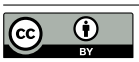
Bash-Commenter: Leveraging Syntax-Aware Preference Optimization to Reinforce Large Language Model for Bash Code Comment Generation

LEI YU*, Institute of Software, Chinese Academy of Sciences, China
JINGYUAN ZHANG*, Institute of Software, Chinese Academy of Sciences, China
XIN WANG*, Institute of Software, Chinese Academy of Sciences, China
LI YANG†, Institute of Software, Chinese Academy of Sciences, China
FENGJUN ZHANG†, Institute of Software, Chinese Academy of Sciences, China
PENG WANG*, Institute of Software, Chinese Academy of Sciences, China
JIA XU*, Institute of Software, Chinese Academy of Sciences, China
JIAJIA MA, Institute of Software, Chinese Academy of Sciences, China

Bash script comprehension is a significant challenge in Linux environments due to Bash's syntactic freedom and complex command structures. Despite its critical role in system administration and development, Bash scripts often lack adequate comments, hindering code readability and maintainability. Existing approaches to automated Bash comment generation face two main challenges: (1) Limited training datasets that inadequately represent real-world Bash usage patterns, particularly for complex multi-line scripts; and (2) Insufficient understanding of Bash-specific concepts by Large Language Models (LLMs). Our empirical analysis shows that even after standard training, LLMs still struggle to precisely understand complex Bash command semantics, leading to inaccurate comments. To address these challenges, we propose Bash-Commenter, an advanced comment generation method based on LLaMA-3.1-8B. First, to overcome data limitations (Challenge 1), we construct a comprehensive dataset of complex, multi-line Bash scripts with high-quality comments. Second, to enhance semantic understanding (Challenge 2), we conduct Continual Pre-training (CPT) on large-scale Bash script data, followed by Supervised Fine-tuning (SFT) on our annotated dataset, strengthening the model's foundational knowledge of Bash syntax and semantics. Finally, to resolve the subtle semantic errors that persist, we introduce Syntax-Aware Preference Optimization (SAPO). This method automatically constructs preference pairs by applying single, atomic operations (e.g., modifying a command option or removing an argument) to a script's Abstract Syntax Tree (AST), creating minimal pairs of correct and subtly incorrect scripts. This final optimization stage enables fine-grained command semantics learning and context-dependent quality assessment, significantly improving comment accuracy. We evaluate Bash-Commenter on single-line Bash commands and multi-line Bash scripts. Our method outperforms state-of-the-art baselines, achieving

*Affiliated with University of Chinese Academy of Sciences, Beijing, China.
†Li Yang and Fengjun Zhang are the corresponding authors.

Authors' Contact Information: Lei Yu, Institute of Software, Chinese Academy of Sciences, Beijing, China, yulei2022@iscas.ac.cn; Jingyuan Zhang, Institute of Software, Chinese Academy of Sciences, Beijing, China, zhangjingyuan2023@iscas.ac.cn; Xin Wang, Institute of Software, Chinese Academy of Sciences, Beijing, China, wangxin@iscas.ac.cn; Li Yang, Institute of Software, Chinese Academy of Sciences, Beijing, China, yangli2017@iscas.ac.cn; Fengjun Zhang, Institute of Software, Chinese Academy of Sciences, Beijing, China, fengjun@iscas.ac.cn; Peng Wang, Institute of Software, Chinese Academy of Sciences, Beijing, China, wangpeng232@mails.ucas.ac.cn; Jia Xu, Institute of Software, Chinese Academy of Sciences, Beijing, China, xujia23@mails.ucas.ac.cn; Jiajia Ma, Institute of Software, Chinese Academy of Sciences, Beijing, China, majiajia@iscas.ac.cn.



ACM 2994-970X/2026/7-ARTFSE095
https://doi.org/10.1145/3808102

33.40% BLEU-4, 58.26% METEOR, and 57.03% ROUGE-L for 1,064 single-line commands, and 22.15% BLEU-4, 43.89% METEOR, and 32.80% ROUGE-L for 1,046 multi-line scripts. Moreover, human evaluation and LLM evaluation demonstrate the superior quality of comments generated by Bash-Commenter in terms of correctness, completeness, and naturalness.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Domain specific languages**; • **Computing methodologies → Natural language generation**.

Additional Key Words and Phrases: Bash Scripts, Large Language Models, Syntax-Aware Preference Optimization, Comment Generation

ACM Reference Format:
Lei Yu, Jingyuan Zhang, Xin Wang, Li Yang, Fengjun Zhang, Peng Wang, Jia Xu, and Jiajia Ma. 2026. Bash-Commenter: Leveraging Syntax-Aware Preference Optimization to Reinforce Large Language Model for Bash Code Comment Generation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE095 (July 2026), 24 pages. https://doi.org/10.1145/3808102

1 Introduction

Bash, the default domain-specific language (DSL) in Linux systems for tasks like file management, network control, and process management [32], remains indispensable for Linux system development and maintenance despite its narrower application range compared to general-purpose programming languages (GPPLs) [23]. Known for its high flexibility and syntactical freedom, Bash is a powerful tool, but these very characteristics also pose significant challenges for code comprehension [23]. This challenge is particularly pronounced for developers unfamiliar with Bash, who often struggle to understand the functionality and semantics of Bash scripts. Based on our statistical analysis of Stack Overflow, as of March 10, 2025, there are 93,046 Q&A posts related to the keyword “shell” and 156,721 Q&A posts related to the keyword "bash" highlighting the urgent need for tools to assist in understanding Bash scripts [5, 6]. Studies suggest that developers spend up to 59% of their time on code comprehension, and high-quality comments are critical for improving code readability and maintainability [30, 49]. Many developers often fail to provide sufficient comments due to oversight or time constraints, which leads to missing or outdated annotations [23, 39]. Longitudinal analysis of Stack Overflow activity (1134 days post-ChatGPT release [40] vs 3000 day baseline [41], collected January 7, 2026) shows Bash retains 14.39% of baseline volume, 6.54% relatively higher than SQL and 18.07% relatively higher than Java, indicating LLMs struggle more with Bash-specific syntax than general programming. This persistent demand stems from Bash’s safety-critical nature: unlike GPPLs, Bash’s syntactic freedom allows direct OS manipulation where semantic misinterpretation (e.g., quoting in `rm -rf "$VAR"` vs `rm -rf $VAR`, block sizes in `dd` commands) can cause irreversible data loss. High-quality Bash comments serve four critical applications: (1) DevOps maintenance (where developers spend 59% of time on comprehension [49]), (2) security auditing to identify risky operations (recursive deletion, permission changes, network operations), (3) developer assistance for shell-related posts, and (4) legacy code modernization: comprehending undocumented infrastructure scripts for safe migration to containerized platforms or rewriting in modern languages. This challenge reflects a broader issue for domain-specific languages (DSLs). Unlike general-purpose programming languages (GPPLs) like Java and Python where automated comment generation is well-studied[9, 30], DSLs such as Bash remain largely unexplored in this context [54, 55, 58], despite an urgent need to address this gap.

Recent work on Bash code comment generation has explored advanced techniques to enhance the quality and relevance of generated comments. Yu et al. [54] proposed the BASHEXPLAINER framework, which employs a two-stage training strategy. In the first stage, CodeBERT is used to semantically encode Bash code, while the second stage integrates an information retrieval module with a deep learning-based generator to produce comments by leveraging semantic and lexical

similarity. Shen et al. [39] introduced Bash2Com, which further enhanced comment generation through adversarial training (NP-GD) and a semantic-aware module (MASA). More recently, Zhang et al. [70] proposed HBCom, which leverages a Heterogeneous Information Graph (HIG) to integrate syntactic and semantic features, providing a more comprehensive understanding of Bash commands.

Recently, Large Language Models (LLMs) have attracted significant interest due to their exceptional ability to understand and generate natural language. They have shown outstanding efficacy across a range of complex applications, such as code comprehension and generation [4, 25, 30, 38, 48, 59, 64], as well as smart contract auditing and generation [55–58, 61–63]. General LLMs show promise in adapting to new patterns [31]. However, they often encounter significant difficulties in handling Bash scripting-specific concepts and command execution nuances. As shown in Fig. 1, when comparing comments from a reference description, a general LLM (LLaMA-3.1-8B-Instruct), and a specialized model after continual pre-training and supervised fine-tuning (Bash-Commenter with CPT+SFT), both models still exhibit critical misunderstandings. The general LLM fundamentally misinterprets multiple aspects of the command, including the `awk` field splitting mechanism, the purpose of `uniq -i`, and the recursive nature of `find`. Bash-Commenter (CPT+SFT), while improving in certain aspects such as correctly identifying the main purpose of finding duplicate files (`uniq -d`) and accurately describing the output format (`awk`), still presents significant deficiencies. It continues to omit the recursive traversal functionality of the `find` command, fails to accurately express the role of case-insensitive matching (`uniq -i`), and omits the field splitting functionality of `awk`. These persistent errors demonstrate that traditional training approaches (CPT+SFT) are insufficient for developing models with accurate understanding of complex Bash commands.

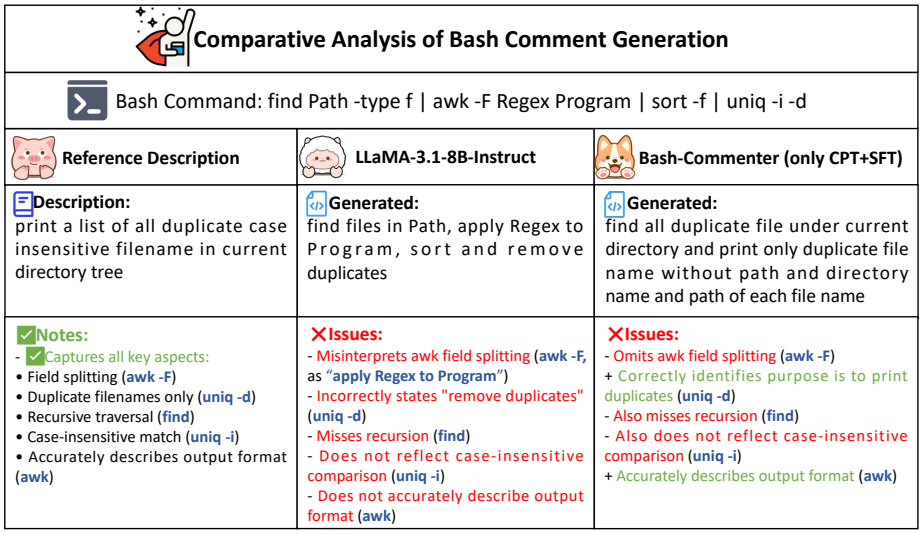


Fig. 1. A motivation example to illustrate the limitations of LLM-based Bash comment generation, motivating the need for SAPO.

