amount of tokens utilizable by OpenAI's GPT at any given moment. Another large challenge taking notice is quite similar, being how many tokens can be used within a minute by the LLM's API. The solution to these problems has been through chunking all of the documents that do not have specific separators - such as the Rust Programming Language book (Klabnik and Nichols, 2018).

In our approach, we originally separated the documents by chunks of 200 characters with a 20 character overlap. However, changing the separation to a 500 character chunk with 50 character overlaps lowered the output time significantly since there were fewer chunks to sort through for every prompt.

Some of the original coreutils programs originally translated into significantly larger sections of Rust, so they were lead through an additional process of chunking, so as not to overtake the token limitation. This chunking was done by a 5000 character limit for the original transpiled code so the LLM would have enough room for the segments and it's context. If the segment exceeded the 5000 character limit, we will rechunk it to chunks of 4000 characters with a 15 character overlap.

For this research, we had access to ample OpenAI models - since the focus of the research is to find if there is a general improvement with LLMs and RAG-pipelines, which necessitates the usage of multiple language models, we decided to use 3 OpenAI models with different focuses in their training. We used 2 LLMs, gpt-4o and gpt-4-turbo, and 1 SLM, o3-mini.

5 Experimental Results

The main focus of this paper is to find a way to transpile C++ code into Rust without sacrificing memory safety, as well as testing if a RAG pipeline on an LLM can minimize hallucinations. The results for this paper came from two techniques: using a script to find an instances of UTCs and RPDs and asking the LLM to calculate its estimation of UTCs and RPDs.

We have these two validation techniques so we can get accurate results of the efficacy of the pro-

```
def llmResponse(query):  1 usage
    systemPrompt = """You are a software engineer attempting to transpile C++ code into Rust.
    Your main goals are memory safety and idiomacy.
    Respond ONLY with your summary"""
    userPrompt = query
    response = client.chat.completions.create(
        model=model_id,
        messages=[
            {"role": "system", "content": systemPrompt},
            {"role": "user", "content": userPrompt}
        ],
        temperature=0
    )
    answer = response.choices[0].message.content
    return answer
```

Figure 4: One of our functions that garnered responses from the LLMs.

gram from our script that goes through the compiled code, and so we can see how accurate the LLM is in its attempt to notice and quantify UTCs and RPDs.

5.1 LLM - gpt-4o

In Table 2, we can see how many RPDs (raw point dereferences) and UTCs (unsafe type casts) were changed in the coreutils C++ code, from the first transpilation into Rust into the final, supposedly memory safe transpilation, based on the LLM's own verification and validation.

Of the seven coreutils programs, gpt-4o believes it has:

- created three programs with no RPDs and UTCs by the end,
- had two programs with an increase in UTCs, but none with an increase in RPDs,
- greatly decreased the amount of RPDs compared to UTCs.

It is accurate in all three of those generalizations, and in fact has ended with four programs having zero UTCs and RPDs. Gpt-4o is least accurate with the tail program, where it misses 26.5% of the RPDs in the Original Rust, overestimates the RPDs in the Final Rust, and hallucinates an increase in UTCs, when it's transpilation truly halved the UTCs.

5.2 LLM - gpt-4 turbo

In Table 3, we can see how many RPDs and UTCs were changed in the coreutils C++ code, from the first transpilation into Rust to the final transpilation into Rust with more memory safety measures added. The LLM's verification and validation can be compared with the results in our later table, Table 5.

Of the seven coreutils programs, gpt-4-turbo believes it has:

- created two programs with no RPDs and UTCs by the end,
- had one program with an increase in RPDs, but none with an increase in UTCs,
- gotten rid of all but one or two RPDs and UTCs in all but one program.

It is only inaccurate in its prediction that it has increased the amount of RPDs in one of its programs. It's specific counts of RPDs and UTCs are highly inaccurate, most likely because it has cleared the code of all RPDs and UTCs in 5 of the 7 programs, and has only left 1 UTC in the tail program. It seems to hallucinate a bit with its misinterpretation, and yet has created the most memory safe programs of Rust.

5.3 SLM - o3-mini

When we applied our architecture to the o3-mini model, we had to get rid of our temperature parameter. We can confirm that o3-mini should still produce decently reproducible results due to a paper by (Zeng et al., 2025). In this study, o3-mini consistently did well in reproducibility compared to other state-of-the-art models. This has given us

Model	Program	# of RPDs in Original Rust	# of RPDs in Final Rust	# of UTCs in Originl Rust	# of UTCs in Final Rust
gpt-4o	uniq	0	0	3	0
	cat	27	2	2	2
	pwd	0	0	0	0
	truncate	0	0	0	2
	head	0	0	0	0
	split	18	2	5	1
	tail	25	3	3	4

Table 2: Changes in number of RPDs and UTCs from first generated Rust to final generated Rust, according to gpt-4o.

Model	Program	# of RPDs in Original Rust	# of RPDs in Final Rust	# of UTCs in Originl Rust	# of UTCs in Final Rust
gpt-4 turbo	uniq	0	0	0	0
	cat	8	0	8	1
	pwd	1	2	1	0
	truncate	0	0	0	0
	head	n/a	n/a	n/a	n/a
	split	8	1	9	0
	tail	3	2	9	5

Table 3: Changes in number of RPDs and UTCs from first generated Rust to final generated Rust, according to gpt-4 turbo.

the confidence that we can work with it despite it's architecture rejecting the temperature parameter.

One thing we noticed immediately from the SLM o3-mini, is that it is much less accurate than the LLMs in the task of security and memory safety, yet very competent at the task of code transpilation. The o3-mini model gives itself extra security measures throughout its transpilation by adding

```
#![forbid(unsafe_code)]
```

and

```
#![deny(unsafe_code)]
```

as ways to stop UBs from compiling without the programmer's knowledge. This added step in memory safety makes it different from the way the previous LLMs had compiled the code.

However, it is apparent that o3-mini is hallucinating the most with how inaccurate its predictions are. None of the predictions match the compiler's actual outputs, except for the count of UTCs in the final Rust.

5.4 Rust Compiler

We can use rustc, the rust compiler, to count how many Error Codes connected to Raw Point Dereferences and Unsafe Type Casts we found. This can be used as a verification of the LLMs accuracy, as well as a test for hallucination within the LLM, since only specific codes align with raw point dereferences and unsafe type casts.

- RPD Error Codes: E0133, E0392, E0793
- UTC Error Codes: E0604, E0605, E0606, E0607

With this knowledge, we use a bash script to check for error codes of RPDs and UTCs. The script is also useful to check the code for unsafe blocks and functions, which may contain compilable RPDs and UTCs. In Rust, *unsafe* is a key word

Model	Program	# of RPDs in Original Rust	# of RPDs in Final Rust	# of UTCs in Original Rust	# of UTCs in Final Rust
o3-mini	uniq	9	4	11	0
	cat	7	9	9	9
	pwd	3	3	1	0
	truncate	3	1	9	0
	head	16	5	19	4
	split	18	2	5	1
	tail	38	8	31	16

Table 4: Changes in number of RPDs and UTCs from first generated Rust to final generated Rust, according to OpenAI's o3-mini.

to allow the compiler to ignore memory safety violations within the code.

Unsafe blocks of code can be small or large. The code below is an example of an unsafe block from our original output of cat's Rust program, from gpt-4o transpilation:

```
unsafe {
    libc::write(
        io::stdout().as_raw_fd(),
        wp as *const _, outsize
        )
    }
```

It is important to note that this example has an UTC within it. The UTC is boldened within the UB.

In Table 5, we can see how many RPDs and UTCs were left according to the compiler, despite the LLM's chosen outputs. This table gives us accurate results that we can use to compare with the state-of-the-art results from (Emre et al., 2021; Nitin et al., 2025; Sim et al., 2025). The previous papers calculated their accuracy in the percentage changes between the original Rust compilation and final Rust compilation.

By allowing the LLM, with help from the RAG's specific context, to define it's own efficacy before verifying the accuracy through the Rust compiler, we were able to validate the accuracy of the RAG pipeline, and check for hallucinations. We are able to see in a comparison of Tables 2, 3, and 4 with Table 5, we can see there is a slight variation of difference, but nothing remarkable.

Since our approach has allowed multiple of our programs to be memory safe enough to have 0 RPDs and/or 0 UTCs in both their original and final versions, we have represented them in Tables 6, 7, and 8, through "–". Any that have "!–" represent a change that went from 0 in the original version to a positive number in the final version. This is to show that while it cannot be calculated, it is not for the lack of change.

Both RPDs and UTCs are unsafe because they occur in unsafe blocks of rust code - allowing them to pass by the compiler's safety measures. Thus, how many unsafe blocks (UBs) there are within the code is also a measure of safety. C2SaferRust (Nitin et al., 2025) calculates this by counting the amount of unsafe lines of code (ULoCs) in their programs, so we shall do the same. The count for ULoCs helps show more information about the safety in the code.

Table 8 shows us the difference made within the entire program by showing how many ULoCs were fixed. Sometimes the UB cannot be removed, but moving information out of the block is still beneficial, as it minimizes the amount of information that exists in an unsafe area.

Since the amount of unsafe code is what is important, we will compare the change in unsafe lines of code. To get the change in ULoCs, we have averaged our changes, from the Original compiled Rust to the Final compiled Rust, in each program between the three models that we tested the code on.

From these tables, we can see that when we had unsafe code, with the RAG-pipeline assisted LLMs, we generally performed better than the state-of-the-art research has in adding memory safety and security.

6 Threats to Validity

With the usage of LLMs, there are many threats to validity, as well as risks that must be considered.

Model	Program	# of RPDs in Original Rust	# of RPDs in Final Rust	# of UTCs in Originl Rust	# of UTCs in Final Rust	# of UB in Original Rust	# of UB in Final Rust	# of ULoc in Original Rust	# of ULoc in Final Rust
gpt-4o	uniq	0	0	1	0	3	0	3	0
	cat	29	0	1	1	34	2	97	2
	pwd	0	0	0	0	0	0	0	0
	truncate	0	0	0	1	2	2	2	2
	head	0	0	0	0	2	0	11	0
	split	12	0	4	0	5	7	99	19
	tail	34	1	6	3	25	11	72	19
gpt-4turbo	uniq	0	0	0	0	1	0	3	0
	cat	5	0	0	0	10	1	35	1
	pwd	0	0	1	0	1	2	7	5
	truncate	0	0	0	0	0	0	0	0
	head	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	split	0	0	3	0	15	1	114	1
	tail	0	0	1	1	17	14	45	24
o3-mini	uniq	1	4	0	0	7	5	120	10
	cat	7	8	9	11	39	20	52	66
	pwd	0	3	0	0	0	4	0	12
	truncate	0	0	0	0	4	5	4	5
	head	0	7	11	4	24	14	51	38
	split	4	0	13	14	37	27	307	132
	tail	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5: Count of RPDs and UTCs in rustc compiled code.

LLMs can be inaccurate and easily misused, as well as falsifiers, with their hallucinations. These may make our results difficult to reproduce. While we have confidence that using this approach will generate very similar results, due to the precautions taken, we are well aware that LLMs can make results that are difficult to recreate, which can also be a threat to our validity in this research. While we are attempting to minimize any threats to validity, it is essential to acknowledge these potential challenges and risks that may befall us.

Ways in which we have attempted to limit the potential threats to the validity of our research include:

1. using multiple LLMs to vary our results,
2. setting the temperature of the model to zero,
3. using a similarity search to give relevant context to the LLM,
4. taking measurements in a multitude of areas.

Each of these focuses on validating our research have been implemented throughout our architecture's continuous process.

While we believe these have limited the potential threats to the validity in our research, we understand that there are still challenges and risks we may not have anticipated or been able to stop.

7 Conclusion and Future Work

Memory safety is one of the most popular topics in computer science right now, with there being a constant effort to translate unsafe C++ legacy code into safer Rust code. Other memory-safe languages, such as Python and Java have been considered, but Rust's similarity to C allows it to work at a quicker pace, making it more efficient and useful. With a new focus on whether or not LLMs can make this less costly, more time-efficient, and more effective, this paper has focused on the ability to utilize RAG-assisted LLMs to translate C++ into Rust, using the benchmarks of (Nitin et al., 2025) to verify the

Program	Us (gpt-4o)	Us (gpt-4 turbo)	Us (o3-mini)	C2SaferRust	LAC2R
uniq	–	–	-300%	27%	60%
cat	100%	100%	-14%	24%	51%
pwd	–	–	!–	24%	54%
truncate	–	–	–	19%	58%
head	–	n/a	!–	21%	43%
split	100%	–	–	18%	43%
tail	97%	–	n/a	22%	27%

Table 6: Average Change in RPDs by Different Approaches

Program	Us (gpt-4o)	Us (gpt-4 turbo)	Us (o3-mini)	C2SaferRust	LAC2R
uniq	–	–	–	11%	48%
cat	0%	–	-22%	12%	48%
pwd	–	100%	–	0%	56%
truncate	!–	–	–	8%	47%
head	–	n/a	64%	14%	48%
split	100%	100%	-8%	6%	37%
tail	50%	0%	n/a	6%	38%

Table 7: Average Change in UTCs by Different Approaches

accuracy of our method. RAG has been added to the LLM method to both minimize hallucinations and maximize the efficacy of the LLM.

We have approached this by segmenting the C++ code, and then sending it through an LLM multiple times - first to transpile it into idiomatic Rust code, then to add specific memory safety measures, then to attempt to count the potential errors it would find in it.

Our results have shown that LLMs are adept at improving the memory safety of a program, as shown in Table 5. Our results are also consistently better than the state-of-the-art in fixing RPDs in the programs, and minimizing lines of unsafe code. While there is a visible difference between the models assumed errors and the calculated errors, the gpt-4o model is usually within an error range of up to 3 miscalculated per 10 actually recorded, whilst gpt-4-turbo assumes much higher amounts than existent.

While the LLMs have shown consistent improvement and great results, our SLM has had very different results. It has done very well at disabling the usage of unsafe code in the Rust program. However, despite this benefit, it has also demonstrated a consistent will to increase how many UBs are in a program and generate more RPDs, UTCs, and ULoCs. Both of which are only usable if the programmer turns off the added safety step and allows unsafe code.

Our results sometimes include an increase in UBs of Rust, but with a decrease in total ULoCs, and thus a decrease in total code within all unsafe blocks. We believe that the ULoCs are more important than the UBs, since it is a measure that shows how much information is truly not being checked for safety measures and could contain security issues.

For future work, we would like to reconfigure this experiment with larger sets of code to verify its ability to be useful for larger programs. We would like to start with the 10 LAERTES benchmark datasets we were unable to perform this on. However, the beginning of the process would have to be modified to allow it to reconfigure Rust code, rather than transpile straight C++ code.

Data and Source Code Availability

The source code is available in a Github repository [1].

8 Acknowledgements

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant No. 2349452. Any opinions, findings, conclusions, or recommendations expressed in this

[1] https://github.com/qas-lab/reu-sarah-bedell/tree/main/

Program	Us (gpt-4o)	Us (gpt-4 turbo)	Us (o3-mini)	C2SaferRust	LAC2R
uniq	100%	100%	92%	28%	48%
cat	98%	97%	-27%	26%	48%
pwd	–	29%	!–	25%	56%
truncate	0%	–	-25%	20%	47%
head	100%	n/a	25%	25%	48%
split	81%	99%	57%	18%	37%
tail	74%	47%	n/a	24%	38%

Table 8: Change in ULoCs by Different Approaches

material are those of the authors and do not necessarily reflect the views of the NSF.

References

Defense Advanced Research Projects Agency. TRACTOR: Translating All C To Rust. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 07/18/2025.

Garima Agrawal, Tharindu Kumarage, Zeyad Alghamdi, and Huan Liu. 2024. Mindful-rag: A study of points of failure in retrieval augmented generation. In *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*, pages 607–611.

Jaehyeon Bae, Seoryeong Kwon, and Seunghwan Myeong. 2024. Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques. *Electronics*, 13(13).

Emery D Berger and Benjamin G Zorn. 2006. Diehard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 41(6):158–168.

Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. 2024. Verified code transpilation with LLMs. *Advances in Neural Information Processing Systems*, 37:41394–41424.

Kenneth Ward Church, Jiameng Sun, Richard Yue, Peter Vickers, Walid Saba, and Raman Chandrasekar. 2024. Emerging trends: a gentle introduction to rag. *Natural Language Engineering*, 30(4):870–881.

Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the effectiveness of llm-as-a-judge for code generation and summarization. *IEEE Transactions on Software Engineering*, pages 1–17.

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating c to safer rust. *Proc. ACM Program. Lang.*, 5(OOPSLA).

FareedKhan-dev. 2025. Fareedkhan-dev/all-rag-techniques: Implementation of all rag techniques in a simpler way. Accessed: 06/16/2025.

Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. 2024. Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives. *IEEE Transactions on Artificial Intelligence*, 5(12):5873–5893.

Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An automatic c-to-rust lock api translator for concurrent programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 716–728.

Jaemin Hong and Sukyoung Ryu. 2024a. To tag, or not to tag: Translating c's unions to rust's tagged unions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 40–52, New York, NY, USA. Association for Computing Machinery.

Jaemin Hong and Sukyoung Ryu. 2024b. Type-migrating c-to-rust translation using a large language model. *Empirical Softw. Engg.*, 30(1).

Larry Huynh, Yinghao Zhang, Djimon Jayasundera, Woojin Jeon, Hyoungshick Kim, Tingting Bi, and Jin B. Hong. 2025. Detecting code vulnerabilities using llms. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–414.

Immunant Inc. c2rust. Accessed: 07/18/2025.

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

Robin Ko÷, Mustafa Kağan Gürkan, and Fatoş T. Yarman Vural. 2024. Rerag: A new architecture for reducing the hallucination by retrieval- augmented generation. In *2024 9th International Conference on Computer Science and Engineering (UBMK)*, pages 961–965.

Rasmus Krebs and Somnath Mazumdar. 2025. Deploy, but verify: Analysing llm generated code safety. In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 13–16.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Yo-Seob Lee. 2024. Analysis of small large language models(llms). *International Journal of Advanced Smart Convergence*, 13(4):155–160.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. 2022. In rust we trust – a transpiler from unsafe c to safer rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 354–355.

Suqing Liu, Zezhu Yu, Feiran Huang, Yousef Bulbulia, Andreas Bergen, and Michael Liut. 2024. Can small language models with retrieval-augmented generation replace large language models when learning computer science? In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, page 388–393, New York, NY, USA. Association for Computing Machinery.

Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques. Accessed: 07/28/2025.

Ahmet Okutan, Samuel Merten, Christoph C. Michael, and Ben Ryjikov. 2024. Leveraging rag-llm to translate c++ to rust. In *2024 International Conference on Assured Autonomy (ICAA)*, pages 102–105.

Dominik Palla and Antonin Slaby. 2025. Evaluation of generative ai models in python code generation: A comparative study. *IEEE Access*, 13:65334–65347.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9248–9274, Singapore. Association for Computational Linguistics.

HoHyun Sim, Hyeonjoong Cho, Yeonghyeon Go, Zhoulai Fu, Ali Shokri, and Binoy Ravindran. 2025. Large language model-powered agent for c to rust code translation. Accessed: 07/25/2025.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2727–2735, New York, NY, USA. Association for Computing Machinery.

Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–25.

Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024a. Vert: Verified equivalent rust transpilation with large language models as few-shot learners.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024b. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608.

Qiuhai Zeng, Claire Jin, Xinyue Wang, Yuhan Zheng, and Qunhua Li. 2025. An analyst-inspector framework for evaluating reproducibility of llms in data science. Accessed: 07/25/2025.

Large Language Models for Software Security Weakness and Vulnerability Mitigation

Michael Conner[1], **Terrance E. Boult**[2], and **Armin Moin**[3]

[1]Department of Computer Science
[1]University of Colorado Colorado Springs (UCCS), Colorado, United States, mconner@uccs.edu
[2]Department of Computer Science
[2]University of Colorado Colorado Springs (UCCS), Colorado, United States, tboult@uccs.edu
[3]Department of Computer Science
[3]University of Colorado Colorado Springs (UCCS), Colorado, United States, amoin@uccs.edu,

1 Abstract

Automated vulnerability mitigation methods are a highly desired system for developers and maintainers of large code-bases. These methods may potentially have significant impact on the effectiveness and efficiency of the work of software developers and software security experts tasked with these roles. However, current solutions are typically designed for a single language and seldom produce code that ”best fits” within a developer’s project. In addition, existing methods suffer from a typical over-fitment problem that potentially reduces their expansion potential and accuracy. As such, current solutions often fail to identify and correct potential vulnerabilities when exposed to new and unseen code. Existing methods also lack a significant ability to provide new information and context to improve the success rate of generated patches. Using recent advancements in code analysis and Large Language Models (LLMs), we are proposing a new technique to not only identify vulnerabilities but also offer possible vulnerability patches and suggestions based on in-depth code analysis and generation rather than code matching. In fact, our current approach has demonstrated an improvement of over 10% more valid and applyable patches than existing techniques and methods.

2 Introduction

Software security is a key component of modern Cybersecurity. In fact according to the Identity Theft Resource Center (ITRC, [n. d.]) the amount of databreach notices increase 211% each year. As such, maintaining the security of cyberspace is of crucial importance to all interests, from national security to economic stability and prosperity. However, at the heart of any Cybersecurity initiative are the backbone programs themselves. In fact, according to Statista(CVE-Details, [n. d.]), there were over 25,000 vulnerabilities identified in 2024, and although lower than 2023, the trend points to increased vulnerabilities each year. With growing code-bases, patching vulnerabilities is not a trivial task. As such, in this work we aim to deploy Artificial Intelligence (AI) to help carry out automated repair for code snippets that are known to be prone to specific vulnerabilities or weaknesses that may open a program to exploitation and subsequent malicious activity. Specifically, we will be leveraging the AI subdiscipline of machine learning by using and evaluating current state-of-the-art pre-trained models known as LLMs, Frontier AI Models or Foundation AI models.

Because of their large knowledge-base developed through extensive training datasets, current state-the-of-art LLMs such as GPT4.1(OpenAI-GPT41, [n. d.]) are more than capable of different coding tasks from code-completion to code-generation. In fact, using LLMs for different Software Engineering (SE) is not a new concept either. Over the recent years, LLMs have established a proven track record in a number of automated Software Engineering (SE) and software security tasks, including automated program repair methods such as (Ribeiro, 2023) and vulnerability analysis methods (Gokkaya, 2025) and (Huynh et al., 2025).

As such, Automated Vulnerability Repair (AVR) is also not a new concept, with different approaches to create patches. However, major concern with several current approaches to AVR, such as VulnFix(Zhang et al., 2022) is the over-fitment problem which raises a major scalability concern with larger, more diverse code projects. In addition, the limited vulnerabilities and/or weaknesses each method is trained on further reduces the diversity of possible patches each method may generate. In this paper, we are interested in mitigating security exposures that result from known security

weaknesses and/or security vulnerabilities.

Our key contribution is to propose a novel approach based on a number of different LLMs that can be deployed for Automated Program Repair (APR) to mitigate security vulnerabilities. By using "conventional" state-of-the-art LLMs such as GPT4.1, we also aim to avoid the over-fitment problem(Bridewell et al., 2005) by leveraging the diverse datasets modern LLMs are trained. As such key novelty of this work is enhancing the performance of LLMs and preventing their so-called *hallucination* by using a Retrieval Augmented Generation (RAG) pipeline to generate a patch file in the Unified Format. To validate and evaluate the proposed approach, we use available benchmark datasets, such as CVEfixes (Bhandari et al., 2021). In particular, we answer the following Research Questions (RQs): RQ1. Can LLMs perform code repair for software security vulnerability mitigation effectively and efficiently? RQ2. Can a RAG pipeline enhance the LLMs' performance in code repair for vulnerability mitigation?

The rest of this paper is structured as follows. Section 3 provides some background information on program repair, the CVE (cve, [n. d.]), NVD (nvd, [n. d.]), and CWE (cwe, [n. d.]) datasets, LLMs, and RAGs. Afterward, Section 4 reviews the literature of current state-of-the-art methods of APR. Further, we propose our novel approach in Section 5. The experimental study in Section 6 validates and evaluates the proposed approach. Additionally, Section 8 elaborates on the potential threats to validity of the research outcomes. Finally, we conclude and suggest future work in Section 9.

3 Background

Large Language Models (LLMs) are one of the most well-known applications of generative AI. Typically, these models are trained on a large corpus of text, containing billions of phrases or complete sentences to identify patterns such as grammar and word relationships. Using these patterns, LLMs are designed to generate or predict text from given sentences or prompts. In fact, recent breakthroughs such as the transformer model allow LLMs such as GPT4.1 and Llama 4 to generate text that is remarkably close to "regular" speech. By including other forms of text, such as source code, models such as CodeT5 (Yue Wang, 2021) have even been trained to analyze and generate source code in different programming languages such as C/C++ or Python.

While general-purpose LLMs have demonstrated exceptional capability at answering questions or writing stories using training data, specialized LLMs, Ranking Model have also been developed. Ranking LLMs represent a slightly different approach by leveraging supervised learning to help categorize and sort information based on identified features. In turn, this information is used to calculate a relevance score based on a given prompt to select the best information to use when generating a reply. These models can be trained for a variety of purposes such as customer recommendations, information gathering and processing and even categorizing security vulnerabilities and weaknesses.

Within the complexity of current large codebases, there is always a chance that some code may introduce a bug, vulnerability or weakness. In the context of computer programming, a vulnerability is typically a bug that can be exploited by a malicious actor for a nefarious purpose. A weakness, on the other hand, is a bug that may be used by a malicious actor in the future to perform malicious activity. Weaknesses can typically be combined to create a complete vulnerability within the target program. Typically the behavior such as attack method and end result are categorized as CVE or CWE codes for vulnerabilities and weaknesses respectively by organizations such as Mitre or NIST. Identifying and repairing these code defects quickly and effectively remains an important aspect of modern software programming.

