On the Effectiveness of Large Language Models for GitHub Workflows

Xinyu Zhang
Purdue University
West Lafayette, IN, USA
zhan5085@purdue.edu

Siddharth Muralee
Purdue University
West Lafayette, IN, USA
smuralee@purdue.edu

Sourag Cherupattamoolayil
Purdue University
West Lafayette, IN, USA
scherupa@purdue.edu

Aravind Machiry
Purdue University
West Lafayette, IN, USA
amachiry@purdue.edu

ABSTRACT

GitHub workflows or GitHub CI is a popular continuous integration platform that enables developers to automate various software engineering tasks by specifying them as workflows, i.e., `YAML` files with a list of jobs. However, engineering valid workflows is tedious. They are also prone to severe security issues, which can result in supply chain vulnerabilities. Recent advancements in Large Language Models (LLMs) have demonstrated their effectiveness in various software development tasks. However, GitHub workflows differ from regular programs in both structure and semantics. We perform the first comprehensive study to understand the effectiveness of LLMs on five workflow-related tasks with different levels of prompts. We curated a set of ~400K workflows and generated prompts with varying detail. We also fine-tuned LLMs on GitHub workflow tasks. Our evaluation of three state-of-the-art LLMs and their fine-tuned variants revealed various interesting findings on the current effectiveness and drawbacks of LLMs.

CCS CONCEPTS

• **Security and privacy** → *Vulnerability scanners*; • **Software and its engineering** → *Automatic programming*.

KEYWORDS

GitHub Workflow, Large Language Model, Vulnerability Detection

ACM Reference Format:
Xinyu Zhang, Siddharth Muralee, Sourag Cherupattamoolayil, and Aravind Machiry. 2024. On the Effectiveness of Large Language Models for GitHub Workflows. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30–August 02, 2024, Vienna, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3664476.3664497

1 INTRODUCTION

Continuous Integration (CI) or Continuous Integration and Development (CI/CD) systems [17] play a crucial role in modern software development practices, automating the integration and testing of code to ensure its reliability and security. Among the plethora of CI platforms[1][2][3], GitHub workflows or GitHub CI [4] emerges as a front-runner due to its seamless integration within the GitHub ecosystem, the ability to use third-party modules (i.e., Actions), and flexibility in triggering mechanisms. Developers use GitHub workflows by defining a *pipeline or workflow*, which is a `YAML` file that specifies all the details (Listing 1 shows an example).

However, unlike traditional code, workflows contain a unique blend of configuration and programming logic and can incorporate snippets of multiple programming languages. Valenzuela-Toledo *et al.* [49] demonstrated that despite the popularity of GitHub workflows, the process of engineering these workflows lacks tool support, leading to a high incidence of errors during their development. Furthermore, developers are known to use insecure practices, leading to security vulnerabilities unique to workflows. This underscores the complexity of generating workflows and the need for techniques that can produce syntactically valid and secure workflows.

LLMs [26, 32] are igniting a revolution in the heart of the software development realm, automating various software engineering tasks such as coding [40], crafting test cases [51], and enriching code with documentation [28]. Companies are embracing LLMs at an unparalleled pace [48], making artificial intelligence-guided development the standard in the industry. Significant research has been conducted to assess the effectiveness of LLMs for code generation tasks and to delve into the security aspects [13, 35] of the code they produce. The findings from these studies on effectiveness of LLMs in generating code from a given prompt and the strategies to engineer effective prompts have practical implications for software development. They provide insights into how LLMs can be harnessed effectively in real-world scenarios.

GitHub workflows, although similar in their intent, vary in structure, semantics, and format (§ 2.1) from traditional code written using various programming languages. Also, vulnerabilities in GitHub workflows differ from regular code-level vulnerabilities because of the difference in the desired security properties of GitHub workflows [21]. OWASP has even created a new list for the Top 10 CI/CD security risks [33] to raise awareness of CI/CD vulnerabilities. Previous studies [5, 15, 21, 27] have meticulously examined the security

ARES 2024, July 30–August 02, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3664497

[1] https://travis-ci.org/
[2] https://circleci.com/
[3] https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/
[4] https://github.com/features/actions

characteristics of the GitHub CI platform, enumerating potential weaknesses inherent in GitHub workflows.

With the increase in their adoption, it is imperative to understand the effectiveness of LLMs for workflow-related tasks. Prior works have explored the effectiveness of LLMs on software development. However, the difference in the structure, semantics, and security properties of workflows raises concerns regarding the generalizability of observations made by prior works for workflows.

In this paper, we tackle this problem by evaluating the effectiveness of LLMs on workflow-related tasks. Specifically, we intend to evaluate the effectiveness of LLMs for:

- **RQ1: Workflow Generation (§ 4.1).** How effective are LLMs in generating workflows? How secure are these generated workflows?
- **RQ2: Defect Detection in workflows (§ 4.2).** How effective are LLMs in finding different classes of defects in workflows?
- **RQ3: Defect Repair in workflows (§ 4.3).** How effective are LLMs in repairing defective workflows?

We selected (our selection criteria in § 3.1) three state-of-the-art LLMs as our subjects, i.e., GPT-3.5 [31], CodeLlama [40], and StarChat [47]. We curated a large (~400K) workflow dataset by enhancing an existing dataset (provided by ARGUS [27]) with various prompts and syntactic defects. We created fine-tuned variants of all our LLMs by using a small subset of our dataset. We evaluated both off-the-shelf and fine-tuned variants for each LLM along different modes (i.e., 0-shot, 1-shot, etc). We organized each research question into various tasks (details in Table 3). For each task, we designed prompts with varying levels of detail and contextual information. For a given LLM variant (i.e., off-the-shelf or fine-tuned) and a task, we first perform calibration on a small subset of data, i.e., to identify the temperature value and prompt that performs the best. Second, we perform the final evaluation using the best-performing configuration on the large dataset.

Our study revealed various interesting findings, such as, unlike regular code generation tasks, LLMs requires detailed prompts to generate desired workflows. However, LLMs have a high likelihood of producing invalid (i.e., with syntactic errors) workflows with detailed prompts. Also, LLMs can produce workflows with code injection vulnerabilities. There is a significant difference in the performance of LLMs in detecting different types of defects. Also, fine-tuning reduced the effectiveness of StarChat for detecting syntactic errors. Currently, LLMs are ineffective at repairing workflow defects, eliciting the need for novel LLMs assisted techniques.
In summary, our contributions are as follows:

- A systematic evaluation of the capabilities of three state-of-the-art LLMs to generate GitHub workflows and detect, and repair different classes of defects.
- Various prompt engineering techniques aimed at optimizing the performance of LLMs across various tasks related to GitHub workflows.
- Insights about the current state and limits of LLMs when applied to engineering and security of GitHub workflows.
- A curated set of ~400K workflows with various prompts enabling future LLM research on GitHub workflows.

Our code is available at *purs3lab/LLMs4GitHubWorkflows*.

```
name: Deployment
on①:
  # dummy_pull:
  pull_request:
    branches: [main]
jobs②:
  build⑧:
    steps:
     - name: Checkout Code
       uses③: actions/checkout@v2
     - name: Build the code
       run④: make
    ...
  test⑨:
    needs⑤: build
    steps:
     - name: Log tests
       run: |
            echo "Running␣tests"
            echo "Commit - ${{␣github.event.pull_request.head.sha␣
                 }}"⑥
            echo "Branch - ${{␣github.event.pull_request.head.
                 ref␣}}"⑦
             # echo "Branch - $BRANCH_NAME"
       # env:
           # BRANCH_NAME: ${{ github.event.pull_request.head.ref
                }}
      ...
```

Listing 1: Example of a workflow file which is triggered upon the creation of a pull request. The workflow builds the submitted code and runs the existing test-suite.

2 BACKGROUND AND RELATED WORK

In this section, we will provide the necessary background on GitHub workflows, LLMs, and discuss related work.

2.1 GitHub workflows

GitHub workflows can be created by adding a `YAML` file (i.e., workflow file) to the `.github/workflows` folder in the target GitHub repository. The Listing 1 shows an example of a workflow file with markings representing different components. A workflow file needs to have event triggers (which trigger the execution of workflow, i.e., ① in Listing 1), jobs (②) to be executed (e.g., ⑧, ⑨), where each job is a sequence of steps (e.g., ③, ④). A job can be dependent on other jobs, e.g., the job `test` depends on `build` as indicated by ⑤. Each step represents a unit of work, which can be performed either through running shell commands (e.g., ⑥, ⑦), programs (e.g., ④), invoking other modules (i.e., Actions) e.g., ③. A workflow starts execution when one of the triggers occurs. Each job is independent, and all jobs execute in parallel unless there is a dependency where a job waits for all its dependents. Within each job, steps are executed sequentially in the order specified in the workflow.

Several works studied GitHub workflows along various aspects, such as most common automation practices [8], common patterns to perform various tasks [20], and changes made by developers over time [49]. Valenzuela-Toledo *et al.* [49] highlighted the absence of robust tools that could support GitHub workflows and detect syntactic and functional errors at an early stage in the development process. None of these works involve LLMs or have specifically addressed their use for GitHub workflows.

2.1.1 Defects in workflows. Similar to traditional programs, workflows can also have defects. We focus on two classes of defects: *syntactic errors* and *security vulnerabilities*, specifically code injection vulnerabilities.

Syntactic errors prevent the workflow from being executed. However, identifying syntactic errors in workflows requires complete knowledge of workflow structure and valid values. The mere validity of `YAML` file does not guarantee correct workflow syntax. For instance, in the workflow in Listing 1, changing the trigger (i.e., `pull_request`) to an invalid name (say `dummy_pull` as indicated by) produces a valid `YAML` but syntactically invalid workflow.