To address these challenges, we propose **Bash-Commenter**, a specialized model based on the LLaMA-3.1-8B model, developed through a comprehensive three-stage pipeline. **First**, to address the data scarcity and quality issues (as detailed in Section 2.2), we curate a new high-quality dataset sourced from the large-scale empirical study by Dong et al. [7]. Unlike previous datasets that mainly

consist of single-line commands, our dataset features more complex and multi-line scripts with detailed annotations, better reflecting the practical needs of Bash code comment generation in real-world scenarios. **Second**, to overcome the limitations of general LLMs in understanding Bash-specific syntax and semantics (as empirically demonstrated in Section 2.2), we utilize large-scale Bash script data for domain-specific Continual Pre-training (CPT), followed by Supervised Fine-tuning (SFT) on our curated dataset. This two-step process enhances the model's foundational knowledge of Bash commands and its capability to annotate complex multi-line scripts. **Finally**, to resolve the subtle but critical issues in comment quality that persist even after CPT and SFT (also noted in Section 2.2), we introduce our primary innovation: **Syntax-Aware Preference Optimization (SAPO)**. Instead of using generic preference pairs for Direct Preference Optimization (DPO), SAPO employs a fully automated pipeline to generate "Minimal Syntactic Pairs" by applying single "atomic operations" to a command's Abstract Syntax Tree (AST). This targeted approach transforms DPO into a "knowledge injection" process, achieving the goal of **Fine-grained Command Semantics Learning** by forcing the model to master the precise semantics of command-specific details, such as correctly recognizing `awk` field splitting, `find`'s recursive traversal functionality, and `uniq -i`'s case-insensitive comparisons. This process also enables **Context-dependent Quality Assessment**, learning to provide the most relevant comment based on specific analysis needs as demonstrated in Fig. 2, ensuring the final comments are not only technically accurate but also conceptually insightful.

In the data construction process, we handled single-line and multi-line Bash scripts separately. For single-line commands, SFT data was drawn from the BASHEXPLAINER dataset [54], with typographical errors corrected. For multi-line scripts, initial comments were generated by DeepSeek-V3-0324 and Qwen2.5-Max, scored by DeepSeek-R1 for correctness, completeness, and naturalness, and then reviewed and refined by bash experts (average over 5 years Linux experience). For the SAPO dataset, we designed a fully automated pipeline to generate "Minimal Syntactic Pairs." This method systematically applies single, atomic operations (e.g., modifying a command option or removing an argument) to a script's Abstract Syntax Tree (AST), creating a pair consisting of the original script and a subtly incorrect variant. Specifically, we programmatically modify a single node within the AST and then reconstruct the script string from the modified tree; scripts that fail the initial parsing are discarded to ensure robustness. To form the preference pair, the 'chosen' comment and the 'rejected' comment were both generated by a single powerful LLM, DeepSeek-V3-0324. Unlike the multi-model voting process in the SFT stage, this single-model approach was intentionally used to ensure stylistic consistency, thereby forcing the model to learn the precise semantic difference caused by the atomic operation rather than superficial stylistic variations.

Experimental results on 1,064 single-line Bash commands and 1,046 multi-line Bash scripts confirm Bash-Commenter's superior performance. It achieved the best performance (BLEU-4, METEOR, ROUGE-L): 33.40%, 58.26%, 57.03% for single-line, and 22.15%, 43.89%, 32.80% for multi-line scripts. LLM evaluation show positive ratings (score 3 or 4) for correctness, completeness, and naturalness: for single-line commands (76.5%, 80.0%, 86.5%) and multi-line scripts (70.0%, 68.0%, 73.0%). Human evaluation further underscore its quality, with positive ratings (score 3 or 4) for correctness, completeness, and naturalness: for single-line commands (79.5%, 82.0%, 89.0%) and multi-line scripts (73.0%, 71.0%, 76.0%), significantly outperforming baselines.

The main contributions of this paper are as follows:

- To the best of our knowledge, we are the first to introduce Syntax-Aware Preference Optimization in the bash code comment generation task.
- We constructed and publicly released comprehensive datasets for training and evaluating Bash comment generation, uniquely featuring complex multi-line scripts.

- We propose Bash-Commenter, a new method combining CPT, SFT and SAPO for bash code comment generation, achieving state-of-the-art performance through automatic metrics, LLM evaluation, and human evaluation.

2 Background and Motivation

2.1 Problem Statement

We propose an automated approach for Bash script comment generation. Given a Bash command or script x, our system generates a descriptive comment $\hat{y}$ that accurately captures the script's functionality. The task is formalized as finding an optimal mapping function $f : X \rightarrow Y$ where X represents the space of Bash scripts and Y represents comments.

2.2 Motivations

In this section, we analyze the motivations behind our research on bash code comment generation.

Table 1. Comparison of BASHEXPLAINER and Bash-Commenter Training Datasets

Dataset	Samples	SL/ML (%)	AvgLen	AvgLn	CmtLen	Cmds	P/R (%)	DistCmds	Anno
BASHEXP [54]	8,469	100.0/0.0	8.5	1	12	1.5	40.5	466	Expert
Ours	18,657	49.4/50.6	54.6	17.2	123	9.7	55.8	17,122	LLM+Exp

Notes: SL/ML: Single-line Commands / Multi-line Scripts (%); SL/ML shows the percentage ratio of single-line commands to multi-line scripts; AvgLen/AvgLn: avg tokens/lines; CmtLen: avg comment length; Cmds: commands per sample; P/R: pipe/redirection (%); DistCmds: distinct commands; Anno: annotation type.

Following the taxonomy proposed by Chen et al. [3] and Geng et al. [9], code comments serve two distinct purposes: (1) **Descriptive comments** explain command-level functionality and parameters (e.g., "-mtime +7 finds files modified more than 7 days ago"), and (2) **Intent-level comments** articulate high-level goals (e.g., "This script automates log rotation to manage disk space"). For Bash, this naturally maps to script complexity: **single-line commands (49.4%) primarily need descriptive comments** for syntax and parameters, while **multi-line scripts (50.6%) need both types**, with descriptive comments for individual commands and intent-level comments for overall workflow. As shown in Table 1, this dual-category design enables Bash-Commenter to generate both precise parameter explanations and comprehensive workflow documentation.

Motivation 1: Limitations of Existing Bash Code-Comment Dataset. Previous research on Bash code comment generation has focused almost exclusively on single-line Bash commands, resulting in datasets that are short, simple, and fail to capture the complexity of real-world scripting. For example, as shown in TABLE 1, the BASHEXPLAINER dataset [54] consists entirely of single-line commands (100%), with an average length of only 8.5 tokens, average comment length of 12 tokens, and just 1 line per sample. It covers only 466 distinct Bash commands and 40.5% of samples contain pipe or redirection, making it difficult for models to generalize to practical, professional scenarios. The BASHEXPLAINER dataset is also used by recent works such as Bash2Com [39] and HBCom [70].

To better reflect real-world scripting practices, our dataset combines two sources with distinct annotation strategies: **(1) Single-line commands (reused with corrections):** All 8,469 commands from BASHEXPLAINER [54], with factual errors (e.g., "directori"→"directory") and syntactic mistakes corrected before SFT (Section 3.2.3). **(2) Multi-line scripts (newly annotated):** 8,154 scripts sampled from Dong et al. [7], for which we generated and expert-verified all new annotations via LLM+expert pipeline (Section 3.2.3). The construction process was built upon the large-scale empirical study of Dong et al. [7], which demonstrated that multi-line scripts and complex structures are

```
find /data -type f -name "*.log" | grep -i "error" | awk '{print $1,$5}' >
                                errors_summary.txt
```

Fig. 2. Example of a Bash command pipeline for error log processing.

common in professional Bash development. Inspired by this finding, we excluded auto-generated and trivial scripts from Dong et al. [7] (e.g., scripts with fewer than 3 lines, fewer than 20 tokens, or containing patterns like "DO NOT EDIT"), and further removed near-duplicates using a Jaccard Index threshold of 0.9. We then sampled a diverse range of examples from domains such as system administration, DevOps, data processing, and network management to ensure practical coverage. **(3) SAPO preference data:** All 2,034 preference pairs are constructed from the same code sources as SFT (748 single-line, 1,286 multi-line). Chosen comments for single-line data reuse corrected BASHEXPLAINER annotations; all other annotations (chosen comments for multi-line and all rejected comments) are newly generated via AST-based mutation (Section 3.2.4).

As shown in TABLE 1, the resulting Bash-Commenter dataset is much larger and more representative of real-world Bash usage than previous datasets. It contains 18,657 samples, of which half (50.6%) are multi-line scripts. The average sample length reaches 54.6 tokens, with 17.2 lines and 9.7 Bash commands per sample. Notably, 55.8% of samples involve pipes or redirection, and the dataset includes 17,122 distinct Bash commands. The average comment length is as high as 123 tokens, and for single-line data, annotations are expert-written with our corrections; for multi-line data, all annotations are LLM-generated and expert-verified (Section 3.2.3).

Motivation 2: Limitations in Comment Quality of LLM-based Bash Comment Generation Methods. As shown in Fig. 1, general LLMs (LLaMA-3.1-8B-Instruct) fail in two aspects: they fundamentally misinterpret multiple aspects of Bash commands and provide inaccurate comment. Although specialized LLMs like Bash-Commenter (trained through CPT and SFT) perform better in command analysis, they still struggle to provide fully accurate comment. The specialized LLM correctly identifies the main purpose of finding duplicate files (`uniq -d`) and accurately describes the output format (`awk`). However, its comment contains critical misunderstandings of command functionality. It omits the recursive traversal functionality of the `find` command, fails to accurately express the role of case-insensitive matching (`uniq -i`), and overlooks the field splitting functionality of `awk`. This limitation in comment quality persists despite the improvements from CPT and SFT. It underscores the need for further refinement in the LLM's understanding of Bash command execution flow and operational semantics. While general LLMs like LLaMA-3.1-8B-Instruct struggle with Bash semantics (Fig. 1), this limitation extends to commercial models. GPT-4.1 and Claude-3.7-Sonnet achieve only 11.89% and 11.50% BLEU-4 on multi-line scripts respectively (Table 4), with human evaluations showing 28% and 27% perfect correctness ratings (Table 6). Case analysis (Fig. 4) reveals systematic failures across all methods in recognizing safety mechanisms (`-print0`/`xargs -0`) and regex types (`posix-egrep`).