Currently, several organizations freely identify, categorize and log all potential and exploited vulnerabilities and weaknesses. A service provided by the Mitre Corporation, CVE (cve, [n. d.]), or Common Vulnerabilities and Exposures, is the most widely used catalog for vulnerabilities that have been identified and disclosed. Each CVE record in the catalog identifies one vulnerability in a specific piece of software, as well as potential methods of exploit and end results of a successful exploit. In addition, each CVE record may also contain associated CWE codes that contributed to the vulnerability. CWE (cwe, [n. d.]), or Common Weakness Enumeration, is another service provided by the Mitre Corporation. Instead of documenting vulnerabilities identified in software programs, CWE records document weak-

nesses that can potentially build up to a vulnerability. These weaknesses can typically be studied to identify a root cause for an exploited system to better understand and repair potentially exploitable code. Lastly, NVD (nvd, [n. d.]) or National Vulnerability Database is a U.S. government sponsored repository provided by the National Institute of Standards and Technology (NIST). Similar to CVE, the NVD provides records that with information such as descriptions of the vulnerability and potential exploit methods for identified in a software program. Ultimately, each of these repositories also assign a CVSS, Common Vulnerability Scoring System, rating for each vulnerability based its ease of exploitation and potential impact on a scale of 0 to 10.

Lastly a Unified Format diff patch (diff, [n. d.]), or simply Unified Format, is the most commonly used code patch format. Almost all conventional and commonly used version control software, such as Apache Subversion (SVN, [n. d.]) and Git (GIT, [n. d.]) support this format for patching repositories. A unified diff consists of two main parts: the initial header and a series of groups of changes called hunks. The initial header contains the original file to be patched, denoted by the three dashes, and the new file, denoted with three plusses. Typically, the new file retains the same name as the original file. This header typically contains an optional timestamp after the filename in the "YYYY-DD-MM HH-MM-SS.MS - timezone" format. Followed by this header are a series of hunks that with two pairs of '@' symbols that surround a reference to the line number in the original source code that the hunk refers to and how many lines are referenced. This header also contains the line number of the new code, which may be different depending on previous lines changed, and how many lines the new code refers to. An example of a diff patch is highlighted by 1 showing lines that are to be removed in red and new lines in green. Lines that remain unaffected, and typically used to provide context, are not highlighted. With its widespread support, ease of use, and documentation, the Unified Format represents a superior solution for automated LLM-based vulnerability and weakness mitigation.

4 Related Work

Due to the significant demand for Automated Vulnerability Repair (AVR), multiple possible solutions have been proposed and demonstrated. Although these methods have shown promising results, they are not without shortcomings.

VRepair is one of the most commonly used, and compared to, methods of Automated Vulnerability Repair proposed by Chen *et al.*(Chen et al., 2022). By analyzing two years of commits on GitHub and combining the data with information from common vulnerability databases, VRepair was able to create a sizable dataset suitable for training a deep learning neural model. In turn, VRepair attains almost double the accuracy compared to models that are not trained with this method. However, VRepair is designed only to provide patches for existing vulnerabilities that it was trained on, thus limiting its direct use in the future. In addition, VRepair's method relies on matching vulnerable code tokens to matched pairs of vulnerable/patched code. This creates a high possibility of inaccurately generated patches when it is provided code that does not relate to code it has been trained with. Lastly, being designed only for C, VRepair may not be usable in large projects that make use of multiple languages.

VulRepair is a method proposed by Fu *et al.* (Fu et al., 2022) that employs CodeT5 pre-trained on a database with more than 8 million functions. Using Byte Pair Encoding (BPE) to create smaller character pair tokens and a neural network, VulRepair has demonstrated an accuracy improvement of over 10% compared to other proposals such as VRepair (Chen et al., 2022). However, VulRepair shows severe shortcomings with longer functions and repair patches, dropping from an average prediction rate of 70% to 32%. In addition, when processing functions, it is unable to process functions with more than 512 tokens, typically truncating the additional tokens and further reducing the accuracy of the algorithm. Similarly, VulRepair has a limit of 20 repair patch tokens, with accuracy dropping below 60% with token lengths greater than 10. These shortcomings limit VulRepair's feasability on large codebases which may contain numerous, lengthy functions.

FAVOR (Dong et al., 2025) is one of the most promising AVR proposals, using a different approach to analyze and patch potentially vulnerable code. Proposed by Dong *et al.* FAVOR relies on a control flow graph (CFG) to better analyze the context, patterns and operation of provided code for improved vulnerability detection

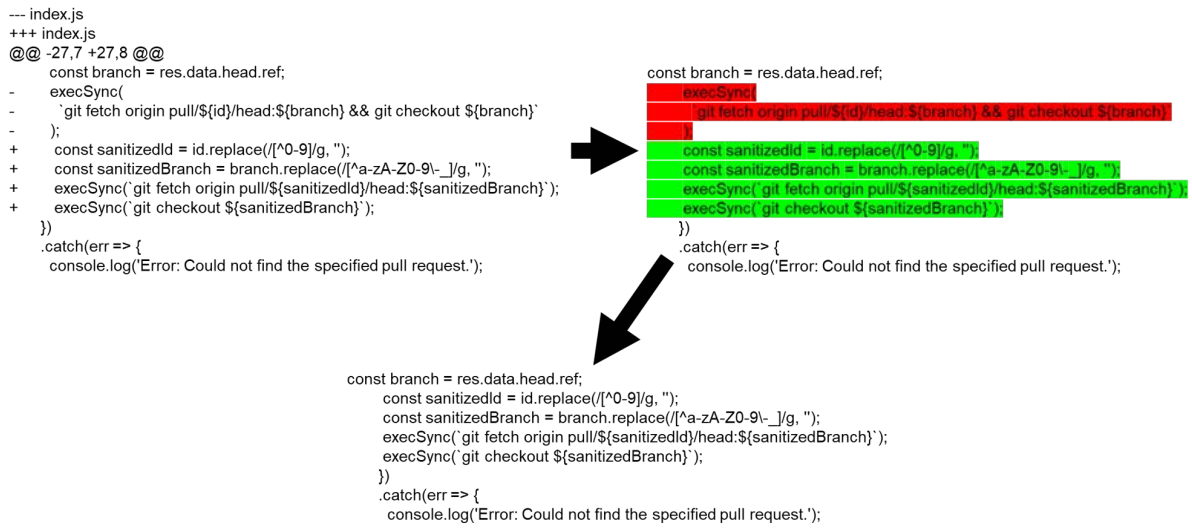

Figure 1: Example of Diff patching using a GPT4-Turbo generated patch for CVE-2018-25083

and patch generation. In addition, FAVOR also analyzes dependencies of the code, such as headers, to better identify the context of the code to produce more meaningful patches. However, FAVOR only relies on one dataset, BigVul, rather than analyzing real-world commits as part of the training method. While this reduces the difficulty in acquiring, curating and training data the side effect is a reduced amount of exposure to real-world coding examples. Secondly, FAVOR is only designed for C/C++, reducing its potential in current development pipelines that are turning toward memory-safe languages.

Automated Vulnerability Repair: Methods, Tools, and Assessments proposed by Hu *et al.* (Hu et al., 2025) is a novel approach to AVR that analyzes different tools and methods to identify and patch potentially weak or vulnerable code. Notably, in the context of our proposal, it mentions different methods of using LLMs to generate patches. Of note are the training, fine-tuning and prompt engineering approach. In addition, this approach also provides a standardized evaluation and validation approach for generated patch analysis. However, this approach is primarily designed for analyzing and patching C/C++ vulnerabilities rather than our Python, Java and Javascript targets.

Proposed by Zhang *et al.* VulAdvisor (Zhang et al., 2024) is a solution that, instead of providing direct code patches, uses natural language to offer repair suggestions for vulnerable code. Using CodeT5 and a Vector Space Model to improve pre-training, VulAdvisor is able to better identify local context within code functions, such as variables, to better identify vulnerabilities. In addition, this improved context identification also improves the accuracy and quality of suggestions, allowing them to be significantly more useful and relevant to the developer. However, being a suggestion only method, VulAdvisor is not suitable for direct use in an automated repair solution. In addition, VulAdvisor is only trained for C/C++, further limiting its use in codebases with a diversity of languages.

Maltracker (Yu et al., 2024) by Wang *et al.* is a novel LLM-based method to identify malicious packages in the Node Package Manager (NPM) for JavaScript. By extracting malicious code from existing packages using the LLM, identifying the behavior, and summarizing the findings, Maltracker is able to attain a false negative rate of around 7%. Maltracker ultimately involves analyzing malicious packages to produce features from packages to train a model to predict vulnerabilities based on features produced from target packages. This ultimately identifies potentially malicious lines of code within publicly available packages stored on NPM. However, this proposal is only designed for JavaScript and does not provide any repair or mitigation solutions.

Proposed by Gokkaya, Leveraging Large Language Models for Security Patch Detection and CWE Classification represents a current state-of-the-art approach to using LLMs to identify potential vulnerabilities or weaknesses (Gokkaya, 2025). This approach used specially designed con-

versations, prompts and roles to examine provided source code for potential weaknesses or vulnerabilities. In addition, the approach examined different LLMs and their performances in regards to accuracy, precision and recall. This approach showed that state-of-the-art models such as GPT-4o were able to achieve over 70% accuracy when identifying vulnerabilities and weaknesses.

5 Proposed Approach

As illustrated by 2, the patching process involves a series of steps to generate a Unified Format diff patch as well as analyze and apply the generated patch. Ultimately, when provided with vulnerable code, the LLM will provide a usable patch for the identify vulnerability or weakness. By providing just a Unified Diff format patch, we can reduce token generation time while providing additional patch possibilities for the developer. This will allow for seamless application of the patch across different projects in a Continuous Integration (CI) fashion. Currently, we will focus on patching Python, Java, and Javascript code due to the prevalence of these languages and their platform-agnostic nature, allowing ease of testing and evaluation. The following is an overview of the patch generation and application process. In addition, we will also evaluate how the addition of a RAG pipeline will influence the results. The RAG context will involve the use of CVE or CWE descriptions while providing real-world patch examples to help ”guide” the LLM. In addition, we will provide the description of a Unified Diff patch. Under the RAG experiment phase, we will use the same steps and conversation prompt generators as before, except with the addition of the necessary context information within the first step. As part of the evaluation of the RAG addition, we will compare the generated patches, source code and any errors or deficiencies between the RAG-influenced and non-RAG approaches.

- **Step 1: Prompt Engineering** This step will involve creating a prompt for the LLM to produce a patch.
- **Step 2: Patch Generation** This step will involve processing the prompt’s response to provide an output patch.
- **Step 3: Patch application and Evaluation** This step involves using the LLM to apply the generated patch to the vulnerable or weak code.
- **Step 4: Patched code testing and Evaluation** In this step, we will evaluate the generated code using a linting utility to check for possible syntactical errors. Passing code will then be executed to verify the code works as intended without runtime errors. This step ultimately verifies the quality and performance of the patched code.

5.1 Step 1: Prompt Engineering

Our proposal involves the use of an LLM to automatically patch and repair vulnerable or weak code. We aim to determine the best method to prompt state-of-the-art LLMs such as GPT4-Turbo, GPT4o, and Hermes-Llama3.5 to repair vulnerable code. During this step, we will identify the best conversation prompts to guide the LLM to create a patch to repair known vulnerabilities in provided code. However, we are not prompting the LLM to identify vulnerabilities. This process will also involve creating a standard and reproducible method of both prompting and providing the source code from existing code repository databases. Each LLM will be provided with the same prompt, the same vulnerability or weakness, and the same source code. During this step, we will also evaluate the implementation of a RAG pipeline by providing necessary context from the datasets in use. This will involve testing three different RAG approaches. The will provide both the explanation of the CVE or CWE to patch and an example of the actual patch for the CVE or CWE. The second will just provide the explanation while the third will provide just a code example. These will be labelled as RAG1, RAG2, and RAG3 respectively. In this case, the context can simply be queried by CVE or CWE number to quickly find the correct descriptions and example patch code without needing to compute embeddings for a similarity search. For each RAG approach, the same prompts used previously will be provided, with the difference being the added context depending on the RAG approach. The exact prompts provided are shown with 2.

5.2 Step 2: Patch Generation

In this step, we will pass the generated prompts to different state-of-the-art LLMs. Each LLM will produce a response as a Unified Format patch.

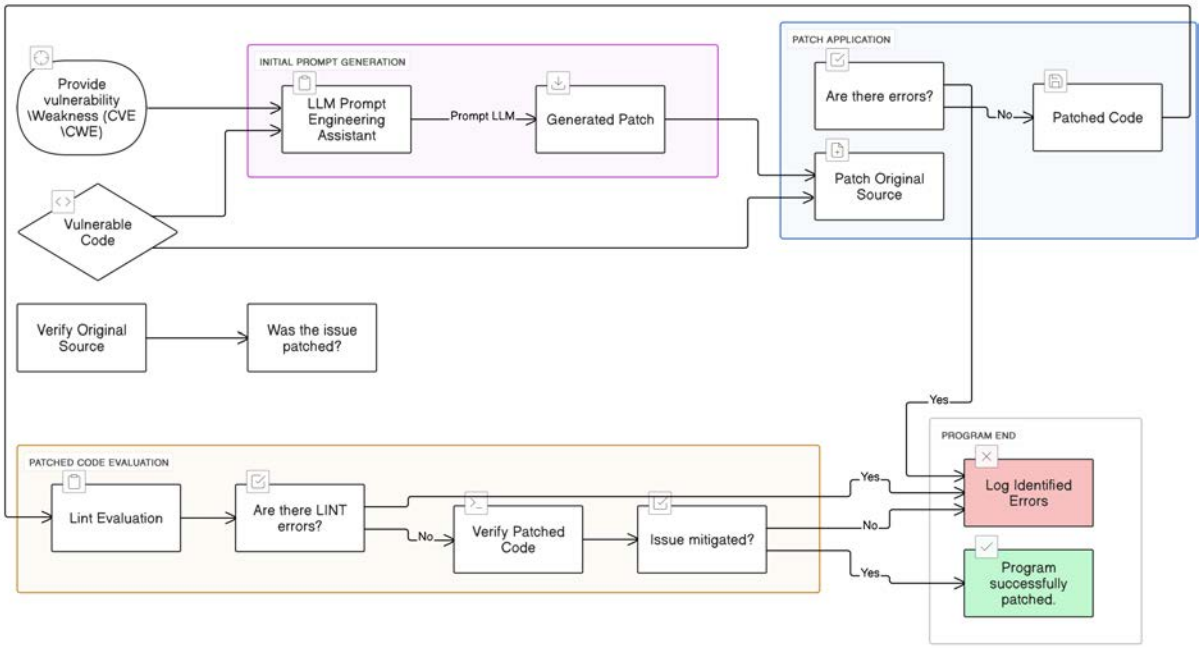

Figure 2: Flowchart of our proposed approach.

Table 1: LLM Patch Generation Prompts

System	Generation
”You are an expert in repairing security vulnerabilities and weaknesses. You are well-versed in computer languages. Code repairs are to be provided in the unified diff format with correct range information for the provided source code. Please provide reasoning or explanations as valid comments for the source code’s language, within the unified diff format patch. Do not tell me you patched the code, ONLY provide a patch. When counting lines for patch hunk range information, blank lines must be included.”	”The following CVE or CWE ’{dataPoint[’CVE’]}’ has been identified in some source code. The original filename is: ’{filename}.’ Only patch code that is affected by the identified CVE or CWE. Please interpret ’\n’ as a new line and ’\t’ as a tab.”

The responses of these LLMs will be captured and cleaned of extraneous information such as ”Here is a patch I made” typically prepended in the LLM response. To evaluate the LLM's exact performance when generating patches, patches *will not* be further modified with the exception of patch hunk range recounting and extraneous text removal. Both the original and ”cleaned” versions of the response/patch will be saved for evaluation. The RAG and non-RAG versions of each prompt will be stored separately for evaluation of the generated patches.

5.3 Step 3: Patch application and Evaluation

This step will involve the application and initial evaluation of the patch generation. Information such as the original source code, source language, LLM used, and the generated patch will be saved. Each generated patch will be applied to the original source code using the Git apply utility, replicating a typical patch application process in a real-world environment. Any errors detected during the patch application will be also be recorded. These results will be compared both between models and between the non-RAG and RAG approaches.

In this step, we will also evaluate the patch generation and patch application success and failure rates in a similar metrics as Gokkaya *et al.*(Gokkaya, 2025).

- ***Patch generation rate.*** This metric will evaluate the amount of valid patches per language, per model, that were successfully generated by the LLM, including usable patches. This metric is only to evaluate *if* the LLM could provide a patch that could be processed by Git apply, whether the patch was applied or not. With x indicating the amount of patches successfully generated and y denoting the total prompts, this metric is defined as x/y x 100%.

- ***Patch validity rate.*** This metric will evaluate, of the successfully generated patches, per language and model, which patches were successfully applied to the source code. This metric will not evaluate the usability of the patched source code. With x indicating the amount of patches successfully applied and y denoting the amount of patches that were generate, this metric is defined as x/y x 100%.

Crucial to this evaluation step is a comparison with other current approaches in terms of their published patch generation and validity rates. Notably, VulnFix(Zhang et al., 2022), ExtractFix(Gao et al., 2021), and Senx(Huang et al., 2019) will be used for comparison.

5.4 Step 4: Patched code testing and Evaluation

This is the final and most important evaluation step. Here, the quality of the usable patches will be evaluated using three methods to verify their functionality and performance. First, the original and patched versions will be statically analyzed using a linter utility. Linter utilities can analyze the semantics of the code to identify obvious errors, formatting inconsistencies and even potential bugs such as vulnerabilities. The amount of deficiencies identified by the linter will be compared with that of the original and ground-truth ”actual” patch to identify if any new errors were added. Specifically, RUFF (RUFF, [n. d.]) will be used to evaluate the pre- and post-generated Python code, ESLint (ESLINT, [n. d.]) will be used for Javascript, and Checkstyle (CHECKSTYLE, [n. d.]) for Java. Each of these linting utilities were selected due to their extensive use and widely accepted results. ESLint will use the default configuration, while Checkstyle will use the *sun checks* configuration. While initially considered as a metric in this evaluation, CodeBleu was not selected due to the potential difference between the LLM's approach to mitigation and the actual developer's. In addition, the patches will be compared against the CVE/CWE mitigation specified for the weakness and vulnerability to determine if the correct action was taken to mitigate the bug.

5.5 Step 2: Patch Generation

In this step, we will pass the generated prompts to different state-of-the-art LLMs. Each LLM will produce a response as a Unified Format patch. The responses of these LLMs will be captured and cleaned of extraneous information such as ”Here is a patch I made” typically prepended in the LLM response. To evaluate the LLM's exact performance when generating patches, patches *will not* be further modified with the exception of patch hunk range recounting and extraneous text removal. Both the original and ”cleaned” versions of the response/patch will be saved for evaluation. The RAG and non-RAG versions of each prompt

Table 2: LLM Patch Generation Prompts

System	Generation
”You are an expert in repairing security vulnerabilities and weaknesses. You are well-versed in computer languages. Code repairs are to be provided in the unified diff format with correct range information for the provided source code. Please provide reasoning or explanations as valid comments for the source code’s language, within the unified diff format patch. Do not tell me you patched the code, ONLY provide a patch. When counting lines for patch hunk range information, blank lines must be included.”	”The following CVE or CWE ’{dataPoint[’CVE’]}’ has been identified in some source code. The original filename is: ’{filename}.’ Only patch code that is affected by the identified CVE or CWE. Please interpret ’\n’ as a new line and ’\t’ as a tab.”

will be stored separately for evaluation of the generated patches.

5.6 Step 3: Patch application and Evaluation

This step will involve the application and initial evaluation of the patch generation. Information such as the original source code, source language, LLM used, and the generated patch will be saved. Each generated patch will be applied to the original source code using the Git apply utility, replicating a typical patch application process in a real-world environment. Any errors detected during the patch application will be also be recorded. These results will be compared both between models and between the non-RAG and RAG approaches.

In this step, we will also evaluate the patch generation and patch application success and failure rates in a similar metrics as Gokkaya *et al.*(Gokkaya, 2025).

- ***Patch generation rate.*** This metric will evaluate the amount of valid patches per language, per model, that were successfully generated by the LLM, including usable patches. This metric is only to evaluate *if* the LLM could provide a patch that could be processed by Git apply, whether the patch was applied or not. With x indicating the amount of patches successfully generated and y denoting the total prompts, this metric is defined as x/y x 100%.

- ***Patch validity rate.*** This metric will evaluate, of the successfully generated patches, per language and model, which patches were successfully applied to the source code. This metric will not evaluate the usability of the patched source code. With x indicating the amount of patches successfully applied and y denoting the amount of patches that were generate, this metric is defined as x/y x 100%.

Crucial to this evaluation step is a comparison with other current approaches in terms of their published patch generation and validity rates. Notably, VulnFix(Zhang et al., 2022), ExtractFix(Gao et al., 2021), and Senx(Huang et al., 2019) will be used for comparison.

5.7 Step 4: Patched code testing and Evaluation

This is the final and most important evaluation step. Here, the quality of the usable patches will be evaluated using three methods to verify their functionality and performance. First, the original and patched versions will be statically analyzed using a linter utility. Linter utilities can analyze the semantics of the code to identify obvious errors, formatting inconsistencies and even potential bugs such as vulnerabilities. The amount of deficiencies identified by the linter will be compared with that of the original and ground-truth ”actual” patch to identify if any new errors were added. Specifically, RUFF (RUFF, [n. d.]) will be used to evaluate the pre- and post-generated Python code, ESLint (ESLINT, [n. d.]) will be used for Javascript, and Checkstyle (CHECKSTYLE, [n. d.]) for Java. Each of these linting utilities were selected due to their extensive use and widely accepted results. ESLint will use the default configuration, while Checkstyle will use the *sun checks* configuration. While initially considered as a metric in this evaluation, CodeBleu was not selected due to the po-

tential difference between the LLM's approach to mitigation and the actual developer's.

The second metric will involve actual use of the code. As previously stated, this step will only involve patched code that is deemed "usable" during the initial lint verification. First, the patched code will replace the original, vulnerable code in a local copy of its project repository. The project will then be deployed as instructed and executed. Test cases, whether provided by the developer or created in-house, will be used to verify that the newly patched code both functions as intended and without errors. This is a pass/fail metric, and each pass or fail will be recorded. Code that passes this metric will be used in the final evaluation method.

The third and final quality evaluation method will involve tests to verify that the vulnerability or weakness has been patched. The patched code project will again be executed as designed, however test cases will involve attempts to trigger the vulnerability or weakness that is expected to have been patched. The behavior of the patched code will be compared against the behavior of the reported vulnerability or weakness to determine if the code is still vulnerable or not. This will also involve a comparison between the behavior of the generated patch and the ground-truth "actual" patch. If the vulnerability or weakness is still present (or a new one was created) this will be a failure, otherwise this will be a pass. These results will be recorded for documentation.

6 Experimental Results

6.1 Datasets used

As mentioned, two datasets were used for this experiment. The first, CVEFixes (Bhandari et al., 2021), was used to provide vulnerable source code for the experiment while also providing examples of patches for each identified vulnerability. Specifically, we used CVEFixes 1.0.8 and has data up until July 2024. In addition, this dataset includes data from a variety of current GitHub repositories, which is crucial for testing a wide variety of code-bases and vulnerabilities in different languages such as Python, Java, C/C++. For our purposes, a new dataset will be created based on the original CVEFixes dataset to only include the languages to be tested, as well as other important information such as the CVE code, filename, and provided source codes and patches. In addition, CVEList, provided by CVE (cve, [n. d.]) is used to provide descriptions of each vulnerability as well as any associated weaknesses for use in the RAG context. This dataset is constantly updated as new vulnerabilities and weaknesses are identified and documented. The data is provided as a series of JSON-formatted files for each documented CVE under the CVE Record Format. Due to this dataset containing *every* CVE ever documented, any listing that was marked as "rejected" was omitted from use. Overall, 60 source files from independent source repositories have been evaluated from the CVEFixes dataset. As stated each of these files were tested with the state-of-the-art LLMs GPT4o, GPT4-Turbo, o3-pro, and Hermes-3-Llama-3.2-3B. In general, state-of-the-art LLMs have demonstrated extensive coding capabilities and are capable of generating Unified Format patches of different quality.

6.2 Patch Generation without RAG

To answer part of our first research question, *RQ1: Can LLMs perform code repair for software security vulnerability mitigation effectively and efficiently* we assessed one aspect of the LLM effectiveness by comparing the the patch generation results. Indicated by 5, GPT4-Turbo has demonstrated the best capability to generate both valid and usable patches compared to the other models without RAG. It consistently includes the provided filename within the initial two-line heading of the patch while also capable generating a filename when none is provided. In addition, the filenames in the header also carry the correct file extension for the tested language. GPT-4o, o3-Pro and Hermes-3-Llama-3.2-3B would typically fail to provide this header. This aspect was evaluated due to its importance when used in automated CI workflows where utilities such as Git apply are typically used to apply code patches on-the-fly. Secondly, GPT4-Turbo has been shown to generate the most "apply-able" patches of the LLMs tested, with the most successful patches being for the Python language. In fact, of the LLMs evaluated, GPT4-Turbo was the only LLM to generate successful patches for the Java programming language without RAG context information.

6.3 Patch Generation with RAG

A troubling result is that the LLMs appear to perform worse when generating patches with the RAG context information despite retaining the same patch quality as they did without RAG.

