Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

MABash: Multi-Agent Collaborative Parsing for Bash Command Comment Generation

Feifei Wang
Guizhou University
Guowei Shen
gwshen@gzu.edu.cn
Guizhou University
Yujie Fu
Guizhou University
Chun Guo
Guizhou University
Yunhe Cui
Guizhou University
Yi Chen
Guizhou University

Research Article

Keywords: Bash command, Comment generation, Multi-agent systems, Abstract syntax tree, Large language models

Posted Date: June 1st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9709428/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

MABash: Multi-Agent Collaborative Parsing for Bash Command Comment Generation

Feifei Wang[1,2], Guowei Shen[1,2*], Yujie Fu[1,2], Chun Guo[1,2], Yunhe Cui[1,2], Yi Chen[1,2]

[1*]The State Key Laboratory of Public Big Data, College of Computer Science and Technology, Guizhou University, Guiyang, 550025, China.
[2]The Engineering Research Center of Text Computing & Cognitive Intelligence, College of Computer Science and Technology, Guizhou University, Guiyang, 550025, China.

*Corresponding author(s). E-mail(s): gwshen@gzu.edu.cn;
Contributing authors: gs.ffwang24@gzu.edu.cn; gs.yjfu24@gzu.edu.cn;

Abstract

Bash command comment generation aims to automatically generate natural-language descriptions of Bash commands to facilitate program comprehension and maintenance. Existing approaches typically treat commands as linear token sequences or encode structural information implicitly within monolithic neural architectures, which limits their ability to capture complex logical relations and constraint scopes accurately. This paper proposes MABash, a structured multi-agent collaborative framework with a shared blackboard architecture for generating Bash command comments. The framework explicitly decomposes command understanding into lexical, syntactic, and semantic parsing stages, each handled by specialized agents. Intermediate representations are incrementally constructed and maintained within a shared blackboard, enabling cross-stage reasoning and explicit modeling of semantic constraints. A LoRA-finetuned DeepSeek-Coder-6.7B model is then employed to generate comments conditioned on the final structured semantic state. Experiments conducted on a dataset of 10,592 Bash command–comment pairs demonstrate that MABash consistently outperforms existing approaches. Compared with the state-of-the-art baseline HBCom, MABash improves BLEU-4 by 1.56%, ROUGE-L by 2.85%, and METEOR by 2.73%. Further analysis shows that the proposed multi-agent decomposition is particularly effective in handling commands with complex logical structures.

Keywords: Bash command, Comment generation, Multi-agent systems, Abstract syntax tree, Large language models

1 Introduction

Bash (Bourne Again Shell) is one of the most widely used command-line interpreters and scripting languages in Unix-like operating systems [Milligan and Baker2014]. Its flexible syntax, strong automation capabilities [Mazurak and Zdancewic2007], and support for batch processing make it essential in system development, software deployment, and operations management. However, this flexibility also increases the difficulty of comprehension and maintenance. Bash commands often embody complex execution logic in a compact symbolic form with implicit structure and densely compressed semantics, posing substantial challenges for developers—particularly those without extensive Bash experience [Kidwai et al.2021]. This issue remains prevalent in practice. As of April 2026, more than 156,756 questions tagged "bash" have been posted on Stack Overflow, indicating persistent difficulties in understanding and using Bash commands in real-world scenarios. Fig. 1 illustrates a representative Stack Overflow discussion that highlights the need for automated Bash command comment generation.



Fig. 1 A Stack Overflow discussion on the comprehension of a Bash command

With advances in deep learning, researchers have explored neural approaches to automatic Bash comment generation. Existing studies predominantly adopt encoder–decoder architectures, typically fine-tuning pretrained models such as CodeBERT and incorporating retrieval augmentation or semantic fusion strategies to map Bash commands into continuous representations for natural language generation. Prior studies [Yu et al.2022] and [Shen et al.2024] demonstrate the potential of deep learning models for this task. However, most approaches treat Bash commands as

