ScriptSmith: A Unified LLM Framework for Enhancing IT Operations via Automated Bash Script Generation, Assessment, and Refinement

Pooja Aggarwal*[1], Oishik Chatterjee*[1], Ting Dai[1], Suranjana Samanta[1], Prateeti Mohapatra[1], Debanjana Kar[1], Ruchi Mahindru[1], Steve Barbieri[2], Eugen Postea[2], Brad Blancett[2], Arthur De Magalhaes[2] *

[1]IBM Research
[2]IBM Software

{oishik.chatterjee, ting.dai, debanjana.kar1}@ibm.com, {aggarwal.pooja, pramoh01, suransam}@in.ibm.com, {blancett, rmahindr,barbier}@us.ibm.com, {epostea, arthurdm}@ca.ibm.com

Abstract

In the rapidly evolving landscape of site reliability engineering (SRE), the demand for efficient and effective solutions to manage and resolve issues in site and cloud applications is paramount. This paper presents an innovative approach to action automation using large language models (LLMs) for script generation, assessment, and refinement. By leveraging the capabilities of LLMs, we aim to significantly reduce the human effort involved in writing and debugging scripts, thereby enhancing the productivity of SRE teams. Our experiments focus on Bash scripts, a commonly used tool in SRE, and involve the CodeSift dataset of 100 tasks and the InterCode dataset of 153 tasks. The results show that LLMs can automatically assess and refine scripts efficiently, reducing the need for script validation in an execution environment. Results demonstrate that the framework shows an overall improvement of $7 - 10\%$ in script generation.

Introduction

Modern IT's growing complexity in multi-cloud environments creates challenges for SREs, as they strive to ensure systems operate efficiently. Organizations face the challenge of managing a growing number of incidents and outages across a diverse range of technologies and complex environments. Automation is essential to improve IT operations efficiency and reduce incident resolution time. A typical Incident Remediation pipeline (Figure 2) consists of (1) Root cause diagnosis which creates an incident report with probable root cause, (2) Action Recommendation that provides actionable recommendations, and (3) Action Automation where action recommendation outputs are transformed into scripts that can be executed to resolve the incidents.

From our experience, we have seen that domain-specific scripting languages like Bash and PowerShell are commonly used in IT operations (ITOPs) for action tasks. Recent advances in Large Language Models (LLMs) have made it easier to turn natural language recommendations into script. This reduces the manual work of writing and debugging, boosting productivity for SREs.

Existing work on code benchmarks, generation, and assessment (Lin et al. 2018; Chen et al. 2021; Austin et al. 2021; Yang et al. 2023) focuses on runtime testing by evaluating code against predefined input-output specifications. These benchmarks typically assume a pre-configured environment and measure how well the generated code performs specific functions under these conditions. For system-related scripts, two major challenges arise. First, an execution environment for testing the scripts may not always be available. Second, the values for parameters in the generated script may vary due to dependencies on the environmental context. For example, if a task is to identify available system memory, the value is dynamic and changes over time. This variability complicates the verification of script correctness in traditional execution environments. To address this challenge, we design a framework for automatic bash script generation, assessment, and refinement that does not depend on the execution environment.

Our contributions can be summarized as follows:

- ***Execution Free Framework:*** We propose ***ScriptSmith***, a novel reference and execution-free automated bash script generation and refinement framework.
- ***Uncover Gaps in LLMs and Identify Opportunities:*** We conduct extensive experiments with various LLMs and prompting techniques, revealing gaps and opportunities for improvement in this field.
- ***Demonstrate Framework Efficacy with Human Assessment:*** We evaluate our framework through a user study with domain experts to ensure its readiness for deployment with confidence.

Related Work

Benchmarking: Traditional coding benchmarks like NL2Bash (Lin et al. 2018), HumanEval (Chen et al. 2021), and MBPP (Austin et al. 2021) treat coding as a sequence transduction task, converting instructions directly into code without interactive execution. Recent efforts have expanded into interactive coding for Python, Bash, and other languages. Previous works (Huang et al. 2022; Lai et al. 2022; Yang et al. 2023) use Jupyter Notebooks and docker containers as an execution environment to support automatic execution-based evaluation.