Table 3: Example of a GPT4-Turbo Generated Patch for CVE-2021-3822

```
“‘diff
— util.js
+++ util.js
@@ -73,8 +73,8 @@
/
export function tryJsonRepair (jsonString) {
   try {
-    return jsonrepair(jsonString)
-    } catch (err) {
+    return jsonrepair(jsonString) // Ensure jsonString is properly sanitized or validated before repair attempt
+    } catch (err) { // Handle specific error types and ensure jsonString integrity before returning
       // repair was not successful, return original text
   return jsonString
}
“‘
```

Table 4: Patch Rates

LLM	Python Gen. Rate	Python Val. Rate	JavaScript Gen. Rate	JavaScript Val. Rate	Java Gen. Rate	Java Val. Rate
GPT-4o	92.86%	10.71%	90.48%	9.52%	100%	0.00%
GPT-4o w/ RAG1	75.00%	3.57%	57.14%	0.00%	90.91%	0.00%
GPT-4o w/ RAG2	82.14%	3.57%	80.95%	9.52%	90.91%	18.18%
GPT-4o w/ RAG3	85.71%	0.00%	80.95%	4.76%	100%	0.00%
GPT4-Turbo	92.86%	32.14%	90.48%	33.33%	100%	54.55%
GPT4-Turbo w/ RAG1	96.43%	17.86%	80.95%	23.81%	100%	9.09%
GPT4-Turbo w/ RAG2	96.43%	32.14%	85.71%	23.81%	100%	27.27
GPT4-Turbo w/ RAG3	96.43%	21.43%	80.95%	19.05%	100%	18.18%
o3-Pro	14.29%	7.14%	14.29%	9.52%	0.00%	0.00%
o3-Pro w/ RAG1	25.00%	14.29%	38.1%	28.57%	0.00%	0.00%
o3-Pro w/ RAG2	10.71%	3.57%	4.76%	4.76%	0.00%	0.00%
o3-Pro w/ RAG3	17.86%	7.14%	38.1%	19.05%	18.18%	0.00%
Hermes-3-Llama-3.2	14.29%	0.00%	19.05%	0.00%	81.82%	0.00%
Hermes-3-Llama-3.2 w/ RAG1	57.14%	3.57%	52.38%	4.76%	72.73%	0.00%
Hermes-3-Llama-3.2 w/ RAG2	35.71%	0.00%	38.10%	4.76%	63.64%	0.00%
Hermes-3-Llama-3.2 w/ RAG3	42.86%	3.57%	47.62%	14.29%	63.64%	8.98%

Table 5: Average Patch Rates

LLM	Gen Rate	Applies
GPT-4o	94.45%	6.74%
GPT-4o w/ RAG1	74.35%	1.19%
GPT-4o w/ RAG2	84.67%	10.42%
GPT-4o w/ RAG3	88.89%	1.59%
GPT4-Turbo	94.45%	40.01%
GPT4-Turbo w/ RAG1	92.46%	16.92%
GPT4-Turbo w/ RAG2	94.05%	27.74%
GPT4-Turbo w/ RAG3	92.46%	19.55%
o3-Pro	9.53%	5.55%%
o3-Pro w/ RAG1	21.03%	14.29%
o3-Pro w/ RAG2	5.16%	2.78%
o3-Pro w/ RAG3	24.71%	8.73%
Hermes-3-Llama-3.2	38.39%	0.00%
Hermes-3-Llama-3.2 w/ RAG1	60.75%	2.78%
Hermes-3-Llama-3.2 w/ RAG2	45.82%	1.59%
Hermes-3-Llama-3.2 w/ RAG3	51.37%	8.98%
VulnFix(Zhang et al., 2022)	18.52%	100%
ExtractFix(Gao et al., 2021)	17.39%	100%
Senx(Huang et al., 2019)	22.35%	100%

Table 6: Observed Python Lint Errors, where N/A denotes no patches generated

LLM	Prior	After	Actual
GPT-4o	2	7	2
GPT-4o w/ RAG1	0	0	0
GPT-4o w/ RAG2	0	0	0
GPT-4o w/ RAG3	N/A	N/A	N/A
GPT4-Turbo	4	4	5
GPT4-Turbo w/ RAG1	3	9	3
GPT4-Turbo w/ RAG2	3	7	3
GPT4-Turbo w/ RAG3	5	20	5
o3-Pro	2	2	2
o3-Pro w/ RAG1	2	2	2
o3-Pro w/ RAG2	0	0	0
o3-Pro w/ RAG3	3	3	3
Hermes-3-Llama-3.2	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG1	0	0	0
Hermes-3-Llama-3.2 w/ RAG2	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG3	0	0	0

Table 7: Observed JavaScript Lint Errors, where N/A denotes no patches generated

LLM	Prior	After	Actual
GPT-4o	5	5	6
GPT-4o w/ RAG1	N/A	N/A	N/A
GPT-4o w/ RAG2	13	14	13
GPT-4o w/ RAG3	2	2	3
GPT4-Turbo	75	146	146
GPT4-Turbo w/ RAG1	34	103	104
GPT4-Turbo w/ RAG2	35	39	39
GPT4-Turbo w/ RAG3	216	277	288
o3-Pro	61	61	61
o3-Pro w/ RAG1	101	104	102
o3-Pro w/ RAG2	13	14	13
o3-Pro w/ RAG3	53	127	127
Hermes-3-Llama-3.2	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG1	29	29	29
Hermes-3-Llama-3.2 w/ RAG2	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG3	74	74	144

Table 8: Observed Java Lint Error Rates, where N/A denotes no patches generated

LLM	Prior	After	Actual
GPT-4o	N/A	N/A	N/A
GPT-4o w/ RAG1	N/A	N/A	N/A
GPT-4o w/ RAG2	219	225	222
GPT-4o w/ RAG3	N/A	N/A	N/A
GPT4-Turbo	915	929	985
GPT4-Turbo w/ RAG1	31	32	91
GPT4-Turbo w/ RAG2	581	584	641
GPT4-Turbo w/ RAG3	123	126	190
o3-Pro	N/A	N/A	N/A
o3-Pro w/ RAG1	N/A	N/A	N/A
o3-Pro w/ RAG2	N/A	N/A	N/A
o3-Pro w/ RAG3	N/A	N/A	N/A
Hermes-3-Llama-3.2	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG1	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG2	N/A	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG3	31	34	91

Table 9: Vulnerability/Weakness Repair Rates where N/A indicates no patches generated

Vulnerability/Weakness Repair Rates		
LLM	Vulnerability Fixed	Not Fixed
GPT-4o	0	4
GPT-4o w/ RAG1	1	0
GPT-4o w/ RAG2	0	4
GPT-4o w/ RAG3	0	1
GPT4-Turbo	2	20
GPT4-Turbo w/ RAG1	5	6
GPT4-Turbo w/ RAG2	1	16
GPT4-Turbo w/ RAG3	2	13
o3-Pro	0	T4
o3-Pro w/ RAG1	8	2
o3-Pro w/ RAG2	0	2
o3-Pro w/ RAG3	1	4
Hermes-3-Llama-3.2	N/A	N/A
Hermes-3-Llama-3.2 w/ RAG1	1	1
Hermes-3-Llama-3.2 w/ RAG2	0	1
Hermes-3-Llama-3.2 w/ RAG3	4	2

On average, the LLMs generated 10% *less* valid patches, with even fewer applicable patches. GPT4-Turbo, which performed well without RAG context information, generated almost 30% less *apply-able* patches, despite generating 4% more *valid patches*, meaning that while it can generate more patches with RAG in general, less are actually usable. However, of note GPT-4o actually improved its Java patch generation with this context information and was able to successfully generate Java apply-able patch files when it previously could not. O3-Pro, however, was shown to greatly improve its patch generation by nearly doubling its valid and apply-able patch rates.

6.4 Overall Patch Generation Evaluation

Overall, as mentioned previously, patches do appear to contain correct-appearing syntax including properly formatted hunk headers with range and line information. However, the generated range information is typically incorrect requiring frequent recounting. It was also observed that blank lines are typically a major source of issue for LLM-based patch generation. Blanks lines are frequently not counted when calculating range information, while blank lines are occasionally added when they were not present in the original source code. Similarly, the LLMs do not reliably generate the required two-line file header. Even when provided the filename as part of the prompt and context, the LLMs typically omit this required information. While the range information can be recounted, and recalculated automatically with Git apply and the missing two-line header can be automatically prepended to the patch, these pitfalls result in several patches that are not apply-able to the target source. Of note, when passing the generated patch to another LLM for corrections, the new patch is often degraded in quality. Line count inaccuracies are worsened, hunk headers are sometimes stripped, and the code quality itself is degraded with poor syntax. In addition, two patched files, ”test_http_parser.py” generated by GPT4-Turbo w/ RAG1 and ”wrapEmojiText-PAHdiQbe.js” generated by Hermes-3-Llama-3.2 w/ RAG2 could not be evaluated due to file writing errors.

GPT-4o and o3-Pro are tied in regards to the amount of patches that were able to be applied, being second to GPT4-Turbo. Of note, overall o3-Pro generated more ”invalid” and completely unusable patches on average when compared to GPT-

4o and was not able to generate *any* Java patches. In addition, in one case it generated the response "Patch is not produced – unable to determine affected code for CVE." when requesting a patch for CVE-2022-45907. On the other hand Hermes-3-Llama-3.2-3B has been evaluated to consistently fail to generate any usable patches. It consistently fails to add the range information to patch hunk header when using identical prompts as the former three models, often times providing a patch with invalid syntax (such as missing hunk headers) or generates a complete source file with just the Diff '+' and '-' lines added. In fact, it has been observed to provide an incomplete hunk header every time it generates a patch. However, of note, it does include a method or class reference in the incomplete header, which other models typically do not. By further refining the patches, they may be significantly more usefull.

Nonetheless, these results demonstrate that LLMs are certainly effective at generating code patches *as a whole*. In fact, GPT4-Turbo has shown to be the most effective LLM for generating usable patches for identified vulnerabilities and weaknesses in all three evaluated languages.

6.5 Patch Generation Performance

Overall, patch generation times were checked independently per model. Overall, GPT4-Turbo was able to generate a patch in less than a minute, with o3-Pro requiring 1-2 minutes per prompt. However, Hermes-3-Llama-3.2-3B took the longest to generate patches using an Intel 11700 at manufacturer specs. The fastest was just under 2 minutes (1:51) to patch CVE-2018-7408 in "extract.js" while taking just over 49 minutes (49:22) to patch "llollms_binding_infos.py" with CVE-2024-4078. The inclusion of each RAG method had little difference on the generation times. Using a CPU to handle local LLMs for patch generation is not recommended.

6.6 Patch Quality

As part of the *effectiveness* evaluation, we compared the LINT rates of the original source code, rates of the code after applying the generated patch, and that of the actual patch as shown by 6 - 8. GPT-4o, GPT4-Turbo, o3-Pro and Hermes-3-Llama-3.2 added a negligible amount of LINT errors when compared to the original source code and the ground-truth source code. Of note, while GPT4-Turbo introduced more Java and JavaScript LINT errors, this is considered negligible when compared to code from the published repair. However, it did introduce 3-4 times as much Python errors when compared to the published patch and the original source code. Nonetheless, it can be considered that LLMs can generate code of acceptable quality.

6.7 Vulnerability Mitigation

Our final *effectiveness* evaluation was to compare the generated patches against the actual patches to identify if the LLMs can *effectively* patch identified vulnerabilities. Unfortunately, these results were not as promising compared to the generation phase. Typically, the generated patch code is in no way similar to the actual real-world patch that was written for the specified vulnerability presenting, and unexpectedly ineffective at mitigating the actual issue. In fact, without RAG, the patches typically addresses areas of the code that are not involved with the specified CVE or CWE. The LLMs appear to use literal interpretations of published workarounds when generating patches for the specified vulnerability or weakness, or even misinterpreting them as a whole even with RAG context. In fact, the LLMs have occasionally hallucinated the actual CVE/CWE definition as we observed with the LLM generated code comments. In one case GPT4-Turbo added only code comments that "explain" how to patching CVE-2018-7408. However, these comments only mention package verification, which the CVE did mention that the vulnerable code was in a pre-release package that was accidentally released, this was not the actual issue. In addition, LLMs would add unnecessary code, such as adding a log entry when the patch needed to address communication between child processes. Another example was that GPT4-Turbo *explicitly* stated to use a hypothetical function for a hotfix, rather than performing a verification of a client.

O3-Pro, however, did exhibit an interesting phenomena with the RAG1 approach. When provided the CVE or CWE explanation with an example of the patch, it would actually ignore "extraneous" corrections such as adding spaces, comment typographical corrections, etc. and only address the relevant areas of the code. In addition, despite the CVE/CWE entries lacking extensive details on how to actually fix the issue, o3-Pro actually generated source comments that sufficiently explained

how the patch fixed the vulnerability or weakness preceding the modified code. These demonstrate o3-Pro is performing some ”thinking” with the provided context information, which is crucial to identifying the correct method to fix the identified weakness or vulnerability.

7 Discussion

As part of the evaluation of our approach, we compared our results with that of other well-known novel approaches. VulnFix(Zhang et al., 2022) was selected as the first comparison and uses fuzzing approach similar to approaches such as Daikon (Ernst et al., 2007). It is primarily designed to infer a patch based on ”good” and ”bad” values in relation to the success or failure of vulnerable code to generate a patch to fix the ”bad” values. ExtractFix(Gao et al., 2021) was also selected for comparison due to its novel ”constraint” based approach that identifies valid and invalid inputs, locations of the ”error-triggering” code to improve its accuracy and reduce overfitment. Lastly, Senx(Huang et al., 2019) was selected due to its human-defined ”safety property” approach to improve its patch generation accuracy.

Compared to these methods, it has been observed that GPT-4o and GPT4-Turbo with and without RAG context are able to generate significantly more patches for the provided source code. However, none of the tested LLMs were able to match the other methods' application/restoration rates of 100%. In addition, without RAG, we have determined that our methods are almost as ineffective as previous solutions.

8 Threats to Validity

At this time, it is not possible to determine whether the models used have been exposed to the test cases used due to the large sizes of the training datasets typically used for training. However, in an attempt to reduce the likelihood of repeated code, we will curate the datasets based on similar appearing code patches or identical commits across the datasets used. In addition, due to the size of the datasets, and their sources, it is not possible to thoroughly verify that code patches, or code, are not completely AI-generated. Curation has been done to remove the obvious instances, however we can't vet all of them.

In addition due to the small sample size of code, further reduced by the amount of successful patches, there is also the possibility that LLMs may be more or less successful overall with a larger sample size. However, when compared against other approaches such as VulnFix(Zhang et al., 2022), our sample size is comparable.

Lastly, our approach focused on Java, JavaScript, and Python while other approaches focused on C/C++. This may pose a threat in terms of comparison, for instance a recently published study by *Lucas et al.*(Twist et al., 2025) shows that LLMs may be trained more heavily on Python while showing a preference in that language. However, that same paper indicates that this is primarily the case for *newly generated* programs, and that LLMs are capable of working with other languages such as JavaScript and C/C++.

Lastly, unintentional errors may have been introduced through the handling of source code and other data used in this procedure. As noted previously, for instance, some encoding errors did prevent two generated patched code files from being evaluated.

9 Conclusion and Future Work

Automated Vulnerability Repair is a highly desirable system for modern software development pipelines when maintaining and securing large code-bases. While several methods have already been proposed and implemented, they typically suffer from over-fitment and lack of scalability due to their training and design methods. Unfortunately, despite leveraging state-of-the-art LLMs, we have determined that LLMs without specific RAG context perform poorly at mitigating potential vulnerabilities or weaknesses. However, in the future, by leveraging fine-tuning and pre-training more analytical-based models, such as o3-Pro, we may be able to leverage this capability to generate superior prompts for a code-based model to improve mitigation rates and language support.

Data and Source Code Availability

The source code for the the proposal will be stored on a publicly available Github repository. This repository may be accessed at https://github.com/qas-lab/reu-michael-conner. While the source code to re-generate the CVEFixes databse is not included, it may be accessed using the official download link: https://github.com/secureIT-project/CVEfixes

provided previously under the ”Code” folder as ”create_CVEfixes_from_dump.sh” sh command file for Linux. Included in our source code is the database parser for CVEFixes, the ”curated” database creator, and the code patch generation source code. CVEList may be accessed and downloaded via the official cve.org link, https://www.cve.org/Downloads.

Acknowledgements

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant No. 2349452. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*. ACM, 10. https://doi.org/10.1145/3475960.3475985

Will Bridewell, Narges Bani Asadi, Pat Langley, and Ljupčo Todorovski. 2005. Reducing overfitting in process model induction. In *Proceedings of the 22nd International Conference on Machine Learning* (Bonn, Germany) *(ICML '05)*. Association for Computing Machinery, New York, NY, USA, 81–88. https://doi.org/10.1145/1102351.1102362

CHECKSTYLE [n. d.]. Checkstyle. https://checkstyle.sourceforge.io/. Accessed: 2025-06-01.

Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2019.2940179

cve [n. d.]. Common Vulnerabilities and Exposures: CVE. https://cve.mitre.org. Accessed: 2024-09-03.

CVE-Details [n. d.]. Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2024. https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/. Accessed: 2025-06-01.

cwe [n. d.]. Common Weakness Enumeration: CWE. https://cwe.mitre.org. Accessed: 2024-09-01.

diff [n. d.]. Detailed Description of Unified Format. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html. Accessed: 2025-06-01.

Qingao Dong, Yuanzhang Lin, Hailong Sun, and Xiang Gao. 2025. Enhancing Automated Vulnerability Repair Through Dependency Embedding and Pattern Store. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 193–204. https://doi.org/10.1109/SANER64311.2025.00026

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1–3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

ESLINT [n. d.]. Find and fix problems in your JavaScript code . https://eslint.org/. Accessed: 2025-06-01.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098

Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (Feb. 2021), 27 pages. https://doi.org/10.1145/3418461

GIT [n. d.]. Git –distributed-is-the-new-centralized. https://git-scm.com/. Accessed: 2025-06-01.

Betul Gokkaya. 2025. Leveraging Large Language Models for Security Patch Detection and CWE Classification. In *2025 7th International Congress on Human-Computer Interaction, Optimization and Robotic Applications (ICHORA)*. 1–10. https://doi.org/10.1109/ICHORA65333.2025.11017081

Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. 2025. SoK: Automated Vulnerability Repair: Methods, Tools, and Assessments. arXiv:2506.11697 [cs.SE] https://arxiv.org/abs/2506.11697

Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. 539–554. https://doi.org/10.1109/SP.2019.00071

Larry Huynh, Yinghao Zhang, Djimon Jayasundera, Woojin Jeon, Hyoungshick Kim, Tingting Bi, and Jin B. Hong. 2025. Detecting Code Vulnerabilities using LLMs. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 401–414. https://doi.org/10.1109/DSN64029.2025.00047

ITRC [n. d.]. ITRC 2024 Annual Data Breach Report. https://www.idtheftcenter.org/publication/2024-data-breach-report/. Accessed: 2025-06-01.

nvd [n. d.]. National Vulnerability Database (NVD). https://nvd.nist.gov/vuln. Accessed: 2024-09-03.

OpenAI-GPT41 [n. d.]. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/. Accessed: 2025-06-01.

Francisco Ribeiro. 2023. Large Language Models for Automated Program Repair. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Cascais, Portugal) *(SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 7–9. https://doi.org/10.1145/3618305.3623587

RUFF [n. d.]. Ruff - An extremely fast Python linter and code formatter, written in Rust. https://github.com/astral-sh/ruff. Accessed: 2025-06-01.

SVN [n. d.]. Apache® Subversion®. https://subversion.apache.org/. Accessed: 2025-06-01.

Lukas Twist, Jie M. Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. 2025. LLMs Love Python: A Study of LLMs' Bias for Programming Languages and Libraries. arXiv:2503.17181 [cs.SE] https://arxiv.org/abs/2503.17181

Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. 2024. Maltracker: A Fine-Grained NPM Malware Tracker Copiloted by LLM-Enhanced Dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1759–1771. https://doi.org/10.1145/3650212.3680397

Shafiq Joty Steven C.H. Hoi Yue Wang, Weishi Wang. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP*.

Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. 2024. VulAdvisor: Natural Language Suggestion Generation for Software Vulnerability Repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1932–1944. https://doi.org/10.1145/3691620.3695555

Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 691–702. https://doi.org/10.1145/3533767.3534387

Large Language Models for Software Verification and Validation

Zoe Fingleton[1,2], Himon Thakur[2], Nazanin Siavash[2], Armin Moin[2]
[1]Smith College, MA, USA
[2]Department of Computer Science, University of Colorado Colorado Springs (UCCS), CO, USA
{zfingleton}@smith.edu, {hthakur,nsiavash,amoin}@uccs.edu

Abstract

In the software development cycle, Verification (code inspection) and Validation (testing) are essential processes responsible for identifying defects and inefficiencies in code. Although these manual procedures are relatively effective, they can be time-consuming and laborious to perform. In this paper, we propose a novel approach that utilizes Large Language Models (LLMs) for code inspection and test case generation, helping to alleviate the burden of traditional methods. Additionally, to address the *hallucination* problem—in which LLMs produce incorrect outputs—we will implement a Retrieval-Augmented Generation (RAG) pipeline to integrate supplementary knowledge sources.

1 Introduction

Verification and Validation (V&V) is a crucial stage of the Software Development Lifecycle (SDLC), also known as the Software Process. In this procedure, software testing, inspection/review, and formal verification are popular examples of V&V activities.

Among these, inspection (also known as review) of software artifacts is the most cost-effective one. On average, it can find around 60% of software defects or inefficiencies in the design and implementation, which is impressive (Sommerville, 2016). However, manual inspection is laborious and time-consuming.

Another well-established V&V activity is Testing. Similar to inspection, and unlike formal verification, testing can only show the presence of errors, not their absence. Nonetheless, software testing is essential for ensuring the proper functionality of software systems. While testing is generally less expensive than formal verification, it is also less cost-effective compared to inspection.

There are various forms of software testing, but this work focuses explicitly on development testing. When properly performed, there should be at least one test case for each of the functional and non-functional requirements of a software system. This standard for developing test cases is guided by the Software Requirements Specification (SRS) document, which details both functional and non-functional requirements. These requirements are also known as the quality attributes. In this paper, we are interested in the domain of development testing.

In this work, we are concerned with automated code inspection and test case generation using Artificial Intelligence (AI). Specifically, we are interested in the subdiscipline of Machine Learning, and in particular, state-of-the-art pre-trained machine learning models, called Large Language Models (LLMs). The contribution of this paper is to propose a novel approach based on the integration of a RAG pipeline to enhance the performance of LLMs and prevent the hallucination problem. For this task, we will utilize several different LLMs that can be deployed for V&V activities, as explained above.

To validate and evaluate the proposed approach for automated code inspection, we compare the LLM error detection rate with the average reference point for human code inspectors, which is 60%. Moreover, to validate and evaluate the proposed approach for automated test case generation, we use the benchmark dataset provided by Wang et al. (Wang et al., 2025). In particular, we answer the following Research Questions (RQs):

- RQ1. Can LLMs perform code inspection effectively and efficiently?
- RQ2. Can LLMs perform test case generation effectively and efficiently?

The rest of this paper is structured as follows: Section 2 provides some background information on code inspection, development testing, LLMs, and RAGs. Afterward, Section 3 reviews the literature of automated V&V. Further, we propose

our novel approach in Section 5. The experimental study in Section 6 validates and evaluates the proposed approach. Finally, we conclude and suggest future work in Section 7.

2 Background

2.1 Verification and Validation

When developing software, testing is a necessary step in determining whether the product will reach its intended purpose and meet requirements. There are several approaches to benchmarking program functionality, the most central of which are V&V testing.

In verification testing, we check whether the product meets its functional and non-functional requirements (Sommerville, 2016). This involves determining whether the present code meets all design standards at its current developmental stage. We perform regular verification throughout the design life cycle.

In tandem, validation testing is the process of searching for defects in the code that impede the program's ability to meet user needs (Sommerville, 2016). This entails testing each system feature to confirm its intended functionality, regardless of any stressors. Generally, this form of software testing is performed after the program is completed as a final check before user access.

2.2 Large Language Models

LLMs are a significant advancement in AI, trained on vast corpora containing billions of words. These models are upscaled iterations of language models (LMs), which learn intricate patterns of natural language from a dataset, enabling them to predict and generate contextually relevant information (Zhao et al., 2023).

The driving force behind many LLMs is the self-attention module found in the Transformer architecture (Vaswani et al., 2017), which enables understanding of context. Through a process known as "pre-training", LLMs learn a general knowledge understanding from a dataset. Often, this is accomplished using the aforementioned self-attention module.

Furthermore, beyond this ability to learn a general language understanding from initial training, LLMs can be "fine-tuned" for a specific task. Fine-tuning redirects the model to focus on a specific field for enhanced domain-specific performance (Naveed et al., 2025).

Fine-tuning is a particularly relevant endeavor in the current state-of-the-art, which has gained popularity following the introduction of the Bidirectional Encoder Representations from Transformers (BERT) model in 2019 (Church et al., 2021). This model, developed by Devlin et al., changed the existing LLM landscape through its integration of pre-training and fine-tuning (Devlin et al., 2019).

Subsequently, fine-tuning techniques have been widely adapted for industrial applications (Parthasarathy et al., 2024) and developed for scaling (Zhang et al., 2024).

2.3 Large Language Model Hallucination

As the performance of LLMs in Natural Language Processing (NLP) tasks has rapidly progressed, the generation of non-factual responses has become a pressing concern. Known as hallucination, LLMs have a tendency to produce content that is either unverifiable or contradictory to their knowledge base (Huang et al., 2025). This often occurs when LLMs lack sufficient information to answer a user's query.

We can generally group hallucinations into two categories: factuality hallucinations and faithfulness hallucinations. Factuality hallucination occurs when an LLM generates a result that is factually incorrect within the broader scheme of known knowledge, whereas a faithfulness hallucination goes directly against an LLM's learned knowledge or user instructions (Huang et al., 2025).

Both forms of LLM hallucination have indisputable implications in industry applications, especially in the software domain, where valuable time and resources are wasted by erroneous model output (Li et al., 2025b). As a result, a variety of solutions have been explored for preventing hallucination, including Retrieval Augmented Generation (RAG), prompt engineering, and reinforcement learning (Liu et al., 2024).

2.4 Retrieval Augmented Generation

Introducing a context-specific knowledge source to the LLM's accessible data is one approach to mitigating the hallucination issue. The presence of this supplementary information reduces the likelihood that the LLM will lack context (Béchard and Ayala, 2024). To access knowledge base data, documents in the RAG knowledge base are first encoded into embeddings. The general methodology for performing these embeddings involves using a sentence transformer model or an encoder-based

architecture to find sentence embeddings through contrastive learning (Huang et al., 2025).

With these documents in vector storage, the pipeline can perform a similarity search for a specific query and return relevant context. Using this methodology, an LLM can receive additional context-specific knowledge pertinent to a specific prompt.

Existing results, particularly in the domain of NLP tasks, show that RAG-enhanced models can outperform the state-of-the-art in several language generation tasks (Lewis et al., 2021; Lyu et al., 2025). Subsequently, RAG pipelines have been integrated into domain-specific contexts, such as code generation and summarization (Parvez et al., 2021), mathematics (Levonian et al., 2023), and medicine (Yang et al., 2021), with similarly successful results (Zhao et al., 2024a).

3 Related Work

In recent years, as the capabilities of LLMs have increased, there has been a growing overlap between model strengths and software development applications.