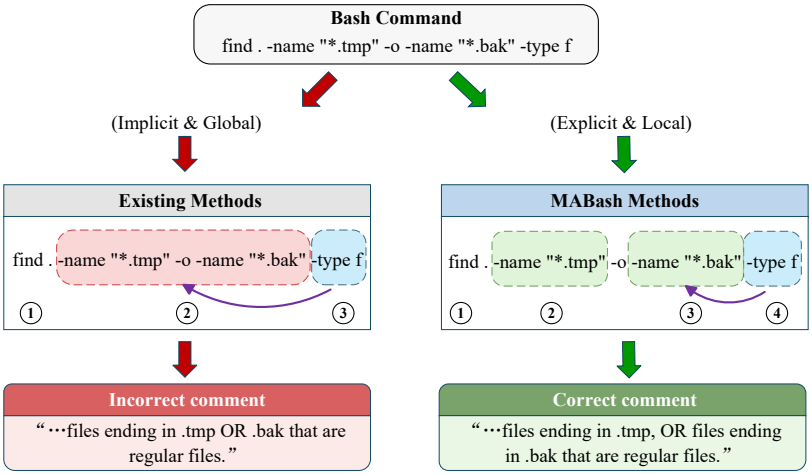


Fig. 2 Comparison between implicit global modeling and explicit local multi-agent modeling for Bash command understanding

plain-text sequences, overlooking their explicit syntactic structures and execution-logic constraints. Even the HBCom model proposed by [Zhang et al.2025], which introduces abstract syntax tree (AST)-based structural information and hierarchical instruction graphs, still follows a monolithic end-to-end paradigm without explicitly decomposing and coordinating different abstraction levels in the program comprehension process. Consequently, these methods struggle to accurately model complex logical compositions and conditional scopes, often leading to semantic misinterpretation and overgeneralization of constraints, particularly for structurally complex Bash commands.

These limitations become particularly evident in Bash commands with complex logical compositions. Consider the example shown in Fig. 2: `find . -name "*.tmp" -o -name "*.bak" -type f`. This command searches for all files or directories ending in `.tmp` and regular files ending in `.bak`. Since the precedence of `-o` is lower than that of the implicit `-a`, the constraint `-type f` applies only to the second branch rather than to the entire expression. However, existing methods, such as HBCom [Zhang et al.2025], encode structural information into implicit vector representations, thereby failing to delineate the scope of conditional constraints precisely. As a result, models may generate incorrect comments such as `find all regular files ending with .tmp or .bak` , which mistakenly generalizes `-type f` as a global constraint and deviates from the command's actual semantics.

To address these limitations, we propose MABash, a multi-agent collaborative framework for Bash command comment generation that explicitly decomposes the program comprehension process. The framework partitions command understanding into several coordinated analysis stages, each handled by a dedicated parsing agent. In the motivating example, the syntactic agent identifies the structural position of the logical operator `-o` and its branch relationships. In contrast, the semantic agent determines the scope of conditional constraints according to Bash execution rules, explicitly distinguishing local predicates from global ones. The derived semantic facts

are stored as structured intermediate states on a shared blackboard and subsequently used to constrain the language model during comment generation, preventing local conditions from being incorrectly generalized. By decomposing reasoning processes and introducing controllable intermediate representations, MABash enables more accurate modeling of multi-level semantics and conditional constraints in complex Bash commands.

The main contributions of this work are summarized as follows:

- We propose MABash, a multi-agent collaborative framework for Bash command comment generation that explicitly decomposes the program comprehension process into lexical, syntactic, and semantic analysis stages, enabling systematic modeling of the multi-level structural characteristics of Bash commands.
- We design three specialized parsing agents for lexical, syntactic, and semantic analysis, and introduce a shared blackboard mechanism that supports structured intermediate states and coordinated reasoning among agents, improving interpretability and structural constraint modeling.
- We integrate a LoRA-finetuned generative large language model as the comment generation module and incorporate beam search during inference, enabling efficient task adaptation and improved generation quality.
- We conduct extensive experiments on a Bash benchmark dataset. Experimental results demonstrate that MABash consistently outperforms strong baselines across multiple automatic evaluation metrics, including retrieval-based approaches, deep learning models, pretrained model-based methods, and Bash-specific comment generation techniques.

2 Related works

Related work can be categorized into five groups: information retrieval–based methods, deep learning–based methods, pretrained model–based approaches, task-specific methods for Bash comment generation, and multi-agent system–based approaches.