Unlike traditional training methods that rely on absolute labels, SAPO learns from relative preferences between paired comments which is especially effective in two key aspects:

(1) **Fine-grained Command Semantics Learning:** SAPO's loss function explicitly models the relative preference between two comment, enabling the capture of subtle command-specific details in Bash scripts. As illustrated in Fig. 1, even Bash-Commenter (CPT+SFT) struggles with critical issues such as omitting `awk` field splitting details, misunderstanding recursion with `find`, and failing to recognize case-insensitive comparisons. Our SAPO approach addresses these limitations by learning from human preferences between command-level comments. By training on paired examples where experts prefer correct interpretations over incorrect ones, the model gradually learns to distinguish subtle aspects of command behavior—properly recognizing when `awk -F`

indicates field splitting rather than regex application, understanding that `find` implies recursive traversal by default, and acknowledging the case-insensitive nature of `uniq -i` operations. This preference-guided learning creates a more nuanced understanding of command semantics, critical for avoiding potentially catastrophic errors in practical scripting scenarios.

(2) **Context-dependent Quality Assessment:** Bash commands often involve complex interactions between utilities and data transformations, making comment quality highly context-dependent. SAPO's preference learning effectively captures how comment quality varies with context through paired comparisons. For example, consider the Bash command in Fig. 2 with two different comments:

Comment A: "This command recursively searches for all `.log` files in the `/data` directory, filters lines containing `'error'` (case-insensitive), extracts the `1st` and `5th` columns from each matching line, and saves the results to `errors_summary.txt`"

Comment B: "This command analyzes logs by finding all log files in the data directory tree and filtering for error messages using case-insensitive matching. It extracts timestamp and error message columns, then saves these details to a summary file."

While both describe the same command, their value differs by context. Comment A suits technical references with precise syntax descriptions, while Comment B fits higher-level documentation like user guides by focusing on conceptual purpose and usage patterns. Through such paired comparisons, SAPO learns context-appropriate comment generation, addressing limitations of CPT and SFT-only training.

3 Approach

Bash-Commenter follows three stages (Fig.3): CPT, SFT, and SAPO, built on LLaMA-3.1-8B for its optimal performance-efficiency balance[30, 44, 55]. **Design Rationale.** SAPO-based fine-tuning is prioritized over alternatives for three reasons: (1) *Retrieval fails on semantics*: RAG baselines like BASHEXPLAINER reach only 29.13% BLEU-4 (Table 3), struggling to distinguish subtle command opposites (e.g., `grep -v` vs. `grep`), while BM25/VSM performs worse (9.40%–19.24%). (2) *Prompting misses syntax nuances*: even GPT-4.1 overlooks critical option combinations and pipeline interactions (Fig.4). (3) *Agents are inefficient*: multi-agent systems incur high latency without guaranteeing AST-level correctness. SAPO instead directly embeds syntax rules into model weights via AST-based minimal pairs, enabling efficient single-pass inference. LLaMA-3.1-8B (July 2024) further supports practical deployment, outperforming comparable models and rivaling those 4×4× larger (Table3).

3.1 Open-Source Bash Scripts Collection

3.1.1 Continual Pre-training. For our Continual Pre-training dataset, we followed the approach in [7] and integrated Bash-related data from multiple sources: (1) The complete collection of Linux manual pages (Man Pages), containing detailed command descriptions and usage examples; (2) Stack Overflow question-answer pairs tagged with "bash", "shell", and "linux"; (3) Bash projects from the top 1,000 GitHub repositories by star count (following [7]); (4) Curated shell script discussions from Unix & Linux Stack Exchange. We extracted a total of 1,013,135 Bash commands and scripts from these sources. To ensure uniqueness, scripts were filtered using a Jaccard Index similarity threshold of 0.9, eliminating those with over 90% token similarity. After deduplication, 676,524 unique Bash scripts remained. The threshold of 0.9 was adopted following prior work [42, 55], which used this value to identify and remove near-duplicate code snippets.

3.1.2 Supervised Fine-Tuning and Syntax-Aware Preference Optimization. **For SFT dataset**, we combined data from two primary sources: the BASHEXPLAINER dataset [54] and selected scripts from [7]. The BASHEXPLAINER dataset [54] provided 8,469 professionally annotated single-line Bash commands with high-quality comment. To augment this dataset, we incorporated an

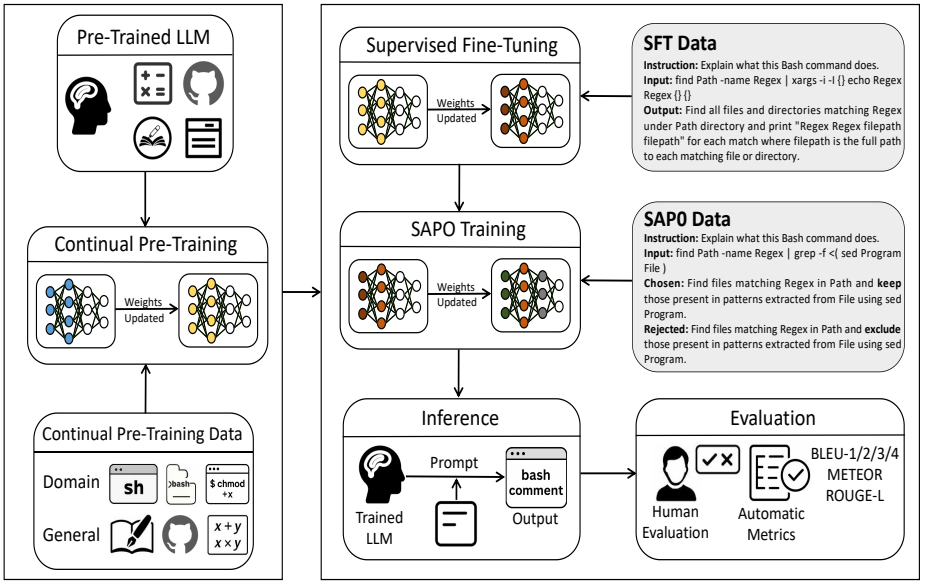


Fig. 3. The Overview of our Bash-Commenter.

additional 8,154 annotated multi-line Bash scripts from [7], focusing on diverse use cases and complexity levels. After quality control and deduplication, our final SFT dataset contained 16,623 high-quality ⟨Bash Code, Comment⟩ pairs. Our quality control employed a four-stage pipeline: (1) Jaccard-based deduplication (threshold 0.9); (2) LLM scoring (DeepSeek-R1) with 50/30/20 weighting for correctness/completeness/naturalness, filtering samples ≥ 7.0/10 and excluding 23.4% low-quality candidates; (3) expert verification by 24 professionals (dual annotation on 300 samples, Fleiss' Kappa=0.78); (4) re-annotation for confidence <6, with 8.7% requiring arbitration (details in Sections 3.2.2 - 3.2.3). **For SAPO dataset**, we constructed paired data comprising preferred and rejected comment for the same Bash commands or Bash scripts. We leveraged 748 pairs from [54]. Additionally, we created 1,286 preference pairs from the [7] collection. We ensured there was no overlap between the SAPO dataset and the SFT dataset. In total, our SAPO dataset consisted of 2,034 preference pairs from single-line commands to multi-line scripts.

3.2 SFT and SAPO Data Construction

3.2.1 Initial Data Generation. We utilized DeepSeek-V3-0324 [24] and Qwen2.5-Max [52] for initial annotation generation, chosen for their performance comparable to models like GPT-4.1 and Claude-3.7-Sonnet at a lower cost, as indicated by their technical reports [24, 52]. The Bash scripts were collected from GitHub, Stack Overflow, and Linux system administrator communities [7]. For each script, corresponding functional comment annotations were generated.

3.2.2 LLM Scoring. DeepSeek-R1 served as the judge model for scoring generated comments, selected for its strong reasoning, code evaluation performance, and cost-effectiveness [14]. It assigned per-aspect scores (1–10) across Correctness, Completeness, and Naturalness (Section 3.7), and computed a weighted overall score (Correctness 50%, Completeness 30%, Naturalness 20%) following Smart-LLaMA-DPO [57], which prioritizes correctness for code explanation. Only annotations exceeding a weighted score of 7.0 advanced to human verification, automatically filtering out 23.4% of

initial candidates. Empirical validation (Table 6) confirms this weighting: when correctness=1, 85.7% of samples (6/7) exhibit low completeness (scores 1–2), while correctness=3–4 samples consistently achieve high scores across all metrics. The 7.0/10 threshold follows prior work [57].

3.2.3 SFT Data Construction. For multi-line scripts, annotations with higher LLM scores were selected and refined by 24 Bash experts (12 senior Linux system administrators averaging 8 years of experience, 7 DevOps engineers averaging 6 years, and 5 computer science researchers averaging 5 years), who verified command functionality accuracy, improved parameter comments, and ensured clarity and completeness. Inter-annotator consistency over 300 dual-annotated samples (200 single-line, 100 multi-line) yielded Fleiss' Kappa = 0.78. Each annotation received a confidence score (1–10); those below 6 were independently re-annotated by a second expert with consensus discussion, and unresolved cases were arbitrated by a third expert (8.7% arbitration rate), ensuring all final annotations exceeded confidence 6, with 89.3% exceeding 7. Independent quality assessment by 5 senior experts (5+ years) on 500 stratified samples (247 single-line [49.4%], 253 multi-line [50.6%]) yielded 8.3/10 (Correctness), 8.1/10 (Completeness), and 8.5/10 (Naturalness), with a 91.2% approval rate (Correctness > 7). Error analysis of the remaining 44 samples (8.8%) revealed 4.4% minor wording issues, 3.0% incomplete parameter explanations, and 1.4% factual errors (removed). For single-line commands, spelling errors in BASHEXPLAINER [54] (e.g., directori→directory) were corrected. The final SFT dataset comprises 16,623 high-quality samples. Following human-LLM collaborative annotation practices [57], this demonstrates that LLM-generated labels with systematic human refinement can match human-written annotation quality. Validation on 500 randomly sampled BASHEXPLAINER commands re-annotated via our methodology, blindly rated by 5 senior experts (5+ years, Fleiss' Kappa = 0.76), showed no significant differences across all metrics (Correctness: 8.6±1.2 vs. 8.5±1.1, p=0.421; Completeness: 8.3±1.4 vs. 8.4±1.2, p=0.587; Naturalness: 8.7±1.1 vs. 8.8±0.9, p=0.312), confirming statistical equivalence.