Notably, we observe a consistent improvement in model performance across several relevant NLP tasks. Particularly, state-of-the-art models have achieved high accuracy in areas such as abstraction and reasoning (ARC)—which concerns a model's ability to approach a novel problem using general intelligence (Chollet, 2019)—and massive multitask language understanding (MMLU)—problem solving related to a wide range of subject matter from STEM fields to the humanities (Hendrycks et al., 2021). Several models have achieved over 90% accuracy in at least one of the two challenges, with the 2023 releases of GPT-4 and PaLM 2 surpassing 90% in both (Hagos et al., 2024).

The demonstrated potential of LLMs in natural language understanding has led to research into whether these models possess the ability to comprehend and generate code. In this domain, state-of-the-art proprietary LLMs have shown an ability to understand code context and generate related suggestions (Hagos et al., 2024; Zhuo et al., 2025), a promising indicator for software applications.

Beyond general NLP task-solving aptitude and general code processing skill, LLMs have shown strong potential in software development tasks (Wang et al., 2024; Zheng et al., 2024b,a). In particular, LLMs have applications in code inspection and test case generation, which will be outlined in the discussion of related works on code inspection and test generation. Additionally, there are several "best practices" within the emerging field of LLM-developer interaction that overlap with software life cycle goals. These methods will be discussed in the context of overlaps with popular LLM methodology. Finally, there exist many novel applications of LLMs for software tasks. A selection of these developments is included in this segment.

Code inspection is a crucial activity within the V&V process; however, manual processing is time-consuming and laborious. As a result, the possibility of integrating LLMs into this workflow is of high interest. Recently, state-of-the-art models have performed strongly for automated code inspection (Guo et al., 2024; Hao et al., 2023), especially when fine-tuned for software applications (Pornprasit and Tantithamthavorn, 2024). Furthermore, throughout testing, LLMs have consistently demonstrated their ability to comprehend code inspection comments and make subsequent revisions (Lin et al., 2025). However, they struggle with an inability to express gaps in their understanding (Zhao et al., 2024b). Obfuscated code in particular provides a challenge for LLMs in both general performance and ability to make meaningful analysis (Fang et al., 2024).

Beyond applications for code inspection, LLMs show potential for adaptability in other areas of software development. One such area is computational test case generation, a well-established field (Miller and Spooner, 1976) that has recently begun to incorporate LLMs to facilitate complete automation. Recent research into automated unit test generation and evaluation suggests that LLMs have surpassed the current state-of-the-art in some forms of test case generation, such as JavaScript test generation programs (Schäfer et al., 2024). Furthermore, LLMs demonstrate potential for automating test case generation from bug reports (Kang et al., 2023).

However, LLM performance remains inconsistent across datasets and code contexts, with a substantial influence from the LLMs' training set (Siddiq et al., 2024). In some cases, LLMs have a similar performance to current state-of-the-art benchmarks, while in others, they are significantly outperformed. As one solution, empirical research into the performance of specific models, namely the ChatGPT suite, has shown potential for im-

provement upon out-of-the-box model generation ability through the implementation of iterative test refinement (Yuan et al., 2024).

Despite the relative specialization of using LLMs for software, standard LLM practices remain applicable. Within the broader field of LLM research, one popular approach for improving model performance is "fine-tuning," a process in which a model is reweighted for specialization on a small corpus of data. In current research, LLMs have rarely been fine-tuned for specific software applications, with a recent study reporting only 25% of papers in a small sample leveraged this tactic (Boukhlif et al., 2024).

Another prevalent technique is chain-of-thought reasoning, in which models are prompted to explain their rationale step-by-step. General research into chain-of-thought (CoT) prompting for code generation has been conducted, but recent state-of-the-art results indicate that the current accuracy rate for this form of training is not viable for practical applications (Li et al., 2025a). An alternative approach is the application of "structured chain-of-thought prompting."

In the process of "structured chain-of-thought prompting" (SCoT), LLMs follow typical step-by-step CoT procedures within the context of sequential, branching, and looping structures. This guides the LLMs to reason in terms of concise code, rather than verbose text. This approach has been successful, with greater performance over CoT prompting in pass@1 benchmarks (Li et al., 2025a).

Alternative methods for integrating LLMs into software development environments include indirect model interaction, such as GitHub bots that communicate between LLMs and specific repository source code (Martins et al., 2024). These bots leverage repository-specific knowledge to achieve enhanced contextual LLM performance.

These works indicate potential for application of LLMs in the context of V&V activities for software development.

4 Problem Statement

While there have been successful applications of LLMs in software development, there is still considerable potential for improvement in their integration into the V&V process. This is particularly true given the absence of RAG techniques to help mitigate model *hallucinations* in current research. Incorrect model outputs can significantly disrupt this critical stage of the development process, making it pertinent to work towards a solution. As a result, we propose a novel approach to address this issue.

5 Proposed Approach

While there have been successful applications of LLMs in software development, there is still considerable potential for improvement in their integration into the V&V process. This is particularly true given the absence of techniques to help mitigate model *hallucinations* in current research. Incorrect model outputs can significantly disrupt this critical stage of the development process, making it pertinent to work towards a solution. As a result, we propose a novel approach to address this issue in the areas of inspection and test case generation.

Throughout the generation process, our models will be provided with a knowledge base of relevant information, forming a RAG pipeline. This pipeline will target the hallucination issue by providing external knowledge to help mitigate erroneous outputs. A component of this work will focus on compiling related knowledge sources to determine an effective supplement.

5.1 Automated Code Inspection

In Figure 1, we outline our RAG-enhanced model pipeline. This model consists of four distinct components: (1) Pipeline Setup, (2) RAG/ Knowledge Base, (3) Main Loop, and (4) Evaluation.

We begin by initializing our model to work with a user-specified LLM setup. Several arguments are allowed for this configuration, including the model, temperature, max number of tokens, and whether to enable a RAG knowledge base.

Following the processing of user arguments, we instantiate a pre-trained sentence transformer embedding model that will later enable us to encode query data into embeddings. This marks the beginning of the Pipeline Setup component of the model pipeline. In our base configuration, we use all-MiniLM-L6-v2 (noa, 2024) as our embedding model. Once our embedding model is configured, we implement in-memory vector storage. This will store our embeddings for easy access. With the framework for our model now established, we can load data and our knowledge documents.

In parallel, we prepare our knowledge base corpus by loading JSON/JSONL data from a GitHub repository. As a foundational knowledge base, this

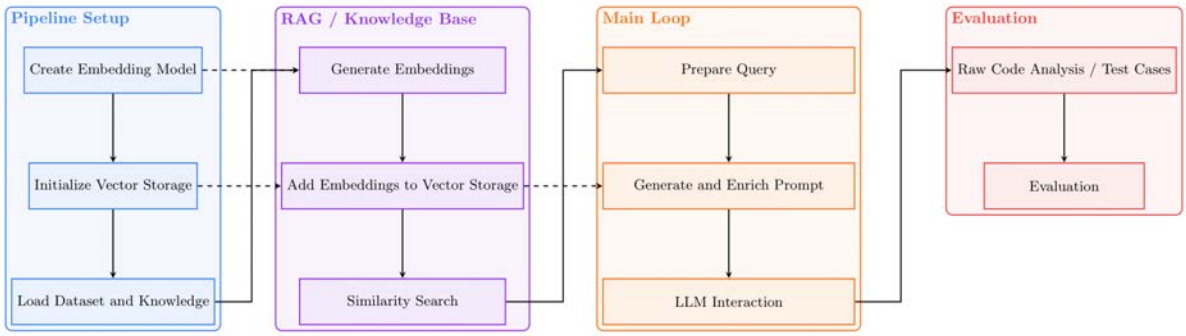

Figure 1: RAG Pipeline

pipeline will be using the Mostly Basic Python Programming (MBPP) dataset developed by Austin et al. (Austin et al., 2021; noa, 2022). This dataset contains 1000 crowd-sourced Python code snippets, useful as examples of well-written and error-free "golden standards."

Our model parses this data line by line for a task ID, content, and metadata; each task and accompanying data are saved as a separate "document."

Following data loading and model activation, we move to the second step: knowledge base configuration. Here, if the RAG flag in the command-line arguments is enabled, we prepare our RAG pipeline. Using the embedding model and loaded documents from our pipeline setup, we generate embeddings for these documents and load the complete embedded corpus into the vector storage.

Following the RAG setup, we enter the main processing loop. The most focal component of this step is the query object. To create an instance of the query class, our model iterates through each data point in the loaded dataset and prepares the data as a query object. To do this, we process the data and save class variables for each relevant piece of metadata. Next, we run a function that randomly selects one of the eight code snippets. This function returns the ground truth of the snippet's validity status (bug or no bug) and the selected data point.

From here, the selected data point is encoded into an embedding, which is input into the vector store for a similarity search. To find relevant documents, we utilize the Python library NumPy to perform a cosine similarity operation and return the top k results.

Using these sourced documents and the previously formatted query, the model generates a prompt with the selected output and enriches it with knowledge base context. From here, the completed prompt (see Figure 2) is fed into an LLM, which then returns a code analysis of the provided snippet.

```
prompt = f"""
You are an expert software engineer performing code inspection.
Use the following knowledge base to help identify potential issues
or bugs in the given code.

Knowledge Base:
{context_text}

Code to inspect:
{query.text}

Please provide a one line code review identifying the type of defect in the code.
If there are no visible bugs, please state "Code is visibly bug-free"

Review:
"""
```

Figure 2: Prompt for code inspection

Once code analysis is produced for each data point, the data are saved and funneled into the evaluation process. With this data, we evaluate our model's code analysis results by contrasting the generated bug type with the labeled bug type collected during the selection process. If the LLM correctly identified the root of the bug (if present), its response is labeled as a "match." Alternatively, if the LLM failed to identify a bug within the code snippet or misclassified the bug type, its response is labeled as a "mismatch." We run this evaluation through a new LLM conversation, with OpenAI's gpt-3.5-turbo as the default model.

To assess the effectiveness of our model's automated code inspection, we first compare the error detection rate of the RAG-enhanced and standard LLMs with the average performance benchmark for human code inspectors, which is previously established at 60% in this paper's introduction (Sommerville, 2016). Then, we compare performance with and without the presence of a RAG knowledge base.

5.2 Automated Test Case Generation

For our automated test case generation pipeline, we continue to utilize the framework depicted in Figure 1. As with our pipeline for automated test code inspection, this model consists of four sections: (1) Pipeline Setup, (2) RAG/ Knowledge Base, (3) Main Loop, and (4) Evaluation.

For pipeline and knowledge base setup, we prepare our model using the same methodology as automated code inspection, creating and configuring our embedding model and vector storage.

In our main processing loop, we use the same general framework as automated code inspection, with some systemic differences. To prepare our queries after processing each data point as a query class object, we save only the program code, function name, and function description. From here, we embed the query and use the same similarity search functionality previously described.

```
prompt = f"""
Please write a test method for the function '{func_name}'
given the following program under test,
the function description, and the knowledge base {context_text}.
Your answer should only contain one test input.

Program under test:
----
{program}
----

Function description for '{func_name}':
----
{description}
----

Your test method should begin with:
def test_{func_name}():
solution=Solution()

You should only generate the test case, without any additional explanation.
"""
```

Figure 3: Prompt for test generation

Following embedding, the pipeline provides the specified LLM with the program under test, the function name and description, and the knowledge base context (see Figure 3). The completed prompt is fed into an LLM, which then returns n unique test cases.

To ensure that each test case is independent of previous generations, we first run the query and document context through a single test generation function. Following the completion of this cycle, the result is fed into a multi-round generation function that takes all previous generations as input.

Once raw test cases are produced for each data point, the data are saved and funneled into evaluation. We pre-process our data by formatting the results into a structured form. With this data, we evaluate our model's test case generation results in terms of overall, line, path, and branch coverage. To achieve this, we utilize TestEval's source code, which cross-references generated test cases with a complementary metadata category, "branches," that details all the task's branches. Coverage is determined by the proportion of lines/branches covered by at least one test case using pytest-cov.

6 Experimental Results

6.1 Experimental Dataset

For our automated code inspection experimental data, we use the Bug In the Code Stack dataset provided by Lee et al (Lee et al., 2024). This dataset comprises 4,010 programming tasks. For each task in this dataset, there are eight code snippets: a bug-free code sample and seven "buggy" versions (e.g., the correct code missing a colon or misusing a keyword as a variable name). The dataset also contains metadata specifying the line at which buggy code occurs.

For our automated test case generation dataset, we locally use the benchmark dataset provided by Wang et al (Wang et al., 2025). This repository's data comprises coding assessments from LeetCode, an online programming platform, and contains code for computing line and branch coverage.

As the central knowledge base, both pipelines will be using the Mostly Basic Python Programming (MBPP) dataset seen in the code inspection model architecture. (Austin et al., 2021; noa, 2022). This dataset is relevant to automated inspection, as it contains 1000 examples of well-written code, and is relevant to automated test case generation, as it contains corresponding sets of Python problems, solutions, and test cases.

6.2 Experimental Setup

In this experimental study, we evaluate three commercial LLMs from the OpenAI GPT line. All experiments are run with a temperature of 0 and an output limit of 256 tokens. We chose 0 as the temperature value to minimize variability across runs, and 256 as the output limit to keep LLM response relatively concise.

6.3 Automated Test Case Generation

For automated test case generation, we tested the coverage of LLM-provided test cases. To evaluate automated test case generation ability, we directly implement a RAG pipeline alongside TestEval's source code. Using command line arguments as a

Table 1: Automated test generation results on the TestEval dataset (210 tasks) (noa, 2025a)

Model	RAG	Compilation success		Test case coverage		Line $cov@k$			Branch $cov@k$		
		syntax	execution	line	branch	$k=1$	$k=2$	$k=5$	$k=1$	$k=2$	$k=5$
GPT-3.5 Turbo	X	99.95	94.61	93.26	89.96	85.66	87.47	89.76	78.05	80.80	84.43
GPT-3.5 Turbo	✓	**100**	**98.76**	**96.64**	**93.20**	**88.35**	**90.03**	**92.63**	**80.35**	**83.02**	**87.10**
GPT-4o	X	**99.90**	98.80	98.27	96.84	89.00	91.13	93.66	81.56	84.90	89.03
GPT-4o	✓	99.80	**99.33**	**98.51**	**96.85**	**89.39**	**91.45**	**94.02**	**81.89**	**85.12**	**89.19**

Table 2: Efficiency of automated test generation on the TestEval dataset (210 tasks) (noa, 2025a)

Model	RAG	Runtime
GPT-3.5 Turbo	X	1:32:30
GPT-3.5 Turbo	✓	**1:28:59**
GPT-4o	X	**1:55:26**
GPT-4o	✓	2:09:43

RAG "toggle", we can compare performance on the same data with and without a supplementary knowledge base. With this toggle determining whether a RAG database will be provided, we prompt our model to generate *N* unique test cases. In this experimental configuration, *N* = 20.

Following the generation of the provided quantity of test cases, we evaluate the model's performance by computing compilation success—in terms of syntax and execution—test case coverage—in terms of line and branch coverage—and a proprietary coverage metric developed by Wang et. al in Testeval, *cov@k*. The *cov@k* metric randomly samples *k* test cases and computes localized coverage.

All evaluation methodology for test case generation remains unchanged from Testeval's source code, as of July 2025. To compute compilation success, the program attempts to execute without syntax errors. For coverage metrics, pytest-cov is used to determine the proportion of branches covered by at least one test case. Finally, *cov@k* randomly samples *k* generated test cases and finds accuracy. Our findings for test case coverage are outlined in Table 1.

Regarding coverage results, several observable trends are apparent. Firstly, under the current experimental configuration, RAG-enhanced LLMs consistently outperform standard models in terms of execution success. Furthermore, RAG-enhanced LLMs have a greater rate of coverage across all areas of measured line and branch coverage, including *cov@k*. Compilation success in terms of syntax appears to be relatively uninfluenced by the presence of a RAG pipeline.

This pattern continues in Table 2, where we can see the influence of RAG on process runtime. Generally, the integration of RAG does not seem to have a substantial influence on runtime and, in fact, leads to a quicker runtime in one of our tested LLMs.

These results strongly indicate that the addition of a RAG pipeline has an overall positive influence on automated test case generation in terms of effectiveness and efficiency, with no substantial drawbacks visible at this time.

6.4 Automated Code Inspection

For automated code inspection, we tested LLM accuracy in correctly identifying bugs in code snippets. Using the same configurable pipeline outlined in automated test case generation, we prompted an LLM to locate and classify code defects in a provided snippet of code. For our evaluation, we compared the LLM-assigned bug label with the ground truth sourced from the Bug In the Code Stack dataset. This comparison was performed using OpenAI's GPT-3.5-turbo model. Our findings are outlined in Tables 3 and 5, and Figure 4.

When comparing the performance of standard models to those enhanced with RAG, several notable patterns emerge. First, in Table 3, we observe that all models show improved accuracy in bug detection when they have access to our RAG database.

Table 3: Automated code inspection results on the Bug In the Code Stack dataset (4,010 tasks) (**?**)

Model	RAG	Matches	Mismatches	Accuracy
GPT-3.5 Turbo	X	2678	1332	66.78
GPT-3.5 Turbo	✓	**3632**	**378**	**90.57**
GPT-4	X	2943	1067	73.39
GPT-4	✓	**3608**	**402**	**89.98**
GPT-4o	X	3388	622	84.49
GPT-4o	✓	**3604**	**406**	**89.88**

Table 4: Efficiency of automated code inspection on the Bug In the Code Stack dataset (4,010 tasks) (**?**)

Model	RAG	Runtime
GPT-3.5 Turbo	X	**1:05:28**
GPT-3.5 Turbo	✓	1:20:59
GPT-4o	X	1:27:35
GPT-4o	✓	**1:19:26**
GPT-4.1	X	1:25:18
GPT-4.1	✓	**1:24:20**

In Figure 4 we can evaluate model performance in classifying specific bug types. RAG-enhanced LLMs consistently perform better in identifying "mismatched comma", "mismatched quotation", "mismatched bracket", and "keyword as identifier" type bugs. The "bug-free code", "missing colon", "missing parenthesis", and "missing quotation" bug types are more variable, although RAG-enhanced LLMs have a better overall performance.

Table 5 shows us overall trends in which bug types are more challenging for our LLMs to classify. There are fewer consistent patterns present in these data, although we do see a uniformly stronger performance by RAG-enhanced models in identifying "missing quotation", "mismatched quotation", "mismatched bracket", and "keyword as identifier" types of bugs. Consistent with our findings at this point, the "bug-free code", "missing colon", and "missing parenthesis" bug-types are less evenly distributed, as well as the "missing comma" label. However, again, we see that RAG-enhanced LLMs have a better overall performance.

Along with increased accuracy and strong performance in bug identification, we see in Table 4 that applying RAG results in no significant drawbacks in terms of runtime. Instead, in two of our three tested models, RAG-enhanced LLMs perform automated code inspection faster than the base models

These results demonstrate that incorporating a RAG pipeline consistently improves accuracy and generally boosts the overall performance of automated code inspection without compromising efficiency. While RAG-enhanced LLMs struggle in specific regions of bug identification, their overall performance is strong.

7 Conclusion

Through analysis of our experimental results, we can conclude that Large Language Models hold significant potential for applications in software development, particularly in verification and validation. Regarding our initial research questions, "Can LLMs perform code inspection effectively and efficiently?" and "Can LLMs perform test case generation effectively and efficiently?", we have validated these questions.

In particular, we have found that the addition of a RAG pipeline generally increases the effectiveness of these models. Notably, RAG-enhanced models consistently outperform their standard counterparts in terms of test case generation coverage and their overall accuracy in identifying and classifying bugs during code inspection.

Efficiency remains largely unchanged with the presence of RAG, motivating the integration of this framework into real-world software solutions.

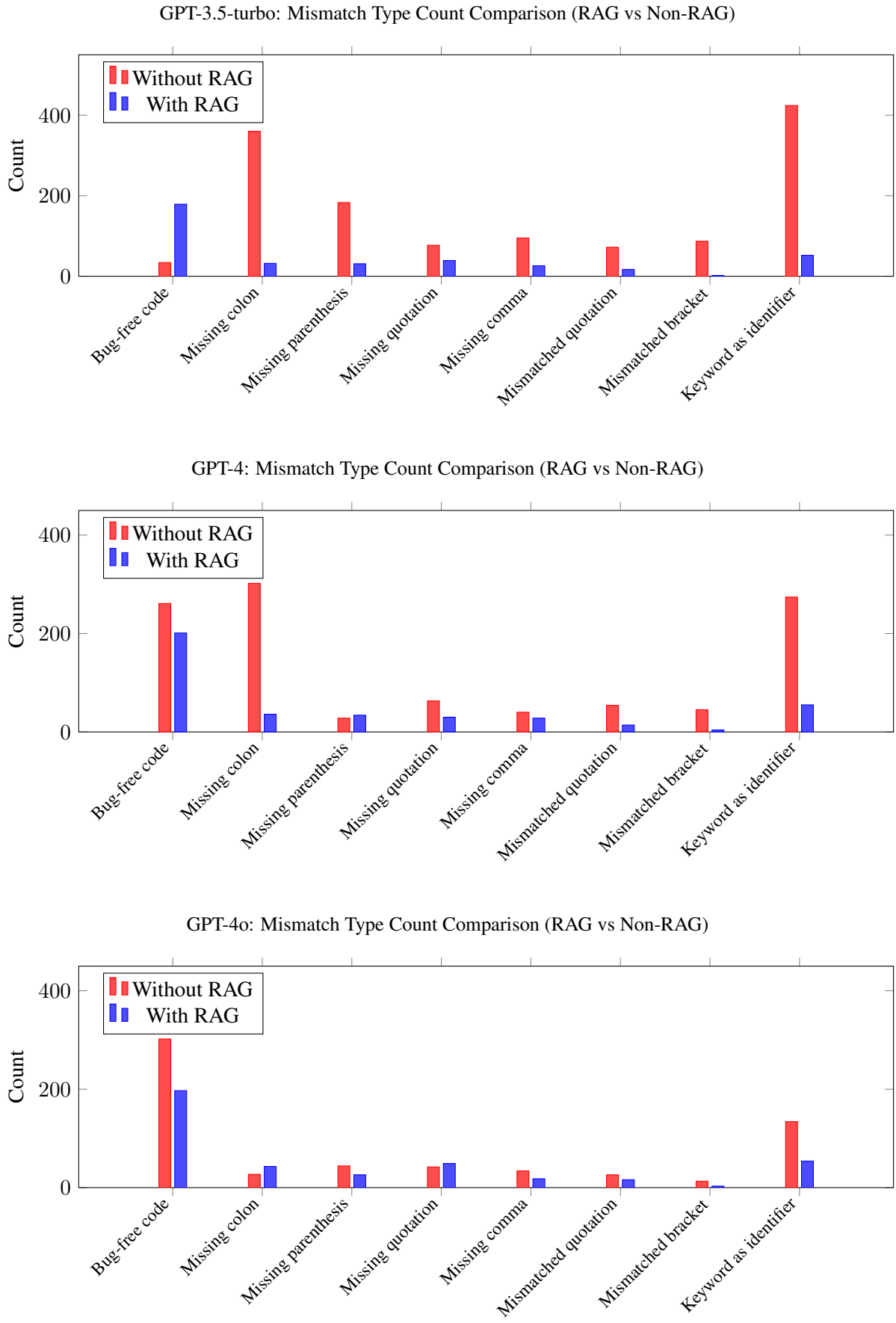

Figure 4: Mismatch Type Count Comparison (GPT-3.5, GPT-4, GPT-4o) between RAG and Non-RAG

Table 5: Mismatch type rate (proportional to overall mismatches)

Model	RAG	Bug-free code	Missing colon	Missing parenthesis	Missing quotation	Missing comma	Mismatched quotation	Mismatched bracket	Keyword as identifier
GPT-3.5-turbo	X	**2.55%**	27.03%	13.74%	**5.78%**	7.13%	5.41%	6.53%	31.83%
GPT-3.5-turbo	✓	47.35%	**8.47%**	**8.2%**	10.32%	**6.88%**	**4.5%**	**0.53%**	**13.76%**
GPT-4	X	**24.46%**	28.3%	**2.62%**	**5.9%**	**3.75%**	5.06%	4.22%	25.68%
GPT-4	✓	50.0%	**8.96%**	8.46%	7.46%	6.97%	**3.48%**	**1.0%**	**13.68%**
GPT-4o	X	48.55%	**4.34%**	7.07%	**6.75**	5.47%	4.18%	2.09%	21.54%
GPT-4o	✓	**48.52%**	10.59%	**6.4%**	12.07%	**4.43%**	**3.94%**	**0.74%**	**13.3%**

8 Threats to Validity

As the first study of its kind, our approach has some limitations. Our primary concerns include the limitations of using LeetCode data as a real-world proxy and our decision to focus on bug review activities to approximate code inspection ability.

In terms of our use of LeetCode data, while we recognize that these programming problems may not perfectly align with industry work, they serve as a strong benchmark for well-written and well-implemented code. In addition, our focus on bug review allows us to empirically evaluate LLM capability in ways not achievable through other approaches.

Beyond work-specific concerns, several technical threats to validity exist. Most notable is the problem of using LLMs without repeatable outputs. To limit spontaneity in LLM generation, we use a standardized prompt and a temperature of 0 to minimize variation in output. Furthermore, we hope that the presence of a RAG database will ensure the model draws from a consistent knowledge source.

9 Data and Source Code Availability

The source code and results for experiments conducted in this paper are accessible at https://github.com/qas-lab/reu-zoe-fingleton (noa, 2025b). This repository contains code to run and evaluate automated test case generation and automated code review using OpenAI, Ollama, and Hugging Face models.

Additionally, this work uses data from Google's Mostly Basic Python Programming dataset (noa, 2022), the Bug In the Code Stack dataset from Lee et al. (Lee et al., 2024), and the TestEval dataset from Wang et al. (Wang et al., 2025). The relevant experimental data used in this study are accessible in the source code repository. To reproduce the experimental setup, users will need to download TestEval's code from source. Details are specified in the README.md for this paper's repository.

10 Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant No. 2349452. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

2022. google-research/mbpp at master · google-research/google-research.

2024. sentence-transformers/all-MiniLM-L6-v2 · Hugging Face.

2025a. LLM4SoftwareTesting/TestEval.

2025b. qas-lab/reu-zoe-fingleton.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. ArXiv:2108.07732 [cs].

Mohamed Boukhlif, Nassim Kharmoum, and Mohamed Hanine. 2024. LLMs for Intelligent Software Testing: A Comparative Study. In *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security*, pages 1–8, Meknes AA Morocco. ACM.