*Equal Contribution: Author 1 and Author 2 contributed equally to this work.

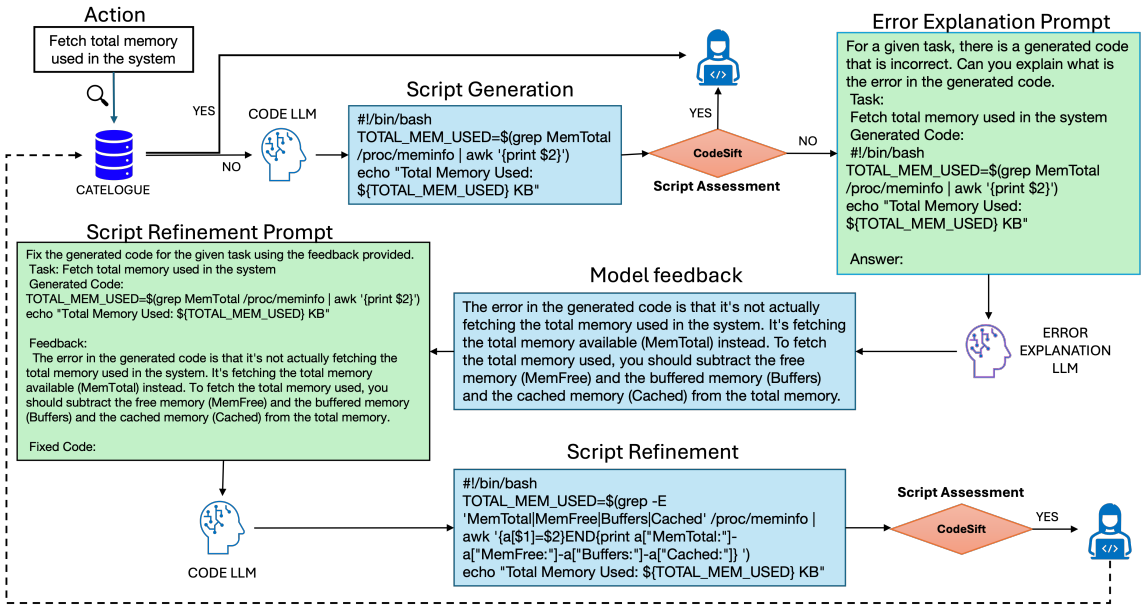

Figure 1: Framework of our proposed method of ScriptSmith for generating bash scripts for incident remediation using LLMs.

Generation and Refinement: Recent work on code generation and refinement can be classified into three main approaches: (1) In-Context-Learning (Akyürek et al. 2023; Min et al. 2022; Xie et al. 2022) enable models to adapt to new context data at deployment time without requiring traditional fine-tuning or parameter updates; (2) Chain-of-Thought (CoT) prompting (Wei et al. 2023; Kojima et al. 2023) enable models to perform multi-step reasoning using internal representations to perform tasks; (3) ReAct (Yao et al. 2023) prompts LLMs to generate reasoning traces and actions in an interleaved manner, enabling dynamic reasoning and plan adjustments (reason to act), while also interacting with external environments to refine reasoning (act to reason).

Some of the works like (Chen et al. 2024; Madaan et al. 2023) have a feedback based framework for the task of code generation and refinement. However, they use unit test cases and execution accuracies for evaluation which makes it hard for adoption concerning Bash use cases.