2.1 Code comment generation

Methods based on information retrieval. Early research on code summarization largely drew upon information retrieval techniques, based on the assumption that functionally similar code fragments share similar descriptions. [Kuhn et al.2007] first applied latent semantic indexing (LSI) to source code clustering. [Haiduc et al.2010] employed vector space models and LSI to generate class- and method-level summaries. [Zhang et al.2020] further incorporated syntactic and semantic similarity to improve retrieval quality, while [Wei et al.2020] proposed reusing retrieved comments as guidance for generation. Although these methods are relatively simple to implement, they heavily depend on the scale and quality of the code repository and struggle to capture complex semantic relationships and structural logic.

Methods based on deep learning. Deep learning–based approaches to code comment generation typically formulate the task as a neural machine translation problem, learning an end-to-end mapping from source code to natural language

descriptions. [Iyer et al.2016] introduced CODE-NN, which employs LSTM with attention for code summarization. Subsequent work incorporated structural and syntactic information to improve performance. Ahmad et al. [Ahmad et al.2020] adopted a Transformer with relative positional encoding to model token relationships. [Hu et al.2018] proposed DeepCom, which integrates semantic and structural features, and later introduced SBT [Hu et al.2020] to serialize AST structures via traversal. [Guo et al.2022] designed positional encodings for AST nodes to model sequential and structural representations jointly. [Shi et al.2023] combined AST and control-flow information, while [Hellendoorn et al.2019] and [Wu et al.2021] integrated structural biases into self-attention mechanisms. In addition, graph neural networks have been widely used for structural modeling. [Choi et al.2021] applied graph convolutional networks to encode ASTs alongside Transformers. Other studies leveraged AST positional information [Gong et al.2022], priority-based traversals [Yang et al.2021], Tree-LSTM variants [Shido et al.2019], and graph-based encoders [Fernandes et al.2018, LeClair et al.2020]. [Zhang et al.2020] further combined retrieval with an encoder–decoder framework, using GCNs to capture AST structure for improved summarization performance.

Methods based on pre-trained models. With the advancement of large-scale pre-trained models, [Feng et al.2020] introduced CodeBERT, a bimodal pretraining model for programming and natural languages. [Ahmad et al.2021] subsequently proposed PLBART, which adopts a denoising sequence-to-sequence pretraining strategy to leverage unlabeled data. [Guo et al.2022] presented UniXcoder, enhancing code representations through masked attention matrices and cross-modal content modeling. [Wang et al.2021] proposed CodeT5, built upon the T5 architecture to support bidirectional transformation between natural and programming languages. More recently, large language models such as GPT-4o and DeepSeek-V2.5, pretrained autoregressively on massive corpora, have demonstrated strong semantic understanding capabilities. Empirical studies by [Geng et al.2024], [Ghaemi et al.2024], [Su and McMillan2024], [Su et al.2026], [Majumdar et al.2026], and [Oli et al.2023] further confirm the superior performance of large language models in code summarization tasks.

2.2 Bash comment generation

Bash command comment generation aims to automatically generate natural-language descriptions of Bash commands or scripts to facilitate code understanding and maintenance. However, publicly available datasets are limited; for example, the dataset constructed by [Yu et al.2022] contains only 10,592 command–comment pairs, restricting large-scale training. [Yu et al.2022] conducted the first systematic study on this task and proposed BASHEXPLAINER. Their approach adopts a two-stage strategy: fine-tuning a CodeBERT-based encoder for Bash commands, followed by retrieval-augmented generation. While effective, it mainly enhances input representations through retrieval and provides limited modeling of complex syntactic and semantic structures within commands. Subsequent work explored richer semantic modeling. [Shen et al.2024] proposed Bash2Com, a dual-module architecture that improves robustness and leverages CodeBERT with an LSTM and attention for generation. [Zhang et al.2025] introduced HBCom, which constructs a heterogeneous information

graph from the AST to integrate token and structural information, using multi-scale graph neural networks and a copy-enhanced Transformer decoder for comment generation. More recently, large language models have introduced a new paradigm. [Shen et al.2025] proposed BashAgent, built on DeepSeek-Coder, which employs self-generated chain-of-thought prompting to enhance reasoning and generates comments without fine-tuning, outperforming traditional baselines.