Patrice Béchard and Orlando Marquez Ayala. 2024. Reducing hallucination in structured outputs via Retrieval-Augmented Generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, pages 228–238. ArXiv:2404.08189 [cs].

François Chollet. 2019. On the Measure of Intelligence. ArXiv:1911.01547 [cs].

Kenneth Ward Church, Zeyu Chen, and Yanjun Ma. 2021. Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering*, 27(6):763–778.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep

bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.

Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large language models for code analysis: Do {LLMs} really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846.

Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, Lisbon Portugal. ACM.

Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. 2024. Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives. *IEEE Transactions on Artificial Intelligence*, 5(12):5873–5893.

Yu Hao, Weiteng Chen, Ziqiao Zhou, and Weidong Cui. 2023. E&V: Prompting Large Language Models to Perform Static Analysis by Pseudo-code Execution and Verification. ArXiv:2312.08477 [cs].

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. ArXiv:2009.03300 [cs].

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems*, 43(2):1–55.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. ISSN: 1558-1225.

Hokyung Lee, Sumanyu Sharma, and Bing Hu. 2024. Bug In the Code Stack: Can LLMs Find Bugs in Large Python Code Stacks. ArXiv:2406.15325 [cs].

Zachary Levonian, Chenglu Li, Wangda Zhu, Anoushka Gade, Owen Henkel, Millie-Ellen Postle, and Wanli Xing. 2023. Retrieval-augmented Generation to Improve Math Question-Answering: Trade-offs Between Groundedness and Human Preference. ArXiv:2310.03184 [cs].

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. ArXiv:2005.11401 [cs].

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025a. Structured Chain-of-Thought Prompting for Code Generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23.

Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W. Eric Wong. 2025b. Evaluating large language models for software testing. *Computer Standards & Interfaces*, 93:103942.

Hong Yi Lin, Chunhua Liu, Haoyu Gao, Patanamon Thongtanunam, and Christoph Treude. 2025. CodeReviewQA: The Code Review Comprehension Assessment for Large Language Models. ArXiv:2503.16167.

Aiwei Liu, Qiang Sheng, and Xuming Hu. 2024. Preventing and Detecting Misinformation Generated by Large Language Models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3001–3004, Washington DC USA. ACM.

Yuanjie Lyu, Zhiyu Li, Simin Niu, Feiyu Xiong, Bo Tang, Wenjin Wang, Hao Wu, Huanyong Liu, Tong Xu, and Enhong Chen. 2025. CRUD-RAG: A Comprehensive Chinese Benchmark for Retrieval-Augmented Generation of Large Language Models. *ACM Transactions on Information Systems*, 43(2):1–32. Publisher: Association for Computing Machinery (ACM).

Gustavo F. Martins, Emiliandro C. M. Firmino, and Vinicius P. De Mello. 2024. The Use of Large Language Model in Code Review Automation: An Examination of Enforcing SOLID Principles. In Helmut Degen and Stavroula Ntoa, editors, *Artificial Intelligence in HCI*, volume 14736, pages 86–97. Springer Nature Switzerland, Cham. Series Title: Lecture Notes in Computer Science.

W. Miller and D.L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A Comprehensive Overview of Large Language Models. *ACM Transactions on Intelligent Systems and Technology*. Publisher: Association for Computing Machinery (ACM).

Venkatesh Balavadhani Parthasarathy, Ahtsham Zafar, Aafaq Khan, and Arsalan Shahid. 2024. The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research

Challenges and Opportunities. ArXiv:2408.13296 [cs].

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. ArXiv:2108.11601 [cs].

Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology*, 175:107523.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–322, Salerno Italy. ACM.

Ian Sommerville. 2016. *Software engineering*, tenth edition edition. Always learning. Pearson, Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. ArXiv:2406.04531 [cs].

Xingyi Yang, Muchao Ye, Quanzeng You, and Fenglong Ma. 2021. Writing by Memorizing: Hierarchical Retrieval-based Medical Report Generation. ArXiv:2106.06471 [cs].

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2024. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. ArXiv:2305.04207 [cs].

Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. 2024. When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method. ArXiv:2402.17193 [cs].

Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. 2024a. Retrieval-Augmented Generation for AI-Generated Content: A Survey. ArXiv:2402.19473 [cs].

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. Version Number: 16.

Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. 2024b. CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding? ArXiv:2408.10718 [cs].

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2024a. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. ArXiv:2311.10372 [cs].

Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2024b. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. ArXiv:2406.15877 [cs].

A Systematic Literature Review on Quantum Natural Language Processing (QNLP)

CJ Rivas
Department of Computer Science
University of Colorado Colorado Springs &
New Mexico State University
cjriv22@nmsu.edu

Himon Thakur
Department of Computer Science
University of Colorado Colorado Springs
CO, USA
hthakur@uccs.edu

Nazanin Siavash
Department of Computer Science
University of Colorado Colorado Springs
CO, USA
nsiavash@uccs.edu

Armin Moin
Department of Computer Science
University of Colorado Colorado Springs
CO, USA
amoin@uccs.edu

Abstract

Quantum natural language processing is a emerging field that explores the use of quantum computing (QC) to solve limitations in natural language processing (NLP) that arise in classical systems. In this paper, we conduct a systematic literature review and inclusion and exclusion criteria to explore the areas NLP that QC has impacted and find what areas have already shown promise that quantum methods outperform classical methods. We hope to assess which areas in NLP show early promise, and which areas have already shown QC methods have outperformed classical approaches.

1 Introduction

Quantum Computing (QC), which is based on the principles of Quantum Physics, has proven to be disproportionately more efficient for very specific computing tasks and algorithms. For instance, (Hagos et al., 2024) showed that using their Noisy Intermediate-Scale Quantum (NISQ) processor, they were able to solve a problem that would otherwise take more than 10,000 years on the world's best classical supercomputer, in only 200 seconds. This level of disruption holds for certain algorithms that are quantum-by-design, meaning that they can inherently benefit from the characteristics of quantum computers, such as superposition and entanglement. Examples include, but are not limited to, the Grover and Shor algorithms.

QC has been applied to Artificial Intelligence (AI), leading to the emerging field of Quantum AI (QAI). AI has different sub-disciplines, such as Machine Learning (ML) and Natural Language Processing (NLP). Over the past decade, ML has contributed a lot to NLP. State-of-the-art NLP methods and techniques at the present time use pre-trained machine learning models, called Large Language Models (LLMs), also known as Frontier AI Models or Foundation AI Models. Also, other ML models, such as Long Short-Term Memories (LSTMs) and Convolutional Neural Networks (CNNs), are widely used in NLP. Although these are classical models, their success in capturing the complexity of linguistics such as semantic ambiguities or contextual dependencies often rely on vector space representations or computational mechanisms, which are inherently compatible with the mathematical properties of quantum mechanics.

Research over the past years has shown promising results in using Quantum ML (QML) for NLP tasks. For instance, (Pandey et al., 2023a) used both classical and quantum-based variants of LSTMs or Part of Speech (PoS) tagging - a crucial NLP task on social media code-mixed language text. Their experiments indicated that quantum-based LSTMs outperformed the classical variant.

The contribution of this paper is to conduct a Systematic Literature Review (SLR) in the landscape of research on quantum and hybrid quantum classical approaches to NLP. This includes works that focus on one or more of the specific NLP tasks, regarding both automated syntax and semantics analysis. Examples of relevant areas include, but are not limited to, PoS tagging, Named-Entity Recognition (NER), anaphora resolution, word-sense disambiguation, context-aware question answering, sentiment analysis, Machine Translation (MT), text summarization, text rephrasing, and approaches inspired by quantum algorithms and quantum information science, which are not necessarily using quantum computers. In particular, we answer the following Research Questions (RQs): RQ1. What areas of NLP can QC advances impact? RQ2. In what NLP areas have QC approaches already outperformed classical approaches?

2 Background

2.1 NLP

NLP is a subfield of Artificial Intelligence (AI). It focuses on the mathematical and computational

modeling of natural language and the development of various systems (Joshi, 1991). One crucial subset of NLP is preprocessing. Pre-processing converts the natural language in a way that is understandable, predictable, and analyzable by the machine (Tabassum and Patil, 2020). This could include tasks such as tokenization, removing stop words, stemming, and lemmatization. Tokenization splits a sentence into words known as "tokens", using delimiters such as spaces and punctuation. Stop words are frequently used words that have little meaning such as "and", "is", "the", "it", and "was". Stemming involves combining the alternative forms of a word into a common representation. (Kannan et al., 2014). For (e.g., information, inform, informative) could all be reduced to a common prefix "inform". Although stemming simply chops off word endings, lemmatization reduces the words to their original vocabulary form (Smelyakov et al., 2020). Lemmatization can be more effective than stemming, but require more time to analyze since it must consider the context of the word and its part of speech. These foundational steps are needed to prepare the data for various downstream NLP tasks.

A fundamental task in NLP is *Part-of-Speech (PoS) tagging*, where sentences are broken down into their corresponding speech, such as nouns, verbs, and adjectives (Pandey and Pakray, 2023). There are several different methods and algorithms in PoS, such as neural network methods (Chiche and Yitagesu, 2022), and probabilistic methods (Pakray et al., 2018). PoS can help clear ambiguity in a sentence where more than one verb is present, or particularly distinguish in social media where code-mixed languages can be full of many verbs and nouns (Pandey et al., 2023a).

Although PoS tagging helps us understand the grammatical role of individual words, *Named Entity Recognition (NER)* is an important method in NLP that allows information to be extracted from the given text. NER is able to detect and categorize the given text into named entities, which hold semantic value such as a person, place, organization, etc (Jehangir et al., 2023). So, while NER can tell us that "Apple" is a company, it can not help us distinguish which "bank" the text is referring to. The text might be about a financial institution, but in reality it is thinking about the side of a river.

Anaphora resolution is a fundamental task in NLP, where it tries to identify the antecedent in an anaphor (Lata et al., 2021). Essentially, it involves identifying the expression or phrase corresponding to a pronoun already mentioned in the text (Prajapati et al., 2024).

So *Word-Sense Disambiguation (WSD)* is a key aspect of ambiguity resolution in NLP. WSD can be used to disambiguate words in sentences and to understand words with more than one meaning (Zhang et al., 2024a). Although WSD can disambiguate words with more than one meaning, it is important to find which words it actually refers to. Another task in NLP that helps us understand the context of the text is semantic representation.

Semantic representation in NLP is a process in which meaning is assigned to words, sentences, and phrases to understand the context behind them (Li et al., 2018). Methods and algorithms such as Statistical Relational Learning (SRL) and Knowledge Bases (KB) have paved the way in NLP by helping computer systems understand the hidden meaning behind words or allowing accurate information extraction (Garg et al., 2019). Essentially once ambiguity is resolved and the text's context is understood, we can use this information for text classification.

Text classification is an important task in NLP where textual data automatically gets assigned predefined labels or categories (Meichanetzidis et al., 2023). There are many problems that arise in NLP where text classification can solve such as email spam filtering, information retrieval, and topic labeling (Joulin et al., 2016). Another crucial task in NLP, is understanding the emotions or attitudes toward a specific entity.

Sentiment analysis, also known as opinion mining, is the computational study of the opinions, attitudes, and emotions of people toward an entity, where an entity can represent individuals or topics (Medhat et al., 2014). While sentiment analysis gives us the emotions or attitudes within a text, text summarization condenses the information into a concise review preserving its core meaning.

Text summarization is a task in NLP that automatically produces a summary of key ideas and includes all relevant information based on the original document (Widyassari et al., 2022). There are two main types of text summarization, extractive and abstractive. Extractive text summarization involves selecting important sentences, paragraphs, etc., and concatenating into a shorter form based on the given input (Moratanch and Chitrakala, 2017).

Abstractive text summarization has the ability to develop new sentences that convey the meaning of the original input given (Moratanch and Chitrakala, 2016). Text summarization might give users some concise information they need, but at times users often have more specific informational needs.

Question Answering (QA) can help with that and is a task in NLP that incorporates systems to answer questions and give accurate answers based on the given natural language (Lende and Raghuwanshi, 2016a). In question answering, two prevalent types are open-domain and closed domain question answering. Open domain question answering deals with the questions that can be related to any domain with low accuracy. Closed domain question answering deals with specific domain related questions that can answer it with high accuracy but limited to one domain (Lende and Raghuwanshi, 2016b). Modern context-aware question answering can recognize answers to questions depending on the surrounding information. A crucial NLP task with broad utility across various fields is machine translation.

Machine translation (MT) is a cornerstone in NLP as it automatically converts text or speech to another language (Abbaszade et al., 2021). Neural machine translation (NMT) stands as the prevailing paradigm in modern machine translation, where the NMT model can read the entire source sentence first. This allows the model to form a good understanding before generating the target sentence word by word (Wang et al., 2022). MT plays a vital role in helping break language barriers, continuing to preserve extinct languages, and empowering multilingual chat-bots to help users communicate (Narayan et al., 2016).

2.2 AI

AI is the development of intelligent machines, especially intelligent computer programs (McCarthy et al., 2007). AI has been a longstanding field that has impacted many areas such as healthcare (Jiang et al., 2017), education (Chen et al., 2020), and business (Bharadiya et al., 2023). AI offers many advantages that helps users improve their daily lives and tasks. These advantages include automation of repetitive tasks (Acemoglu and Restrepo, 2018), enhanced-decision making (Dear, 2019), and better accuracy (Pantanowitz et al., 2020). For example, AI plays a vital role in agriculture, where it is able to identify active pests and suggest control measures, help with general crop management, or diagnose diseases and recommend control measures for ailing plants (Bannerjee et al., 2018). Another key area that AI strives to make a difference is the transportation and logistics area. AI can offer advantages such as developing full autonomous vehicles by using AI algorithms, combined with advanced sensor technologies, which can help improve road safety, mitigating traffic congestion, and increasing overall efficiency (Bharadiya et al., 2023). AI powered logistics and routing systems can contribute to sustainable transportation by prioritizing environmentally friendly options, which helps reduce carbon emissions. AI can be used to help save time, resources, and financial needs when used properly and highly impact areas like agriculture and transportation.

2.3 ML

An important branch of AI that enables computer systems to learn and give better predictions based on input is ML. ML is a process where computers are able to automatically learn from experience, which can enhance some measure of performance when executing a task (Jordan and Mitchell, 2015). Many areas have benefited from ML, those include computer vision (Khan et al., 2021), healthcare (Habehh and Gohel, 2021), and bioinformatics (Larranaga et al., 2006). ML has demonstrated its ability to significantly improve tasks previously slow or inefficient for other methods or AI. It is able to provide computational advantages due to its advanced algorithms, which could lead to greater efficiency (Rasheed, 2023). Areas such as finance benefit a lot from ML as they are able to improve bankruptcy prediction, stock price prediction, and portfolio management using nonlinear classifiers, such as Support Vector Machines (SVMs) and Artificial neural networks (ANNs) which can outperform traditional methods (Ahmed et al., 2022).

3 Related Work

3.1 QNLP in Bioinformatics: Existing Methodologies, and Future Directions

(Pallavi and Prasanna Kumar, 2025) discuss and review the branches of bioinformatics that been influenced by QNLP such as genomic sequence analysis, and protein structure prediction. Although the paper focuses on bioinformatics and the impact of QNLP within the domain, the research methodology is a strong reference for our work to conduct a

strong systematic literature review on QNLP.

3.2 SLR: QML and its Applications

(Peral-García et al., 2024b) mention the impacts of quantum machine learning and how quantum devices can be leveraged in several areas such as finance and chemistry, despite their limitations, such as fault-tolerant and limited qubits.

3.3 QML in Disease Detection: a Survey of Applications

(Upama et al., 2023) discuss the potential impact quantum machine learning (QML) has on disease detection leveraging QC techniques and algorithms to help analyze medical datasets. The authors identify patterns to predict the appearance or progression of the disease with greater results than classical methods.

3.4 The State of the Art of NLP

(Sawicki et al., 2023) discuss the state-of-the-art NLP using NLP techniques in the study. They used methods such as a named entity recognition model, which extracts named entities like databases and languages. They used an API for ArXiv to automate the process of collecting papers to ensure that reproducibility is met.

4 Problem Statement

Quantum natural language processing has been emerging as a prominent field as it has been able to solve some of the limitations that natural language processing (NLP) faces in classical systems. There has been many areas in NLP that have been impacted by quantum computing (QC) and in this paper, we will systematically find the areas of NLP that have impacted by QC, which areas in NLP have QC already outperform classical approaches, and find any research gaps.

5 Proposed Approach

5.1 Research Questions

This paper will address the systematic literature review with the following research questions (RQs):
RQ1: What areas of NLP can QC impact? The emerging field of quantum natural language processing (QNLP) has attracted the attention of many researchers to using quantum computing as a means of solving the challenges that NLP faces in classical systems.

RQ2: In what NLP areas have QC approaches already outperformed classical approaches? Many areas in NLP have been improved by QC methods. Quantum-inspired models, such as quantum long short-term memory, has been used to improve machine translation (Abbaszade et al., 2021), or tensor product representations to reduce memory usage during training for word embeddings, which support many NLP tasks (Aliakbar Panahi, 2020).

5.2 Inclusion and Exclusion Criteria

Inclusion criteria. If the papers meet any of these requirements, they will be included:

- If they are published over 2020-2025.
- If they are peer-reviewed conferences or journal papers indexed by IEEE Xplore, ACM, Springer, SCOPUS, or Science Direct.
- If they are written in English.

Exclusion criteria. If the papers meet any of these conditions, they will be excluded:

- Papers that do not involve the use of QC in NLP or papers that talk about QC and mention NLP as an example field but are not focused on it.
- Papers that are preprints, for example, on ArXiv, except for preprints published over 2024-2025.
- Papers that are surveys or literature reviews.

5.3 Search Strategy

We follow a systematic search strategy that will allow consistency and reproducibility. We adopted the procedures from (Raharjana et al., 2021) and (Keele et al., 2007) to ensure a rigorous and consistent SLR was met.

5.3.1 Search Keywords

Category	Keywords
Quantum natural language processing (main)	"quantum nlp" OR "quantum natural language processing" OR qnlp
Quantum concepts	qc OR "quantum computing" OR "quantum machine learning" OR nisq OR noisy intermediate-scale quantum OR "quantum algorithms" OR "quantum annealing" OR "quantum-inspired" OR "quantum circuits"
Natural language processing (core)	nlp OR "computational linguistics" OR "natural language processing" OR "natural language understanding" OR nlu
Artificial Intelligence / Advanced NLP Models	"artificial intelligence" OR ai OR "machine learning" OR ml OR "deep learning" OR dl OR "neural networks" OR "large language models" OR llms OR "generative ai" OR "generative artificial intelligence" OR "embeddings" OR "tensor networks" OR DisCoCat
Natural language processing tasks/methods	"part-of-speech tagging" OR "pos tagging" OR "text classification" OR "ambiguity resolution" OR "sentiment analysis" OR "question answering" OR qa OR "machine translation" OR "text summarization" OR "semantic representation" OR "anaphora resolution" OR "word-sense disambiguation" OR wsd OR "named entity recognition" OR ner

Table 1: Keywords

In order to select the keywords that will be used to build search strings, we found top-relevant papers from strong peer-reviewed academic databases and search engines such as Google Scholar to come up with our keywords.

The set of keywords in table 1 was determined by focusing on our research questions, the adjacent areas NLP is affected by such as machine learning, and the areas that can be and are impacted by quantum approaches.

5.3.2 Search Strings

We were able to obtain relevant studies based on keyword selection, and we used those keywords to build our string that we would use for our search process. We started with a broad combination such as (e.g., "quantum natural language processing" OR qnlp OR "quantum nlp") which yields high results but soon narrowed it by adding AND operators to refine our search.

The search string we were able to come up with is

(("quantum natural language processing" OR qnlp OR "quantum nlp" OR "quantum-inspired natural language processing" OR "quantum-inspired nlp") AND ("quantum computing" OR "quantum machine learning" OR "quantum deep learning" OR "noisy intermediate-scale quantum" OR nisq OR "quantum algorithms" OR "quantum annealing" OR "quantum-inspired" OR "quantum circuits" OR "quantum-classical")) AND ("natural language processing" OR nlp OR "computational linguistics" OR "natural language understanding" OR nlu) AND ("artificial intelligence" OR "ai" OR "large language models" OR "llms" OR "deep learning" OR "dl" OR "neural networks" OR "embeddings" OR "tensor networks" OR "machine learning" OR "ml" OR DisCoCat OR "sentiment analysis" OR "machine translation" OR "text generation" OR "text summarization" OR "question answering" OR qa OR "part-of-speech tagging" OR "pos tagging" OR "text classification" OR "ambiguity resolution" OR "semantic representation" OR "anaphora resolution" OR "word-sense disambiguation" OR wsd OR "named entity recognition" OR ner)

Although our core search string remained consistent across all databases used, minor adjustments were made in ScienceDirect, due to the unique query syntax, which limited to using no more than 8 Boolean operators. The following string

("quantum natural language processing" OR qnlp) AND ("large language models" OR DisCoCat OR "deep learning" OR "neural networks" OR "machine learning" OR "nisq")

was modified to fit within the constraints of the limited boolean operators allowed.

5.4 Selection Process

Our process of selecting our papers will be inspired by the PRISMA flow diagram (Liberati et al., 2009), which will allow us to document our process to ensure consistency and reproducibility. First, we apply our identification process, which takes care of duplicates that might arise in our search of multiple databases.

Subsequently, we moved onto the screening process where our inclusion and exclusion mentioned in 5.2, was predefined, which allows us to read the title and abstract and apply the criteria. The criteria we focused on applying at this stage were: 1.) The paper must be written in English, and 2.) The paper must explicitly address QC approaches in NLP in the title, abstract, or keywords. Papers that did not meet this criteria were excluded. Moving on through the process, we proceeded to our eligibility process, which assesses the papers that made it through the screening process. This stage involves reading the full text of papers that successfully passed our initial screening. The purpose of this stage, is to thoroughly assess the papers relating to our inclusion and exclusion, defined in 5.2, to determine its suitability in the SLR. Finally, all papers that have successfully passed the screening and eligibility process will be included for data extraction. We extract necessary and meaningful information from the studies to compile statistics and provide valuable insights for the SLR.

6 Systematic Literature Review Results

We were able to capture data from our systematic literature review by introducing our table 2. Our table provides a detailed overview that shows key characteristics such as the publication year, the quantum approaches used, the data set used, and the frameworks/libraries/SDKs the authors used. This allows us to synthesize the information from the 103 included studies.

6.1 Overview of Study Selection

We identified 103 primary studies based on our 5.4 process. Our initial search process identified a total of 370 papers from multiple databases: 10 from ACM Digital Library, 16 from ScienceDirect, 28 from IEEE Xplore, 52 from SpringerLink, and 263 from SCOPUS. Moving onto our identification phase, we removed 75 duplicates to ensure that a set of unique papers would be included for the subsequent stages.

Throughout the selection process, 172 papers were excluded, while 123 papers were considered suitable for the eligibility assessment during the screening phase. During the eligibility phase, 20 papers were further excluded, leaving a final set of 103 included papers for the Systematic Literature Review (SLR).

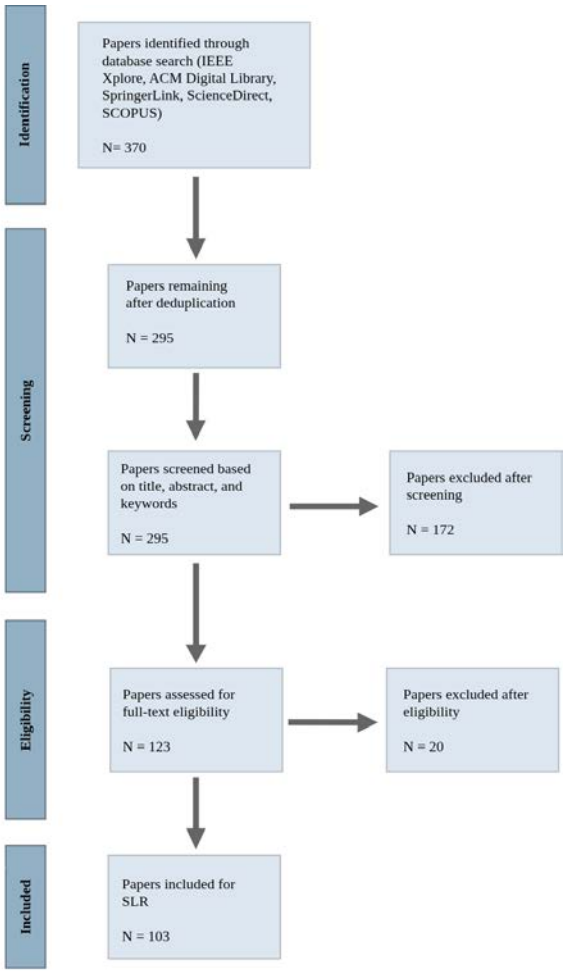

Figure 1: SLR Flowchart Results

6.2 Publication Trends in Quantum Natural Language Processing

Addressing QNLP publications demonstrates the expanding influence that QC has on various NLP domains. Our systematic literature review led us to identify the following publication distribution: 0 papers from 2020, 5 papers from 2021, 12 papers from 2022, 28 papers from 2023, 48 papers from 2024, and 17 papers from 2025.

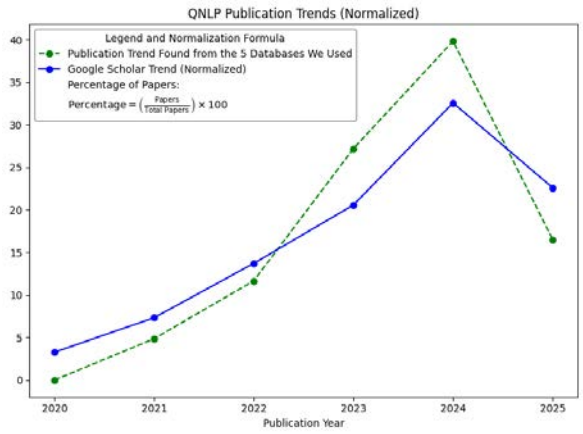

Figure 2: QNLP Publication Trend: (Ours Vs Google Scholar)

Figure 2 showcases the publication trend of our included studies. Our analysis of the publication data also compares it with Google Scholar. These external data were collected using the search string detailed in Section 5.3.2. We refined our searches to a single-year range (e.g., 2020-2020) up to 2025 to collect the yearly publication count. The total publications from each year from Google Scholar goes as follows: 21 papers from 2020, 47 papers from 2021, 88 papers from 2022, 132 papers from 2023, 209 papers from 2024, and 145 papers from 2025 for a total of 642 publications.