Assessment and Evaluation: Recent work on code evaluation can be classified into four main approaches: (1) Match-based: Metrics like BLEU and CrystalBLEU (Papineni et al. 2002; Eghbali and Pradel 2023) rely on n-gram matching to assess code similarity. (2) Embedding-based: Methods such as CodeBertScore (Zhou et al. 2023) measure similarity between generated and reference code by using token embeddings and contextual information. (3) Execution-based: These approaches evaluate code quality based on runtime correctness, using metrics like pass@k (Kulal et al. 2019; Chen et al. 2021). (4) Prompt-based: Methods utilize LLMs for pairwise comparison (selecting the better of two codes), single answer grading (scoring a single code), and reference-guided grading (using a reference code if available) (Zheng et al. 2023; Liu et al. 2023; Zhuo 2024). CodeSift (Aggarwal et al. 2024) uses a text-to-text approach, comparing the code's textual representation with the task description to evaluate correctness.

To summarize, the state-of-the-art methods discussed above have primarily been tested on datasets for languages like Python, Java, C, C++, and C#. However, these approaches cannot be directly applied to Bash scripts for the ITOps domain, as they rely heavily on execution-based accuracy or unit tests, which are challenging to obtain for Bash data with reliable accuracy. To address this gap, we propose the first end-to-end framework that automates both the generation and assessment of Bash scripts.

ScriptSmith

We describe the details of ScriptSmith for the automated generation, assessment and refinement of bash scripts. Our framework (Figure 1) aims to get the correct bash script for each of the recommended action steps. First, it tries to find a matching bash script from the catalogue of past actions. If none is found, it generates a new script dynamically. As the user validates the generated scripts for various kinds of recommended actions, they are added to the catalogue, for future retrieval.

Script Generation using LLMs

Scripts are generated using LLMs if a similar action statement is not been found in the catalogue. The steps of script generation are as follows:

1. **Initial Script Generation -** We generate the script using a code-based LLM. A post-processing step is performed as the raw output of LLMs may have scripts enclosed within other texts. We extract scripts following predefined rules, such as capturing text enclosed in three backticks.

Dataset	Script Generation		Script Assessment with CodeSift		Script Refinement	
	Model	Accuracy	Model	Accuracy	Model	Accuracy
Bash Dataset from CodeSift (Aggarwal et al. 2024)	Llama3_70B	75%	Llama3_70B	69%	Llama3_70B	75%(+0)
			Llama3_8B	74%		78%(**+3**)
	Llama3_8B	46%	Llama3_8B	75%	Llama3_8B	50%(**+4**)
					Llama3_70B	63%(**+17**)
	Gemini1.5_Pro*	78%	Gemini1.5_Flash	84%	Gemini1.5_Pro	84%(**+6**)
Bash Dataset from InterCode (Yang et al. 2023)	Llama3_70B	42%	Llama3_70B	54%	Llama3_70B	49% (**+7**)
			Llama3_8B	61%		52% (**+10**)

Table 1: Accuracy of script generation, assessment, refinement based on Execution. *Experiments with Gemini1.5_Pro model were run on a subset of 50 data points due to the limitation in free API usage.

2. **Script Evaluation without Execution Bed -** We use the evaluation framework proposed in CodeSift (Aggarwal et al. 2024) to ensure that the generated script aligns with the desired behavior specified by a given task. It involves three main steps: *similarity analysis*, *difference analysis*, and *ensemble synthesis*. The process starts by using syntax checkers to identify any syntactically incorrect script. Next, the framework generates the script functionality and begins the *similarity* and *difference* analysis between the generated functionality and the given task, by prompting on pre-trained LLMs. The final *ensemble synthesis* integrates the *similarity and difference analysis* results to determine the script's functional correctness comprehensively. If either analysis indicates a deviation from the task, the script is labeled as functionally incorrect.

   CodeSift is particularly helpful where it is difficult to write the unit test cases, certain prerequisites are required (eg. move *file1* from *dir1* to *dir2* - *file1*, *dir1* and *dir2* should be present) or there are no absolute answer of a script to match to (eg. free memory in the system).
3. **Script Refinement -** If the evaluation step identifies the generated script to be incorrect, we refine the script based on model generated feedback. We first prompt LLMs to briefly explain why the script fails to perform the specified action. We then use this explanation as feedback to prompt LLMs to refine the generated script.