Security vulnerabilities could be exploited by attackers to perform various malicious activities (e.g., exfiltrating repository secrets), leveraging the permissions assigned to the workflow causing broader supply chain attacks [18, 45]. For instance, in Listing 1, one of the steps (indicated by ⑦) prints the source branch name of a pull request and is prone to code injection vulnerability (indicated by). Note that the branch name (`github.event.pull_request.head.ref`) is determined by the creator of the pull request rather than the repository owner. An attacker can craft a branch name that includes the desired shell command and raise a pull request. The print command will interpret the branch name as a shell command and execute the attacker-provided command. The ✔ marker shows the correct way to print, i.e., using an intermediate environment variable for the branch.

Security aspects of the GitHub CI platform have also been explored during prior research [16, 21, 27]. These works primarily focus on designing static analysis tools to detect different classes of security vulnerabilities in GitHub workflows. For instance, Muralee *et al.* [27] developed Argus, a static taint tracking tool aimed at identifying command injection vulnerabilities in GitHub workflows. However, no work tries to use LLMs for security tasks in GitHub workflows.

2.2 Large Language Models (LLMs)

LLMs have emerged as transformative tools capable of understanding and generating human-like text based on vast amounts of data they have been trained on. To elicit better responses from LLMs, various strategies have been formulated. Among these, **instruction fine-tuning** [24, 36, 53, 55] stands out as a notable approach. This method involves augmenting existing pre-trained models by further training them on smaller, domain-specific, and multi-task datasets and providing detailed instructions. Another effective strategy to elicit better responses involves the engineering of more refined prompts [10], i.e., *prompt engineering*, provided to the models. The usage of LLMs can be broadly classified into the following three modes [6] based on the amount of task-specific information provided:

- **Zero-shot mode** involves presenting the LLM with no task specific information. The expectation is that the model, leveraging its extensive pre-training, will generate relevant outputs for entirely novel problems.
- **One-shot mode:** Here, we provide a single example of the prompt and the desired outcome. The example serves to guide the model's response by providing a context or template for the task at hand.
- **Few-shot mode** extends the concept of one-shot mode by providing multiple labeled examples.

It is crucial to understand that the above prompting strategies are **Tuning-free prompting** [23], i.e., we do not change the parameters of the pre-trained LLMs.

2.2.1 Using LLMs for Automated Code Generation. Driven by the effectiveness of LLMs, there has been significant interest in designing LLMs for code-related tasks. For instance, close-source GPT-3.5 [31] and GPT-4 [32], inheriting the capabilities of Codex [7] designed specifically for programming tasks, have been extensively utilized. Other open-source code LLMs including CodeT5[52], CodeGen[30], StarCoder[22], CodeLlama[40], etc., have been successively introduced, and have demonstrated remarkable performance in software development tasks.

One of the important tasks is the text-to-code generation (i.e., generating code based on the natural language description). However, most works focus on programming languages such as `Java`, `C/C++`, and `Python`. As mentioned in § 2.1, GitHub workflows are engineered in `YAML` files. Only few works [37, 56] focus on using LLMs for generating `YAML` files. Pujar *et al.*[37] fine-tuned the CodeGen LLM, and evaluated its performance in generating YAML scripts for Ansible. Although GitHub workflows follow the `YAML` syntax, they differ significantly from Ansible scripts (§ 2.1).

2.2.2 Using LLMs for Automated Defect Detection. Many works investigated the effectiveness of LLMs in defect detection in regular programs. Thapa *et al.* [46] fine-tuned various transformer-based language models (e.g., BERT [9], GPT-2 [38], DistilBERT [42], etc.) on binary and multi-classification tasks using software vulnerability datasets from C/C++ applications. Similarly, Gao *et al.* [14] evaluated defect detection capabilities in CTF (Capture-the-Flag) challenges and real-world applications. Fu *et al.* [12] introduced LineVul, a line-level vulnerability predictor leveraging BERT to predict the presence of vulnerabilities in a dataset composed of C/C++ applications.

All these works focus on vulnerabilities in regular programs. However, as we explained in § 2.1.1, defects in GitHub workflows differ from those in regular programs. Furthermore, none of the existing works try to evaluate the accuracy of the detection, i.e., line number of the defect. In this work, we focus on holistically assessing LLMs capabilities to detect workflow defects.

2.2.3 Using LLMs for Automated Program Repair (APR). Sobania *et al.* [44] performed a comparative evaluation of Python program repair effectiveness of ChatGPT [31], Codex and CoCoNuT [25]. Ahmad *et al.* [2] employed an ensemble of LLMs, specifically Codex and CodeGen, to automatically rectify hardware security vulnerabilities in Verilog. Wu *et al.* [54] studied the capabilities of LLMs in Java vulnerability repair and compared them with those of deep-learning-based APR models. However, no studies focus on CI/CD platforms, specifically GitHub workflows, which contain a blend of configuration steps (potentially) involving various programming languages.

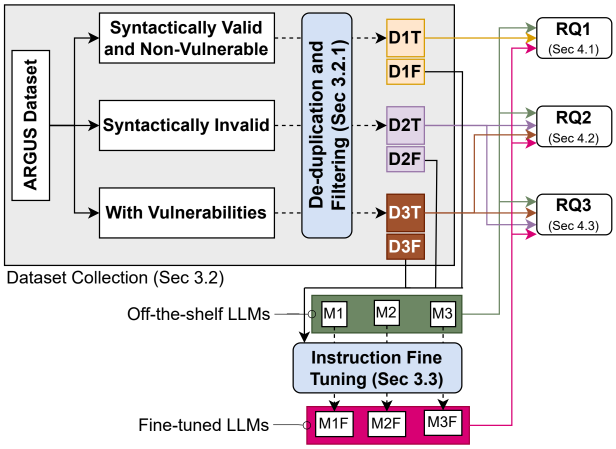

Figure 1: Overview of Our Study.

3 STUDY DESIGN

The Figure 1 shows the overview of our study. We created three GitHub workflow datasets, $D1$, $D2$, and $D3$ to investigate our research questions. We selected three state-of-the-art LLMs and fine-tuned them with a mixed subset of our datasets. We performed our investigation with both off-the-shelf LLMs and their fine-tuned versions.

3.1 LLMs Selection

We aim to select state-of-the-art LLMs that are specifically designed for programming tasks (e.g., code completion, code generation, defect detection, etc.). We focus on instruction-following LLMs, i.e., which perform a task based on provided instructions. Finally, we should also be able to fine-tune the models, e.g., we exclude GPT-4 [32] as we have no access to fine-tune it. Based on the above criteria, we selected three LLMs, i.e., GPT-3.5 Turbo [31], StarChat-β [47], and CodeLlama-7B-Instruct [40] as summarized in Table 1.

Table 1: Models considered in our studies

ID	Model	Model Version	Parameters	Context Length	Provider
M1	GPT-3.5	GPT-3.5 Turbo	-	4,096 tokens	OpenAI
M2	StarChat	StarChat-β	16B	8,192 tokens	Hugging Face
M3	CodeLlama	CodeLlama-7B-Instruct	7B	16,384 tokens	Meta

3.2 Dataset Collection

We used the GitHub workflow dataset from Argus [27], a recent work that tries to find vulnerabilities in GitHub workflows. The dataset has 2,778,483 GitHub workflows, collated over a period from November to December 2022. The dataset also includes 7,640 GitHub workflows with manually confirmed vulnerabilities. We split the dataset into three mutually exclusive sets:

- **Dataset II (D2):** This set contains an equal number of GitHub workflows with one syntax error and workflows with no syntax errors. Specifically, we ran `actionlint` [1], a syntax checker tool to find workflows with syntax errors, and picked the same number of syntactically valid workflows to create $D2$.
- **Dataset III (D3):** Similarly, this set contains an equal number of GitHub workflows with at least one vulnerability and workflows with no vulnerabilities. We used the vulnerable workflows from Argus dataset and collected the same number of non-vulnerable workflows to create $D3$.
- **Dataset I (D1):** All the remaining workflows, i.e., syntactically valid and contain no vulnerabilities, are collected to form $D1$.

3.2.1 *De-duplication and Filtering.*

We deduplicated Argus dataset and ignored workflows with more than 1,024 tokens (2,048 tokens [5] for vulnerable workflows) considering the context length supported by the selected LLMs and our designed prompts. Also, for $D1$, we performed the following two additional filtering steps to ensure that it contains mostly representative and realistic workflows. First, we ignored workflows that lack names or have steps without names. As we will discuss in § 3.4.1, these names are needed to create prompts and are important to understand the objectives of workflows. Second, we classified workflows using structural complexity metrics and filtered out outliers as they are not representative of realistic workflows. We provide the details of this in our extended report [58].

3.2.2 *Fine-Tuning Dataset.*

We also created a fine-tuning split for each dataset by randomly picking the same number (3,200) of workflows from the corresponding dataset. We capped at 3,200 as we did not find any significant increase in effectiveness with a larger number of workflows. For $D2$ and $D3$, we picked 1,600 positive and negative workflows. As we will discuss in § 4, we used the fine-tuning split for each dataset to create fine-tuned LLMs.

The Table 2 shows the summary of different datasets and statistics of the corresponding workflows.

Table 2: Summary of the different datasets used in our study.

Datasets			Num Workflows	Size (Bytes) Min/Mean/Median/Max
D1	Dataset I	FT (D1F)	3,200	155/1,388/1,212/4,060
		Test (D1T)	287,876	84/1,247/1,068/4,450
D2	Dataset II	FT (D2F)	3,200	55/1,362/1,147/4,338
		Test (D2T)	122,640	20/1,352/1,126/4,751
D3	Dataset III	FT (D3F)	3,200	203/2,017/1,709/8,711
		Test (D3T)	2,006	194/2,049/1,748/7,854

3.3 Instruction Fine-Tuning

Several works[19, 46] show the effectiveness of fine-tuning LLMs and demonstrate that they perform better than original models. We also used fine-tuned models as part of our study.