We notice similar results with our search string on the 5 selected databases that we used mentioned in 5.2. Although our systematic search yielded no publications for 2020, this aligns with the trend Google Scholar displays, as the number of QNLP publications for that year was also low. Approximately 3.271% of 642 publications were reported by Google Scholar throughout all years, indicating that QNLP was still in its infancy in 2020. Moving on through the years, we notice an exponential growth of publications from 2021 to 2024 (approximately 4.854% to 39.805%) for our trend and (approximately 7.320% to 32.554%) for Google Scholar. The growth of QNLP shows the earlier years being more theoretical explorations such as proof-of-concepts to small early implementations and experiments on applicable areas of NLP. Although 2025 shows a significant drop from 2024 for both trends, this is likely due to the year not being complete yet, which could still pull in a significant amount of papers.

6.3 (RQ1) What areas of NLP can QC impact?

The results of our included papers demonstrated the areas of NLP that have been impacted by QC and show areas that are partially touched.

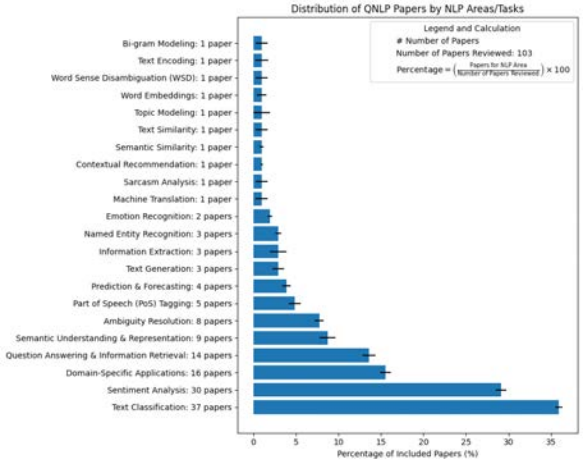

Figure 3: Note: NLP Areas/Tasks may overlap (some papers cover multiple areas/tasks)

Figure 3 shows the included QNLP papers by their respective NLP tasks, offering potential insight into the current landscape. It shows some of the most promising areas in QNLP is text classification and sentiment analysis. This is evidently true, as the increase in digital communication such as emails, texts, and social networks has become a standard for consumers. NLP tasks such as sentiment analysis, question answering, text classification, and Part-of-Speech (PoS) leverage these capabilities. For example, they can extract emotional tones from social networks (Thota and Dash, 2025), using PoS tagging to assign labels in code-mixed languages (Pandey et al., 2023b), or using text classification to separate questions into 20 different categories (Day et al., 2023).

Our search string has successfully identified various NLP areas influenced by QC. This allows us to gain insights into areas that show strong promise

and to other areas which still need significant research. While sentiment analysis and text classification are prominent areas where QC has impacted, it has also made a significant impact on domain-specific applications. Areas such as the medical field (Arora et al., 2025) and agriculture field (Bhargavi et al., 2024) have been leveraging the use of QC to improve NLP related tasks within the fields. In contrast, our search string showed areas like text encoding, word embeddings, and bi-gram modeling have seen minimal impact by QC. This could suggest the lack of research into applying QC to these fundamental areas are still in its early stages. This allows opportunities for researchers to find potential breakthroughs and pioneer advancements in these less-explored fundamental areas, free from the current saturation seen in other areas. If QC can offer fundamental improvements at this level, then it could have a ripple effect in other more complex areas of NLP.

6.4 (RQ2) In what NLP areas have QC approaches already outperformed classical approaches?

We identified which areas of NLP have been most and least affected by QC, but our next step involves evaluating the performance of quantum implementations against classical methods. Several prominent areas show significant improvement using quantum implementations over classical ones. A classical neural network has its limitations such as their attention mechanism, where their inability lies to model fine-grained, sememe-level semantic interactions within words and across different dimensions of words. (Gao et al., 2025) propose an approach called Quantum-inspired hierarchical Semantic Interaction Model (QSIM), which utilizes quantum entanglement theory to capture hierarchical semantic interaction information in Hilbert space. Experimental results on 15 text classification datasets show the proposed QSIM outperform classical methods in several areas. A key metric that was used during the experiments was the Macro-F1, which measures how well the model identifies each category and how well it recalls all instances of that category. The results show the QSIM model outperform classical models during the test such as MLP, TextCNN, DPCNN, RCNN, LSTM, HAN, and ATT. On the the SST dataset, QSIM+MLP showed a significant improvement on the Macro-F1 score from a 0.498 to a 0.537. Another key metric that was used to test QSIM was accuracy, which was measured by the ratio of correct predictions to the total number of predictions. The results show on the Reuters dataset, QSIM+DPCNN (0.984) outperforms DPCNN(0.903) by 0.081 or 8.1% on the accuracy metric. This represents a significance difference, indicating quantum implementations have outperformed classical ones in text classification. QSIM also shows significant improvements over traditional quantum models that utilize density matrix representations, further solidifying its capabilities within state-of-the-art.

A prominent field where quantum methods have shown promise outperforming classical methods is sentiment analysis. Classical models often face limitations tackling the complexities of the human language, despite being widely used. Some limitations include high computational scalability for high dimensional data, high resource intensiveness, and processing nuance in emotion and intensity (e.g., delight vs eagerness) where classical models often simplify sentiment into broad categories, struggling with the complexity of human emotions. (Masum et al., 2023) propose a methodology for sentiment analysis using hybrid quantum-classical machine learning algorithms. The proposed methodology is evaluated with two distinct datasets, based on English and Bengali languages. For the Twitter dataset, the proposed hybrid quantum-classical algorithms outperform the classical support vector machine (SVM) in certain scenarios. For the classical SVM classifier, it scored 72% on the Bengali dataset and 84% on the English dataset. For the proposed hybrid quantum-classical SVM models performed worse using the PCA and Haar wavelet transform compared to the classical SVM model at a 71.23% accuracy. This however was not the full picture, specifically, when the Haar transform was utilized for data compression, the classical SVM model test accuracy dropped to about 58% after one level of decomposition. In contrast, the hybrid quantum-classical algorithms provided better and steady classification accuracy of 71.23%. For the Bengali dataset, the hybrid quantum-classical model did not show significant improvement over the classical SVM model prior to dimensionality reduction, with both models showing similar accuracy at approximately 72%. The classical SVM model remained constant in terms of accuracy up

to the maximum decomposition level because the dataset used in the experiment is balanced. Meanwhile, the hybrid quantum-classical model showed the same level of performance as classical systems without dimensionality reduction. This shows the hybrid quantum-classical model with dimensionality reduction was highly effective, leading for sentiment analysis to be perform well on a large dataset like the Bengali dataset. The outcome demonstrates that quantum-classical implementations can offer advantages not completely based in superior accuracy or precision, but rather in their capacity to handle more data, which allows for better computational overhead and efficiency.

Beyond text classification and sentiment analysis, our initial research suggests that quantum and hybrid quantum-classical implementations have also shown potential in other subareas of NLP. For instance, areas such as NER (Day et al., 2023), PoS Tagging (Pandey et al., 2023b), and Binary Classification (Harvey et al., 2025) have shown that quantum approaches can outperform classical methods in one way or another. These works aim to leverage quantum properties for (e.g., more accurate named entity recognition, enhanced text tagging, better generalization capabilities) which could lead to improved (e.g., time complexity, computations, and resource compression) in these specific areas.

7 Threats to Validity

While our research implements rigorous methods, it is important for us to acknowledge the internal and external validity of our results. A potential threat to our validity comes from the reviewed literature itself. Specifically, not all papers included in the study mentioned the libraries, frameworks, SDKs, or datasets utilized. We only included data that was explicitly mentioned in the paper the authors used in their work and marked it 'N/A' if it was not. Consequently, our data extraction for our table 2 may not be entirely exhaustive or a 100% accurate representation of the literature included in the study. Another threat to our validity, was the use of 'toy datasets' in some of the included literature for our study. Smaller datasets can lead to overfitting of the model and can inflate the results, which do not generalize to unseen or different data distribution. This can impact the generalizability and real world applicability of their results, which we draw conclusions from.

8 Conclusion

We laid out the groundwork for a systematic literature review to ensure our coverage on selected NLP tasks impacted by QC. Our results show the areas where QC demonstrates the most impact and where it can offer potential benefits. We also revealed the trends of QNLP publications within the lat five years, indicating the few recent years show great progress for advancement.

Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant No. 2349452. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

Mina Abbaszade, Vahid Salari, Seyed Shahin Mousavi, Mariam Zomorodi, and Xujuan Zhou. 2021. Application of quantum natural language processing for language translation. *IEEE Access*, 9:130434–130448.

Daron Acemoglu and Pascual Restrepo. 2018. Artificial intelligence, automation, and work. In *The economics of artificial intelligence: An agenda*, pages 197–236. University of Chicago Press.

N Roshan Ahmed and S Shridevi. 2024. Deepket-quantum space-efficient word embedding layer for steganalysis. In *2024 3rd International Conference on Artificial Intelligence For Internet of Things (AIIoT)*, pages 1–6. IEEE.

Shamima Ahmed, Muneer M Alshater, Anis El Ammari, and Helmi Hammami. 2022. Artificial intelligence and machine learning in finance: A bibliometric review. *Research in International Business and Finance*, 61:101646.

Mst Shapna Akter, Hossain Shahriar, and Zakirul Alam Bhuiya. 2022. Automated vulnerability detection in source code using quantum natural language processing. In *International Conference on Ubiquitous Security*, pages 83–102. Springer.

Aaranya Alexander and Dominic Widdows. 2022. Quantum text encoding for classification tasks. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, pages 355–361. IEEE.

Tom Arodz Aliakbar Panahi, Seyran Saeedi. 2020. word2ket: Space-efficient word embeddings inspired by quantum entanglement. In *Proceedings of the 2020 International Conference on Learning Representations (ICLR)*.

Alexander P Alodjants, Anna E Avdyushina, Dmitriy V Tsarev, Igor A Bessmertny, and Andrey Yu Khrennikov. 2024. Quantum approach for contextual search, retrieval, and ranking of classical information. *Entropy*, 26(10):862.

Ebrahim Ardeshir-Larijani and Mohammad Mahdi Nasiri Fatmehsari. 2024. Hybrid classical-quantum transfer learning for text classification. *Quantum Machine Intelligence*, 6(1):19.

Chandani Arora, Ramandeep Sandhu, and Sardar MN Islam. 2025. Quantum linguistics in patient-clinical trial relations for improved healthcare. In *2025 International Conference on Pervasive Computational Technologies (ICPCT)*, pages 186–190. IEEE.

Gouravmoy Bannerjee, Uditendu Sarkar, Swarup Das, and Indrajit Ghosh. 2018. Artificial intelligence in agriculture: A literature survey. *international Journal of Scientific Research in computer Science applications and Management Studies*, 7(3):1–6.

Niyazi Furkan Bar, Musa Yenilmez, Sedef Aksu, and Mehmet Karakose. 2024. A quantum computing based approach for sentiment analysis in bilateral conversations. In *2024 28th International Conference on Information Technology (IT)*, pages 1–4. IEEE.

Hacene Belhadef, Hala Benchiheb, and Lynda Lebdjiri. 2023. Exploring the capabilities and limitations of vqc and qsvc for sentiment analysis on real-world and synthetic datasets. In *European Conference on Advances in Databases and Information Systems*, pages 415–424. Springer.

Jasmin Praful Bharadiya, Reji Kurien Thomas, and Farhan Ahmed. 2023. Rise of artificial intelligence in business and industry. *Journal of Engineering Research and Reports*, 25(3):85–103.

G Bhargavi, Manisha Das, S Banumathi, E Malarvizhi, and R Surendran. 2024. Leveraging nlp and quantum computing for advanced agricultural solutions. In *2024 8th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 900–906. IEEE.

Suman Bharti, Dan Chia-Tien Lo, and Yong Shi. 2024. Enhancing contextual understanding in knowledge graphs: Integration of quantum natural language processing with neo4j llm knowledge graph. In *2024 IEEE International Conference on Big Data (BigData)*, pages 8628–8630. IEEE.

Lukas Bischof, Stefan Teodoropol, Rudolf M Füchslin, and Kurt Stockinger. 2025. Hybrid quantum neural networks show strongly reduced need for free parameters in entity matching. *Scientific Reports*, 15(1):4318.

Yousra Bouakba and Hacene Belhadef. 2023. Ensemble learning based quantum text classifiers. In *European Conference on Advances in Databases and Information Systems*, pages 407–414. Springer.

Giuseppe Buonaiuto, Raffaele Guarasci, Giuseppe De Pietro, and Massimo Esposito. 2025. Multilingual multi-task quantum transfer learning. *Quantum Machine Intelligence*, 7(1):46.

Giuseppe Buonaiuto, Raffaele Guarasci, and Massimo Esposito. 2024a. Quantum transfer learning for sentiment analysis: an experiment on an italian corpus. In *Proceedings of the 2024 Workshop on Quantum Search and Information Retrieval*, pages 25–30.

Giuseppe Buonaiuto, Raffaele Guarasci, Aniello Minutolo, Giuseppe De Pietro, and Massimo Esposito. 2024b. Quantum transfer learning for acceptability judgements. *Quantum Machine Intelligence*, 6(1):13.

Damir Cavar and Koushik Reddy Parukola. 2025. Word and text similarity using classical word embeddings in quantum nlp systems. In *2025 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)*, pages 1–5. IEEE.

Damir Cavar and Chi Zhang. 2024. Semantic similarities using classical embeddings in quantum nlp. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 2, pages 450–451. IEEE.

Fu Chen, Qinglin Zhao, Li Feng, Chuangtao Chen, Yangbin Lin, and Jianhong Lin. 2025. Quantum mixed-state self-attention network. *Neural Networks*, 185:107123.

Lijia Chen, Pingping Chen, and Zhijian Lin. 2020. Artificial intelligence in education: A review. *IEEE access*, 8:75264–75278.

Yixiong Chen and Weichuan Fang. 2024. Multi-scale feature fusion quantum depthwise convolutional neural networks for text classification. *arXiv preprint arXiv:2405.13515*.

Alebachew Chiche and Betselot Yitagesu. 2022. Part of speech tagging: a systematic review of deep learning and machine learning approaches. *Journal of Big Data*, 9(1):10.

A. D. Correia, M. Moortgat, and H. T. C. Stoof. 2022. Quantum computations for disambiguation and question answering. *Quantum Information Processing*, 21(4):126.

Wei Day, Hao-Sheng Chen, and Min-Te Sun. 2023. Qnet: A quantum-native sequence encoder architecture. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 246–255. IEEE.

Giovanni de Felice, Alexis Toumi, and Bob Coecke. 2020. Discopy: Monoidal categories in python. *arXiv preprint arXiv:2005.02975*.

Keith Dear. 2019. Artificial intelligence and decision-making. *The RUSI Journal*, 164(5-6):18–25.

Arijit Dey and Jitendra Nath Shrivastava. 2024. Quantum inspired zeroshot classification for adverse drug reactions detection from social media reviews. In *2024 International Conference on Artificial Intelligence and Emerging Technology (Global AI Summit)*, pages 953–958. IEEE.

Riccardo Di Sipio, Jia-Hong Huang, Samuel Yen-Chi Chen, Stefano Mangini, and Marcel Worring. 2022. The dawn of quantum natural language processing. In *ICASSP 2022-2022 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 8612–8616. IEEE.

J Ismael Díaz-Ortiz, Axel Villanueva, and Francisco Delgado. 2023. Strongly entangling neural network: Quantum-classical hybrid model for quantum natural language processing. In *International Conference on Mathematical Modeling in Physical Sciences*, pages 503–514. Springer.

Jurek Eisinger, Ward Gauderis, Lin de Huybrecht, and Geraint A Wiggins. 2025. Classical data in quantum machine learning algorithms: Amplitude encoding and the relation between entropy and linguistic ambiguity. *Entropy*, 27(4):433.

Zipeng Fan, Jing Zhang, Peng Zhang, Qianxi Lin, Yizhe Li, and Yuhua Qian. 2024. Quantum-inspired language models based on unitary transformation. *Information Processing & Management*, 61(4):103741.

Srinjoy Ganguly, Sai Nandan Morapakula, and Luis Miguel Pozo Coronado. 2022. Quantum natural language processing based sentiment analysis using lambeq toolkit. In *2022 Second International Conference on Power, Control and Computing Technologies (ICPC2T)*, pages 1–6. IEEE.

Hui Gao, Peng Zhang, Jing Zhang, and Chang Yang. 2025. Qsim: a quantum-inspired hierarchical semantic interaction model for text classification. *Neurocomputing*, 611:128658.

Dinesh Garg, Shajith Ikbal, Santosh K. Srivastava, Harit Vishwakarma, Hima Karanam, and L Venkata Subramaniam. 2019. Quantum embedding of knowledge for reasoning. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Muhammad Mustafa Umar Gondel, Kyung Min Byun, and Hyundong Shin. 2024. Advancing textual semantic understanding with quantum lstm networks. In *2024 15th International Conference on Information and Communication Technology Convergence (ICTC)*, pages 512–517. IEEE.

Raffaele Guarasci, Giuseppe Buonaiuto, Giuseppe De Pietro, and Massimo Esposito. 2023. Applying variational quantum classifier on acceptability judgements: a qnlp experiment. In *International Conference on Numerical Computations: Theory and Algorithms*, pages 98–112. Springer.

Hafsa Habehh and Suril Gohel. 2021. Machine learning in healthcare. *Current genomics*, 22(4):291–300.

Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. 2024. Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives. *IEEE Transactions on Artificial Intelligence*, 5(12):5873–5893.

Kathleen Hamilton, Mayanka Chandra Shekar, John Gounley, Dhanvi Bharadwaj, Prasanna Date, Eduardo Antonio Coello Pérez, In-Saeng Suh, and Georgia Tourassi. 2023. Characterizing quantum classifier utility in natural language processing workflows. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 2, pages 369–370. IEEE.

Carys Harvey, Richie Yeung, and Konstantinos Meichanetzidis. 2025. Sequence processing with quantum-inspired tensor networks. *Scientific Reports*, 15(1):7155.

Junyuan He, Yin Kan, and Cheng Xue. 2024. Training quantum self-attention model in near-term quantum computer. In *2024 16th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 139–144. IEEE.

Xiaokai Hou, Yingli Yang, and Xiaoting Wang. 2022. Realization of long short-term memory networks on quantum circuits. In *2022 13th Asian Control Conference (ASCC)*, pages 2360–2366. IEEE.

Yu-Chao Hsu, Tai-Yu Li, and Kuan-Cheng Chen. 2025. Quantum kernel-based long short-term memory. In *2025 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)*, pages 1–5. IEEE.

Basra Jehangir, Saravanan Radhakrishnan, and Rahul Agarwal. 2023. A survey on named entity recognition—datasets, tools, and methodologies. *Natural Language Processing Journal*, 3:100017.

Fei Jiang, Yong Jiang, Hui Zhi, Yi Dong, Hao Li, Sufeng Ma, Yilong Wang, Qiang Dong, Haipeng Shen, and Yongjun Wang. 2017. Artificial intelligence in healthcare: past, present and future. *Stroke and vascular neurology*, 2(4).

Michael I Jordan and Tom M Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.

Aravind J. Joshi. 1991. Natural language processing. *Science*, 253(5025):1242–1249.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

Subbu Kannan, Vairaprakash Gurusamy, S Vijayarani, J Ilamathi, Ms Nithya, S Kannan, and V Gurusamy. 2014. Preprocessing techniques for text mining. *International Journal of Computer Science & Communication Networks*, 5(1):7–16.

Amin Karamlou, Marcel Pfaffhauser, and James Wootton. 2022. Quantum natural language generation on near-term devices. *arXiv preprint arXiv:2211.00727*.

Pragya Katyayan and Nisheeth Joshi. 2022. Supervised question classification on selqa dataset using variational quantum classifiers. In *International Conference on Innovative Computing and Communications: Proceedings of ICICC 2022, Volume 3*, pages 695–706. Springer.

Pragya Katyayan and Nisheeth Joshi. 2023. Implications of deep circuits in improving quality of quantum question answering. In *Quantum Computing: A Shift From Bits To Qubits*, pages 457–479. Springer.

Staffs Keele et al. 2007. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, ver. 2.3 ebse technical report. ebse.

Abdullah Ayub Khan, Asif Ali Laghari, and Shafique Ahmed Awan. 2021. Machine learning in computer vision: A review. *EAI Endorsed Transactions on Scalable Information Systems*, 8(32).

Debarshi Kundu, Archisman Ghosh, Srinivasan Ekambaram, Jian Wang, Nikolay Dokholyan, and Swaroop Ghosh. 2024. Application of quantum tensor networks for protein classification. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, pages 132–137.

Tuomas Laakkonen, Konstantinos Meichanetzidis, and Bob Coecke. 2024. Quantum algorithms for compositional text processing. *arXiv preprint arXiv:2408.06061*.

Wei Lai, Jinjing Shi, and Yan Chang. 2023. Quantum-inspired fully complex-valued neutral network for sentiment analysis. *Axioms*, 12(3):308.

Pedro Larranaga, Borja Calvo, Roberto Santana, Concha Bielza, Josu Galdiano, Inaki Inza, José A Lozano, Rubén Armananzas, Guzmán Santafé, Aritz Pérez, et al. 2006. Machine learning in bioinformatics. *Briefings in bioinformatics*, 7(1):86–112.

Kusum Lata, Pardeep Singh, and Kamlesh Dutta. 2021. A comprehensive review on feature set used for anaphora resolution. *Artificial Intelligence Review*, 54:2917–3006.

Sweta P Lende and Mukesh M Raghuwanshi. 2016a. Question answering system on education acts using nlp techniques. In *2016 world conference on futuristic trends in research and innovation for social welfare (Startup Conclave)*, pages 1–6. IEEE.

Sweta P Lende and Mukesh M Raghuwanshi. 2016b. Question answering system on education acts using nlp techniques. In *2016 world conference on futuristic trends in research and innovation for social welfare (Startup Conclave)*, pages 1–6. IEEE.

Guangxi Li, Xuanqiang Zhao, and Xin Wang. 2024. Quantum self-attention neural networks for text classification. *Science China Information Sciences*, 67(4):142501.

Qiuchi Li, Sagar Uprety, Benyou Wang, and Dawei Song. 2018. Quantum-inspired complex word embedding. In *Proceedings of the Third Workshop on Representation Learning for NLP*, pages 50–57, Melbourne, Australia. Association for Computational Linguistics.

Yanan Li, Zhimin Wang, Rongbing Han, Shangshang Shi, Jiaxin Li, Ruimin Shang, Haiyong Zheng, Guoqiang Zhong, and Yongjian Gu. 2023. Quantum recurrent neural networks for sequential learning. *Neural Networks*, 166:148–161.

YaoChong Li, Yi Qu, Ri-Gui Zhou, and Jing Zhang. 2025. Qmlsc: A quantum multimodal learning model for sentiment classification. *Information Fusion*, 120:103049.

Alessandro Liberati, Douglas G Altman, Jennifer Tetzlaff, Cynthia Mulrow, Peter C Gøtzsche, John PA Ioannidis, Mike Clarke, Philip J Devereaux, Jos Kleijnen, and David Moher. 2009. The prisma statement for reporting systematic reviews and meta-analyses of studies that evaluate healthcare interventions: explanation and elaboration. *Bmj*, 339.

Tingyu Liu, Yingying Wei, and Jingtao Wang. 2024. Research on distributional compositional categorical model in both classical and quantum natural language processing. In *2024 IEEE/ACIS 27th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 66–71. IEEE.

Kin Ian Lo, Mehrnoosh Sadrzadeh, and Shane Mansfield. 2022. A model of anaphoric ambiguities using sheaf theoretic quantum-like contextuality and bert. *arXiv preprint arXiv:2208.05720*.

Robin Lorenz, Anna Pearson, Konstantinos Meichanetzidis, Dimitri Kartsaklis, and Bob Coecke. 2023. Qnlp in practice: Running compositional models of meaning on a quantum computer. *Journal of Artificial Intelligence Research*, 76:1305–1342.

Maximilian Balthasar Mansky, Franziska Wörle, Jonas Korbinian Stein, Robert Müller, and Claudia Linnhoff-Popien. 2023. Adapting the discocat-model for question answering in the chinese language. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 591–600. IEEE.

Yoshihiro Maruyama. 2021. The categorical integration of symbolic and statistical ai: quantum nlp and applications to cognitive and machine bias problems. In *Intelligent Systems Design and Applications: 19th International Conference on Intelligent Systems Design and Applications (ISDA 2019) held December 3-5, 2019 19*, pages 466–476. Springer.

Abu Kaisar Mohammad Masum, Anshul Maurya, Dhruthi Sridhar Murthy, Naveed Mahmud, et al. 2023. Hybrid quantum-classical machine learning for sentiment analysis. In *2023 International Conference on Machine Learning and Applications (ICMLA)*, pages 1006–1011. IEEE.

John McCarthy et al. 2007. What is artificial intelligence.

Walaa Medhat, Ahmed Hassan, and Hoda Korashy. 2014. Sentiment analysis algorithms and applications: A survey. *Ain Shams engineering journal*, 5(4):1093–1113.

Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni De Felice, Nicolo Chiappori, Alexis Toumi, and Bob Coecke. 2020. Quantum natural language processing on near-term quantum computers. *arXiv preprint arXiv:2005.04147*.

Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. 2023. Grammar-aware sentence classification on quantum computers. *Quantum Machine Intelligence*, 5(1):10.

Maha A Metawei, Mohamed Taher, Hesham ElDeeb, and Salwa M Nassar. 2023. A topic-aware classifier based on a hybrid quantum-classical model. *Neural Computing and Applications*, 35(25):18803–18812.

Eduardo Reck Miranda, Richie Yeung, Anna Pearson, Konstantinos Meichanetzidis, and Bob Coecke. 2022. A quantum natural language processing approach to musical intelligence. In *Quantum Computer Music: Foundations, Methods and Advanced Concepts*, pages 313–356. Springer.

N Moratanch and S Chitrakala. 2016. A survey on abstractive text summarization. In *2016 International Conference on Circuit, power and computing technologies (ICCPCT)*, pages 1–7. IEEE.