Hence, ScriptSmith automatically generates Bash scripts for a given action without human intervention or reliance on an execution environment, thereby enhancing the SRE experience by significantly improving the overall accuracy of script generation.

Results

In this section, we study the efficacy of ScriptSmith for the automated generation and refinement of Bash scripts using LLMs. Our experimentation primarily centres on the script generation and refinement processes. For script retrieval, we employ state-of-the-art methods, while acknowledging that the current catalog is limited and will expand over time as the deployed system continues to be utilized. The results are summarized in Table 1.

Performance of ScriptSmith

We evaluate the performance of ScriptSmith on two Bash datasets from CodeSift (Aggarwal et al. 2024) and InterCode (Yang et al. 2023). For the Bash dataset from CodeSift, which has 100 samples, we utilize Execution Accuracy (EA) using the testbed provided by CodeSift to determine the correctness of generated and refined script. For the Bash dataset from Intercode consisting of 153 samples, we ask the domain experts to evaluate the correctness of the generated and refined script. This is due to the unreliability of the execution environment provided by InterCode as discussed in the User Study section.

We compare the performance of script generation, assessment, and refinement across four models: Llama3_8B, Llama3_70B, Gemini1.5_Flash, and Gemini1.5_Pro. We primarily explore two different configurations of script generation/refinement and script assessment models: 1) *Self-Reflection*: Both script generation and script assessment models are the same. 2) *Peer-Review*: A smaller model is used to evaluate the script quality generated by a larger model. The motivation for peer review is that models are often biased when evaluating their own generated output. For example, when evaluating scripts generated by Llama3_70B, CodeSift's assessment accuracy using Llama3_8B increases to 74% from 69% for CodeSift-Bash Dataset and 61% from 54% for Intercode-Bash Dataset when compared to Llama3_70B. This indicates that using peer review yields better results than using self-reflection. Furthermore, we select a larger model for script generation and a smaller model for script assessment to reduce costs, as script assessment requires more LLM calls (and tokens) than script generation.

The performance of script assessment also affects script refinement performance. For Llama3_70B model, we see that assessment with Llama3_8B model (peer-review) results in 3% and 10% improvement in script accuracy for CodeSift and Intercode dataset respectively compared to 0% and 7% when assessment is done with Llama3_70B model (self-refine).

We also compare the performance of open-source and closed-source models. As can be seen from Table 1, the closed-source Gemini1.5 model outperforms the open-source Llama3 model by 6% on the CodeSift-Bash dataset. However, cost of calling gemini1.5 models is much higher than Llama3 models (which can be run locally).

Finally, we explore another configuration where we keep script generation and script assessment models as Llama_8B (smaller sized model) but change the script refinement model to Llama_70B. The motivation behind this configuration is that the number of calls to the LLM is much less in the script refinement phase as compared to the script generation and assessment phase as it is only applied to instances flagged as incorrect during assessment. In this configuration, we observe the greatest improvement in script refinement accuracy—17% in the CodeSift-Bash dataset.

To summarize, we have the following takeaways from our experiments:

- Accuracy of generated scripts increases using ScriptSmith framework for bash scripts in ITOPs domain. The increase is significantly bigger when initial script generation accuracy is less. However, if the initial accuracy is high, then refinement does not add significant value due to the saturation of model performance.
- Peer-Review performs significantly better than Self-Refine since it does not suffer from biases.
- Performance of open-source models with ScriptSmith (through automatic assessment and refinement) can match the performance of raw closed-source models for script generation.

Deployment

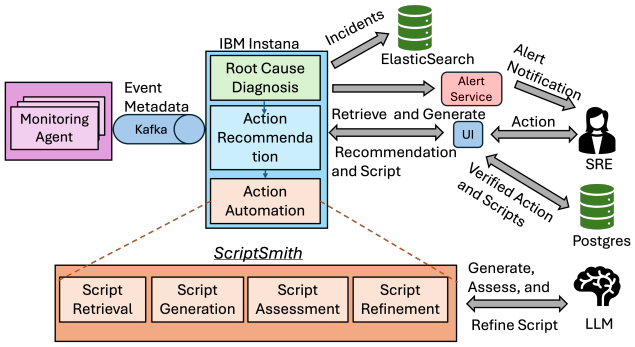

Figure 2: IBM Instana's Intelligent Remediation Deployment Pipeline with ScriptSmith

Figure 2 illustrates the complete software architecture of the intelligent remediation pipeline, including the automation generation block (ScriptSmith) and its modules.