Fine-tuning requires a dataset of input and expected output pairs. Specifically, for instruction fine-tuning, we need (instruction, output) pairs, i.e., natural language instruction to perform a task and the expected output. We created the fine-tuning dataset for three of our tasks, i.e., Workflow Generation (T1), Syntactic Error Identification (T2), and Code Injection Vulnerability Detection (T3) by using the corresponding fine-tuning splits (§ 3.2.2), i.e., D1F, D2F, and D3F, respectively. For each task, we use the expected user

[5]The number of vulnerable workflows is limited.

Table 3: Workflow-related tasks and corresponding prompts (§ 3.4.1), and metrics (§ 3.4.2) that are evaluated as a part of the study.

Research Question	Task	Prompt Engineering (§ 3.4.1)				Evaluation Metrics (§ 3.4.2)
		System Prompt		User Prompt		
		Persona	Output format	ID	Description	
RQ1	Workflow Generation (T1)	software engineer	```yaml <Workflow>```	P1	workflow-level information and all job IDs	Accuracy@K BLEU score Manual validation
				P2	workflow-level information, all job IDs and all step names	
				P3	workflow-level information, all job IDs, all step names and all dependencies that can be used	
				P4	workflow-level information, all job IDs, job-level information and all step names	
				P5	workflow-level information, all job IDs, job-level information and step-level information	
RQ2	Syntactic Error Identification (T2)	software engineer	<Yes *or* No> | line number: ...	P1	Is there a syntactic error in the following GitHub workflow? ```yaml <Workflow>```	Accuracy@K F1 score
				P2	Is there <syntactic error type> in the following GitHub workflow? ```yaml <Workflow>```	
	Code Injection Vulnerability Detection (T3)	security engineer	No *or* Yes | line number: ... | tainted variable: ... | taint source: ...	P1	Is there any code injection vulnerability in the following GitHub workflow? ```yaml <Workflow>```	
				P2	Is there any <vulnerability type> in the following GitHub workflow? ```yaml <Workflow>```	
				P3	Is there any code injection vulnerability in the following GitHub workflow? <hint message>. ```yaml <Workflow>```	
RQ3	Syntactic Error Fixing (T4)	software engineer	```yaml <Workflow>```	P1	Please fix syntactic errors in the following GitHub workflow. ```yaml <Workflow>```	Accuracy@K
				P2	Please fix syntactic errors in the following GitHub workflow. <location>. ```yaml <Workflow>```	
				P3	Please fix syntactic errors in the following GitHub workflow. <location>. <error message>. ```yaml <Workflow>```	
	Code Injection Vulnerability Repair (T5)	security engineer	```yaml <Workflow>```	P1	Please repair code injection vulnerabilities in the following GitHub workflow. ```yaml <Workflow>```	
				P2	Please repair code injection vulnerabilities in the following GitHub workflow. <location >. ```yaml <Workflow>```	
				P3	Please repair code injection vulnerabilities in the following GitHub workflow. <location>. <fix strategy>. ```yaml <Workflow>```	

prompt (Table 3) as its instruction and the corresponding workflow (T1) or defect location (T2 and T3) as the output. We use the suffix *F* to indicate the fine-tuned variant of the model. For instance, GPT-3.5F indicates fine-tuned variant of GPT-3.5 (Table 1). Note that we used three generation tasks (instead of all five tasks) for fine-tuning. This is because generating expected output for repair tasks (T4 and T5) is tedious, especially when there can be multiple valid but semantically equivalent repairs for a given defect. Nonetheless, as shown by the recent work [53, 55], the fine-tuned models on generation tasks will also perform better on other related but unseen tasks. Based on this, our fine-tuned models are expected to perform better even on unseen defect repair tasks.

3.3.1 Implementation Details. We use OpenAI's APIs to fine-tune the GPT-3.5. As for StarChat and CodeLlama, we utilize the Hugging Face implementation version of the models and fine-tune each model using the PyTorch framework with the parameter-efficient fine-tuning (PEFT) method. The fine-tuning processes for StarChat and CodeLlama are executed on a single NVIDIA A100 GPU with 80GB memory and on a cluster node running CentOS 7, utilizing Slurm (Simple Linux Utility for Resource Management) as the batch scheduler for resource and job management. Each model is fine-tuned for 5 epochs. We mixed D1F, D2F, and D3F as the training set and randomly selected 8,00 samples from each of D1T, D2T, and D3T for testing, maintaining a train-to-test ratio of 8:2.

3.4 Methodology

The aim of our study is to evaluate the effectiveness of LLMs in performing various tasks related to GitHub workflows. Our study is organized into the following three research questions:

- **RQ1: Workflow Generation:** What is the effectiveness of LLMs in generating GitHub workflows (T1)? How secure and valid are the generated workflows?
- **RQ2: Defect Detection:** How effectively can LLMs detect defects? Both syntactic errors (T2) and code injection vulnerabilities (T3)?
- **RQ3: Defect Repair:** What is the effectiveness of LLMs in repairing defects? Both syntactic errors (T4) and code injection vulnerabilities (T5)?

The Table 3 summarizes tasks associated with each research question. We followed the same methodology to investigate all our research questions. Specifically, for each task and workflow, we provide a prompt to LLMs and compare their outputs with the expected output using various metrics (§ 3.4.2).

3.4.1 Prompt Engineering. Several works [3, 50] show that prompts greatly influence the effectiveness of LLMs. For each task, we created prompts (mimicking *user* instructions) with varying levels of detail describing the desired output from a LLM.

Salewski *et al.* [41] demonstrated that assigning a specific persona (e.g., domain expert) to LLMs will result in better results. Based on this, we create a persona prompt or *system* prompt for each task that sets up the desired persona of a LLM. *We prepend the system prompt to the user prompt to create the final prompt, which we provide to LLMs*. The details of the prompts are depicted in Table 3. Our extended report [58] provides examples of the prompts and explains every definition (e.g., vulnerability type, error message, fix strategy, etc.) in user prompts.

User Prompts for Workflow Generation (T1). In this task, we evaluate the capability of LLMs in generating well-formatted workflows from a natural language description. As described in § 2.1, a workflow has a name, trigger, and set of jobs, each with a sequence of steps. In addition, Job and Step have a *name* field describing its functionality, e.g., *"Build the project"*. We create five types of prompts (P1-P5) for this task, with each prompt providing more description about the target workflow. P1 has the minimal description needed to create the workflow, i.e., name, trigger, and the set of job IDs. However, it does not provide any details about the steps in each job. Meanwhile, P2 (in addition to information from P1) provides information about the steps. Similarly, P3-P5 provides an increasing level of detail.

User Prompts for Syntactic Error (T2) and Code Injection Vulnerability (T3) Identification. Here, we evaluate the defect detection capability

of LLMs. We create prompts, each of which provides more information about the target defect. The corresponding rows in Table 3 provide more details. The basic prompt asks for the existence of the desired defect (i.e., syntactic error or code injection vulnerability). Other prompts provide more details about the target defect, i.e., specific type or hint message.
User Prompts for Syntactic Error (T4) and Code Injection Vulnerability (T5) Repair. Here, we evaluate the defect repair capabilities of LLMs. We provide LLMs with varying degrees of information related to the target defect. Indicative information ranges from minimal information (i.e., defect type) to comprehensive hints, encompassing details such as defect locations, error messages, or fix strategies.

3.4.2 Evaluation Metrics. We used the following three evaluation metrics to assess the output produced by LLMs across various tasks.
BLEU (Bilingual Evaluation Understudy) score [34] is a value ranging from 0 to 1, indicating how similar the candidate text is to the reference text, with values closer to 1 representing higher similarity. We use BLEU-4 (i.e., the geometric average of 1-gram, 2-gram, 3-gram, and 4-gram precision) to compare the generated workflow with the expected workflow because of the need to preserve the ordering of tokens.
Accuracy@K [57] enables measuring accuracy of results when multiple (i.e., K) responses are provided. Specifically, given a test t (or a sample s), we consider the responses of a LLM as a match (i.e., score 1) when any one of the K responses satisfies t (or matches s), else we consider it as no match (i.e., score 0). We use $K = 5$ in all our experiments. For n tests (or samples), we average the matching score (i.e., 1 or 0) across all the n samples to get *Accuracy@K*.
F1-Score [43] is calculated as a harmonic mean of precision and recall. This score (ranging from 0-1) provides a single metric to evaluate binary classification. We use this to evaluate defect detection effectiveness.

The last column of Table 3 shows the summary of metrics used to evaluate each task.
Workflow Generation Task (T1): Here, we want to evaluate whether the workflow generated by a LLM performs the functionality as the expected workflow. However, precisely accessing this requires semantic equivalence checking [29] — infeasible in the general case. Instead, (i) we utilize `actionlint` to check that the generated workflow is valid (i.e., no syntactic errors), and calculate *Accuracy@K* to measure correctness; and (ii) compare how similar (content-wise) the generated workflow is to the expected workflow by computing *BLEU* score; and (iii) manually validate 270 randomly sampled workflows.
Defect Detection Tasks (T2 and T3): Here, we verify two aspects: detection capability and accuracy of the detection. Specifically, we use *F1-Score* to measure detection capability and measure detection accuracy (i.e., line number for T2, line number, tainted value and taint source for T3) using *Accuracy@K*.
Defect Repair Tasks (T4 and T5): Here, we want to evaluate whether a LLM correctly repaired a workflow. However, automatically checking whether LLMs produced the correct repaired workflow requires semantic checking — similar to the workflow generation task (T1). Instead, we check whether the generated workflow is non-vulnerable and use *Accuracy@K* to measure the repair capabilities of LLMs.