2.3 Multi-Agent Systems

A Multi-Agent System (MAS) consists of multiple autonomous agents that collaborate to accomplish complex tasks. It has been widely applied in distributed computing, automated planning, and cybersecurity due to its strengths in modeling complex environments and dynamic decision-making [Wooldridge2009]. Early MAS research was largely based on symbolic reasoning and game theory [Shoham and Leyton-Brown2008], relying on predefined state spaces and utility functions. Although interpretable, such approaches suffer from state-space explosion, high rule-design costs, and limited adaptability in open or high-dimensional environments.

With advances in deep learning, large language models (LLMs) have provided a new foundation for multi-agent collaboration [Brown et al.2020]. LLM-based agents communicate and coordinate through natural language, enabling role specialization and cooperative reasoning without explicitly designed protocols [Park et al.2023]. These systems typically employ message passing, shared memory, or blackboard mechanisms for coordination. Compared with traditional MAS, natural language interaction offers greater flexibility and semantic richness, making it suitable for tasks with structural uncertainty or complex semantics [Hong et al.2023]. Moreover, the contextual reasoning capabilities of LLMs support reflection and strategy adjustment during interaction, thereby improving coordination and decision quality [Yao et al.2023]. In summary, LLM-enhanced multi-agent systems provide a flexible and scalable paradigm for task decomposition, collaborative reasoning, and complex semantic modeling, offering a foundation for multi-agent frameworks targeting sophisticated command understanding and interpretation [Xi et al.2023].

3 Method

3.1 MABash Framework

MABash is a two-stage framework for automatic Bash command comment generation, consisting of a command parsing stage and a comment generation stage. The framework explicitly models the structural and semantic properties of Bash commands and produces interpretable intermediate representations that guide natural language generation. An overview of the framework is illustrated in Fig. 3.

In the parsing stage, MABash performs structured analysis of the input command through a multi-agent collaboration mechanism. Specialized agents are responsible for lexical, syntactic, and semantic analysis, respectively. The outputs of these agents are incrementally integrated into a unified structured representation that captures both syntactic dependencies and execution semantics.

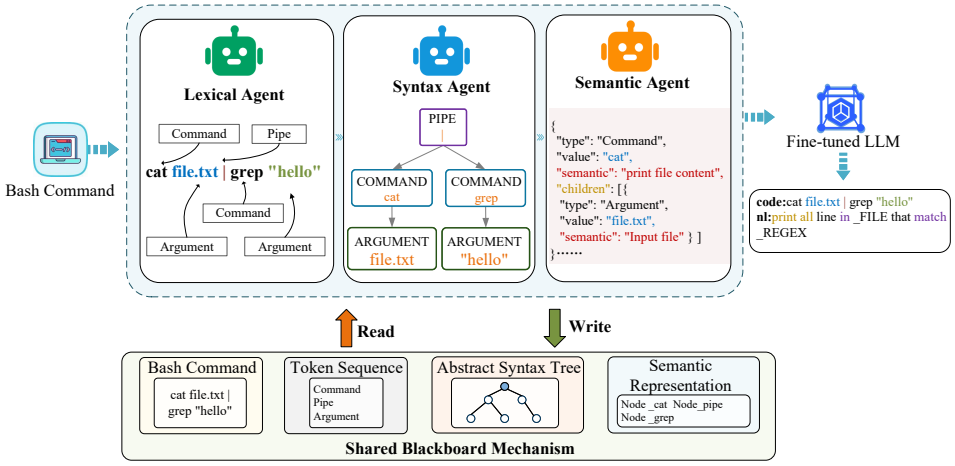


Fig. 3 Overview of the MABash multi-agent collaborative framework with shared blackboard mechanism

In the generation stage, the structured representation produced by the parsing agents is used as conditional input to a generative large language model to generate the corresponding natural-language comment. The generation model is initialized from a pretrained code-oriented LLM and further fine-tuned for the Bash comment generation task. This design enables the model to effectively leverage structural and semantic information extracted during parsing, thereby achieving an end-to-end mapping from Bash commands to descriptive comments.

3.2 Multi-Agent System

Lexical Agent

The lexical parsing agent, denoted as A_{lex}, performs symbol-level analysis of the input Bash command and transforms the raw character sequence into a structured token sequence. The agent reads the original command layer L_1 from the shared parsing state and writes the resulting tokens to the token layer L_2, which serves as input for subsequent syntactic analysis.