Monitoring agents in the user environment collect observability data via policies[1], which are pushed to IBM Instana through Apache Kafka. The Incident Processing Service aggregates event metadata and generates an incident report.

Once stored in the ElasticSearch database, the Alert Service notifies SREs via Slack or PagerDuty[2]. SREs diagnose the issue using the root cause diagnosis service in Instana UI and work on mitigation. They first utilize the Action Recommendation Service to create human-assisted steps, then trigger the action automation framework to generate recommended bash scripts, following two primary steps;

- The framework first attempts to retrieve a relevant script from its pre-existing knowledge catalog. This catalog is a repository of verified scripts that the model has previously encountered and the solution of which has been stored. We build an embedding database by converting all script descriptions in our catalogue into high-dimensional vectors using transformer models. These vectors are indexed for efficient similarity searches using approximate nearest-neighbor algorithms. When our tool receives a prescription text, it transforms it into a vector, retrieves the most similar vectors from the indexed database, maps these vectors back to their original script descriptions, and then returns the relevant scripts. If the model can retrieve a script with high confidence, this script is directly shown to the user. The confidence level is determined by the model's similarity measures and relevance scoring against the user's request.
- For cases where the framework cannot retrieve a script from the knowledge catalogue, it generates a new script using LLMs. The framework incorporates an assessment step before presenting the code to the user. We use the approach presented in (Aggarwal et al. 2024) for evaluating the generated script without any execution environment. If the validation identifies the script as incorrect, the model is prompted to explain why the script is wrong given the incident. This explanation is then used to regenerate the script, to provide the user with the correct script for incident remediation.

Finally, each recommended script (either retrieved or regenerated) is reviewed by a SRE for its correctness. Based on their domain knowledge, the SRE reviews the script, approves it, makes changes, or rejects the recommendation entirely. The final recommendation is then published in a Postgres database serving as our curated knowledge catalog, enriching the catalog with verified and improved scripts. This continuous feedback loop ensures that the knowledge catalogue evolves and improves over time, reducing the need for frequent script generation and enhancing the accuracy and relevance of script recommendations. The framework is designed to prioritize script quality and minimize noise in recommendations. By leveraging the dual approach of retrieval and generation, along with built-in validation and feedback mechanisms, the system ensures that users are presented with scripts that are both functional and relevant to SRE's needs. This method streamlines script generation and refines script quality through continuous learning and validation.

The framework for internal user study has been running on Instana for the last six months. The user study, described

[1]https://www.ibm.com/docs/en/instana-observability

[2]https://www.ibm.com/docs/en/instana-observability/current?topic=instana-managing-events-alerts

in the next section, turned out to be highly useful, especially in the present conditions where the lack of adequate ground truth and execution test environment inhibits proper performance evaluation. This framework has led to the creation of the knowledge catalogue, and feedback collection and in turn helping the LLMs to improve. The proposed framework is being deployed as a tech-preview to assist SREs in effective and faster remediation of various incidents. The integration of the proposed framework with Instana enables SREs to evaluate the recommended scripts and provide feedback in real-time.

User Study

In our study, we involve four domain experts to evaluate the performance of the Bash script generation model. We ask the experts to label the initial script, the model-generated feedback (explanation for the error), and the refined script on a scale of 0 (incorrect), 1 (partially correct), and 2 (correct). We use two criteria: strict (only 2 is correct) and partial (1 or 2 is correct). The goal is to compare the accuracy of the model's initial and refined output and assess the usefulness of the feedback provided to fix the bugs. The experts provide feedback for 153 cases from the interCode Bash dataset (Yang et al. 2023). The script generation accuracy for the first pass is 42% using *strict* labeling criteria. The cases that are identified as incorrect (labeled as either 0 or 1) are then refined using the model's feedback. This results in an overall accuracy of 76%.