3.4.3 LLMs Configuration and Experimental Setup. As mentioned in § 2.2, there are three basic modes (i.e., zero-shot, one-shot, and few-shot) of using a LLM model. However, during our experiments with off-the-shelf variants, we found no difference in effectiveness between the one-shot mode and the few-shot mode. *We will only present the results of zero-shot and one-shot modes for off-the-shelf variants. For Defect Repair Tasks, we used both zero-shot mode and one-shot mode for fine-tuned variants, as these tasks are unseen to the fine-tuned models.* (§ 2.2). For a given mode, the performance of a LLM model might vary with different configurations. For every task, we want to assess a LLM mode using its best-performing configuration and the most effective prompt.

LLMs have a temperature parameter, indicating the desired level of randomness. Specifically, higher temperature values indicate a higher degree of non-determinism. The values from 0 to 1 are recommended to prompt a LLM to produce responses that are acceptable to humans. For example, the temperature range of GPT-3.5 is from 0 to 2. However, temperature values above 0.9 make the responses technically useless. Also, as mentioned in § 3.4.1, we generate several prompts for each task.
Calibration (Identifying Effective Configuration): For a given LLM and task, we use a small but representative subset (i.e., calibration set (CAset)) of samples to identify which temperature and prompt combination gives the best result. Specifically, we use 0.1, 0.3, 0.5, 0.7, and 0.9 as our temperature values and combine them with prompts with varying levels of detail (Table 3). *The best-performing temperature value and prompt will be used to evaluate the final set of samples.*

For T1, we collected 266 workflows as our CAset. We performed a random sampling and collected two workflows each for 133 effective combinations of complexity metrics [58], ensuring that our CAset is representative. Also, given the large number (0.28 million) of workflows in D1T, we picked 20 workflows along each of the 133 complexity metrics combinations as our evaluation dataset.

For T2, we randomly selected 200 workflows (100 syntactically valid, 100 syntactically invalid) to construct our CAset and sampled 5,000 GitHub workflows (2,500 syntactically valid, 2,500 syntactically invalid) to form a larger evaluation dataset. Similarly, for T3, we randomly selected 80 vulnerable workflows (with a total of 108 vulnerabilities), and then sampled 80 non-vulnerable GitHub workflows to construct CAset. We utilized all remaining vulnerable GitHub workflows (923 GitHub workflows with 1,586 vulnerabilities) and 923 non-vulnerable GitHub workflows to form a larger evaluation dataset.

For T4, we randomly chose 200 GitHub workflows with syntactic errors as our CAset and sampled an additional syntactically invalid 2,500 GitHub workflows to constitute a larger evaluation dataset. For T5, we randomly selected 100 and 375 GitHub workflows containing code injection vulnerabilities that can be fixed within workflows to form our CAset and larger evaluation dataset, respectively.

4 RESULTS

In this section, we present the results of the study along with our three research questions. For each task (under a research question), we first present the calibration results, aiming to identify the most

Table 4: BLEU scores of workflow generation on CAset.

Model	t	off-the-shelf										fine-tuned				
		0-shot					1-shot									
		P1	P2	P3	P4	P5	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
GPT-3.5	0.1	13.60	35.57	40.87	40.56	74.79	25.69	46.20	54.31	54.26	78.99	25.83	47.25	53.64	53.33	80.84
	0.3	15.21	36.92	42.10	41.63	76.00	27.77	47.76	55.90	55.87	79.71	27.51	49.05	54.43	54.72	82.56
	0.5	16.00	38.04	43.00	42.58	76.98	28.84	48.57	56.61	56.47	80.49	27.46	49.05	54.60	55.04	83.14
	0.7	16.39	38.08	43.35	43.17	76.93	28.95	49.49	57.23	57.22	81.24	26.69	47.79	53.75	55.13	82.97
	0.9	16.84	38.39	43.69	43.22	**77.45**	29.21	49.67	57.15	57.31	**81.61**	24.36	46.43	52.54	52.80	**83.49**
CodeLlama	0.1	15.59	35.32	40.38	41.98	74.03	36.14	51.22	55.81	57.15	79.84	25.38	45.54	51.06	53.92	82.15
	0.3	17.12	37.84	42.56	44.98	75.47	37.42	52.68	57.34	58.50	81.58	27.43	48.38	53.12	56.29	83.73
	0.5	17.91	37.56	43.15	45.35	**76.41**	37.77	53.20	57.81	59.48	**82.53**	27.42	48.60	52.90	56.54	**84.11**
	0.7	17.65	37.61	42.65	44.43	75.81	38.42	53.41	57.72	58.86	81.75	26.71	47.08	53.16	55.58	83.85
	0.9	16.43	36.18	40.38	41.65	73.31	37.84	51.83	55.99	57.64	81.00	25.08	45.37	51.19	53.53	83.25
StarChat	0.1	17.22	36.87	40.21	41.07	62.75	34.61	49.72	53.91	55.53	75.38	26.98	48.72	53.57	54.47	80.71
	0.3	18.54	38.21	41.82	43.38	64.83	36.51	51.38	55.30	57.36	76.89	29.07	50.67	55.75	57.43	81.81
	0.5	19.46	39.10	42.93	43.81	66.26	37.36	52.33	56.58	58.10	77.49	29.45	50.74	56.20	58.39	**82.79**
	0.7	19.34	39.04	42.69	43.81	**66.45**	37.36	52.30	55.86	58.40	**77.88**	28.72	50.12	55.42	57.15	82.37
	0.9	18.91	39.03	42.46	43.68	66.31	37.14	51.98	56.27	58.00	77.48	26.09	47.88	53.35	55.78	82.34

Table 5: Accuracy@K of workflow generation on CAset.

Model	t	off-the-shelf										fine-tuned				
		0-shot					1-shot									
		P1	P2	P3	P4	P5	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
GPT-3.5	0.1	69.55	66.17	57.89	60.53	78.95	89.10	82.71	70.68	83.46	88.35	92.86	87.22	79.32	85.71	83.83
	0.3	77.07	74.06	62.41	67.29	84.21	91.35	84.96	77.82	87.22	88.72	95.49	93.23	86.09	92.11	89.85
	0.5	84.59	78.57	66.54	68.80	87.59	90.98	87.97	77.07	87.97	89.47	**96.62**	94.36	85.71	93.61	88.72
	0.7	85.71	82.33	65.79	69.92	87.22	92.86	87.97	79.70	87.97	92.86	94.74	93.98	87.97	93.98	92.86
	0.9	89.10	78.95	68.42	76.32	**89.85**	91.73	90.98	83.46	89.85	**93.61**	90.98	92.48	84.96	91.73	92.11
CodeLlama	0.1	84.09	78.41	70.83	76.52	73.11	90.91	79.55	75.00	77.65	80.45	96.21	93.18	88.26	90.91	87.12
	0.3	92.42	84.85	75.76	83.33	79.17	**95.49**	88.35	80.08	83.83	87.97	97.73	94.70	92.05	93.94	90.91
	0.5	92.80	88.64	78.03	85.23	80.30	**95.49**	86.47	81.95	85.71	89.47	**99.24**	96.21	92.05	96.21	90.91
	0.7	**93.56**	88.26	76.14	79.92	82.58	93.98	91.73	79.70	91.35	86.09	97.73	96.97	94.32	96.21	93.18
	0.9	83.71	76.52	63.64	71.97	75.76	90.98	83.83	78.20	81.58	87.59	96.97	93.94	92.80	92.80	93.56
StarChat	0.1	74.62	61.74	56.06	57.58	54.55	70.08	62.50	59.09	62.12	60.53	73.11	70.45	64.02	66.29	58.33
	0.3	84.85	70.08	62.88	63.26	58.33	77.82	71.80	63.53	65.79	61.65	83.33	79.17	70.45	71.59	59.47
	0.5	85.23	74.24	69.70	70.83	61.36	78.95	73.68	66.92	70.30	63.91	88.64	80.30	72.73	76.52	60.23
	0.7	87.88	79.92	71.97	74.24	62.50	83.46	74.06	68.80	72.56	64.66	**90.15**	84.85	76.14	78.03	61.74
	0.9	**89.77**	76.52	75.76	75.00	65.15	**89.10**	74.44	70.30	73.31	63.53	83.33	79.55	74.62	77.65	64.77

effective configuration for each LLM variant in each mode, i.e., (i) off-the-shelf variants in zero-shot mode, (ii) off-the-shelf variants in one-shot mode, and (iii) fine-tuned variants. An exception is for T4 and T5, unseen tasks for fine-tuned variants, where we seek the optimal configuration for fine-tuned variants in both zero-shot and one-shot modes. Second, we present the final assessment of each LLM mode using its most effective configuration.

4.1 RQ1: Workflow Generation

Here, we evaluate the effectiveness of LLMs in generating workflows. As shown in Table 3 and described in § 3.4.2, this research question has one task, and we use three evaluation metrics.

4.1.1 Calibration. We use *BLEU* (Table 4) and *Accuracy@K* (Table 5) scores for calibration.

BLEU Scores. The Table 4 shows that across all temperature (t) values and LLMs modes. The trend of *BLEU* scores across different temperature values changes across different LLMs. For GPT-3.5, the largest temperature value of 0.9 (i.e., greater non-determinism) is better. Whereas for CodeLlama and StarChat, temperature values of 0.5 and 0.7, respectively, are the best. Interestingly, the *BLEU* score increases with more detailed prompts. This indicates that users should provide detailed prompts to get the expected workflow. This differs from the standard code generation tasks, where LLMs are shown to perform well even with a very simple prompt [11]. This is because a simple prompt can precisely describe the desired code generation task, e.g., *"generate sort function"*. Whereas workflows (as explained in § 2.1) are sequences of steps and are hard to describe in a simple prompt. Furthermore, even for a single step, the appropriate way to perform it depends on the target project. For instance, a step to build a project depends on the target project, i.e., C/C++ (make/cmake), python (`setup.py`), java (`ant build`), etc. More contextual information is needed to generate appropriate steps and workflows.