3.2.4 Automated Generation of Minimal Syntactic Pairs for SAPO. To enable our Syntax-Aware Preference Optimization (SAPO) framework, we designed and implemented a **fully automated data generation pipeline**. This pipeline addresses a key limitation of standard DPO: the reliance on generic or manually created preference pairs, which provide diffuse and inefficient learning signals. Our approach, instead, systematically generates **"Minimal Syntactic Pairs"** to provide the model with highly targeted, interpretable, and scalable feedback.

We analyzed 856 Stack Overflow posts (January 2020 to November 2022, pre-LLM era to ensure human-verified corrections) with six coders (Fleiss' κ = 0.82), identifying five atomic operations covering 73.4% of errors: Option Deletion (28.7%), Argument Modification (22.1%), Command Replacement (12.6%), Redirection Removal (6.0%), Pipeline Adjustment (4.0%).

Formal Definition of a Minimal Syntactic Pair: A preference tuple `<C, chosen, rejected>` is defined as a "Minimal Syntactic Pair" if it satisfies the following conditions:

(1) **Semantic Anchor:** The *chosen* comment is a correct and high-quality description of an original Bash command `C`.
(2) **Syntactic Perturbation:** The *rejected* comment is a correct description of a variant command `C'`, which is derived from `C` by applying a single, predefined **"Minimal Atomic Operation"** to its Abstract Syntax Tree (AST).
(3) **Surface Similarity:** The *chosen* and *rejected* comments are naturally similar in language, isolating the semantic difference caused exclusively by the atomic operation.

A "Minimal Atomic Operation" is a single, targeted transformation on the command's AST that corresponds to a common but critical developer error. These atomic operations were not chosen arbitrarily. Based on a preliminary manual analysis of real-world Bash errors collected from Stack

Overflow and the failure cases of our initial SFT model, we developed a wide range of such atomic operations to target the most critical error patterns. For brevity, we present and analyze five of the most representative categories, which are detailed as examples in Table 2.

Table 2. Examples of Minimal Atomic Operations for SAPO Data Generation

Atomic Operation	**Description**	**Example `C -> C'`**
Option Deletion	Removes a command-line option (flag).	`grep -v "a" -> grep "a"`
Option Substitution	Replaces an option with a different one.	`head -n 5 -> tail -n 5`
Argument Mod.	Modifies an argument (*e.g.*, quote type).	`echo "$VAR" -> echo '$VAR'`
Redirection Del.	Removes a file redirection operator.	`ls > f.txt 2>&1 -> ls > f.txt`
Pipeline Simp.	Removes a component from a pipeline.	`sort | uniq -c -> sort | uniq`

The Automated Generation Pipeline: Our pipeline consists of four automated stages:

(1) **Command Variant Generation:** For each source command `C`, we first use a robust Bash parser (`bash-parser` for Python) to generate its AST. To create the variant command `C'`, we then apply one of the predefined atomic operations, which involves programmatically modifying a single node within the AST and then reconstructing the script string from the modified tree. Any command that fails this initial parsing step is automatically excluded.

(2) **Parallel Annotation Generation:** We use a powerful LLM (DeepSeek-V3-0324) to generate annotations for both commands in parallel. The annotation for `C` becomes the candidate *chosen* comment, and the annotation for `C'` becomes the candidate *rejected* comment.

(3) **SAPO Sample Assembly:** The final SAPO training sample is assembled as `<prompt=C, chosen=Comment_C, rejected=Comment_C'>`. This structure forces the model to learn why `Comment_C'` is incorrect *for the original command* `C`, thereby learning the specific semantics of the atomic operation.

(4) **LLM-as-a-Judge Verification:** To ensure quality without human intervention, we employ an LLM-as-a-Judge mechanism (using DeepSeek-R1). Each generated triplet is automatically verified for accuracy and integrity, with any that fail this check being discarded to ensure the final dataset's quality. To validate our judge, we manually reviewed a random sample of 200 of its decisions and found a 96% agreement rate (192 out of 200) with human experts.

3.3 Continual Pre-Training

To develop Bash-Commenter, we continual pre-train LLaMA-3.1-8B for two epochs on Bash-related data and one epoch on natural language data, enhancing both code and language capabilities. We optimize the language modeling objective used in GPT and LLaMA. Given a sequence x of n tokens $t_0, t_1, ..., t_{n-1}$, we maximize the likelihood of the entire sequence by calculating the product of conditional probabilities for each token:

$$p(x) = \prod_{i=1}^{n-1} p(t_i | t_1, t_2, ...t_{i-1}) \tag{1}$$

Pre-training Dataset [809.79M tokens]. To build a comprehensive and efficient pre-training dataset, we sampled high-quality data of two main types: **(1) Bash-related data** [554.40M tokens]. To enhance Bash comment generation, we gathered Bash scripts and commands from diverse sources [7]: Linux manual pages, Stack Overflow Q&A tagged with "bash", "shell", and "linux", highly-starred Bash projects on GitHub, and curated discussions from Unix & Linux Stack Exchange. This collection forms a comprehensive Bash code repository, spanning simple commands to complex scripts. **(2) Other related data** [255.39M tokens]. Based on the data collection methodology of MAP-Neo [65], we collected diverse high-quality data, including: **1) Natural language data**: FLAN v2 multilingual instruction-tuning dataset [28], UltraTextbooks high-quality textbooks [27], and multi-turn dialogue data cleaned from open networks, enhancing the model's instruction-following and conversational abilities. **2) Code data**: The Stack dataset (v1) [19], CodeGPT [50], and LeetCode datasets, covering various programming languages such as Python, C++, and Java. **3) Mathematics data**: AutoMathText dataset [71] and open-web-math dataset [34], improving the model's mathematical problem-solving and logical reasoning abilities.

3.4 Supervised Fine-Tuning

In the Supervised Fine-Tuning stage, we refine the large language model to produce high-quality comments for Bash scripts. For this purpose, we employ a token-level negative log-likelihood loss optimized over the dataset:

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{|\mathcal{D}_{\text{exp}}|} \sum_{(x,y)\in\mathcal{D}_{\text{exp}}} \sum_{t=1}^{|y|} \log P_\theta(y_t \mid x, y_{<t}) \tag{2}$$

Here, x is the input Bash script, y the target comment sequence, and θ denotes all trainable parameters of the LLM (including attention, feed-forward layers, embeddings, etc.).

This objective enables the model to learn to generate comments that accurately and fluently explain the functional and contextual aspects of Bash scripts. The prior domain-specific knowledge acquired during continual pre-training helps the model better understand Bash syntax and semantics, thus supporting the generation of detailed and correct comments.

3.5 Syntax-Aware Preference Optimization (SAPO)

In the final stage, to bridge the gap between a generally capable model and a domain expert, we introduce our novel framework: **Syntax-Aware Preference Optimization (SAPO)**. SAPO reframes the preference optimization problem for code by moving beyond generic quality signals. Instead, it leverages the highly structured and targeted **"Minimal Syntactic Pairs"** (as generated in Section 3.2.4) to inject deep, granular domain knowledge directly into the model.

While SAPO utilizes the same underlying mathematical formulation as Direct Preference Optimization (DPO) [37], its core innovation lies not in the algorithm itself, but in the **semantic richness of the preference data *D***. The SAPO objective is to optimize a policy π_θ that best satisfies a set of preferences (x, y_p, y_n).

The core of SAPO is based on the insight that the optimal policy π^* for a reward function r^* under a KL-constrained optimization objective can be expressed as:

$$\pi^*(y|x) = \frac{1}{Z(x)}\pi_{\text{ref}}(y|x)\exp\left(\frac{1}{\beta}r^*(x,y)\right) \tag{3}$$

where $Z(x)$ is a normalization factor, π_{ref} is a reference policy, and β is a temperature parameter. In our Bash comment generation scenario:

- x: represents the input Bash script.

- y: represents the LLM-generated comment.
- π_{ref}: the reference policy, typically the model from the SFT stage.
- π^*: the final optimized model.

By rearranging this equation, we can express the reward function in terms of the optimal policy:

$$r^*(x, y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x) \tag{4}$$

This allows us to reformulate the Bradley-Terry preference model in terms of policies rather than rewards:

$$p^*(y_p \succ y_n|x) = \sigma\left(\beta \log \frac{\pi^*(y_p|x)}{\pi_{\text{ref}}(y_p|x)} - \beta \log \frac{\pi^*(y_n|x)}{\pi_{\text{ref}}(y_n|x)}\right) \tag{5}$$

where σ is the logistic function, and y_p and y_n represent the preferred and non-preferred comment outputs.

To simplify, we introduce an intermediate variable $\Delta(x, y_p, y_n)$ representing the log ratio difference:

$$\Delta(x, y_p, y_n) = \beta \log \frac{\pi_\theta(y_p|x)}{\pi_{\text{ref}}(y_p|x)} - \beta \log \frac{\pi_\theta(y_n|x)}{\pi_{\text{ref}}(y_n|x)} \tag{6}$$

Using this, the SAPO loss function can be expressed as:

$$\mathcal{L}_{\text{SAPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_p, y_n)\sim D} \log \sigma(\Delta(x, y_p, y_n)) \tag{7}$$

where (x, y_p, y_n) are triples from our curated dataset D.

In the context of SAPO, this optimization is significantly enhanced. By training on minimal syntactic pairs, $\mathcal{L}_{\text{SAPO}}$ directly maximizes the model's ability to distinguish between correct and incorrect interpretations of specific syntactic elements (e.g., command-line flags or quoting conventions), rather than optimizing for vague "betterness." This transforms DPO into a precise tool for **targeted knowledge injection**.

3.6 Prompt Design for LLM Inference

To ensure fair reproducibility, we standardized prompt designs across all baseline evaluations (GPT-4.1, Claude-3.7-Sonnet, DeepSeek-R1 series, Qwen2.5/3 series, LLaMA-3.1-8B-Instruct) using greedy decoding (temperature=0). **For Single-line Command Prompting**, following prior work [39, 70], we employ 3-shot prompting by selecting three diverse examples covering file operations, filtering, and conditionals; crucially, these examples are strictly drawn from the training set only to prevent data leakage and ensure no overlap with the test set. Conversely, **for Multi-line Script Prompting**, we adopt zero-shot prompting to maximize context window utilization, directly instructing models to analyze the script's purpose and workflow.