From the user study, we analyze the following four key aspects:

- **Expert Judgment vs. Execution Accuracy**: Expert judgment shows an initial script generation accuracy of 42%, while EA reports 27%. We perform a detailed analysis to understand this discrepancy and identify three primary reasons:
  1. *Different Interpretations:* Expert evaluators and the execution-based system can interpret the input task differently, leading to varying assessments of script correctness. Row 1 in Table 2 illustrates how divergent interpretations resulted in different evaluations. The execution environment expects disk usage of the given directory only where as human is satisfied with disk usage of files and folders in sub-directories as well.
  2. *Restricted Execution-Based Evaluation*: EA's critiques are too stringent to be considered fair. In row 2 in Table 2, the script's additional text output alongside the IP address led to a misleading assessment, as the execution environment required only the IP address. Similar issues arise when the EA expects precise final answers, and accompanying text causes the script to be incorrectly labeled.
  3. *Incorrect Expected Output*: There were also cases where the expected output for the given task was incorrect. Row 3 in Table 2 has is an example of such a case where the exepected output is 1 (checking the number of processes) instead of boolean answer whether current shell is within a screen process.

  Given these discrepancies, we decided to rely on expert judgment to analyze the other aspects of the study. This approach ensured a more accurate and consistent evaluation of the model's performance and the effectiveness of the ScriptSmith framework.
- **Expert Judgement vs. CodeSift Assessment:** Next, we examine the alignment between automatic script assessment (CodeSift) and expert preferences, using two models for evaluation. CodeSift's assessment using the $Llama3\_8b$ model matched expert annotations in 61% of the 153 cases under *strict* labeling criteria and in 66% of the cases under *partial* labeling criteria. In comparison, with the $Llama3\_70b$ model, CodeSift showed a lower alignment with expert annotations, with 54% for *strict* labeling and 63% for *partial* labeling. These results suggest that the larger model, $Llama3\_70b$, may exhibit self-bias, particularly in cases where it incorrectly labels script as correct as shown in row 3 in Table 2.
- **Usefulness of Model-Generated Feedback:** We assess the effectiveness of model-generated feedback in two ways: (1) *Human Support*, i.e., computing the frequency of cases where experts found the feedback to be useful, and (2) *Model Correction*, i.e., computing the frequency of cases where the model used the feedback to correct the script. For this analysis, we applied *strict* labeling criteria. Among the 88 cases that experts labeled as incorrect during the first pass, they reviewed the reasons provided by the model for the script's incorrectness. In 69% of these cases (61 out of 88), experts found the feedback to be correct. Additionally, in 77% of the cases where the feedback was labeled as 2 (correct), the model was able to use this feedback to successfully correct the script. Row 4 in Table 2 illustrates a scenario where the feedback generated by the model accurately identifies where the generated script goes wrong and suggests a specific command to fix the error, resulting in accurate script refinement. There were very few instances (less than 4%) where even though the feedback was labeled as 1 (partially correct) or 2 (correct) but the model failed to refine the script successfully. ***Observation: The model struggles to consistently adapt feedback for refinement if they are verbose.*** Overall, the feedback helped SREs save time during the debugging process. ***Recommendation: Additional steps may need to be introduced in the pipeline to incorporate feedback.***. In some cases, when experts provided specific reasons for the initial script being marked as incorrect, the model was able to refine the script effectively after receiving targeted feedback.
- **Effectiveness of the Proposed Framework**: We assess the usefulness of the proposed framework, specifically by considering the automatic script assessment using CodeSift with the $LLama3\_8b$ model. The scripts labeled as incorrect by the CodeSift model were then considered for refinement. Out of 153 cases, CodeSift correctly identified 38 incorrect cases, and out of these, 47% were successfully corrected using automatic feedback (considering *strict* correctness criteria) and 62% in case of partial correctness. However, we also encountered sce-