Finding 1.1: Unlike for regular code generation tasks, LLMs require detailed prompts to generate desired workflows.

Accuracy@K Scores. The Table 5 shows the trend of *Accuracy@K* scores. It is interesting to see that detailed prompts do not always improve the *Accuracy@K* scores. In fact, detailed prompts reduce the *Accuracy@K* scores, as shown by the decrease in scores across the P2 and P3 columns. In other words, detailed prompts result in LLMs producing defective workflows.

Interestingly, *Accuracy@K* score follows a inverse bell curve for GPT-3.5 and CodeLlama. Specifically, for low-detail prompts, the *Accuracy@K* score decreases as the prompt becomes more detailed (till P3). However, the *Accuracy@K* slowly rises as the prompt becomes increasingly detailed (P4 and P5). The case is slightly different for StarChat, where *Accuracy@K* always decreases with the increase in the details of the prompt.

The trend is different for *BLEU* score where detailed prompts provide better results. Upon investigation, we found that LLMs generate smaller workflows with simpler prompts and consequently reduces the chances of having defects resulting in higher *Accuracy@K* score. However, simpler prompts are unlikely to generate the desired workflows, as shown by the lower *BLEU* scores (Table 4). On the other hand, detailed prompts to LLMs produce workflows closer to the desired workflows, but the generated workflows might have defects. Listing 2 shows two GitHub workflows generated by fine-tuned CodeLlama. A detailed prompt (P5) produces the left workflow, which is closer to the desired workflow[6] but contains a syntactic error (), while a simple prompt (P1) generates the right one which is syntactically valid but incorrect.

```
name: Generate Docs
on:
  push: {branch: 'main'}
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout sources
        uses: actions/checkout@v2
      - name: Install stable toolchain
        uses: actions-rs/toolchain@v1
        with:
          profile: minimal
          target: wasm32-wasi
          toolchain: stable
          override: true
      - name: Run cargo build
        uses: actions-rs/cargo@v1
        with:
          command: doc
          args: --workspace --no-deps
      - name: Prepare docs folder
        run: |
          sudo chown -R $(whoami) target
              /doc
          touch target/doc/.nojekyll
      - name: Deploy documentation
          branch
        uses: JamesIves/github-pages-
            deploy-action@3.7.1
        with:
          GITHUB_TOKEN: ${{ secrets.
              GITHUB_TOKEN }}
          BRANCH: gh-pages
          FOLDER: target/doc
```

```
name: Generate Docs
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Set up Rust
        uses: actions-rs/toolchain@v1
        with:
          profile: minimal
          toolchain: stable
          override: true
          components: clippy
          target: x86_64-unknown-linux-gnu
      - name: Cargo packages
        uses: actions-rs/cargo@v1
        with:
          command: build
          args: --all
      - name: Publish to gh-pages
        uses: crazy-max/ghaction-github-
            pages@v2
        with:
          target_branch: gh-pages
          build_dir: target/doc
```

Listing 2: Two GitHub workflows generated by fine-tuned CodeLlama with 0-shot prompting at temperature 0.9. The left workflow was generated using P5, whereas the right one was generated by P1. The expected (i.e., ground truth) workflow is docs.yaml[6] in the neoeinstein/cj4-fadec repo.

Finding 1.2: LLMs have a high likelihood of producing invalid (i.e., with syntactic errors) workflows with detailed prompts.

4.1.2 Final Evaluation. We selected the best configuration of each LLM across different modes and performed our final evaluation. The Figure 2 shows the cumulative results across different modes.

Finding 1.3: For all LLMs, the fine-tuned variant (i.e., with F suffix) performs better than the corresponding off-the-shelf variant. For all LLMs, except for StarChat, one-shot mode performs better than zero-shot.

Effectiveness in generating expected workflows: Higher *BLEU* score indicates greater similarity between the generated and expected workflow. For off-the-shelf variants, GPT-3.5 achieves the best *BLEU*

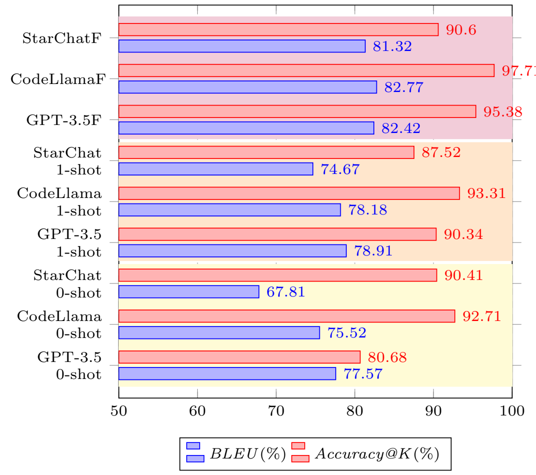

Figure 2: Final evaluation for workflow generation

score across all the modes. For fine-tuned variants, CodeLlama has the best *BLEU* scores. The left subfigure of Figure 3 shows the trend of *BLEU* scores against the size (in KB) of expected workflows. As we show in Table 2, most of the workflows are less than 4KB. Specifically, the majority (> 85%) of workflows are less than 3KB. For our size-related comparisons (i.e., Figure 3), we only considered workflows up to 3KB. We can see from Figure 3 (left subfigure) that as the workflow size increases, *BLEU* score initially increases rapidly and then remains unchanged.

Finding 1.4: The ability of LLMs to generate expected workflows does not vary much with the size of workflows.

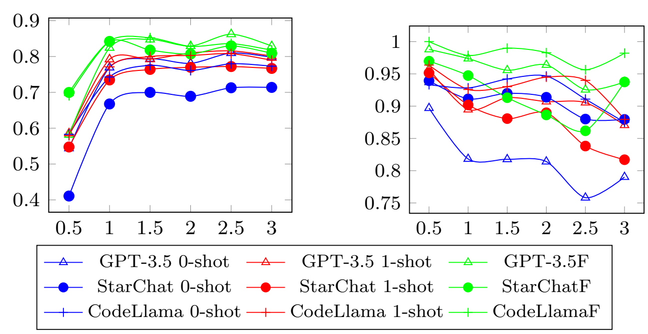

Figure 3: *BLEU* score (left) and *Accuracy@K* (right) against the size (in KB) of expected workflows.

Ability to generate valid (i.e., syntactically correct) workflows: Higher *Accuracy@K* score indicates a greater chance of generating valid workflows. For off-the-shelf and fine-tuned variants, CodeLlama achieves the best *Accuracy@K* score across all the modes. Unlike *BLEU* scores, the *Accuracy@K* scores slowly decrease as workflow size becomes larger (right subfigure of Figure 3). This is expected as workflow size increases, LLMs need to generate more

[6]https://github.com/neoeinstein/cj4-fadec/blob/main/.github/workflows/docs.yaml

tokens, increasing the likelihood of generating syntactic errors resulting in lower *Accuracy@K* scores.

Summary of RQ1: Although GPT-3.5 is highly likely to produce expected workflows. It might produce invalid or defective workflows. On the other hand, CodeLlama has a lesser likelihood of generating expected workflows but has a high probability of generating valid workflows.

Effectiveness in generating semantically correct workflows: We want to assess the LLMs' capability to generate semantically correct workflows. However, automatically determining semantic correctness is impossible. We decided to perform a manual validation by randomly sampling 30 valid workflows for each model and mode's best-performing *BLEU* configurations. In total, we manually checked 270 (30*9) workflows and determined the correctness of each workflow, i.e., a workflow is semantically correct when all the generated steps are semantically correct. The Table 6 shows the percentage of generated workflows with semantic correctness for each model. 30 samples generated by GPT-3.5 across all the modes and CodeLlama in 1-shot mode are all semantically correct workflows. Other models also reach a high percentage. The results indicate that generated workflows have a high semantic correctness.

Table 6: The percentage of semantically correct workflows among all the syntactically valid samples.

Model	off-the-shelf		fine-tuned
	0-shot	1-shot	
GPT-3.5	100%	100%	100%
CodeLlama	97.67%	100%	86.67%
StarChat	86.67%	93.33%	96.66%

How Secure are the Generated Workflows? Here, we want to evaluate how secure the workflows generated by LLMs are. Specifically, we run Argus on each of the workflows to assess the number of syntactically valid workflows generated by LLMs that contain security issues. The Table 7 shows the results. StarChat produced the most number of insecure workflows while GPT-3.5 produced the least. The Listing 3 shows an example of a workflow generated by GPT-3.5 that has a code injection vulnerability.

```
name: Receive PR
on:
  ...
jobs:
  test-pr:
    runs-on: ubuntu-latest
     ...
      - name: Set Outputs
        id: set-outputs
        run: echo ""::set-output name=is_valid::${{ steps.check-pr.outputs.VALID }}\n::set-output name=MSG::${{ steps.check-pr.outputs.MSG }}""
  save-pr-number:
    needs: test-pr
    ...
```

Listing 3: Example of a workflow generated by GPT-3.5 that has a code injection vulnerability. The output *check-pr.outputs.MSG* is tainted (i.e., controlled by non-repository owner).

Table 7: The number of syntactically valid workflows containing security issues.

Model	off-the-shelf		fine-tuned	Total
	0-shot	1-shot		
GPT-3.5	21	10	66	97
CodeLlama	26	35	213	274
StarChat	42	51	252	345

Finding 1.5: LLMs can produce workflows with code injection vulnerabilities. Developers should be careful while using workflows generated by LLMs.