3.7 Human Evaluation and LLM Evaluation of Bash Comments

Human Evaluation. For human evaluation, following [7, 39, 70], we recruited 24 professional evaluators with over 5 years of Bash programming experience and good English reading skills, divided into 12 groups of 2. We randomly sampled 300 code snippets (200 single-line commands and 100 multi-line scripts) from the test set, assigning 25 unique samples per group. Both group members independently annotated their samples, with a 20% cross-group overlap (5 samples) ensuring some samples were evaluated by four evaluators. Each evaluator annotated 30 samples in total, with a maximum of 15 samples per 8-hour workday to avoid fatigue. Evaluators were blinded to comment sources, and presentation order and format were randomized and standardized. Inter-evaluator agreement was measured using Fleiss' Kappa over all overlapping samples (rather than averaging group values), yielding 0.71, confirming evaluation reliability. Discrepancies were resolved through

group discussion. Each comment was rated on a 1–4 Likert scale across three metrics: Correctness (accurate reflection of code functionality and intent), Completeness (coverage of all key elements without omission), and Naturalness (fluency and readability of language).

LLM Evaluation. To complement human evaluation, we employed **DeepSeek-R1** as the LLM judge, validated on 100 diverse samples where it achieved **93% agreement** with human experts, outperforming GPT-4.1 (89.0%), Claude-3.7-Sonnet (87%), LLaMA-3.1-70B (84%), and Qwen2.5-72B (83.0%). This performance is attributed to its reinforcement learning-based training [14], which strengthens reasoning in technical documentation tasks. Following LLM-as-a-Judge best practices [11, 21, 46, 57], we designed a rigorous evaluation prompt covering *correctness, completeness, and naturalness*, incorporating precise task definitions, granular 1–4 scale scoring criteria, three few-shot anchor examples, and structured JSON output constraints.

4 Experiments

4.1 Research Questions

To evaluate our Bash-Commenter, we conduct experiments to answer the research questions:

- **RQ1**: How does Bash-Commenter perform in the Bash comment generation task compared to existing methods?
- **RQ2**: How effective are the individual components of Bash-Commenter (CPT, SFT, and SAPO)?
- **RQ3**: How effective are the comments generated by Bash-Commenter in terms of Correctness, Completeness, and Naturalness, as measured by human evaluation and LLM evaluation?
- **RQ4**: What are the characteristic error patterns of Bash-Commenter on scripts of varying complexity?

4.2 Dataset

Our dataset comprises continual pre-training (CPT), supervised fine-tuning (SFT), syntax-aware preference optimization (SAPO), and evaluation sets. **CPT** uses 676,524 Bash scripts from GitHub and Linux documentation (554.40M tokens) plus 300,000 instances from general code, mathematics, and bilingual text (255.39M tokens), totaling 976,524 instances (809.79M tokens). **SFT** combines 8,469 single-line commands from BASHEXPLAINER [54] and 8,154 multi-line scripts from Dong et al. [7], totaling 16,623 high-quality pairs (5.43M tokens). **SAPO** contains 2,034 preferred/rejected pairs: 748 from BASHEXPLAINER and 1,286 newly annotated from Dong et al., totaling 1.13M tokens with no SFT overlap. **Evaluation** uses 2,110 Bash instances: 1,064 single-line commands from BASHEXPLAINER and 1,046 multi-line scripts from Dong et al., with no training overlap. Human evaluation and LLM Evaluation uses 200 single-line commands and 100 multi-line scripts.

4.3 Baselines and Metrics

Baselines. Our evaluation covers five categories of baselines: **Information retrieval-based methods** retrieve comments from similar code snippets based on semantic or lexical similarity, such as LSI [15], VSM [16], BM25 [67], and NNGen [26]. **Neural network-based methods** generate comments using sequence-to-sequence models, treating the problem as neural machine translation. Examples include CopyNet [12], Transformer [45], CODE-NN [18], and HBCom [70]. **Pre-trained model-based methods** leverage models like CodeBERT [8], UniXcoder [13], CoTexT [35], PLBART [1], CodeT5 [47], BASHEXPLAINER [54], and Bash2Com [39], which are pre-trained on large code corpora and fine-tuned on Bash data. **Hybrid methods** combine retrieval and neural techniques, as seen in Hybrid-DeepCom [17] and Rencos [67], aiming to integrate retrieval results

into generation. **LLM-based methods** employ general-purpose LLMs such as GPT-4.1 (gpt-4.1-2025-04-14) [33], Claude-3.7-Sonnet (claude-3-7-sonnet-20250219) [2], LLaMA-3.1-8B-Instruct [10], Qwen2.5 [52] and Qwen3 series [51], and DeepSeek-R1-Distill-Qwen models [24].

Metrics. We evaluated our model from two dimensions: automatic metrics and human evaluation. For automatic evaluation, we used BLEU-1/2/3/4 (n-gram precision), METEOR (precision/recall with synonyms and stemming), and ROUGE-L (longest common subsequence). Additionally, we employed three key metrics: Correctness, Completeness, and Naturalness (Section 3.7).

Table 3. Comparison results between our proposed method Bash-Commenter and baselines on Single-line Bash Commands. DeepSeek-R1-7B/14B/32B refer to the distilled versions. LLaMA-3.1-8B and Qwen2.5-7B refer to LLaMA-3.1-8B-Instruct and Qwen2.5-7B-Instruct respectively. MET denotes METEOR metric.

Method Type	Method Name	BLEU-1	BLEU-2	BLEU-3	BLEU-4	MET	ROUGE-L
Retrieval-based	LSI	30.18	18.07	12.48	9.40	18.30	28.82
	VSM	36.16	24.47	18.62	15.25	22.04	34.58
	BM25	42.08	30.41	23.58	19.24	26.35	38.49
	NNGen	50.62	38.75	32.11	27.85	27.69	45.88
Deep Learning	CopyNet	38.11	27.06	20.67	16.43	22.06	40.18
	Transformer	46.39	33.37	25.42	19.97	25.22	44.01
	CODE-NN	49.60	37.18	29.53	24.17	26.85	47.21
	HBCom	56.45	44.30	37.52	32.76	30.18	53.44
Pre-trained Methods	CodeBERT	48.65	37.02	29.84	24.83	27.16	47.36
	UniXcoder	49.99	38.52	31.80	27.25	29.03	48.24
	CoTexT	49.17	37.29	30.36	25.75	28.56	48.00
	PLBART	50.79	39.10	32.21	27.55	28.82	47.91
	CodeT5	51.75	40.04	33.25	28.70	29.49	48.36
	BASHEXPLAINER	51.74	40.41	33.73	29.13	28.78	48.81
	Bash2Com	54.74	43.79	37.19	32.57	30.26	51.80
Hybrid Method	Hybrid-DeepCom	47.78	35.45	27.91	22.75	26.27	45.36
	Rencos	46.27	35.11	28.66	24.39	25.82	45.06
General LLMs	GPT-4.1	20.32	8.16	4.31	2.67	27.67	19.64
	Claude-3.7-Sonnet	13.62	5.04	2.72	1.74	25.03	15.07
	DeepSeek-R1-7B	23.15	10.53	5.91	4.10	23.24	24.27
	DeepSeek-R1-14B	25.03	12.59	7.20	4.85	26.98	27.28
	DeepSeek-R1-32B	26.35	13.23	7.48	5.03	28.53	28.92
	LLaMA-3.1-8B	28.37	13.99	7.83	5.12	28.63	30.33
	Qwen2.5-7B	25.68	13.10	7.48	4.96	28.31	27.96
	Qwen3-8B	15.64	6.68	3.59	2.22	22.74	17.19
Our Method	**Bash-Commenter**	56.21	**46.63**	**39.65**	**33.40**	**58.26**	**57.03**

Table 4. Performance Comparison on Multi-line Bash Scripts.

Model	B-1	B-2	B-3	B-4	MET	R-L
GPT-4.1	42.84	26.65	17.70	11.89	35.83	23.40
Claude-3.7-Sonnet	44.49	26.74	17.44	11.50	34.53	22.69
LLaMA-3.1-8B-Instruct	35.97	22.58	15.09	10.11	31.08	20.91
DeepSeek-R1-7B	45.57	26.76	16.65	10.54	33.23	22.15
DeepSeek-R1-32B	45.63	27.84	18.16	12.03	34.79	22.86
Qwen2.5-7B-Instruct	40.74	25.02	16.66	11.26	33.18	20.73
Qwen2.5-72B-Instruct	28.79	18.99	13.22	9.19	39.56	21.37
Qwen2.5-Coder-32B	34.45	22.89	15.96	11.17	41.93	23.55
Qwen3-8B	30.96	18.87	12.19	7.93	36.89	17.72
Qwen3-14B	29.00	17.85	11.65	7.65	37.18	18.09
Bash-Commenter	**53.11**	**38.26**	**28.99**	**22.15**	**43.89**	**32.80**

Note: B-1/2/3/4=BLEU-1/2/3/4, MET=METEOR, R-L=ROUGE-L. All values are in %.

4.4 Implementation Details

All models (CPT, SFT, SAPO) are trained using `LlamaFactory` and DeepSpeed with fp16, cross-entropy loss, and AdamW optimizer ($\beta = (0.9, 0.99)$, $\epsilon = 10^{-8}$), with full parameter tuning. Training is conducted on 8 NVIDIA H800 GPUs (80GB memory each). For CPT: batch size 64, gradient accumulation 16, 3 epochs, learning rate 1×10^{-5} (cosine decay, no warmup), cutoff length 2048, saving every 500 steps. For SFT: batch size 8, gradient accumulation 8, 3 epochs, same learning rate schedule, cutoff length 2048, saving every 200 steps. For SAPO: batch size 8, gradient accumulation 4, learning rate 1×10^{-5} (cosine decay, no warmup), cutoff length 1024, 1 epoch. Evaluation uses greedy decoding (temperature = 0). For baselines of general-purpose LLMs, following previous work [39, 70], we use few-shot (3-shot) prompting for single-line commands. For multi-line scripts, we use zero-shot prompting since they are too long for few-shot examples.