Formally, given the shared parsing state B_t at the time step t, the original command is represented as a sequence of characters:

$$L_1^t = (c_1, c_2, \ldots, c_n), \quad c_i \in \Sigma \tag{1}$$

where Σ denotes the alphabet of characters.

The lexical parsing process can be formalized as a mapping function:

$$\mathcal{T} : \Sigma^n \to \mathcal{X}^m \tag{2}$$

where $\mathcal{X}$ denotes the lexical token space. The parsing result is an ordered token sequence:

$$L_2^{t+1} = (x_1, x_2, \ldots, x_m) \tag{3}$$

where each token $x_i = (\tau_i, v_i)$, with τ_i representing the token type and v_i denoting the corresponding string value.

To facilitate the recovery of implicit structural dependencies in the subsequent syntactic analysis stage, the lexical agent performs additional structural annotation. Specifically, it explicitly marks logically related symbols and introduces placeholder tokens for implicit conjunctions or omitted structural relations that frequently occur in Bash commands. This enhancement process can be formally expressed as:

$$L_2^{t+1} = \phi(L_2^{t+1}) \tag{4}$$

where $\phi(\cdot)$ denotes the implicit-structure annotation function that preserves the original token order while augmenting the sequence with additional structural markers.

The overall state transition of the lexical parsing process can therefore be expressed as:

$$B_{t+1} = F_{\text{lex}}(B_t) \tag{5}$$

where $F_{\text{lex}}(\cdot)$ updates only the token layer L_2 in the shared parsing state while leaving other layers unchanged. Through this process, the original Bash command is transformed into a type-aware, structurally recoverable token sequence that serves as standardized input for subsequent syntactic analysis.

Syntactic Agent

The syntactic parsing agent, denoted as A_{syn}, constructs a structured syntactic representation of a Bash command based on the results of lexical analysis. Its primary objective is to capture the hierarchical organization of operators, conditional composition relationships, and structural constraints within the command. The agent takes the token layer L_2 from the shared parsing state as input and writes the derived syntactic structure into the syntax layer L_3.

Given the shared parsing state B_t at time step t, the token sequence generated by lexical parsing is represented as:

$$L_2^t = (x_1, x_2, \ldots, x_m) \tag{6}$$

where x_i denotes the lexical tokens obtained during the lexical parsing stage.

The syntactic parsing process can be formalized as a mapping from the token sequence to an abstract syntactic structure:

$$P : \mathcal{X}^m \to \mathcal{T} \tag{7}$$

where $\mathcal{T}$ denotes the abstract syntax tree (AST) space.

During parsing, the syntactic agent follows Bash grammar rules and operator precedence principles to construct the AST incrementally, as illustrated in Fig. 4. Different types of operators are mapped to distinct syntax nodes, and their hierarchical positions in the tree explicitly encode the binding order of logical operations. This hierarchical representation eliminates structural ambiguity that arises from purely linear token sequences and provides an explicit structural model of command composition.

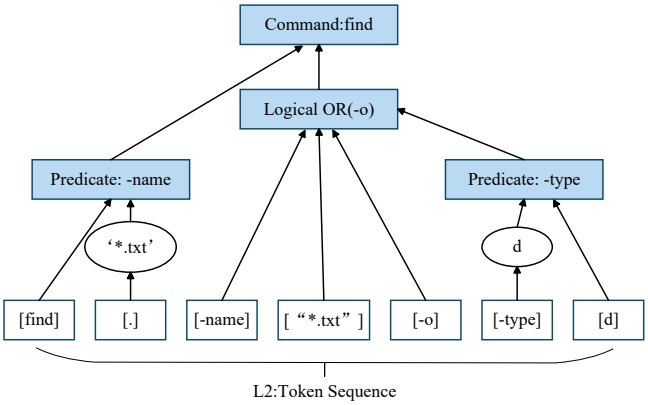


Fig. 4 Abstract Syntax Tree Construction by the syntactic agent from token sequence

The overall state transition of the syntactic parsing process is defined as:

$$B_{t+1} = F_{\text{syn}}(B_t) \tag{8}$$

where $F_{\text{syn}}(\cdot)$ updates only the syntax layer L_3 in the shared parsing state while leaving other layers unchanged. Through syntactic parsing, the token-level linear representation is transformed into an explicit hierarchical structure, which provides the structural foundation for determining the scope of conditional constraints in the subsequent semantic analysis stage.