4.2 RQ2: Defect Detection

As mentioned before, we are interested in LLMs' capability to detect two types of defects: syntactic errors and code injection vulnerabilities. As mentioned in § 2.1, detecting syntactic errors requires reasoning about the format of workflows. In other words, a well-formatted and syntactically valid `yaml` can be an invalid workflow. As mentioned in § 3.4.2, we use *F1-Score* to measure detection capability and *Accuracy@K* to measure detection accuracy (i.e., line number).

4.2.1 Syntactic Error Identification (T2). We evaluate this task using two prompts with varying details (Table 3).

Calibration: Table 8 shows the *F1-Score* and *Accuracy@K* of different models and their variants across different modes. Unlike Workflow Generation Task (T1), detailed prompts (P1 v/s P2) seem to have less effect on syntactic error detection.

F1-Score*:* In 0-shot mode, GPT-3.5 performs the best in detecting syntactic errors with the highest *F1-Score* of 72.25%. The performance of GPT-3.5 and CodeLlama dropped in 1-shot mode — contrary to previous works [4, 39] which show that 1-shot mode provides better performance than 0-shot mode. As expected, in GPT-3.5 and CodeLlama, the fine-tuned variants performed better than off-the-shelf variants. The case is different for StarChat, where the 1-shot mode of the off-the-shelf variant performs the best, even better than the fine-tuned variant.

Accuracy@K*:* The detection accuracy of off-the-shelf variants follows the same trend as the detection capability. In other words, GPT-3.5 performs the best in 0-shot mode, and 1-shot mode hurts the performance of GPT-3.5 and CodeLlama but improves that of StarChat. As expected, fine-tuned variants perform better than off-the-shelf variants.

Final Evaluation: The Figure 4 shows the evaluation of the best-performing configuration on the final large dataset. Overall, StarChat 1-shot mode is the best at detecting syntactic errors as indicated by the highest *F1-Score*, i.e., 100%. However, fine-tuned GPT-3.5 has the highest accuracy. In other words, StarChat is good at detecting whether a workflow has a syntactic error or not. But, fine-tuned GPT-3.5 is good at detecting where (i.e., line number) the syntactic error is. Listing 4 in our extended report [58] shows an example where StarChat correctly identified a syntactic error but GPT-3.5 failed.

Table 8: Effectiveness of syntactic error detection on CAset.

Model	t	F1-Score						Accuracy@K					
		off-the-shelf				fine-tuned		off-the-shelf				fine-tuned	
		0-shot		1-shot				0-shot		1-shot			
		P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2
GPT-3.5	0.1	61.40	**72.25**	3.170	2.560	87.76	90.45	27.00	39.00	3.000	1.000	81.00	85.00
	0.3	56.62	69.43	3.230	1.270	88.21	90.45	37.00	41.00	4.000	3.000	83.00	86.00
	0.5	58.56	68.69	4.260	3.730	87.18	90.45	39.00	42.00	5.000	5.000	83.00	86.00
	0.7	62.78	70.41	**4.320**	1.270	88.21	**91.46**	**46.00**	44.00	7.000	**8.000**	86.00	89.00
	0.9	54.13	65.98	3.240	3.800	86.87	90.55	42.00	43.00	6.000	6.000	86.00	**90.00**
CodeLlama	0.1	17.09	31.88	**32.00**	7.270	71.97	80.37	1.000	6.000	1.000	0.000	48.00	51.00
	0.3	33.09	37.42	25.21	3.740	72.80	**80.75**	1.000	8.000	2.000	2.000	51.00	55.00
	0.5	45.16	44.30	25.21	9.090	70.39	79.64	4.000	**11.00**	**8.000**	5.000	50.00	**59.00**
	0.7	**46.05**	40.99	25.81	3.850	70.18	79.82	3.000	**11.00**	6.000	3.000	54.00	57.00
	0.9	40.25	41.51	26.45	7.340	70.94	78.57	2.000	10.00	**8.000**	5.000	**59.00**	56.00
StarChat	0.1	67.34	66.67	**100.0**	**100.0**	64.20	**84.47**	6.000	12.00	12.00	10.00	49.00	62.00
	0.3	67.34	68.03	**100.0**	**100.0**	62.65	84.16	10.00	13.00	17.00	14.00	52.00	63.00
	0.5	65.68	**69.82**	98.49	99.50	68.26	82.76	**15.00**	14.00	13.00	20.00	55.00	65.00
	0.7	51.16	60.68	96.04	97.98	65.90	83.58	10.00	14.00	14.00	**29.00**	54.00	**69.00**
	0.9	47.13	57.45	89.11	91.98	64.80	80.98	7.000	14.00	21.00	14.00	51.00	65.00

Table 9: Effectiveness of code injection vulnerability detection on CAset.

Model	t	F1-Score									Accuracy@K								
		off-the-shelf						fine-tuned			off-the-shelf						fine-tuned		
		0-shot			1-shot						0-shot			1-shot					
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
GPT-3.5	0.1	7.140	17.98	80.00	1.770	0.000	24.14	93.49	92.86	**99.38**	0.000	0.000	8.570	0.000	0.000	3.850	86.67	89.52	89.52
	0.3	7.140	20.00	78.57	0.000	0.000	25.42	92.94	92.40	**99.38**	0.000	0.000	9.520	0.950	0.000	9.620	86.67	90.48	90.48
	0.5	9.410	21.98	79.52	0.000	0.000	24.78	92.94	92.31	**99.38**	0.000	0.000	12.38	0.950	0.000	**12.50**	89.52	91.43	91.43
	0.7	2.440	17.78	**81.93**	0.000	6.520	25.86	92.94	92.31	**99.38**	0.000	0.000	9.520	0.000	0.000	**12.50**	88.57	93.33	89.52
	0.9	11.63	17.78	79.04	5.130	0.000	**27.12**	93.49	93.49	**99.38**	0.000	0.950	**12.38**	0.950	0.000	**12.50**	90.48	**96.19**	90.48
CodeLlama	0.1	66.11	68.67	66.67	**87.86**	74.88	66.67	89.53	**89.66**	88.34	0.950	0.950	**3.810**	0.000	0.000	0.000	60.00	55.24	53.33
	0.3	66.10	67.54	66.67	85.54	73.89	66.67	89.02	89.02	87.80	0.950	0.000	2.860	0.000	0.000	0.000	64.76	66.67	58.10
	0.5	63.72	**68.81**	66.67	77.50	70.59	67.80	87.78	89.02	88.20	0.000	0.000	**3.810**	0.000	0.000	0.000	65.71	69.52	60.95
	0.7	60.66	59.11	67.56	74.17	67.05	68.09	84.39	87.86	87.80	0.950	0.000	1.900	0.000	0.000	0.000	67.62	**71.43**	62.86
	0.9	63.64	67.01	67.59	69.74	56.18	70.27	86.86	88.24	84.66	0.000	0.950	0.000	0.000	0.000	0.000	63.81	68.57	66.67
StarChat	0.1	66.67	66.67	66.67	72.07	87.43	72.90	94.41	93.83	96.86	0.000	0.000	1.900	12.38	17.14	20.95	68.57	68.57	64.76
	0.3	66.67	66.11	66.67	72.07	**89.89**	72.48	94.41	95.00	96.20	0.000	0.000	0.950	18.10	19.05	25.71	74.29	76.19	65.71
	0.5	66.67	**67.24**	65.25	76.56	84.66	73.49	93.08	93.17	95.54	0.000	0.000	**2.860**	12.38	22.86	**28.57**	78.10	75.24	68.57
	0.7	66.09	64.55	65.50	77.61	88.89	73.93	93.67	93.83	**97.50**	0.000	0.000	**2.860**	12.38	16.19	25.71	73.33	77.14	72.38
	0.9	59.81	56.84	60.00	74.40	84.44	75.12	91.82	92.59	**97.50**	0.000	0.000	1.900	6.670	10.48	26.67	**80.95**	78.10	74.29

Finding 2.1: Contrary to the observations for other applications, for GPT-3.5 and CodeLlama, the 1-shot mode is less effective than 0-shot in identifying syntactic errors in workflows. StarChat is best at detecting syntactic errors but GPT-3.5 can accurately identify the location of syntactic error.

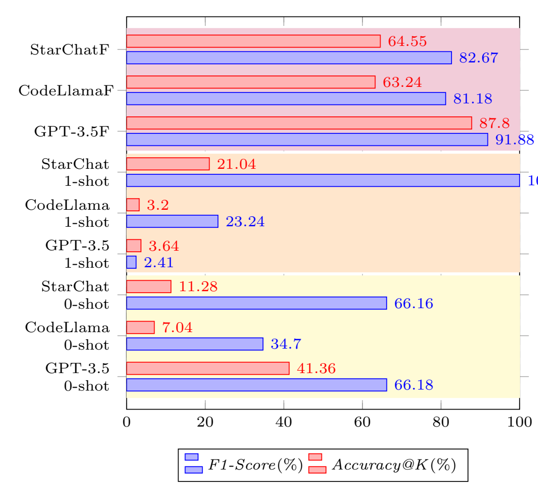

Figure 4: Final evaluation for syntactic error detection

4.2.2 Code Injection Vulnerability Detection (T3). As shown in Table 3, we use three prompts to evaluate this task.

Calibration: The left part of Table 9 shows the *F1-Score* of code injection vulnerability detection of different models and their variants across different modes. The fine-tuned variants perform best for all LLMs and a given prompt. For off-the-shelf variants of CodeLlama and StarChat, simpler prompts (i.e., P1 and P2) provide the best *F1-Score*. However, for GPT-3.5, the detailed prompt (i.e., P3) provides the best *F1-Score*. For off-the-shelf variants of CodeLlama and StarChat, smaller temperature values (i.e., low non-determinism) provide the best *F1-Score*. In contrast, higher temperature (i.e., higher non-determinism) works well for GPT-3.5.