4.5 Experimental Results

4.5.1 RQ1. Overall Performance. We compared Bash-Commenter's performance against baselines.

Quantitative Results. **For single-line commands (TABLE 3)**, Bash-Commenter demonstrated strong performance. It achieved a BLEU-4 of 33.40% (vs. HBCom's 32.76%) and its ROUGE-L was 57.03%, surpassing HBCom (53.44%). Notably, its METEOR score was an exceptionally high 58.26%, significantly outperforming Bash2Com (30.26%). This substantial METEOR score is primarily attributed to our three-stage post-training process (CPT, SFT, and SAPO), which greatly enhances the model's ability to capture semantic nuances and generate fluent, contextually relevant comments. Additionally, using LLaMA-3.1-8B as the base model further enhances performance due to its stronger semantic understanding compared to earlier pre-trained models like CodeBERT. **For multi-line scripts (TABLE 4)**, Bash-Commenter also outperformed other compared LLMs. It achieved a BLEU-4 of 22.15% (vs. DeepSeek-R1-32B's 12.03%). Its METEOR was 43.89% (vs. Qwen2.5-Coder-32B's 41.93%), and ROUGE-L was 32.80% (vs. Qwen2.5-Coder-32B's 23.55%). Earlier CodeBERT-based methods (BASHEXPLAINER, Bash2Com) and deep learning approaches (HBCom) were not used as baselines due to token limits (512 for CodeBERT) and less suitable architectures for complex scripts.

Case Study Analysis. To complement the quantitative results, we conducted case studies across four representative scenarios. **In the first command**, GPT-4.1's comment fails to recognize the

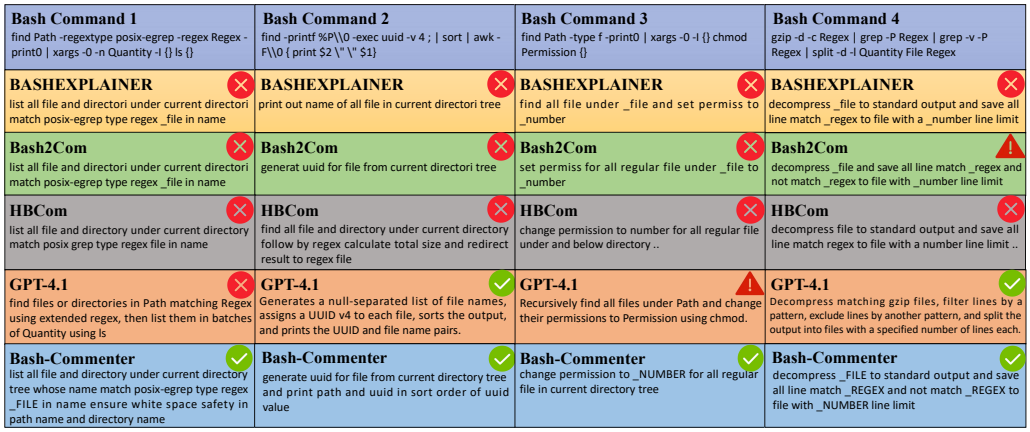


Fig. 4. Case Study of Bash Code Comment Generation Using Bash-Commenter.

specific regex matching pattern and ignores the critical safety parameters (`-print0` and `xargs -0`) designed to handle filenames with spaces; BASHEXPLAINER and Bash2Com similarly overlook this mechanism and contain spelling errors ("directori"). HBCom incorrectly describes the regular expression type as "posix grep" instead of the actual "posix-egrep", showing a misunderstanding of the command parameters. In contrast, Bash-Commenter accurately identifies both the regex type and matching pattern, while specifically highlighting the filename space handling safety mechanism. **For the second command**, BASHEXPLAINER's comment misses the primary functionality: UUID generation, describing the command as listing filenames. Bash2Com recognizes the UUID feature but omits sorting and output format details. HBCom severely misinterprets the command, incorrectly claiming it calculates file sizes and redirects results to a file, neither of which exist. GPT-4.1 provides a technically accurate comment but focuses excessively on processing mechanisms over practical purpose. Bash-Commenter accurately captures the core purpose and result organization, specifically noting that results are sorted by UUID value. **In the third command**, BASHEXPLAINER and Bash2Com fail to accurately convey the permission setting mechanism, HBCom's "under and below directory" is vague and imprecise, and GPT-4.1 overemphasizes the search process with "recursively find", making it verbose. Bash-Commenter concisely describes the permission modification purpose while correctly identifying it operates on regular files in the directory tree. **For the fourth command**, BASHEXPLAINER and HBCom both miss the dual filtering mechanism (`grep -P` and `grep -v -P`), describing only single filtering, which is a significant functional omission. Bash2Com captures both filters but omits that decompression outputs to standard output, while GPT-4.1 is complete but verbose. Bash-Commenter achieves optimal balance by accurately capturing decompression, dual filtering, and line limit while maintaining conciseness.

Answer to RQ1: Bash-Commenter outperformed existing state-of-the-art methods across main automatic evaluation metrics (BLEU-4, METEOR, and ROUGE-L) for both single-line Bash commands and multi-line Bash scripts. Case studies further demonstrate that these quantitative improvements stem from superior semantic understanding of command-specific behaviors (e.g., regex types, safety mechanisms).

4.5.2 RQ2. Ablation Study. Our ablation study (TABLE 5) demonstrates the crucial roles of SAPO and CPT. **For single-line Bash commands**, removing SAPO training significantly degraded performance: BLEU-4 decreased by 10.36 percentage points, ROUGE-L by 10.10, and METEOR by 7.83, indicating SAPO's crucial role in refining comment generation. Removing CPT had greater impact, with BLEU-4 dropping by 14.16 points, ROUGE-L by 11.40, and METEOR by 8.07, highlighting the importance of domain-specific knowledge acquired during this phase. Without both, we observed the most severe degradation: BLEU-4 declined by 17.01 points, ROUGE-L by 16.03, and METEOR by 12.80, demonstrating their complementary nature. **For multi-line Bash scripts**, performance impacts were less dramatic but significant. Without SAPO training, BLEU-4 decreased by 1.41 points, ROUGE-L by 0.13, and METEOR by 3.89. CPT removal had greater influence, reducing BLEU-4 by 2.96 points, ROUGE-L by 3.16, and METEOR by 4.39. Removing both decreased BLEU-4 by 3.10 points, ROUGE-L by 3.00, and METEOR by 3.47. This reveals a context-dependent pattern: while both components significantly enhance performance for simpler single-line commands, CPT emerges as more critical for complex multi-line scripts, suggesting that domain expertise becomes increasingly paramount as code complexity increases.

Table 5. Ablation Study of Bash-Commenter on Single-line Bash Commands and Multi-line Bash Scripts.

Scenario	Metric	Base	w/o SAPO	w/o CPT	w/o Both
Single-line	BLEU-1	**56.21**	46.04	42.12	37.09
	BLEU-2	**46.63**	35.60	32.75	28.40
	BLEU-3	**39.65**	28.58	25.40	21.64
	BLEU-4	**33.40**	23.04	19.24	16.39
	METEOR	**58.26**	50.43	50.19	45.46
	ROUGE-L	**57.03**	46.93	45.63	41.00
Multi-line	BLEU-1	**53.11**	51.62	49.50	49.65
	BLEU-2	**38.26**	36.67	34.61	34.75
	BLEU-3	**28.99**	27.43	25.62	25.61
	BLEU-4	**22.15**	20.74	19.19	19.05
	METEOR	**43.89**	40.00	39.50	40.42
	ROUGE-L	**32.80**	32.67	29.64	29.80

Answer to RQ2: Our findings confirm that both CPT and SAPO aid performance with context-dependent importance. CPT offers domain knowledge and has greater impact, especially on multi-line scripts, while SAPO enhances comment quality.

4.5.3 RQ3. Human and LLM Evaluation. TABLE 6 presents the human and LLM evaluation results. For single-line Bash commands, Bash-Commenter significantly outperformed all baselines in both human and LLM evaluations. In human evaluation, positive ratings (scores 3 and 4) for Correctness, Completeness, and Naturalness reached 79.5%, 82.0%, and 89.0%, respectively. In comparison, HBCom achieved 66.5%, 64.5%, and 72.0%; Bash2Com scored 61.5%, 60.5%, and 69.0%; and BASHEXPLAINER had 55.0%, 56.5%, and 62.0% on the three metrics. LLM evaluation showed consistent trends with positive ratings of 76.5%, 80.0%, and 86.5% for our method, compared to

Table 6. Human and LLM Ratings of Correctness, Completeness, and Naturalness.

Model	Correctness				Completeness				Naturalness			
	1	2	3	4	1	2	3	4	1	2	3	4
Single-Line Bash Commands (Human Evaluation)												
BEXPLAIN	26	64	73	37	22	65	83	30	14	62	94	30
B2Com	20	57	77	46	18	61	87	34	10	52	99	39
HBCom	20	47	78	55	16	55	93	36	8	48	102	42
Ours	7	34	50	109	4	32	58	106	5	17	76	102
Single-Line Bash Commands (LLM Evaluation)												
BEXPLAIN	29	66	71	34	25	67	80	28	16	64	92	28
B2Com	23	59	75	43	21	62	85	32	13	52	97	38
HBCom	22	51	76	51	18	58	91	33	11	49	101	39
Ours	10	37	52	101	7	33	60	100	7	20	78	95
Multi-Line Bash Scripts (Human Evaluation)												
GPT-4.1	8	23	41	28	11	20	41	28	4	24	33	39
Claude-3.7	9	25	39	27	9	22	43	26	5	23	38	34
DeepSeek	13	27	37	23	14	23	40	23	11	18	42	29
Qwen2.5	15	27	37	21	13	23	39	25	13	19	40	28
Ours	7	20	42	31	11	18	42	29	5	19	42	34
Multi-Line Bash Scripts (LLM Evaluation)												
GPT-4.1	10	24	39	27	13	21	40	26	6	25	34	35
Claude-3.7	11	26	38	25	11	23	42	24	7	24	37	32
DeepSeek	15	28	36	21	16	24	39	21	13	20	40	27
Qwen2.5	17	28	36	19	15	24	38	23	15	21	38	26
Ours	9	21	41	29	13	19	41	27	7	20	41	32

Note: BEXPLAIN: BASHEXPLAINER, B2Com: Bash2Com, Claude-3.7: Claude-3.7-Sonnet, DeepSeek: DeepSeek-R1-Distill-Qwen-32B, Qwen2.5: Qwen2.5-Coder-32B.