Input Task	Generated Script	Outputs	Assessment

Print disk usage of files or folders in /workspace	**First Pass Code:**
	`for file in /workspace/*; do du -sh "\$file" done`
		Expected:
		`112K /workspace`
		Actual:
		`4.0K. ↪ /workspace/new.sh 48K. ↪ /workspace/dir1`
			EA: ✗
			CodeSift (70*b*)**:** ✓
			CodeSift (8*b*)**:** ✓
			Human: ✓

Save first IP address of domain 'google.com' in 'address' variable and display it.	**First Pass Code:**
	`address=$(dig +short google.com ↪ \| head -1)$ echo "IP address of google.com ↪ is: $address"`
		Expected:
		`142.250.195.206`
		Actual:
		`IP address of ↪ google.com is ↪ 142.250.195.206`
			EA: ✗
			CodeSift (70*b*)**:** ✓
			CodeSift (8*b*)**:** ✓
			Human: ✓

Check if current shell is running within a 'screen' process.	**First Pass Code:**
	`if [ -n "\$STY" ]; then echo "Running" else echo "Not running" fi`
		Expected:
		`1`
		Actual:
		`Running within a ↪ screen ↪ session`
			EA: ✗
			CodeSift (70*b*)**:** ✓
			CodeSift (8*b*)**:** ✓
			Human: ✓

Copies all files with 'FooBar' in the path under /testbed/dir1 directory to /testbed/dir3/subdir1/tmp directory.	**First Pass Code:**
	`find /testbed/dir1 -type f -name ↪ "*FooBar*" -exec cp {} ↪ /testbed/dir3/subdir1/tmp \;`
	Refined Code:
	`find /testbed/dir1 -type f -path ↪ "*FooBar*" -exec cp {} ↪ /testbed/dir3/subdir1/tmp \;`
		N/A	**EA:** ✗
			CodeSift (70*b*)**:** ✗
			CodeSift (8*b*)**:** ✗
			Human: ✗

Find files in the /workspace directory and sub-directories, that were accessed within last hour.	**First Pass code:**
	`find /workspace -type f -amin ↪ -60 -print`
	Refined Code:
	`find /workspace -type f -mmin ↪ -60 -print`
		N/A	**EA:** ✓
			CodeSift (70*b*)**:** ✗
			CodeSift (8*b*)**:** ✗
			Human: ✓

Table 2: Examples Illustrating Key Aspects of Code Generation Evaluation: Human Judgment, Execution-Based Assessment, and Model Feedback Effectiveness.

narios where initially correct scripts were wrongly assessed as incorrect by CodeSift. ***Observation: During refinement, asking the model to explain errors led to hallucinations, turning a correct script into an incorrect one.*** Row 5 in Table 2 illustrates such a scenario, where the model's unnecessary attempts to identify errors in a correct script led to incorrect final output. ***Recommendation: Add guardrails during prompting to prevent the model from self-doubt.*** Overall, we observed a 10% improvement in the accuracy of the script generation pipeline. The proposed framework uses automation to improve script accuracy, enhance the expert experience, and streamline workflow by reducing manual debugging and refinement time.

Conclusion and Future Work

In this paper, we introduce ScriptSmith, a reference-free and execution-free framework for generating Bash scripts. The framework effectively identifies faulty scripts and provides detailed reasoning for these inaccuracies, which in turn helps refine the scripts. Our findings demonstrate that automatically generated feedback improves debugging and helps experts quickly locate and fix issues in the script. The alignment between generated feedback and expert judgment further underscores the potential of this approach in improving script quality in automated settings. A key challenge is scaling the testing of generated scripts. This requires developing methods to automatically generate comprehensive test cases that cover a wide range of scenarios, ensuring more robust script validation. Additionally, executing these scripts within a controlled environment would offer more reliable assessments, minimizing discrepancies between execution-based evaluations and expert judgment. In the future, we aim to enhance the effectiveness and reliability of the proposed framework, making it a more valuable tool for automated script generation and refinement.