The right part of Table 9 shows the *Accuracy@K* of code injection vulnerability detection of different models and their variants across different modes on CAset. The off-the-shelf variant performs better when receiving detailed prompts, but it has a poor performance in pinpointing vulnerabilities across all modes. On the contrary, the fine-tuned variant does not benefit from detailed prompts and performs much better than the corresponding off-the-shelf variant.

Final Evaluation: The Figure 5 shows the evaluation of the best-performing configuration on the final large dataset. Overall, fine-tuned variants perform better, demonstrating the importance of fine-tuning in detecting code injection vulnerabilities. The fine-tuned

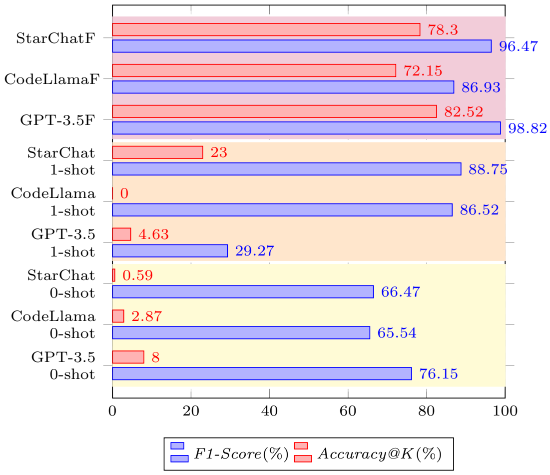

Figure 5: Final evaluation for code injection vulnerability detection.

variant of GPT-3.5 (i.e.,GPT-3.5F) performs the best. *Interestingly, off-the-shelf GPT-3.5 performs the worst in 1-shot mode.* Listing 5 in our extended report [58] shows an example where GPT-3.5 correctly identified a code injection vulnerability missed by other LLMs.

Summary of RQ2: Across the tested LLMs, there is a significant difference in the effectiveness of syntactic error detection and code injection vulnerability detection. Off-the-shelf StarChat in 1-shot mode is best at detecting syntactic errors, whereas fine-tuned GPT-3.5 is best at detecting code injection vulnerabilities.

We also discuss the effectiveness of detection against the size of workflows in the extended report [58].

4.3 RQ3: Defect Repair

Similar to defect detection, we focus on repairing two kinds of defects: syntactic errors and code injection vulnerabilities. We use *Accuracy@K* to assess the effectiveness of defect repair. As described in § 3.3, we do not include repair examples in our fine-tuning dataset. Hence defect repairs (T4 and T5) can be considered as unseen (but related) tasks for LLMs.

4.3.1 *Syntactic Error Repair (T4).*

We evaluate this task using three prompts (P1, P2, P3) with increasing detail (Table 3).
Calibration: The Table 10 shows the *Accuracy@K* of different LLMs on our calibration dataset (CAset). Across all prompts, higher temperatures yield better results. This is expected as higher temperature value allows LLMs to be more creative, consequently increasing the likelihood of generating repaired workflow. Listing 6 in our extended report [58] shows an example of a syntactically invalid workflow due to the use of an invalid step name (🐞). In this instance, setting a higher temperature value successfully corrected the syntactic error, whereas a lower temperature setting failed to do so.

Also, detailed prompts provide better results, as indicated by the increasing trend across P1 to P3. For simpler prompts, i.e., P1 and P2, fine-tuned variant of GPT-3.5 perform better on syntactic error repair tasks (unseen tasks) than the off-the-shelf variant. However, the case is different with CodeLlama and StarChat, where the fine-tuned variant performed poorly. These results demonstrate that fine-tuning GPT-3.5 on certain tasks helps in improving its effectiveness on other unseen but related tasks. However, this is not the case with other LLMs, where fine-tuned variants can perform poorly on unseen (but related) tasks. Intuitively, this makes sense as GPT-3.5 is trained on diverse datasets and has higher generalization capability. Whereas specialized LLMs (i.e., CodeLlama and StarChat) have less generalization capability. Our observations are in line with prior work [53], which showed that fine-tuned large models (e.g., GPT-3.5) generalize to unseen (but related) tasks. In contrast, smaller models (e.g., CodeLlama and StarChat) suffer as all model capacity is used for tasks used in fine-tuning.
Final Evaluation: The Figure 6 shows *Accuracy@K* of syntactic error fixing on the large dataset. We did not include the results for fine-tuned variants of CodeLlama and StarChat as they are extremely poor (i.e., < 40%). GPT-3.5 in 1-shot mode performs the best across all LLMs and their variants.

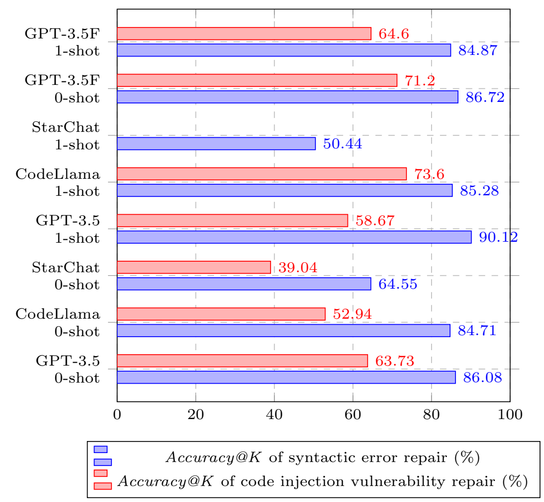

Figure 6: Final evaluation for defect repair.

4.3.2 *Code Injection Vulnerability Repair (T5).*

We evaluate this task using three prompts (P1, P2, P3) with increasing detail (Table 3).
Calibration: The Table 11 shows the *Accuracy@K* of different LLMs on our calibration dataset (CAset). It follows a similar pattern as repairing syntactic errors, i.e., higher temperatures yield better results across all prompts. Also, detailed prompts provide better results, as indicated by the increasing trend across P1 to P3. Fine-tuned variant of GPT-3.5 performs better on code injection vulnerability tasks (unseen tasks) than the off-the-shelf variant. However, the case is different with CodeLlama and StarChat, where the fine-tuned variant performs poorly.
Final Evaluation: Figure 6 shows the evaluation on the final large dataset. We did not include the results for StarChat in 1-shot mode and fine-tuned variants of CodeLlama and StarChat since they are extremely poor. CodeLlama in 1-shot mode performs the best across all LLMs and their variants.

Table 10: Accuracy@K of syntax error fixing on CASET.

Model	t	off-the-shelf						fine-tuned					
		0-shot			1-shot			0-shot			1-shot		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
GPT-3.5	0.1	46.50	50.00	80.50	51.27	56.85	82.23	65.00	65.50	82.50	70.53	59.26	80.65
	0.3	47.50	52.50	82.50	52.79	57.87	84.77	67.00	69.50	87.00	72.63	65.08	85.48
	0.5	48.50	52.50	86.00	54.31	59.90	90.86	67.50	71.00	89.00	74.21	68.78	86.56
	0.7	50.00	57.00	88.50	55.33	61.42	93.40	68.50	75.50	90.00	71.21	71.43	87.63
	0.9	53.00	58.00	**91.50**	56.85	66.50	**93.91**	73.00	75.50	**91.00**	76.32	73.54	**88.17**
CodeLlama	0.1	4.040	17.17	62.12	9.000	52.50	71.50	0.000	0.510	0.510	5.000	7.000	7.500
	0.3	3.540	36.87	87.88	11.50	35.00	88.50	0.000	0.510	0.000	11.50	10.50	12.00
	0.5	6.060	35.86	87.37	14.00	33.00	86.00	0.000	0.000	1.520	17.50	15.00	17.00
	0.7	8.590	40.91	**89.39**	18.00	33.00	86.50	2.530	1.520	2.530	27.50	23.00	25.50
	0.9	15.15	36.36	86.36	19.00	37.00	**92.50**	3.030	3.030	**3.540**	**35.50**	25.50	30.00
StarChat	0.1	44.44	49.49	60.10	42.00	43.50	45.00	0.000	0.000	1.010	0.000	0.500	0.000
	0.3	44.44	51.01	62.12	43.50	44.00	47.00	0.000	0.510	1.010	0.000	2.500	0.000
	0.5	44.95	52.02	64.14	43.50	44.50	48.50	0.000	0.510	1.010	0.000	5.000	0.000
	0.7	44.44	53.54	**64.65**	32.50	29.50	54.00	0.000	0.510	1.520	0.500	8.000	0.000
	0.9	42.42	52.02	**64.65**	42.00	36.50	**55.00**	1.520	0.000	**2.530**	1.000	**9.000**	0.500

Table 11: Accuracy@K of code injection vulnerability repair on CASET.

Model	t	off-the-shelf						fine-tuned					
		0-shot			1-shot			0-shot			1-shot		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
GPT-3.5	0.1	2.020	2.020	20.41	8.050	4.600	31.40	21.21	49.49	50.00	26.67	25.68	36.99
	0.3	3.030	6.060	26.53	12.50	5.620	41.86	38.38	52.53	56.12	42.67	31.08	49.32
	0.5	7.070	9.090	32.65	10.11	7.870	47.62	40.40	62.63	62.24	42.67	41.89	54.79
	0.7	6.060	8.080	43.88	13.79	6.980	**66.67**	49.49	63.64	69.39	46.67	45.95	**64.38**
	0.9	14.14	24.24	**44.90**	20.69	15.12	55.95	52.53	**69.70**	67.35	53.33	48.65	**64.38**
CodeLlama	0.1	36.00	48.00	26.00	33.00	62.00	41.00	0.000	0.000	0.000	0.000	0.000	1.000
	0.3	35.00	52.00	36.00	46.00	**73.00**	60.00	0.000	0.000	0.000	0.000	1.000	1.000
	0.5	35.00	42.00	38.00	46.00	72.00	67.00	0.000	0.000	0.000	0.000	1.000	1.000
	0.7	32.00	44.00	50.00	47.00	59.00	67.00	0.000	0.000	0.000	0.000	2.000	3.000
	0.9	45.00	**60.00**	45.00	48.00	57.00	68.00	0.000	**2.000**	0.000	4.000	4.000	**7.000**
StarChat	0.1	1.000	4.000	11.00	2.000	2.000	3.000	0.000	0.000	0.000	0.000	0.000	0.000
	0.3	2.000	8.000	17.00	3.000	3.000	5.000	0.000	0.000	0.000	0.000	0.000	0.000
	0.5	7.000	10.00	20.00	3.000	6.000	11.00	0.000	0.000	0.000	0.000	0.000	0.000
	0.7	9.000	20.00	29.00	7.000	5.000	13.00	0.000	0.000	0.000	0.000	0.000	0.000
	0.9	21.00	19.00	**37.00**	10.00	10.00	**19.00**	0.000	0.000	0.000	**2.000**	0.000	0.000

Summary of RQ3: LLMs perform well (at higher temperatures) in repairing syntactic errors but suffers at repairing code injection vulnerabilities. Fine-tuning CodeLlama and StarChat hurts their performance on unseen (but related) workflow tasks.