N Moratanch and S Chitrakala. 2017. A survey on extractive text summarization. In *2017 international conference on computer, communication and signal processing (ICCCSP)*, pages 1–6. IEEE.

Marco Mordacci, Davide Ferrari, and Michele Amoretti. 2024. Multi-class quantum convolutional neural networks. In *Proceedings of the 2024 Workshop on Quantum Search and Information Retrieval*, pages 9–16.

Ravi Narayan, S. Chakraverty, and V.P. Singh. 2016. Quantum neural network based machine translator for english to hindi. *Applied Soft Computing*, 38:1060–1075.

Phuong-Nam Nguyen. 2024. Quantum word embedding for machine learning. *Physica Scripta*, 99(8):086004.

Ahmed Omar and Tarek Abd El-Hafeez. 2023. Quantum computing and machine learning for arabic language sentiment classification in social media. *Scientific Reports*, 13(1):17305.

Nour El Houda Ouamane and Hacene Belhadef. 2023. Proposed model for qcnn-based sentimental short sentences classification. In *International Conference of Reliable Information and Communication Technology*, pages 214–223. Springer.

Partha Pakray, Goutam Majumder, and Amarnath Pathak. 2018. An hmm based pos tagger for pos tagging of code-mixed indian social media text. In *Social Transformation–Digital Way: 52nd Annual Convention of the Computer Society of India, CSI 2017, Kolkata, India, January 19-21, 2018, Revised Selected Papers 52*, pages 495–504. Springer.

Leela Krishna Kumar Pallapothu, Veda Manohara Sunanda Vulavalapudi, Poorna Chand Evuru, Pramoda Medisetty, Kolla Bhanu Prakash, and Gandharba Swain. 2023. Semantic analysis of auto-generated sentences using quantum natural language processing. In *2023 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE)*, pages 623–628. IEEE.

Gundala Pallavi and Rangarajan Prasanna Kumar. 2025. Quantum natural language processing and its applications in bioinformatics: a comprehensive review of methodologies, concepts, and future directions. *Frontiers in Computer Science*, 7:1464122.

Shyambabu Pandey, Nihar Jyoti Basisth, Tushar Sachan, Neha Kumari, and Partha Pakray. 2023a. Quantum machine learning for natural language processing application. *Physica A: Statistical Mechanics and its Applications*, 627:129123.

Shyambabu Pandey, Nihar Jyoti Basisth, Tushar Sachan, Neha Kumari, and Partha Pakray. 2023b. Quantum machine learning for natural language processing application. *Physica A: Statistical Mechanics and its Applications*, 627:129123.

Shyambabu Pandey, Pankaj Dadure, Morrel VL Nunsanga, and Partha Pakray. 2022. Parts of speech tagging towards classical to quantum computing. In *2022 IEEE Silchar Subsection Conference (SILCON)*, pages 1–6. IEEE.

Shyambabu Pandey and Partha Pakray. 2023. Bi-quantum long short-term memory for part-of-speech tagging. In *Proceedings of the 20th International Conference on Natural Language Processing (ICON)*, pages 301–307, Goa University, Goa, India. NLP Association of India (NLPAI).

Shyambabu Pandey, Partha Pakray, and Riyanka Manna. 2024. Quantum classifier for natural language processing applications. *Computación y Sistemas*, 28(2):695–700.

Liron Pantanowitz, Douglas Hartman, Yan Qi, Eun Yoon Cho, Beomseok Suh, Kyunghyun Paeng, Rajiv Dhir, Pamela Michelow, Scott Hazelhurst, Sang Yong Song, et al. 2020. Accuracy and efficiency of an artificial intelligence tool when counting breast mitoses. *Diagnostic pathology*, 15(1):80.

Esteban Payares, Edwin Puertas, and Juan C Martinez-Santos. 2023. Quantum n-gram language models for tweet classification. In *2023 IEEE 5th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 69–74. IEEE.

David Peral-García, Juan Cruz-Benito, and Francisco José García-Peñalvo. 2023a. Showcasing sentiment classification and word prediction in the quantum natural processing area. In *International Conference on Information Society and University Studies*.

David Peral-García, Juan Cruz-Benito, and Francisco José García-Peñalvo. 2023b. Using quantum natural language processing for sentiment classification and next-word prediction in sentences without fixed syntactic structure. In *International Conference on Information and Software Technologies*, pages 235–243. Springer.

David Peral-García, Juan Cruz-Benito, and Francisco José García-Peñalvo. 2024a. Comparing natural language processing and quantum natural processing approaches in text classification tasks. *Expert Systems with Applications*, 254:124427.

David Peral-García, Juan Cruz-Benito, and Francisco José García-Peñalvo. 2024b. Systematic literature review: Quantum machine learning and its applications. *Computer Science Review*, 51:100619.

Arpan Phukan, Anas Anwarul Haq Khan, and Asif Ekbal. 2024. Qumin: quantum multi-modal data fusion for humor detection. *Multimedia Tools and Applications*, pages 1–18.

Martin J Pickering and Steven Frisson. 2001. Processing ambiguous verbs: evidence from eye movements. *Journal of experimental psychology: Learning, memory, and cognition*, 27(2):556.

Priyanka Prajapati, Vishal Goyal, and Kawaljit Kaur. 2024. A detailed study on anaphora resolution system for asian languages. *SN Computer Science*, 5(7):811.

Wenbo Qiao, Peng Zhang, Jiaming Zhao, and Chang Yang. 2024. Quantum topic model: Topic modeling using variational quantum circuits. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5895–5899. IEEE.

Zhiguo Qu, Yunyi Meng, Ghulam Muhammad, and Prayag Tiwari. 2024. Qmfnd: A quantum multimodal fusion-based fake news detection model for social media. *Information Fusion*, 104:102172.

Indra Kharisma Raharjana, Daniel Siahaan, and Chastine Fatichah. 2021. User stories and natural language processing: A systematic literature review. *IEEE access*, 9:53811–53826.

Altaf Rahman and Vincent Ng. 2012. Resolving complex cases of definite pronouns: The Winograd schema challenge. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 777–789, Jeju Island, Korea. Association for Computational Linguistics.

KMM Rajashekharaiah, Satyadhyan Chickerur, Goutam Hegde, Subrahmanya L Bhat, and Shubham Annappa Sali. 2022. Sentence classification using quantum natural language processing and comparison of optimization methods. In *International Advanced Computing Conference*, pages 85–98. Springer.

A Abdul Rasheed. 2023. Improving prediction efficiency by revolutionary machine learning models. *Materials today: proceedings*, 81:577–583.

Keith Rayner and Susan A Duffy. 1986. Lexical complexity and fixation times in reading: Effects of word frequency, verb complexity, and lexical ambiguity. *Memory & cognition*, 14(3):191–201.

Fariska Z Ruskanda, Muhammad Rifat Abiwardani, Muhammad Akram Al Bari, Kinantan Arya Bagaspati, Rahmat Mulyawan, Infall Syafalni, and Harashta Tatimma Larasati. 2022. Quantum representation for sentiment classification. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 67–78. IEEE.

Fariska Zakhralativa Ruskanda, Muhammad Rifat Abiwardani, Rahmat Mulyawan, Infall Syafalni, and Harashta Tatimma Larasati. 2023a. Quantum-enhanced support vector machine for sentiment classification. *IEEE Access*, 11:87520–87532.

Fariska Zakhralativa Ruskanda, Muhammad Rifat Abiwardani, Infall Syafalni, Harashta Tatimma Larasati, and Rahmat Mulyawan. 2023b. Simple sentiment analysis ansatz for sentiment classification in quantum natural language processing. *IEEE Access*, 11:120612–120627.

Fariska Zakhralativa Ruskanda, Michael Jonathan Halim, Farizki Kurniawan, Infall Syafalni, Rahmat Mulyawan, and Akio Higo. 2024. Enhancing quantum nlp robustness-analysis on noisy models for quantum sentiment classification. In *2024 11th International Conference on Advanced Informatics: Concept, Theory and Application (ICAICTA)*, pages 1–6. IEEE.

Jan Sawicki, Maria Ganzha, and Marcin Paprzycki. 2023. The state of the art of natural language processing—a systematic automated review of nlp literature using nlp techniques. *Data Intelligence*, 5(3):707–749.

Jinjing Shi, Tian Chen, Wei Lai, Shichao Zhang, and Xuelong Li. 2024. Pretrained quantum-inspired deep neural network for natural language processing. *IEEE Transactions on Cybernetics*.

Daniel Silver, Aditya Ranjan, Rakesh Achutha, Tirthak Patel, and Devesh Tiwari. 2024. Lexiql: Quantum natural language processing on nisq-era machines.

In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

Jaiteg Singh, Kamalpreet Singh Bhangu, Abdulrhman Alkhanifer, Ahmad Ali Alzubi, and Farman Ali. 2025. Quantum neural networks for multimodal sentiment, emotion, and sarcasm analysis. *Alexandria Engineering Journal*, 124:170–187.

Kirill Smelyakov, Danil Karachevtsev, Denis Kulemza, Yehor Samoilenko, Oleh Patlan, and Anastasiya Chupryna. 2020. Effectiveness of preprocessing algorithms for natural language processing applications. In *2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T)*, pages 187–191. IEEE.

Zhihui Song, Xin Zhou, Jinchen Xu, Xiaodong Ding, and Zheng Shan. 2024. Recurrent quantum embedding neural network and its application in vulnerability detection. *Scientific Reports*, 14(1):13642.

Jonas Stein, Ivo Christ, Nicolas Kraus, Maximilian Balthasar Mansky, Robert Müller, and Claudia Linnhoff-Popien. 2023. Applying qnlp to sentiment analysis in finance. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 2, pages 20–25. IEEE.

Jayaprakash Sunkavalli, B Madhav Rao, M Trinath Basu, Harish Dutt Sharma, P Ashok Kumar, and Ketan Anand. 2024. Experimentation analysis of vqc and qsvm on sentence classification in quantum paradigm. In *2024 International Conference on Computing, Sciences and Communications (ICCSC)*, pages 1–5. IEEE.

Ayisha Tabassum and Rajendra R Patil. 2020. A survey on text pre-processing & feature extraction techniques in natural language processing. *International Research Journal of Engineering and Technology (IRJET)*, 7(06):4864–4867.

Michael K Tanenhaus, James M Leiman, and Mark S Seidenberg. 1979. Evidence for multiple stages in the processing of ambiguous words in syntactic contexts. *Journal of verbal learning and verbal behavior*, 18(4):427–440.

Srirajarajeswari Thota and Sandeep Kumar Dash. 2025. A comparison of traditional natural language processing methods for emotion recognition using quantum computing. In *2025 International Conference on Machine Learning and Autonomous Systems (ICMLAS)*, pages 669–673. IEEE.

Valter Uotila. 2024. Quantum natural language processing application for estimating sql query metrics. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 2, pages 392–393. IEEE.

Paramita Basak Upama, Anushka Kolli, Hansika Kolli, Subarna Alam, Mohammad Syam, Hossain Shahriar, and Sheikh Iqbal Ahamed. 2023. Quantum machine learning in disease detection and prediction: a survey of applications and future possibilities. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1545–1551. IEEE.

Senthilkumar Vijayakumar and Filious Louis. 2023. Revolutionizing staffing and recruiting with contextual knowledge graphs and qnlp: An end-to-end quantum training paradigm. In *2023 IEEE International Conference on Knowledge Graph (ICKG)*, pages 45–51. IEEE.

R Vinay et al. 2023. Quantum video classification leveraging textual video representations. In *2023 4th International Conference on Communication, Computing and Industry 6.0 (C216)*, pages 01–06. IEEE.

Daphne Wang, Mehrnoosh Sadrzadeh, Samson Abramsky, and Victor H Cervantes. 2021. On the quantum-like contextuality of ambiguous phrases. *arXiv preprint arXiv:2107.14589*.

Haifeng Wang, Hua Wu, Zhongjun He, Liang Huang, and Kenneth Ward Church. 2022. Progress in machine translation. *Engineering*, 18:143–153.

Hadi Wazni, Kin Ian Lo, Lachlan McPheat, and Mehrnoosh Sadrzadeh. 2024. Large scale structure-aware pronoun resolution using quantum natural language processing. *Quantum Machine Intelligence*, 6(2):60.

Hadi Wazni and Mehrnoosh Sadrzadeh. 2023. Towards transparency in coreference resolution: A quantum-inspired approach. *arXiv preprint arXiv:2312.00688*.

Dominic Widdows, Willie Aboumrad, Dohun Kim, Sayonee Ray, and Jonathan Mei. 2024a. Quantum natural language processing. *KI-Künstliche Intelligenz*, pages 1–18.

Dominic Widdows, Aaranya Alexander, Daiwei Zhu, Chase Zimmerman, and Arunava Majumder. 2024b. Near-term advances in quantum natural language processing. *Annals of Mathematics and Artificial Intelligence*, 92(5):1249–1272. Type: Article.

Adhika Pramita Widyassari, Supriadi Rustad, Guruh Fajar Shidik, Edi Noersasongko, Abdul Syukur, Affandy Affandy, and De Rosal Ignatius Moses Setiadi. 2022. Review of automatic text summarization techniques & methods. *Journal of King Saud University-Computer and Information Sciences*, 34(4):1029–1046.

Huaiguang Wu, Delong Kong, Lijie Wang, Daiyi Li, Jiahui Zhang, and Yucan Han. 2025. Multimodal sentiment analysis method based on image-text quantum transformer. *Neurocomputing*, 637:130107.

Wenduan Xu, Stephen Clark, Douglas Brown, Gabriel Matos, and Konstantinos Meichanetzidis. 2024. Quantum recurrent architectures for text classification. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18020–18027.

Chao-Han Huck Yang, Jun Qi, Samuel Yen-Chi Chen, Yu Tsao, and Pin-Yu Chen. 2022. When bert meets quantum temporal convolution learning for text classification in heterogeneous computing. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8602–8606. IEEE.

Xi Yang, Jia Zhu, and Pasquale De Meo. 2025. A quantum-like zero-shot approach for sentiment analysis in finance. *Journal of Intelligent Information Systems*, 63(3):705–721.

Ben Yao, Prayag Tiwari, and Qiuchi Li. 2025. Self-supervised pre-trained neural network for quantum natural language processing. *Neural Networks*, 184:107004.

Wenbin Yu, Lei Yin, Chengjun Zhang, Yadang Chen, and Alex X Liu. 2024. Application of quantum recurrent neural network in low resource language text classification. *IEEE Transactions on Quantum Engineering*.

Chi Zhang, Akriti Kumari, and Damir Cavar. 2024a. Entangled meanings: Classification and ambiguity resolution in qnlp. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, pages 97–102.

Hui Zhang, Qinglin Zhao, and Chuangtao Chen. 2024b. A light-weight quantum self-attention model for classical data classification. *Applied Intelligence*, 54(4):3077–3091.

Jin Zheng, Qing Gao, and Zibo Miao. 2023. Design of a quantum self-attention neural network on quantum circuits. In *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1058–1063. IEEE.

Xin Zhou, Jianmin Pang, Feng Yue, Fudong Liu, Jiayu Guo, Wenfu Liu, Zhihui Song, Guoqiang Shu, Bing Xia, and Zheng Shan. 2022. A new method of software vulnerability detection based on a quantum neural network. *Scientific Reports*, 12(1):8053.

Jia Zhu, Xiaodong Ma, Zhihao Lin, and Pasquale De Meo. 2023. A quantum-like approach for text generation from knowledge graphs. *CAAI Transactions on Intelligence Technology*, 8(4):1455–1463.

Xianchao Zhu, Yashuang Mu, Xuetao Wang, and William Zhu. 2024. Efficient relation extraction via quantum reinforcement learning. *Complex & Intelligent Systems*, 10(3):4009–4018.

A First Appendix

Table 2: Characteristics of Reviewed Literature (Part 1)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
1	2025	A Comparison of Traditional Natural Language Processing Methods for Emotion Recognition Using Quantum Computing	Quantum machine learning	Emotion Recognition, Sentiment Analysis	TensorFlow Quantum	Football Tweets 2022 from Kaggle	(Thota and Dash, 2025)
2	2025	Self-supervised pre-trained neural network for quantum natural language processing	Quantum Compatible BERT (QCBERT) model	Text Classification	N/A	BOOKCORPUS, English WIKIPEDIA (pre-training), GLUE benchmark, IAC-V1, IAC-V2, Twitter hate speech detection (fine-tuning)	(Yao et al., 2025)
3	2025	A quantum-like zero-shot approach for sentiment analysis in finance	Quantum text representation/embedding	Sentiment Analysis	PyTorch 2.0.1, Huggingface Transformers library	dolly-v2-12b (Databricks), h2ogpt-oasst1-512-12b (H2O.ai), ~5000 phrases/sentences from financial news texts and company press releases	(Yang et al., 2025)
4	2025	Multimodal sentiment analysis method based on image-text quantum transformer	Image and Text Quantum Transformer (ITQT-MSA)	Sentiment Analysis	Flax, TensorCircuit	MVSA, HFM	(Wu et al., 2025)
5	2025	Quantum neural networks for multimodal sentiment, emotion, and sarcasm analysis	Quantum Neural Networks, Variational Quantum Eigensolver (VQE)	Sentiment, Emotional, Sarcasm Analysis	librosa, OpenCV library (cv2)	MUStARDxt, Memotion, CMU-MOSEI, MELD	(Singh et al., 2025)
6	2025	QMLSC: A quantum multimodal learning model for sentiment classification	Quantum multimodal learning for sentiment classification (QMLSC)	Sentiment Analysis	PennyLane, PyTorch	IEMOCAP, MELD	(Li et al., 2025)
7	2025	Quantum Kernel-Based Long Short-term Memory	Quantum Kernel-Based Long Short-Term Memory (QK-LSTM)	Part of Speech-Tagging (PoS)	PennyLane, PyTorch	Manually annotated sentence samples	(Hsu et al., 2025)
8	2025	Sequence processing with quantum-inspired tensor networks	Quantum-inspired Tensor Networks	Binary classification	tensornetwork, DisCoPy, JAX, lambeq, Quantinuum's TKET, Bobcat (CCG parser)	Short news titles (Clickbait), Longer movie reviews (Rotten Tomatoes), DNA sequences (Bioinformatics)	(Harvey et al., 2025)

Table 2: Characteristics of Reviewed Literature (Part 2)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
9	2025	Applying Variational Quantum Classifier on Acceptability Judgements: A QNLP Experiment	Variational Quantum Classifier (VQC)	Acceptability Judgment	PennyLane	ItaCoLa	(Guarasci et al., 2023)
10	2025	QSIM: A Quantum-inspired hierarchical semantic interaction model for text classification	Quantum-inspired hierarchical Semantic Interaction Model (QSIM)	Text classification	PyTorch	Reuters, SST-5, Amazon, 20News, AGNews, IMDB, CNews, Yelp-2, Yelp-5, MPQA, TREC, CR, SST-2, MR, and SUBJ	(Gao et al., 2025)
11	2025	Classical Data in Quantum Machine Learning Algorithms: Amplitude Encoding and the Relation Between Entropy and Linguistic Ambiguity	Variational Quantum Circuits (VQC) with Amplitude Encoding and Density Matrices	Linguistic Ambiguity,	NumPy, PennyLane, Tket	130 sentences (food/IT categories), second dataset extension of first with non-ambiguous subjects	(Eisinger et al., 2025)
12	2025	Multi-scale feature fusion quantum depthwise Convolutional Neural Networks for text classification	MSFF-QDConv (QNN)	Text Classification	VQnet	MC (Meaning Classification), RP (RELPRON)	(Chen and Fang, 2024)
13	2025	Quantum mixed-state self-attention network	Quantum Mixed-State Self-Attention Network (QMSAN)	Text Classification	Tensorcircuit, Tensorflow	MC (Meaning Classification), RP (RELPRON), Sentiment Labeled Sentences (Yelp, IMDb, and Amazon reviews)	(Chen et al., 2025)
14	2025	Word and Text Similarity Using Classical Word Embeddings in Quantum NLP Systems	Amplitude Encoding (Quantum Embeddings)	Text Classification	IBM Qiskit	100 such randomly selected words, hand-crafted a word list of 100 words that belong to different semantic topics or fields	(Cavar and Parukola, 2025)

Table 2: Characteristics of Reviewed Literature (Part 3)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
15	2025	Multilingual multi-task quantum transfer learning	Quantum Transfer Learning with a Variational Quantum Classifier (VQC)	Sentence Acceptability Judgments (focusing on syntax), Sentiment Analysis (focusing on semantics)	PyTorch, PennyLane	CoLa, ItaCoLa, SentiPolc, SST-2	(Buonaiuto et al., 2025)
16	2025	Hybrid quantum neural networks show strongly reduced need for free parameters in entity matching	Hybrid Quantum Neural Network (HQNN)	Entity Matching	SpaCy, PyTorch	generated synthetic 2,000 training pairs and 492 testing pairs, resulting in a train/test split of approximately 80%/20% with 75% of the pairs being labeled as non-matching and 25% as matching	(Bischof et al., 2025)
17	2025	Quantum Linguistics in Patient-Clinical Trial Relations for Improved Healthcare	Quantum Convolutional Neural Networks (QCNN), Variational Quantum Circuits (VQC)	Patient-Clinical Trial Matching	N/A	few thousand patient profiles and descriptions of clinical trials (quantitative analysis), smaller sample of healthcare workers say 20 to 30 (qualitative)	(Arora et al., 2025)
18	2024	Efficient relation extraction via quantum reinforcement learning	Quantum Hierarchical Reinforcement Learning for Relation Extraction (QHRL-RE)	Entity Mention Detection & Relation Extraction	N/A	NTY10, NTY11	(Zhu et al., 2024)
19	2024	A light-weight quantum self-attention model for classical data classification	Quantum Self-Attention Model	Text Classification & Sentiment Analysis	N/A	Reviews from Yelp, IMDb, Amazon. CV dataset on the MNIST and Fashion MNIST	(Zhang et al., 2024b)
20	2024	Entangled Meanings: Classification and Ambiguity Resolution in QNLP	Amplitude Encoding and Divide-and-Conquer Encoding using Quantum Machine Learning	Text Classification & Ambiguity Resolution	scikit-learn, PennyLane, Qiskit, umap-learn, spaCy, gensim	lambeq, Amazon review (for classification), 50 nouns and 18 transitive verbs in English (for ambiguity)	(Zhang et al., 2024a)

Table 2: Characteristics of Reviewed Literature (Part 4)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
21	2024	Application of Quantum Recurrent Neural Network in Low-Resource Language Text Classification	Batch Uploading Quantum Recurrent Neural Network (BUQRNN), Parameter Nonshared Batch Upload Quantum Recurrent Neural Network (PN-BUQRNN)	Text Sentiment Analysis	PennyLane	BOOK-Reviews, YouTube-B	(Yu et al., 2024)
22	2024	Quantum Recurrent Architectures for Text Classification	Quantum RNNs with cells based on Parametrised Quantum Circuits (PQCs)	Text Classification	TorchQuantum	Rotten Tomatoes	(Xu et al., 2024)
23	2024	Near-term advances in quantum natural language processing	Quantum Word-Based Topic Classification, Quantum Support Vector Machine (QSVM), Quantum Circuit Born Machine (QCBM), Quantum Circuits for Ambiguity Resolution	Ambiguity Resolution, Bigram Modeling, Topic Classification, Sentiment Analysis	lambeq, Qiskit	lambeq, IMDb movie reviews, University of South Florida Free Association Norms	(Widdows et al., 2024b)
24	2024	Quantum Natural Language Processing	Quantum Positional String (QPOSTR) utilizing a "position" register and a "alphabet" register	Text Encoding	lambeq	7 sentences using a vocabulary of 11 unique words	(Widdows et al., 2024a)
25	2024	Large scale structure-aware pronoun resolution using quantum natural language processing	Variational Quantum Circuit (DisCoCat-VQC)	Pronoun Resolution	lambeq, SpaCy, Qiskit, (NumPyModel, TketModel, AerBackend (lambeq))	16,400 sentence pairs (larger dataset) and 144 sentence pairs (smaller dataset)	(Wazni et al., 2024)
26	2024	Quantum Natural Language Processing Application for Estimating SQL Query Metrics	Quantum Machine Learning (QML) utilizing Parameterized Quantum Circuits (PQCs)	Estimating SQL Query Metrics	N/A	Join Order Benchmark, IMDB database	(Uotila, 2024)

Table 2: Characteristics of Reviewed Literature (Part 5)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
27	2024	Advancing Textual Semantic Understanding with Quantum LSTM Networks	Quantum Long Short-Term Memory (QLSTM) Network	Textual Semantic Understanding, Sentiment Analysis and Subjective Content Classification	N/A	UCI News Aggregator (UCI), Subjectivity (SUBJ)	(Gondel et al., 2024)
28	2024	Experimentation Analysis of VQC and QSVM on Sentence Classification in Quantum Paradigm	Quantum SVM and (QSVM) Variational Quantum Classifier (VQC)	Sentence Classification	Qiskit, lambeq toolkit	Restaurant industry SMS texts (positive/negative remarks), 29-word vocabulary, four to five words long	(Sunkavalli et al., 2024)
29	2024	Recurrent quantum embedding neural network and its application in vulnerability detection	Recurrent Quantum Embedding Neural Network (RQENN)	Vulnerability Detection	lambeq, MindSpore, MindQuantum , Joern	NVD (National Vulnerability Database) & SARD (Software Assurance Reference Dataset) (C/C++ source codes; buffer overflow and resource management vulnerabilities)	(Song et al., 2024)
30	2024	LEXIQL: Quantum Natural Language Processing on NISQ-era Machines	LEXIQL (Incremental Data Injection, Encode-Inject-Process (EIP) Cells, Ensemble of Diverse Learners, Multiple-Axes Projection)	Text Classification	Pennylane, scikit-learn, nltk, PyTorch, lambeq, Qiskit	Yelp, IMDB, Amazon, LambeqMC and LambeqRP	(Silver et al., 2024)
31	2024	Pretrained Quantum-Inspired Deep Neural Network for Natural Language Processing	Quantum-Inspired Pretrained Feature Embedding (QPFE) & QPFE-ERNIE	Sentiment Classification & Word Sense Disambiguation (WSD)	PyTorch, paddle,	CR, MPQA, MR, SST, SUBJ, ChnSentiCorp, SemCor, SemEval-2007 (SE07), SemEval-2013 (SE13), SemEval-2015 (SE15), Senseval-2 (SE2), Senseval-3 (SE3)	(Shi et al., 2024)