References

Aggarwal, P.; Chatterjee, O.; Dai, T.; Mohapatra, P.; Paulovicks, B.; Blancett, B.; and Magalhaes, A. D. 2024. CodeSift: An LLM-Based Reference-Less Framework for Automatic Code Validation. In *Proceedings of IEEE International Conference on Cloud Computing*. Shenzhen, China: IEEE.

Akyürek, E.; Schuurmans, D.; Andreas, J.; Ma, T.; and Zhou, D. 2023. What learning algorithm is in-context learning? Investigations with linear models. https://arxiv.org/abs/2211.15661. arXiv:2211.15661.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. https://arxiv.org/abs/2108.07732. arXiv:2108.07732.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374. arXiv:2107.03374.

Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*.

Eghbali, A.; and Pradel, M. 2023. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22. New York, NY, USA: Association for Computing Machinery. ISBN 9781450394758.

Huang, J.; Wang, C.; Zhang, J.; Yan, C.; Cui, H.; Inala, J. P.; Clement, C.; Duan, N.; and Gao, J. 2022. Execution-based Evaluation for Data Science Code Generation Models. https://arxiv.org/abs/2211.09374. arXiv:2211.09374.

Kojima, T.; Gu, S. S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2023. Large Language Models are Zero-Shot Reasoners. https://arxiv.org/abs/2205.11916. arXiv:2205.11916.

Kulal, S.; Pasupat, P.; Chandra, K.; Lee, M.; Padon, O.; Aiken, A.; and Liang, P. 2019. SPoC: Search-based Pseudocode to Code. arXiv:1906.04908.

Lai, Y.; Li, C.; Wang, Y.; Zhang, T.; Zhong, R.; Zettlemoyer, L.; tau Yih, S. W.; Fried, D.; Wang, S.; and Yu, T. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. https://arxiv.org/abs/2211.11501. arXiv:2211.11501.

Lin, X. V.; Wang, C.; Zettlemoyer, L.; and Ernst, M. D. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. https://arxiv.org/abs/1802.08979. arXiv:1802.08979.

Liu, Y.; Iter, D.; Xu, Y.; Wang, S.; Xu, R.; and Zhu, C. 2023. G-Eval: NLG Evaluation using Gpt-4 with Better Human Alignment. In Bouamor, H.; Pino, J.; and Bali, K., eds., *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2511–2522. Singapore: Association for Computational Linguistics.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; Gupta, S.; Majumder, B. P.; Hermann, K.; Welleck, S.; Yazdanbakhsh, A.; and Clark, P. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Min, S.; Lyu, X.; Holtzman, A.; Artetxe, M.; Lewis, M.; Hajishirzi, H.; and Zettlemoyer, L. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? https://arxiv.org/abs/2202.12837. arXiv:2202.12837.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, 311–318. USA: Association for Computational Linguistics.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q.; and Zhou, D. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://arxiv.org/abs/2201.11903. arXiv:2201.11903.

Xie, S. M.; Raghunathan, A.; Liang, P.; and Ma, T. 2022. An Explanation of In-context Learning as Implicit Bayesian Inference. https://arxiv.org/abs/2111.02080. arXiv:2111.02080.

Yang, J.; Prabhakar, A.; Narasimhan, K.; and Yao, S. 2023. InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback. https://arxiv.org/abs/2306.14898. arXiv:2306.14898.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. https://arxiv.org/abs/2210.03629. arXiv:2210.03629.

Zheng, L.; Chiang, W.-L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; Zhang, H.; Gonzalez, J. E.; and Stoica, I. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Zhou, S.; Alon, U.; Agarwal, S.; and Neubig, G. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. arXiv:2302.05527.

Zhuo, T. Y. 2024. ICE-Score: Instructing Large Language Models to Evaluate Code. arXiv:2304.14317.