5 THREATS TO VALIDITY

We identified the following potential (generalizability) threats to the validity of our study.

- **Generalizability to Other Tasks:** We investigated three categories of tasks. However, there could be other related tasks (e.g., refactoring) on which the effectiveness of LLMs might differ. We tried to handle this in RQ3 (§ 4.3), where all the tasks are unseen but related.
- **Generalizability to Other LLMs:** We have investigated three LLMs, and the observations may not generalize to other LLMs that are architected differently. Our datasets and experimentation scripts will enable easy evaluation of any given LLM and compare against our results.
- **Generalizability to Other CI platforms:** We anticipate that our observations will generalize to other CI platforms as well because most of the CI platforms follow the same syntax (i.e., `YAML`) and have a similar structure [21].

6 CONCLUSION

We perform the first large-scale study to investigate the effectiveness of three state-of-the-art LLMs and their fine-tuned variants on five tasks related to GitHub workflows. We curated a set of ~400K workflows with various prompts with varying details across different tasks. Our study revealed various interesting findings and open problems in using LLMs for workflows. For instance, LLMs suffer at generating large and valid workflows. LLMs are not effective at repairing code injection vulnerabilities.

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation (NSF) under Grants CNS-2247686, Amazon Research Award (ARA) on "Security Verification and Hardening of CI Workflows" and Defense Advanced Research Projects Agency (DARPA) under contract numbers N6600120C4031 and N660012224037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, Amazon, or the United States Government.

REFERENCES

[1] 2023. actionlint. https://github.com/rhysd/actionlint.

[2] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On Hardware Security Bug Code Fixes by Prompting Large Language Models. *IEEE Transactions on Information Forensics and Security* 19 (2024), 4043–4057. https://doi.org/10.1109/TIFS.2024.3374558

[3] Simon Arvidsson and Johan Axell. 2023. Prompt engineering guidelines for LLMs in Requirements Engineering. (2023).

[4] Weiheng Bai, Qiushi Wu, Kefu Wu, and Kangjie Lu. 2024. Exploring the Influence of Prompts in LLMs for Security-Related Tasks. In *Workshop on Artificial Intelligence System with Confidential Computing (AISCC 2024)* (San Diego, CA). USA. https://dx.doi.org/10.14722/aiscc.2024.23015

[5] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Automatic Security Assessment of GitHub Actions Workflows. In *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 37–45.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[8] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 235–245. https://doi.org/10.1109/ICSME55016.2022.00029

[9] 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)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Sabit Ekin. 2023. Prompt engineering for ChatGPT: a quick guide to techniques, tips, and best practices. *Authorea Preprints* (2023). https://doi.org/10.36227/techrxiv.22683919.v2

[11] Ionut Daniel Fagadau, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2024. Analyzing Prompt Influence on Automated Method Generation: An Empirical Study with Copilot. *arXiv preprint arXiv:2402.08430* (2024).

[12] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-Based Line-Level Vulnerability Prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 608–620. https://doi.org/10.1145/3524842.3528452

[13] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. 2023. Security Weaknesses of Copilot Generated Code in GitHub. arXiv:2310.02059 [cs.SE]

[14] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How Far Have We Gone in Vulnerability Detection Using Large Language Models. arXiv:2311.12420 [cs.AI]

[15] GitHub Security Code Injection Finder [n. d.]. GitHub Security Code Injection Finder. https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-094/ExpressionInjection.ql.

[16] Yacong Gu, Lingyun Ying, Huajun Chai, Chu Qiao, Haixin Duan, and Xing Gao. 2023. Continuous Intrusion: Characterizing the Security of Continuous Integration Services. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1561–1577. https://doi.org/10.1109/SP46215.2023.10179471

[17] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[18] Adnan Khan. 2023. One Supply Chain Attack to Rule Them All – Poisoning GitHub's Runner Images. https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/.

[19] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. arXiv:2311.16169 [cs.CR]

[20] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. 2021. How Do Software Developers Use GitHub Actions to Automate Their Workflows?. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 420–431. https://doi.org/10.1109/MSR52588.2021.00054

[21] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the Security of Github CI Workflows. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2747–2763. https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev

[22] Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=KoFOg41haE Reproducibility Certification.

[23] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (Jan. 2023), 35 pages. https://doi.org/10.1145/3560815

[24] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, and Adam Roberts. 2023. The Flan Collection: Designing Data and Methods for Effective Instruction Tuning. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 22631–22648. https://proceedings.mlr.press/v202/longpre23a.html

[25] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[26] Meta. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[27] Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. 2023. ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6983–7000. https://www.usenix.org/conference/usenixsecurity23/presentation/muralee

[28] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. https://doi.org/10.1145/3597503.3639187

[29] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

[30] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

[31] OpenAI. 2022. *Introducing ChatGPT*. https://openai.com/blog/chatgpt

[32] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[33] OWASP. 2022. OWASP Top 10 CI/CD Security Risks. https://owasp.org/www-project-top-10-ci-cd-security-risks/.

[34] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318.

https://doi.org/10.3115/1073083.1073135

[35] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[36] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction Tuning with GPT-4. arXiv:2304.03277 [cs.CL]

[37] Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matt Jones, Alessandro Morari, and Ruchir Puri. 2023. Invited: Automated Code generation for Information Technology Tasks in YAML through Large Language Models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–4. https://doi.org/10.1109/DAC56929.2023.10247987

[38] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[39] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI EA '21)*. Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages. https://doi.org/10.1145/3411763.3451760

[40] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[41] Leonard Salewski, Stephan Alaniz, Isabel Rio-Torto, Eric Schulz, and Zeynep Akata. 2023. In-Context Impersonation Reveals Large Language Models' Strengths and Biases. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=CbsJ53LdKc

[42] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *The 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing @ NeurIPS 2019*. arXiv:1910.01108 http://arxiv.org/abs/1910.01108

[43] Yutaka Sasaki et al. 2007. The truth of the F-measure. *Teach tutor mater* 1, 5 (2007), 1–5.

[44] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE Computer Society, Los Alamitos, CA, USA, 23–30. https://doi.org/10.1109/APR59189.2023.00012

[45] John Stawinski. 2023. Playing with Fire – How We Executed a Critical Supply Chain Attack on PyTorch. https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/.

[46] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 481–496. https://doi.org/10.1145/3564625.3567985

[47] Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. 2023. Creating a Coding Assistant with StarCoder. *Hugging Face Blog* (2023). https://huggingface.co/blog/starchat.

[48] Ashok Urlana, Charaka Vinayak Kumar, Ajeet Kumar Singh, Bala Mallikarjunarao Garlapati, Srinivasa Rao Chalamala, and Rahul Mishra. 2024. LLMs with Industrial Lens: Deciphering the Challenges and Prospects–A Survey. *arXiv preprint arXiv:2402.14558* (2024).

[49] Pablo Valenzuela-Toledo and Alexandre Bergel. 2022. Evolution of GitHub Action Workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 123–127. https://doi.org/10.1109/SANER53432.2022.00026

[50] Juan David Velásquez-Henao, Carlos Jaime Franco-Cardona, and Lorena Cadavid-Higuita. 2023. Prompt Engineering: a methodology for optimizing interactions with AI-Language Models in the field of engineering. *DYNA* 90, 230 (Nov. 2023), 9–17. https://doi.org/10.15446/dyna.v90n230.111700

[51] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50 (2023), 911–936.

[52] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[53] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*. https://openreview.net/forum?id=gEZrGCozdqR

[54] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1282–1294. https://doi.org/10.1145/3597926.3598135

[55] Xuhai Xu, Bingsheng Yao, Yuanzhe Dong, Saadia Gabriel, Hong Yu, James Hendler, Marzyeh Ghassemi, Anind K. Dey, and Dakuo Wang. 2024. Mental-LLM: Leveraging Large Language Models for Mental Health Prediction via Online Text Data. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 8, 1, Article 31 (March 2024), 32 pages. https://doi.org/10.1145/3643540

[56] Yifei Xu, Yuning Chen, Xumiao Zhang, Xianshang Lin, Pan Hu, Yunfei Ma, Songwu Lu, Wan Du, Zhuoqing Mao, Ennan Zhai, and Dennis Cai. 2023. CloudEval-YAML: A Practical Benchmark for Cloud Configuration Generation. arXiv:2401.06786 [cs.DC]

[57] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We? *IEEE Transactions on Dependable and Secure Computing* (Aug. 2023), 1–18. https://doi.org/10.1109/TDSC.2023.3308897

[58] Xinyu Zhang, Siddharth Muralee, Sourag Cherupattamoolayil, and Aravind Machiry. 2024. On the Effectiveness of Large Language Models for GitHub Workflows.