HBCom's 63.5%, 62.0%, and 70.0%; Bash2Com's 59.0%, 58.5%, and 67.5%; and BASHEXPLAINER's 52.5%, 54.0%, and 60.0%. For multi-line Bash scripts, Bash-Commenter also led in human evaluation with Correctness (73.0%), Completeness (71.0%), and Naturalness (76.0%), surpassing GPT-4.1 and Claude-3.7-Sonnet, both of which scored no higher than 69.0% in Correctness or Completeness, and 72.0% in Naturalness. LLM evaluation corroborated these findings with our method achieving 70.0% in Correctness, 68.0% in Completeness, and 73.0% in Naturalness, while GPT-4.1 and Claude-3.7-Sonnet remained below 66.0%, 66.0%, and 70.0% respectively.

Answer to RQ3: Bash-Commenter has demonstrated its capability to generate comments for Bash commands that are correct, complete, and natural validated by both human and LLM evaluations, outperforming all baselines.

4.5.4 RQ4. Error Analysis. To better understand the limitations of Bash-Commenter, we performed a manual error analysis on 100 incorrect comments, comprising 50 single-line commands and 50 multi-line scripts, that received a 'Correctness' score of 1 or 2 in our human evaluation; two experts independently categorized the primary error in each comment, resolving disagreements through discussion, into six specific categories: **Parameter/Option Omission (PO)**, where the model correctly

identifies the command but omits key parameter functions; **Functionality Misinterpretation (FM)**, which misunderstands the core purpose of a command or pipeline; **Incompleteness (IC)**, denoting comments that are correct at a high level but lack crucial logic details; **Factual Hallucination (FH)**, where the model invents functionality not present in the script; **Over-generalization (OG)**, which is generic and fails to capture specific context; and **Language/Fluency Issues (LFI)**, covering comments containing grammatical errors or awkward phrasing.

Table 7. Distribution of Error Types for Bash-Commenter (%)

Error Category	Single-line Scripts (%)	Multi-line Scripts (%)
Parameter/Option Omission (PO)	38	18
Functionality Misinterpretation (FM)	24	28
Incompleteness (IC)	16	34
Factual Hallucination (FH)	8	6
Over-generalization (OG)	10	12
Language/Flsues (LFI)	4	2
Total	**100**	**100**

Our analysis (Table 7) identifies **Parameter/Option Omission (PO)** as the dominant error for **single-line commands** (38%), indicating the model grasps the main utility but misses flag-induced nuances. Conversely, **multi-line scripts** suffer chiefly from **Incompleteness (IC)** (34%), reflecting the difficulty of summarizing complex logic steps. To address PO, we propose Group Relative Policy Optimization (GRPO). By evaluating a group of candidates against a reward function, GRPO provides granular signals to master subtle flag impacts. For IC, Chain-of-Thought (CoT) fine-tuning offers a solution by enforcing a step-by-step logical breakdown, ensuring all crucial script components are captured in the final summary.

Answer to RQ4: Bash-Commenter's main errors are parameter omission (PO) for single-line commands and incompleteness (IC) for multi-line scripts. Future work can address these using GRPO for finer optimization and CoT fine-tuning.

5 Related Work

5.1 Bash Code Generation from Natural Language

Mapping natural language to Bash commands is challenging. Lin et al.[22] first addressed this with a corpus of over 9,000 Bash code-comment pairs covering 100+ utilities, evaluating Seq2Seq[43], CopyNet [12], and Tellina [22]. DocCGen [36] improved NL-to-DSL generation through a two-stage process: retrieving relevant library documentation, then generating code from syntax rules and templates, reducing syntax and semantic errors in complex DSLs like Bash and Ansible YAML. Bridge-Coder [69] addressed low-resource language code generation via "Code-Bridge," leveraging code and comments from high-resource languages to guide generation in low-resource languages.

5.2 Bash Code Comment Generation

For Bash code comment generation, Yu et al.[54] proposed BASHEXPLAINER via CodeBERT[8]-based encoding and information retrieval, Shen et al.[39] introduced Bash2Com with adversarial

training and a semantic-aware module, and Zhang et al.[70] proposed HBCom using a Heterogeneous Information Graph for syntactic-semantic integration. Despite this progress, these methods share three limitations: CodeBERT's 512-token ceiling prevents handling long multi-line scripts; datasets cover only hundreds of distinct commands (e.g., 466 in BASHEXPLAINER [54]), limiting practical diversity; and the absence of parameter-level semantic modeling causes errors such as misinterpreting `uniq -d` or missing `awk -F` and `uniq -i` semantics (Fig.1). Our work addresses these gaps by constructing a dataset of **17,122 distinct commands** (36.7×36.7× larger) with 50.6% multi-line scripts (Table1), adopting **LLaMA-3.1-8B** [10] with a 2048-token context and CPT on 676,524 Bash scripts, and proposing **SAPO**, the first preference learning framework using AST-based minimal pairs for fine-grained command semantics.

5.3 Automatic Code Comment Generation

Code comment generation has evolved from rule-based templates [16] to neural models like CodeNN [18]. Recent LLM-based approaches have achieved significant breakthroughs: Geng et al.[9] introduced intent-guided generation; Lu et al. [29] established the DeepCRCEval framework for reliable evaluation; while others improved performance through parameter-efficient fine-tuning [30]. Additionally, to better capture the structural semantics of source code, researchers have extensively explored graph neural networks (GNNs) [53, 60, 66, 68] to represent code structures [20]. However, these works primarily focus on general-purpose languages (e.g., Java, Python) rather than domain-specific languages like Bash.

6 Threats to Validity

Internal Validity: Internal validity threats include occasional output inconsistencies such as repetitive phrases or unintended inclusion of the input script in generated comments; these are mitigated by rule-based post-processing in the inference pipeline. Inference runs efficiently on a single 24GB VRAM GPU. The risk of systematic bias from LLM-generated reference text is addressed through 24-expert verification (Fleiss' Kappa = 0.78) and 500-sample validation. Future work will explore refined training objectives and architectural adjustments to further improve output quality.

External Validity: SAPO generates preference pairs via atomic mutations applied to a script's AST, but the semantic correctness of these mutations is implicitly learned from training data patterns. If the model lacks exposure to a command's valid argument structure, SAPO cannot construct meaningful correct-vs-incorrect pairs for it. Thus, while SAPO is fully automated, its effectiveness in new or specialized domains remains contingent on sufficient foundational data coverage for that domain.

7 Conclusion

We propose Bash-Commenter, an LLM-based approach for Bash code comment generation that employs a three-stage post-training pipeline: continual pre-training (CPT), supervised fine-tuning (SFT), and syntax-aware preference optimization (SAPO), supported by a comprehensive dataset constructed for each stage. Bash-Commenter significantly outperforms state-of-the-art methods on BLEU-4, METEOR, and ROUGE-L, generating comments that are correct, complete, and natural.

8 Data Availability

All the experimental data and source code is online available at https://zenodo.org/records/18743947

Acknowledgements

This work was supported by the National Key Research and Development Program of China (No.2023YFB3307203).

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, Online, 2655–2668. doi:10.18653/v1/2021.naacl-main.211

[2] Anthropic. 2025. Claude-3.7-Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet.

[3] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 25 (Feb. 2021), 29 pages. doi:10.1145/3434280

[4] Shiqi Cheng, Chenjie Shen, Li Yang, Lei Yu, Fengjun Zhang, and Chun Zuo. 2025. AUVANA: An Efficient and Automatic Approach to Variable Rename Refactoring via Large Pre-trained Language Model. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. 288–299. doi:10.1109/ISSRE66568.2025.00038

[5] Stack Overflow Community. 2025. Newest 'bash' Questions - Stack Overflow. https://stackoverflow.com/questions/tagged/bash.

[6] Stack Overflow Community. 2025. Newest 'shell' Questions - Stack Overflow. https://stackoverflow.com/questions/tagged/shell.

[7] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. 2023. Bash in the Wild: Language Usage, Code Smells, and Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 8 (Feb. 2023), 22 pages. doi:10.1145/3517193

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[9] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 39, 13 pages. doi:10.1145/3597503.3608134

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (July 2024). arXiv:2407.21783 [cs.AI] doi:10.48550/arXiv.2407.21783

[11] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Zhouchi Lin, Bowen Zhang, Lionel Ni, Wen Gao, Yuanzhuo Wang, and Jian Guo. 2026. A survey on LLM-as-a-judge. *The Innovation* 7, 6 (2026), 101253. doi:10.1016/j.xinn.2025.101253

[12] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1631–1640. doi:10.18653/v1/P16-1154

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.499

[14] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *Nature* 645 (2025), 633–638. doi:10.1038/s41586-025-09422-z

[15] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 223–226. doi:10.1145/1810295.1810335

[16] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering (WCRE '10)*. IEEE Computer Society, USA, 35–44. doi:10.1109/WCRE.2010.13

[17] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217. doi:10.1007/s10664-019-09730-9

[18] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Katrin Erk and Noah A. Smith (Eds.). Association for Computational Linguistics, Berlin, Germany, 2073–2083. doi:10.18653/v1/P16-1195

[19] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (Nov. 2022). arXiv:2211.15533 [cs.CL] doi:10.48550/arXiv.2211.15533

[20] Li Kuang, Cong Zhou, and Xiaoxian Yang. 2022. Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. *Automated Software Engineering* 29, 2 (2022), 43. doi:10.1007/s10515-022-00341-1

[21] Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. 2025. From Generation to Judgment: Opportunities and Challenges of LLM-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 2757–2791. doi:10.18653/v1/2025.emnlp-main.138

[22] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. 2017. *Program Synthesis from Natural Language Using Recurrent Neural Networks.* Technical Report UW-CSE-17-03-01. University of Washington, Department of Computer Science and Engineering, Seattle, WA, USA. https://dericpang.com/tellina-tr170510.pdf

[23] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018).* https://aclanthology.org/L18-1491

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] doi:10.48550/arXiv.2412.19437

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[26] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18).* Association for Computing Machinery, New York, NY, USA, 373–384. doi:10.1145/3238147.3238190

[27] Locutusque. 2024. UltraTextbooks Dataset. https://huggingface.co/datasets/Locutusque/UltraTextbooks.