Semantic Agent

The semantic parsing agent, denoted as A_{sem}, performs structured modeling of the execution semantics of a Bash command based on the syntactic representation produced in the previous stage. It takes the syntax layer L_3 in the shared parsing state as input and determines the scope and binding relationships of conditional constraints according to Bash execution rules. The derived semantic information is then written into the semantic layer L_4.

Given the shared parsing state B_t at time step t, where L_3^t denotes the abstract syntactic structure constructed during the syntactic parsing stage, the semantic parsing process can be formalized as a mapping from syntactic structures to a set of semantic constraints:

$$S : \mathcal{T} \rightarrow \mathcal{C} \tag{9}$$

where $\mathcal{C}$ denotes the semantic constraint space.

To explicitly characterize the applicability of conditional constraints during command execution, the semantic agent identifies conditional nodes and logical composition nodes within the syntactic structure. For each conditional node v, it defines a semantic scope $\text{Scope}(v)$, which represents the syntactic subtree constrained by that condition during execution. The determination of the condition scope is jointly governed by the syntactic hierarchy and the type of logical operator, as illustrated in Fig. 5.

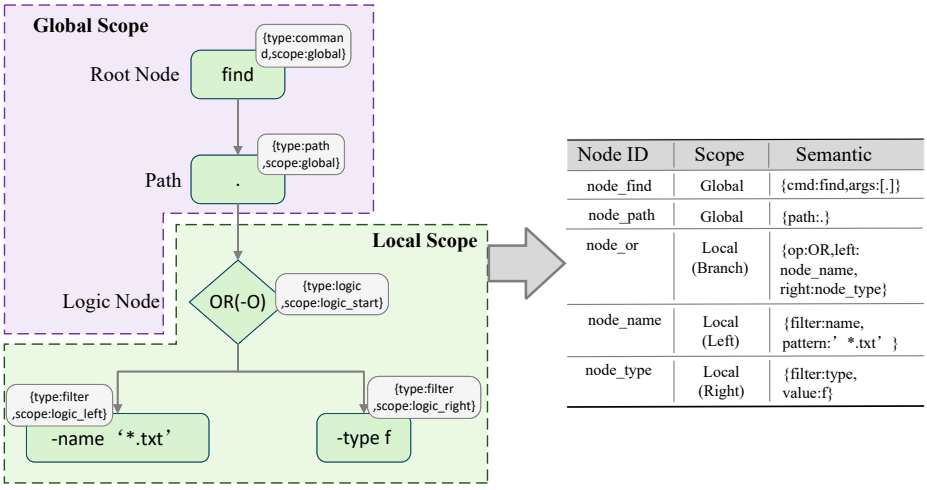


Fig. 5 Scope and execution path analysis in the semantic agent

Furthermore, to capture the semantic differences introduced by various logical compositions, the semantic agent models execution paths by operator type. For logical OR relations, distinct branches correspond to mutually exclusive execution paths. In contrast, for implicit AND relations, associated conditions hold simultaneously within the same execution path. Based on this distinction, a set of effective execution paths $\Pi(v)$ is constructed for each conditional node v, representing the contexts in which the condition applies across alternative branches.

By integrating the conditional node, its scope, and its execution paths, each semantic constraint can be represented as an explicit semantic triple:

$$c = \langle v, \text{Scope}(v), \Pi(v) \rangle \tag{10}$$

All such constraints are aggregated to form the updated semantic state L_4^{t+1}. The semantic parsing stage updates only the semantic layer in the shared state:

$$B_{t+1} = F_{\text{sem}}(B_t) \tag{11}$$

Through this process, the implicitly embedded conditional relationships in the command are transformed into an explicit, structured semantic representation. This prevents unintended merging or incorrect generalization of condition scopes during generation. The resulting semantic state is subsequently provided as explicit input to the explanation generation module, thereby imposing structural constraints on the decoding of natural-language comments.