Table 2: Characteristics of Reviewed Literature (Part 6)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
32	2024	Enhancing Quantum NLP Robustness - Analysis on Noisy Models for Quantum Sentiment Classification	QNLP Pipeline (DisCoCat & Lambeq's IQP anzatz)	Sentiment Classification	Lambeq	Synthetic 290 sentences consisting 170 sentences (for training), 50 sentences (for development), 60 sentences (for test), each sentence 4-5 words long, based on 29 words in the restaurant domain	(Ruskanda et al., 2024)
33	2024	DeepKet-Quantum Space-Efficient Word Embedding Layer for Steganalysis	DeepKet: Quantum Word Embedding utilizing Variational Quantum Circuits (VQC) using Kets (quantum states)	Steganography & Word Embeddings	PennyLane, PyTorch	Text samples (both unigram and bigram, truncated at 100 words per data point) from the "The Black Swan: The Impact of the Highly Improbable"	(Ahmed and Shridevi, 2024)
34	2024	QMFND: A quantum multimodal fusion-based fake news detection model for social media	Quantum Multimodal Fusion-based Model for Fake News Detection (QMFND) that passes through a Quantum Convolutional Neural Network (QCNN)	Fake News Detection	PennyLane, lambeq, XLNet (Hugging Face Transformers library)	Gossip, Politifact	(Qu et al., 2024)
35	2024	Quantum Topic Model: Topic Modeling Using Variational Quantum Circuits	Quantum Topic Model (QTM) based on Variational Quantum Circuits (VQC)	Topic Modeling	OCTIS, PennyLane	20NewsGroup, CNews10k	(Qiao et al., 2024)
36	2024	QuMIN: quantum multi-modal data fusion for humor detection	Quantum Multi-Modal Data Fusion (QuMIN)	Humor Detection	fastText, openSmile, Qiskit	M2H2	(Phukan et al., 2024)
37	2024	Using Quantum Natural Language Processing for Sentiment Classification and Next-Word Prediction in Sentences Without Fixed Syntactic Structure	QNLP Proof of Concepts (Quantum-inspired Tensor Networks (QTNs) using Parameterized Quantum Circuits (PQCs))	Sentiment Classification & Next-Word Prediction	DisCoPy, DisCoCirc, lambeq	N/A	(Peral-García et al., 2023b)

Table 2: Characteristics of Reviewed Literature (Part 7)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
38	2024	Comparing Natural Language Processing and Quantum Natural Processing approaches in text classification tasks	Quantum Natural Processing Model (QNP) based on Parameterized Quantum Circuits (PQCs)	Text Classification	DisCoPy, Numpy, Jax, lambeq	AD2, AD3, AD4, AD5, MILK, KAGG	(Peral-García et al., 2024a)
39	2024	Quantum Classifier for Natural Language Processing Applications	Quantum Recurrent Neural Network (QRNN) utilizing Parameterized Quantum Circuits (PQCs)	Parts of Speech, Named Entity Recognition, Sentiment Analysis	N/A	N/A	(Pandey et al., 2024)
40	2024	Proposed Model for QCNN-Based Sentimental Short Sentences Classification	Quantum Convolutional Neural Network (QCNN)	Sentiment Analysis	N/A	N/A	(Ouamane and Belhadef, 2023)
41	2024	Quantum word embedding for machine learning	Multi-Dimensional, Finite automaton model for quantum word embedding (QWE) via the Galois field	Information Retrieval	numpy, scikit-learn, PyTorch	RPI369, RPI1807, RPI2241	(Nguyen, 2024)
42	2024	Multi-Class Quantum Convolutional Neural Networks	Quantum Convolutional Neural Network (QCNN)	Multi-Class Classification for Information Retrieval	PennyLane	MNIST	(Mordacci et al., 2024)
43	2024	Research on Distributional Compositional Categorical Model in Both Classical and Quantum Natural Language Processing	Tensor networks and Quantum Circuits based on the DisCoCat model	Text Binary Classification	N/A	synthetic (food or IT) (training set, validation set and test set)	(Liu et al., 2024)
44	2024	Quantum self-attention neural networks for text classification	Quantum Self-Attention Neural Network (QSANN)	Text Classification	PaddlePaddle	MC, RP, Yelp, IMDb, Amazon (reviews)	(Li et al., 2024)
45	2024	Quantum Algorithms for Compositional Text Processing	Quantum DisCoCirc (QDisCoCirc)	Text Similarity, Question Answering	DisCoCirc	N/A	(Laakkonen et al., 2024)

Table 2: Characteristics of Reviewed Literature (Part 8)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
46	2024	Application of Quantum Tensor Networks for Protein Classification	Quantum Tensor Networks (QTN) inspired by Recurrent Neural Networks (RNN) & Convolutional Neural Networks (CNN)	Binary Classification	lambeq, JAX, tensornetwork	Protein Sequences from UniProt	(Kundu et al., 2024)
47	2024	Training Quantum Self-Attention Model in Near-Term Quantum Computer	Self-Attention models on near term devices using Variational Quantum Circuits (VQCs)	Text Classification	VQNet	MC, RP	(He et al., 2024)
48	2024	Quantum-inspired language models based on unitary transformation	Quantum-inspired Language Model based on Unitary Transformation (QLM-UT)	Question Answering, Text Classification	fastText	SMAP, MSL, SMD, TRECQA, WIKIQA, YahooQA, MR, SUBJ, CR, MPQA, SST, TREC, AGNEWS	(Fan et al., 2024)
49	2024	Strongly Entangling Neural Network: Quantum-Classical Hybrid Model for Quantum Natural Language Processing	Strongly Entangling Neural Network, a Quantum & Classical Hybrid Model	Binary Classification	Tensorflow, PennyLane	Phrases from English to Spanish, originally used for Deep Learning models with translation tasks (modified from Many-Things.org)	(Díaz-Ortiz et al., 2023)
50	2024	Quantum Inspired Zeroshot Classification For Adverse Drug Reactions Detection From Social Media Reviews	Quantum Transformer that utilizes Variational Quantum Circuits (VQC)	Text Classification	N/A	Source repository of Twitter and other online discussion forums	(Dey and Shrivastava, 2024)
51	2024	Semantic Similarities using Classical Embeddings in Quantum NLP	Mapping embedding vectors to quantum states utilizing Amplitude Encoding	Semantic Similarity	N/A	4400 randomly picked word pairs (from BERT, Numberbatch, GloVe, fastText), a set of 100 words that belong to different semantic topics or fields	(Cavar and Zhang, 2024)

Table 2: Characteristics of Reviewed Literature (Part 9)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
52	2024	Quantum transfer learning for acceptability judgements	Fine tune classical models utilizing Parametrized Quantum Circuit (VQC) via Amplitude Encoding	Acceptability Judgments	SHAP, PennyLane	ItaCoLa	(Buonaiuto et al., 2024b)
53	2024	Quantum Transfer Learning for Sentiment Analysis: an experiment on an Italian corpus	Fine tune classical models utilizing Parametrized Quantum Circuit (VQC) via Amplitude Encoding	Sentiment Analysis	PennyLane	SENTIPOLC 2016	(Buonaiuto et al., 2024a)
54	2024	Enhancing Contextual Understanding in Knowledge Graphs: Integration of Quantum Natural Language Processing with Neo4j LLM Knowledge Graph	Integration of Quantum Natural Language Processing (QNLP) with Neo4j LLM Knowledge Graphs	Enhancing Contextual Understanding and Nuanced Semantic Relationship Modeling	lambeq's BobcatParser, Qiskit's Aer simulator	N/A	(Bharti et al., 2024)
55	2024	Leveraging NLP and Quantum Computing for Advanced Agricultural Solutions	Smart Agriculture Computing based on Quantum Natural Language Processing (SAC-QNLP)	Agricultural data Processing, decision-making, and Resource Management	N/A	N/A	(Bhargavi et al., 2024)
56	2024	A Quantum Computing based Approach for Sentiment Analysis in Bilateral Conversations	Hybrid (Quantum-Classical) Long Short Term Memory (LSTM) utilizing Variational Quantum Circuits (VQC)	Sentiment Analysis	Tensorflow, PennyLane	meticulously curated manually, drawing from interviews, texts, videos, and TV programs featuring dyadic dialogues	(Bar et al., 2024)
57	2024	Hybrid classical-quantum transfer learning for text classification	Hybrid Classical-Quantum Transfer Learning applies Pre-Trained Classical Models utilizing Variational Quantum Circuits (VQC)	Text Classification	PennyLane, Pandas, scikit-learn	Collection of Short Message Service (SMS) messages, each message has a maximum of 175 characters and contains an alphabet and special characters (5572 total messages)	(Ardeshir-Larijani and Nasiri Fatmehsari, 2024)

Table 2: Characteristics of Reviewed Literature (Part 10)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
58	2024	Quantum Approach for Contextual Search, Retrieval, and Ranking of Classical Information	Quantum Inspired Algorithm that uses Heuristic Bell-like testing utilizing Hyperspace Analog to Language (HAL) & Hilbert Space	Document Search, Retrieval, Ranking	spaCy	Synthetic advertising text documents from travel agencies (eight texts describing services and offers in the field of tourism)	(Alodjants et al., 2024)
59	2023	A quantum-like approach for text generation from knowledge graphs	Fusion Model based on Quantum Theory that Optimizes Text Generation from Knowledge Graphs (KGs) by combining Global and Local Graph Encoders	Text Generation	N/A	AGENDA, WebNLG	(Zhu et al., 2023)
60	2023	Design of a Quantum Self-Attention Neural Network on Quantum Circuits	Quantum Self-Attention Neural Network (QSAN) utilizing Parameterized Quantum Circuits (PQCs)	Text Classification	PennyLane	MC & RP	(Zheng et al., 2023)
61	2023	Towards Transparency in Coreference Resolution: A Quantum-Inspired Approach	Variational Quantum Classifier (VQC)	Pronoun Resolution	DisCoPy, NumPyModel (Lambeq), JAX, SpaCy, Stanza, AllenNLP, HuggingFace	Synthetic Winograd-style ambiguous coreference sentences using GPT-3 from Rahman and Ng (2012) (Rahman and Ng, 2012) (consisting of 16,400 entries)	(Wazni and Sadrzadeh, 2023)
62	2023	Revolutionizing Staffing and Recruiting with Contextual Knowledge Graphs and QNLP: An End-to-End Quantum Training Paradigm	Quantum Natural Language Processing (QNLP) model integrating contextual Knowledge Graph Features into Quantum Circuits (IQPAnsatz)	Contextual Information Retrieval, Recommendation, Generative AI Text Generation, Question Answering, Talent Acquisition, Workforce Planning, Named Entity Matching (NER)	lambeq, qiskit	Job descriptions, job postings, unstructured text data, candidate resumes	(Vijayakumar and Louis, 2023)

Table 2: Characteristics of Reviewed Literature (Part 11)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
63	2023	Applying QNLP to Sentiment Analysis in Finance	DisCoCat (Distributional Compositional Categorical), Quantum-Enhanced Long Short-Term Memory (QLSTM)	Sentiment Analysis in Finance	lambeq	Two datasets (low & moderate generated by ChatGPT) simple sentences (maximum length of five words) discussing financial topics or stocks, labeled as positive, neutral, or negative, more complex (both consisting more than 1000 sentences)	(Stein et al., 2023)
64	2023	Simple Sentiment Analysis Ansatz for Sentiment Classification in Quantum Natural Language Processing	Variational Quantum Algorithms (VQA) Framework	Sentiment Analysis	lambeq, DisCoPy	270 English sentences, with a distribution of 160 training sentences, 50 validation sentences, and 60 testing sentences. The vocabulary is comprised of 29 words: 6 nouns, 8 verbs, and 14 adjectives	(Ruskanda et al., 2023b)
65	2023	Quantum-Enhanced Support Vector Machine for Sentiment Classification	Quantum-Enhanced Support Vector Machine (QE-SVM)	Sentiment Classification	lambeq, NLTK	Generated sentences from 29 vocabularies, consisting of both positive and negative sentences, with each sentence having a length of 4-5 words (restaurant domain) (170 training), (50 development), (60 test)	(Ruskanda et al., 2023a)
66	2023	Sentence Classification Using Quantum Natural Language Processing and Comparison of Optimization Methods	Quantum Natural Language Processing (QNLP) using the DisCoCat model to map sentences using ansatz, optimized with SPSA and COMPASS algorithms	Sentence Classification	lambeq, Pytket, Noisyopt	150 sentences (partitioned into 3 subsets namely training (90), test (35) and development (35) sets), mainly of 4 grammatical structures	(Rajashekharaiah et al., 2022)

Table 2: Characteristics of Reviewed Literature (Part 12)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
67	2023	Quantum Video Classification Leveraging Textual Video Representations	Quantum video classification is performed by converting videos into text captions, then using quantum circuits for classification a quantum simulator, making it a quantum text classification problem	Text Classification	lambeq, JAX, NumPy model (from lambeq)	kinetics-700, WebVid-2M	(Vinay et al., 2023)
68	2023	Showcasing sentiment classification and word prediction in the quantum natural processing area	Quantum Circuits derived from string diagrams (proof of concepts)	Sentiment Classification & Missing Word Prediction	DisCoPy, lambeq	N/A	(Peral-García et al., 2023a)
69	2023	Quantum N-Gram Language Models for Tweet Classification	Quantum Kernel Support Vector Machine (qk-svm), Variational Quantum Circuit or Quantum Neural Net-Work (qnn), Hybrid Quantum Convolutional Neural Network (H-QCNN)	Text Classification	Pennylane, Qiskit, spaCy	SemEval-2018 Affect in Tweets (only tweets written in Spanish)	(Payares et al., 2023)
70	2023	Quantum machine learning for natural language processing application	Quantum-Based Long Short-Term Memory (QLSTM) utilizing Variational Quantum Circuits (VQC)	Parts of Speech Tagging (PoS)	PyTorch library (integrated with Pennylane), PennyLane	ICON 2016 Code-Mixed data on Indian languages	(Pandey et al., 2023a)
71	2023	Quantum computing and machine learning for Arabic language sentiment classification in social media	Quantum Support Vector Machine (QSVM)	Sentiment Classification	Qiskit	213,465 Arabic tweets specifically curated for sentiment analysis focusing on the topic of Asthma, collected from Twitter (Kaggle); 44,000 posts and tweets collected from Facebook and Twitter, covering different Arabic sentiment topics	(Omar and Abd El-Hafeez, 2023)

Table 2: Characteristics of Reviewed Literature (Part 13)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
72	2023	Hybrid Quantum-Classical Machine Learning for Sentiment Analysis	Hybrid Quantum-Classical Machine Learning for Sentiment Analysis, integrating Quantum Support Vector Machines (QSVM) and Variational Quantum Classifiers (VQC) with PCA and Wavelet transform	Sentiment Analysis	Qiskit	Bengali News, Twitter US Airline	(Masum et al., 2023)
73	2023	A topic-aware classifier based on a hybrid quantum-classical model	Quantum Natural Language Processing (QNLP) model that utilizes hybrid quantum-classical Parametrized Quantum Circuits (PQC)	Topic-Aware Classification (determine if two sentences are related to the same topic or not)	lambeq, Qiskit, PyTorch, TKet, PennyLane, JAX, DisCoPy	100 pairs of sentences, belongs to 'food' or 'IT' topics with matched/unmatched labels (Quantinuum QNLP team during the Womanium Quantum Hackathon 2022)	(Metawei et al., 2023)
74	2023	Grammar-aware sentence classification on quantum computers	Quantum Natural Language Processing (QNLP) utilizing the Categorical Distributional Compositional (DisCoCat) model to insatiate sentences with Parametrized Quantum Circuits (PQC)	Binary Classification of Sentences (special case of question answering)	DisCoPy, noisyopt, lambeq, JAX, SciPy, TKet	Corpus K_{30}: 30 sentences sampled from the vocabulary of 7 words, corpus K_{16}: 16 labeled sentences sampled from the vocabulary, corpus K_6: 6 labeled sentences sampled from the vocabulary	(Meichanetzidis et al., 2023)

Table 2: Characteristics of Reviewed Literature (Part 14)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
75	2023	QNLP in Practice: Running Compositional Models of Meaning on a Quantum Computer	Quantum Natural Language Processing (QNLP) models, specifically compositional models of meaning being implemented on Noisy Intermediate-Scale Quantum (NISQ) devices utilizing Variational Quantum Circuits (VQCs) (Proof of Concepts)	Sentence Classification	lambeq, DisCoPy, TKet	(130 sentences) was generated automatically by a simple context-free grammar, with half of the sentences related to food and half related to IT (a binary classification task), RP	(Lorenz et al., 2023)
76	2023	Quantum recurrent neural networks for sequential learning	Quantum Recurrent Neural Network (QRNN)	Meteorological Indicators, Stock Price, and Text Categorization	pyQPanda, TensorFlow	(Meteorological indicators (from China Meteorological Data Service Center)), (Stock price (from finance sina)), MC	(Li et al., 2023)
77	2023	Quantum-Inspired Fully Complex-Valued Neutral Network for Sentiment Analysis	Quantum-Inspired Fully Complex-Valued Neural Network (ComplexQNN)	Sentiment Analysis	PyTorch, AllenNLP, NLTK	CR, MPQA, MR, SST, SST-2, SST-5, SUBJ	(Lai et al., 2023)
78	2023	Semantic Analysis of Auto-generated Sentences using Quantum Natural Language Processing	Quantum circuits used for semantic interpretation of autogenerated sentences using the Qasm simulator and IBM quantum experience	Semantic Interpretation	DisCoPy, TKet, Qiskit	synthetic data comprised of small, medium, and large sentences (for training & testing)	(Pallapothu et al., 2023)
79	2023	Supervised Question Classification on SelQA Dataset Using Variational Quantum Classifiers	Quantum Machine Learning using Variational Quantum Circuits (VQC)	Question Answering	Qiskit	SelQA	(Katyayan and Joshi, 2022)
80	2023	Implications of Deep Circuits in Improving Quality of Quantum Question Answering	Quantum Machine Learning using Variational Quantum Classifiers (VQC) and Quantum Support Vector Machines (QSVM)	Question Answering	Qiskit	SelQA	(Katyayan and Joshi, 2023)

Table 2: Characteristics of Reviewed Literature (Part 15)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
81	2023	Characterizing Quantum Classifier Utility in Natural Language Processing Workflows	Quantum Neural Networks (QNN)	Binary Classification, Multi-Class Classification	PennyLane, PyTorch, scikit-learn	MADELON & ECP-CANDLE P3B3	(Hamilton et al., 2023)
82	2023	QNet: A Quantum-Native Sequence Encoder Architecture	QNet, a quantum-native sequence encoder model. ResQNet is a quantum-classical hybrid model using QNet blocks and residual connections, mimicking a Transformer Encoder	Text Classification, Rating Score Prediction, Named Entity Recognition	TensorFlow, TensorQuantum, Cirq	StackOverflow, ColBert, MSRA, RentTheRunaway	(Day et al., 2023)
83	2023	Ensemble Learning Based Quantum Text Classifiers	Quantum Text Classifier utilizing Ensemble Learning	Text Classification	Qiskit Aer simulator, lambeq	MC, RP	(Bouakba and Belhadef, 2023)
84	2023	Exploring the Capabilities and Limitations of VQC and QSVC for Sentiment Analysis on Real-World and Synthetic Datasets	Quantum Support Vector Classifier (QSVC) & Variational Quantum Classifier (VQC)	Sentiment Analysis	Qiskit Aer Simulator	IMDB movie review, generated synthetic 200 short sentences, each with up to 7 words, that are labeled as positive or negative, with a vocabulary of 60 unique words in various combinations	(Belhadef et al., 2023)
85	2023	Adapting the DisCoCat-Model for Question Answering in the Chinese Language	Categorical Distributional Compositional (DisCoCat) utilizing Parameterized Quantum Circuits (PQC)	Question Answering	DisCoPy	Eight Chinese words, along with English translation and pregroup identifications, generated 50 sentences based of the eight words corresponding with a truth value	(Mansky et al., 2023)
86	2023	Automated Vulnerability Detection in Source Code Using Quantum Natural Language Processing	Quantum Long Short Term Memory (QLSTM) with Variational Quantum Circuits (VQCs)	Vulnerability Detection	PennyLane, Keras, TensorFlow	SATE IV Juliet Test Suite, Debian Linux distribution, public Git repositories on GitHub	(Akter et al., 2022)

Table 2: Characteristics of Reviewed Literature (Part 16)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
87	2022	A new method of software vulnerability detection based on a quantum neural network	Quantum Deep Embedding Neural Network (QDENN)	Software Vulnerability Detection	MindSpore, QuantumFlow, MindQuantum	SARD	(Zhou et al., 2022)
88	2022	When BERT Meets Quantum Temporal Convolution Learning for Text Classification in Heterogeneous Computing	Bidirectional Encoder Representations from Transformers - Quantum Temporal Convolution (BERT-QTC)	Text Classification	PennyLane, TensorFlow Hub	Snips, ATIS	(Yang et al., 2022)
89	2022	Quantum Representation for Sentiment Classification	Quantum Representation exploring four Not-Box quantum circuit representations (X-word, Z-X-word, X-end, Z-X-end) for negative sentences	Sentiment Analysis	lambeq, DisCoPy, NLTK	synthetic simple subjective sentences, 4-5 words. 29-word vocabulary for the restaurant domain (positive and negative), 170 training, 50 development, 60 test	(Ruskanda et al., 2022)
90	2022	Parts of speech tagging towards classical to quantum computing	Quantum-Enhanced Long Short-Term Memory (QLSTM)	Part of-Speech Tagging (PoS)	N/A	Mizo	(Pandey et al., 2022)
91	2022	A Quantum Natural Language Processing Approach to Musical Intelligence	Quanthoven, a quantum classifier that utilizes parameterized quantum circuits (PQC)	Text Classification for Sentiment Analysis in music	DisCoPy, lambeq, JAX, Quantinuum's compiler t|ket⟩	generated synthetic corpus of 100 compositions for piano using a bespoke context-free Grammar, labeling as rhythmic or melodic	(Miranda et al., 2022)
92	2022	A Model of Anaphoric Ambiguities using Sheaf Theoretic Quantum-like Contextuality and BERT	BERT for probability extraction within a sheaf-theoretic framework	Anaphoric Ambiguity	N/A	N/A	(Lo et al., 2022)
93	2022	Quantum Natural Language Generation on Near-Term Devices	Hybrid quantum-classical algorithm, based on the annealing technique for combinatorial optimization and utilizes Parametrized Quantum Circuits (PQC) (proof of concepts)	Natural Language Generation (NLG), Sentence Classification	lambeq	Food vs IT, 105 News headlines generated by a CFG	(Karamlou et al., 2022)

Table 2: Characteristics of Reviewed Literature (Part 17)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
94	2022	Realization of Long Short-Term Memory Networks on Quantum Circuits	Duplication-Free Quantum Long Short-Term Memory Neural Network (DQLSTM)	Sentiment Analysis	Jieba	Chinese E-commerce comments, consisting of 10,000 comments and obtained from Taobao, with positive and negative sentiment labels	(Hou et al., 2022)
95	2022	Quantum Natural Language Processing Based Sentiment Analysis Using Lambeq Toolkit	Quantum native model using variational quantum circuits utilizing the Distributional Compositional Categorical (DisCoCat) framework	Sentiment Analysis	lambeq, PyTorch, Qiskit's Aer simulator, TKet (pytket interface), JAX	Positive and negative sentiments of candidates on reading various book genres such as fiction, nonfiction, comics, and classics (consists of 130 sentences; 70 training, 30 development, and 30 test)	(Ganguly et al., 2022)
96	2022	The Dawn of Quantum Natural Language Processing	Quantum-Enhanced Long Short-Term Memory (QLSTM) network & Quantum Enhanced Transformer	Part-of-Speech (PoS) Tagging	PennyLane, Qulacs	IMDB	(Di Sipio et al., 2022)
97	2022	Quantum computations for disambiguation and question answering	Framework that implements word representations as quantum states and performing tensor contractions on quantum circuits using (DisCoCat) framework	Disambiguation & Question Answering	N/A	N/A	(Correia et al., 2022)
98	2022	Quantum Text Encoding for Classification Tasks	Quantum Support Vector Machine (QSVM) classification method	Text Classification	Qiskit	Lambeq, IMDB	(Alexander and Widdows, 2022)
99	2021	On the Quantum-like Contextuality of Ambiguous Phrases	Sheaf-theoretic framework for quantum contextuality applied to model meaning combinations (proof of concept)	Modeling Contextuality of Ambiguous Phrases	N/A	British National Corpus, UKWaC, psycholinguistics studies (Pickering and Frisson, 2001), (Tanenhaus et al., 1979), (Rayner and Duffy, 1986)	(Wang et al., 2021)

Table 2: Characteristics of Reviewed Literature (Part 18)

No.	Year	Title	Quantum Approach/Methods	NLP Areas/Tasks	Frameworks, Libraries, or SDK	Dataset/Input	Ref
100	2021	Quantum Natural Language Processing on Near-Term Quantum Computers	Pipeline for NLP tasks by compositional translation of lexical structures to parametrized quantum circuits utilizing DisCoCat	Semantic Representation/Compositional Distributional Semantics	DisCoPy, TKet (pytket interface)	N/A	(Meichanetzidis et al., 2020)
101	2021	The Categorical Integration of Symbolic and Statistical AI: Quantum NLP and Applications to Cognitive and Machine Bias Problems	Quantum linguistics, integrating Symbolic and Statistical AI (proof of concept)	Semantic Representation/Compositional Semantics	N/A	N/A	(Maruyama, 2021)
102	2021	DisCoPy: Monoidal Categories in Python	Quantum algorithms using rigid functors to Lambek pregroup grammars to quantum circuits (proof of concept)	Semantic Represntation	DisCoPy, TKet, NumPy, networkx, matplotlib, JAX, PyZX	N/A	(de Felice et al., 2020)
103	2021	Application of Quantum Natural Language Processing for Language Translation	Quantum Long Short-Term Memory (Q-LSTM) utilizing both Parameterized Quantum Circuits (PQC) & Variational Quantum Circuits (VQC)	Machine Translation	N/A	Two synonymous simple sentences in English and Persian	(Abbaszade et al., 2021)