[28] 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

[29] Junyi Lu, Xiaojia Li, Zihan Hua, Lei Yu, Shiqi Cheng, Li Yang, Fengjun Zhang, and Chun Zuo. 2025. Deepcrceval: Revisiting the evaluation of code review comment generation. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 43–64. doi:10.1007/978-3-031-90900-9_3

[30] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE).* 647–658. doi:10.1109/ISSRE59848.2023.00026

[31] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A Comprehensive Overview of Large Language Models. *ACM Trans. Intell. Syst. Technol.* 16, 5, Article 106 (Aug. 2025), 72 pages. doi:10.1145/3744746

[32] Cameron Newham, Bill Rosenblatt, and Gigi Estabrook. 1998. *Learning the Bash Shell.* O'Reilly & Associates, Inc.

[33] OpenAI. 2025. GPT-4.1. https://platform.openai.com/docs/models/gpt-4.1.

[34] Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. 2024. OpenWebMath: An Open Dataset of High-Quality Mathematical Web Text. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 20357–20379. https://proceedings.iclr.cc/paper_files/paper/2024/file/5949a8750a110ce1f0631b1776c500a2-Paper-Conference.pdf

[35] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty (Eds.). Association for Computational Linguistics, Online, 40–47. doi:10.18653/v1/2021.nlp4prog-1.5

[36] Sameer Pimparkhede, Mehant Kammakomati, Srikanth G. Tamilselvam, Prince Kumar, Ashok Pon Kumar, and Pushpak Bhattacharyya. 2024. DocCGen: Document-based Controlled Code Generation. In *Proceedings of the 2024 Conference*

on Empirical Methods in Natural Language Processing, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 18681–18697. doi:10.18653/v1/2024.emnlp-main.1040

[37] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 53728–53741. https://proceedings.neurips.cc/paper_files/paper/2023/file/a85b405ed65c6477a4fe8302b5e06ce7-Paper-Conference.pdf

[38] Chenjie Shen, Jie Zhu, Lei Yu, Li Yang, and Chun Zuo. 2024. Dependency-Aware Method Naming Framework with Generative Adversarial Sampling. In *2024 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN60899.2024.10651109

[39] Yiheng Shen, Xiaolin Ju, Xiang Chen, and Guang Yang. 2024. Bash comment generation via data augmentation and semantic-aware CodeBERT. *Automated Software Engineering* 31 (2024), 30. doi:10.1007/s10515-024-00431-2

[40] Stack Overflow. 2026. Bash Questions (1134 days post-ChatGPT). https://stackoverflow.com/questions/tagged/bash?sort=Newest&days=1134. Data collected: 2026-01-07, covering November 30, 2022 to January 7, 2026.

[41] Stack Overflow. 2026. Bash Questions (3000 day baseline). https://stackoverflow.com/questions/tagged/bash?sort=Newest&days=3000. Data collected: 2026-01-07, covering October 21, 2017 to January 7, 2026.

[42] André Storhaug, Jingyue Li, and Tianyuan Hu. 2023. Efficient Avoidance of Vulnerabilities in Auto-completed Smart Contract Code Using Vulnerability-constrained Decoding. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 683–693. doi:10.1109/ISSRE59848.2023.00035

[43] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2014/file/5a18e133cbf9f257297f410bb7eca942-Paper.pdf

[44] Jiayue Tang, Li Yang, Lei Yu, Junyi Lu, Zhirong Huang, Fengjun Zhang, and Chun Zuo. 2025. Breaking Task Isolation: Enhancing Code Review Automation with Mixture-of-Experts Large Language Models. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. 227–238. doi:10.1109/ISSRE66568.2025.00033

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[46] Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025. Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA086 (June 2025), 23 pages. doi:10.1145/3728963

[47] 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*, 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. doi:10.18653/v1/2021.emnlp-main.685

[48] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2025. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 204 (Aug. 2025), 25 pages. doi:10.1145/3714461

[49] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: a large-scale field study with professionals. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 584. doi:10.1145/3180155.3182538

[50] Zhu Xiaoxuan, Xiong Zhuozhi, Zhang Lin, Ye Haoning, Gu Zhouhong, Li Zihan, Jiang Sihang, Feng Hongwei, Xiao Yanghua, Wang Zili, Yang Dongjie, and Wang Shusen. 2023. CodeGPT: A Code-Related Dialogue Dataset Generated by GPT and for GPT. https://github.com/zxx000728/CodeGPT.

[51] An Yang, Anfeng Li, Baosong Yang, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] doi:10.48550/arXiv.2505.09388

[52] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] doi:10.48550/arXiv.2412.15115

[53] Yuanzhe Yang, Li Yang, Lingwei Li, Xiaoxiao Ma, Lei Yu, and Chun Zuo. 2023. DccGraph: Detecting Criminal Communities with Augmented Criminal Network Construction and Graph Neural Network. In *2023 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN54540.2023.10191121

[54] Chi Yu, Guang Yang, Xiang Chen, Ke Liu, and Yanlin Zhou. 2022. BashExplainer: Retrieval-Augmented Bash Code Comment Generation based on Fine-tuned CodeBERT. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 82–93. doi:10.1109/ICSME55016.2022.00016

[55] Lei Yu, Shiqi Chen, Hang Yuan, Peng Wang, Zhirong Huang, Jingyuan Zhang, Chenjie Shen, Fengjun Zhang, Li Yang, and Jiajia Ma. 2024. Smart-LLaMA: Two-Stage Post-Training of Large Language Models for Smart Contract Vulnerability Detection and Explanation. *arXiv preprint arXiv:2411.06221* (2024). doi:10.48550/arXiv.2411.06221

[56] Lei Yu, Shiqi Cheng, Zhirong Huang, Jingyuan Zhang, Chenjie Shen, Junyi Lu, Li Yang, Fengjun Zhang, and Jiajia Ma. 2025. SAEL: Leveraging Large Language Models with Adaptive Mixture-of-Experts for Smart Contract Vulnerability Detection. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 61–72. doi:10.1109/ICSME64153.2025.00016

[57] Lei Yu, Zhirong Huang, Hang Yuan, Shiqi Cheng, Li Yang, Fengjun Zhang, Chenjie Shen, Jiajia Ma, Jingyuan Zhang, Junyi Lu, and Chun Zuo. 2025. Smart-LLaMA-DPO: Reinforced Large Language Model for Explainable Smart Contract Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA009 (June 2025), 24 pages. doi:10.1145/3728878

[58] Lei Yu, Junyi Lu, Xianglong Liu, Li Yang, Fengjun Zhang, and Jiajia Ma. 2023. PSCVFinder: A Prompt-Tuning Based Framework for Smart Contract Vulnerability Detection. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 556–567. doi:10.1109/ISSRE59848.2023.00030

[59] Lei Yu, Peng Wang, Jingyuan Zhang, Xin Wang, Jia Xu, Li Yang, Changzhi Deng, Jiajia Ma, and Fengjun Zhang. 2026. SQL-Commenter: Aligning Large Language Models for SQL Comment Generation with Direct Preference Optimization. *arXiv preprint arXiv:2603.18606* (2026). doi:10.48550/arXiv.2603.18606

[60] Lei Yu, Fengjun Zhang, Jiajia Ma, Li Yang, Yuanzhe Yang, and Wei Jia. 2023. Who Are the Money Launderers? Money Laundering Detection on Blockchain via Mutual Learning-Based Graph Neural Network. In *2023 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN54540.2023.10191217

[61] Lei Yu, Jingyuan Zhang, Xin Wang, Jiajia Ma, Li Yang, and Fengjun Zhang. 2025. Towards Secure and Explainable Smart Contract Generation with Security-Aware Group Relative Policy Optimization. *arXiv preprint arXiv:2509.09942* (2025). doi:10.48550/arXiv.2509.09942

[62] Hang Yuan, Xizhi Hou, Lei Yu, Li Yang, Jiayue Tang, Jiadong Xu, Yifei Liu, Fengjun Zhang, and Chun Zuo. 2025. Leveraging Mixture-of-Experts Framework for Smart Contract Vulnerability Repair with Large Language Model. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1667–1679. doi:10.1109/ASE63991.2025.00140

[63] Hang Yuan, Lei Yu, Zhirong Huang, Jingyuan Zhang, Junyi Lu, Shiqi Cheng, Li Yang, Fengjun Zhang, Jiajia Ma, and Chun Zuo. 2025. Mos: Towards effective smart contract vulnerability detection through mixture-of-experts tuning of large language models. *arXiv preprint arXiv:2504.12234* (2025). doi:10.48550/arXiv.2504.12234

[64] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. 2024. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354* (2024). doi:10.48550/arXiv.2408.14354

[65] Ge Zhang, Scott Qu, Jiaheng Liu, Chenchen Zhang, Chenghua Lin, Chou Leuang Yu, Danny Pan, Esther Cheng, Jie Liu, Qunshu Lin, et al. 2024. MAP-Neo: Highly Capable and Transparent Bilingual Large Language Model Series. arXiv:2405.19327 [cs.CL] doi:10.48550/arXiv.2405.19327

[66] Jingyuan Zhang, Xin Wang, Lei Yu, Li Yang, and Fengjun Zhang. 2026. Binary Message Passing for Generalizable Semi-Supervised Graph Anomaly Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 16334–16342. doi:10.1609/aaai.v40i19.38671

[67] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1385–1397. doi:10.1145/3377811.3380383

[68] Jingyuan Zhang, Lei Yu, Zhirong Huang, Li Yang, and Fengjun Zhang. 2025. Topology Augmented Multi-Band and Multi-Scale Filtering for Graph Anomaly Detection. *ACM Trans. Knowl. Discov. Data* 19, 8, Article 151 (Sept. 2025), 27 pages. doi:10.1145/3748727

[69] Jipeng Zhang, Jianshu Zhang, Yuanzhe Li, Renjie Pi, Rui Pan, Runtao Liu, Ziqiang Zheng, and Tong Zhang. 2024. Bridge-Coder: Unlocking LLMs' Potential to Overcome Language Gaps in Low-Resource Code. arXiv:2410.18957 [cs.CL] doi:10.48550/arXiv.2410.18957

[70] Junsan Zhang, Yang Zhu, Ao Lu, Yudie Yan, and Yao Wan. 2025. Bash command comment generation via multi-scale heterogeneous feature fusion. *Automated Software Engineering* 32 (2025), 28. doi:10.1007/s10515-025-00494-9

[71] Yifan Zhang, Yifan Luo, Yang Yuan, and Andrew C Yao. 2024. Autonomous Data Selection with Language Models for Mathematical Texts. In *ICLR 2024 Workshop on Navigating and Addressing Data Problems for Foundation Models*. https://openreview.net/forum?id=bBF077z8LF

Received 2025-09-04; accepted 2026-03-24