Shared Blackboard

To coordinate the collaborative behavior of the lexical, syntactic, and semantic agents within a unified parsing workflow, this work introduces an explicit shared parsing state in the multi-agent system as the sole carrier for cross-stage information aggregation and transmission. The shared state records the incremental parsing results of a Bash command from the symbolic level to the semantic level. It serves as the

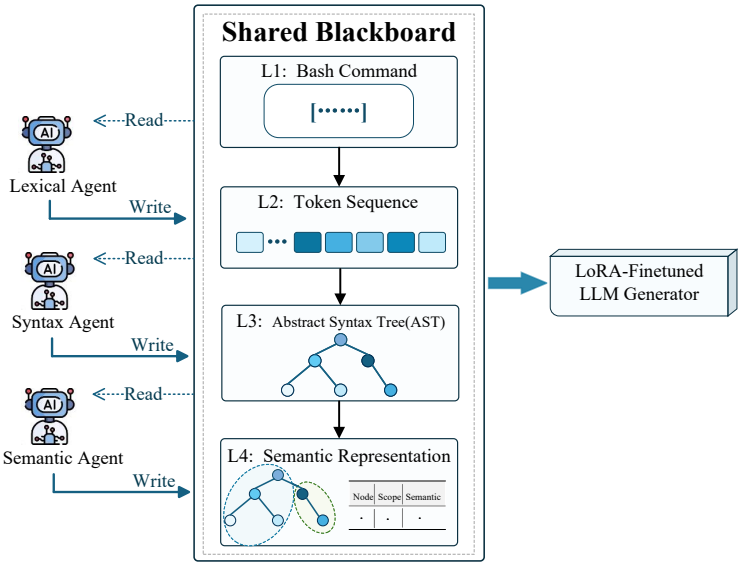


Fig. 6 Shared blackboard with layered parsing state and agent write permissions

structured input for the subsequent generation stage. At any parsing time step t, the global parsing state of the system is defined as:

$$B_t = (L_1^t, L_2^t, L_3^t, L_4^t) \tag{12}$$

where L_1^t denotes the original command sequence, L_2^t represents the token sequence obtained from lexical analysis, L_3^t corresponds to the abstract syntax tree (AST), and L_4^t stores the structured semantic constraint set. As illustrated in Fig. 6, the shared blackboard mechanism enforces stage consistency and result verifiability by restricting each agent's read and write permissions over the state space.

Specifically, the lexical agent A_{lex} has write access only to layer L_2^t; the syntactic agent A_{syn} updates exclusively layer L_3^t; and the semantic agent A_{sem} modifies only layer L_4^t. Meanwhile, all agents may read lower-level parsing results. This constraint can be formally expressed as:

$$\text{Write}(A_i) \subseteq \{L_i\}, \quad \text{Read}(A_i) \subseteq \{L_1, \ldots, L_i\} \tag{13}$$

The multi-agent collaborative parsing process is modeled as a sequence of state transitions:

$$B_{t+1} = F_{a_t}(B_t; \theta_{a_t}), \quad a_t \in \{A_{\text{lex}}, A_{\text{syn}}, A_{\text{sem}}\} \tag{14}$$

where a_t denotes the agent scheduled for execution at time t. The scheduling policy $\pi(\cdot)$ selects the next agent based on the current shared state:

$$a_t = \pi(B_t) \tag{15}$$

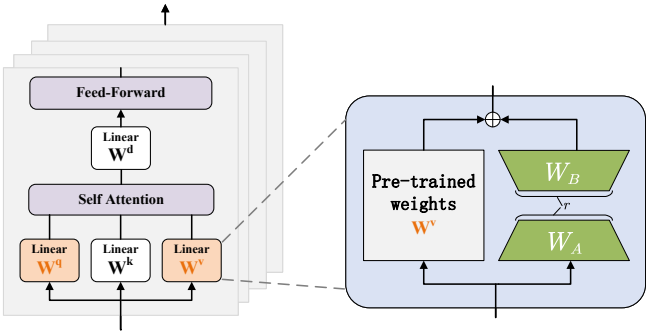


Fig. 7 LoRA Fine-Tuning Mechanism Diagram

The key role of the shared blackboard mechanism is to explicitly preserve and propagate condition scopes. The semantic agent writes structured constraints to layer L_4^t, including the condition itself, its bound syntactic subtree, and the associated execution path set. These semantic states are directly accessed during the generation stage instead of being implicitly encoded in compressed vector representations.

3.3 LLM-Based Comment Generator

This work adopts DeepSeek-Coder-6.7B-Instruct as the backbone model for the explanation generation stage. The model input is constructed by serializing and concatenating the original command sequence C with the shared blackboard parsing state B_t, thereby enabling conditional reasoning during generation.

Model Fine-Tuning

In the explanation generation stage, we employ Low-Rank Adaptation (LoRA) [Hu et al.2022] to perform parameter-efficient fine-tuning of DeepSeek-Coder-6.7B-Instruct, adapting the model to the parsing-constrained conditional comment generation task. The backbone parameters remain frozen, while trainable low-rank adapters are inserted into the linear projection layers of the attention modules. This design enables the model to incorporate structured semantic constraints derived from the multi-agent parsing stage while introducing only a small number of additional parameters. For a linear transformation matrix $W \in \mathbb{R}^{d_{out} \times d_{in}}$, LoRA parameterizes the weight update as:

$$W' = W + \Delta W, \quad \Delta W = BA \tag{16}$$

where $A \in \mathbb{R}^{r \times d}$, $B \in \mathbb{R}^{d \times r}$, and $r \ll d$. Only A and B are optimized during fine-tuning, while W remains fixed.

In our implementation, LoRA adapters are applied to the Query, Key, and Value projection layers of the multi-head self-attention mechanism, as shown in Fig. 7. The model takes the original Bash command sequence C together with the final shared blackboard state B_T as input and generates a natural-language comment sequence Y [Vaswani et al.2017]. The generation process is modeled as a conditional probability distribution:

$$P(Y \mid C, B_T) \tag{17}$$

where the shared state B_T provides explicit semantic constraints derived from the parsing stage. The model is trained by minimizing the negative log-likelihood objective:

$$\mathcal{L}_{\text{LoRA}} = -\sum_{i=1}^{|Y|} \log P(y_i \mid y_{<i}, C, B_T) \tag{18}$$

During inference, the frozen backbone operates jointly with the learned low-rank adapters to generate comments conditioned on the parsing results. By explicitly incorporating the shared blackboard state into the decoding process, the model ensures that the generated explanations remain consistent with the semantic constraints identified by the multi-agent parsing framework.

Inference Optimization

During inference, we employ beam search as the decoding strategy to mitigate the susceptibility of greedy decoding to local optima in Bash command comment generation [Bahdanau et al.2015]. Unlike greedy decoding, which preserves only a single hypothesis at each step, beam search maintains multiple high-probability candidate sequences, enabling the model to explore alternative expression paths and improve both semantic completeness and linguistic fluency.

Given an input Bash command sequence C together with the final shared blackboard state B_T, the generation process is modeled as the conditional probability factorization:

$$P(Y \mid C) = \prod_{i=1}^{m} P(y_i \mid y_{<i}, C) \tag{19}$$

where $Y = \{y_1, \ldots, y_m\}$ denotes the generated natural-language comment sequence [Sutskever et al.2014]. Beam search performs approximate optimal decoding within this conditional probability space.

At each decoding step, the model expands candidate sequences and retains the top-b hypotheses with the highest accumulated log-probability. The decoding process is deterministic and does not involve stochastic sampling, ensuring stable and reproducible inference results.

To reduce bias toward overly short outputs, a length_penalty $= 1.0$ is applied during hypothesis scoring. In our implementation, the beam width is set to $b = 4$, and the maximum generation length is limited to $L_{\max} = 128$ newly generated tokens. Decoding terminates early when all candidate sequences satisfy the stopping criteria, and the highest-scoring hypothesis is selected as the final generated comment.

4 Experimental Setup

4.1 Dataset

The dataset used in this study is adopted from [Yu et al.2022], who constructed it for the Bash code comment generation task. It contains 10,592 pairs of ⟨bash command, comment⟩ instances. The dataset is split into training, validation, and test

sets, comprising 8,474, 1,059, and 1,059 samples, respectively. To ensure experimental consistency and comparability, we follow the original data partition without any modification.

4.2 Evaluation Metrics

To comprehensively assess MABash's performance on the Bash comment generation task, we use seven widely used automatic evaluation metrics, including BLEU-1/2/3/4, overall BLEU, ROUGE-L, and METEOR.

BLEU measures n-gram precision between generated and reference texts and incorporates a brevity penalty to discourage overly short outputs. In addition to reporting the overall BLEU score, we report BLEU-1 through BLEU-4 to reflect performance at different n-gram granularities. BLEU is computed using the NLTK toolkit, with uniform n-gram weights set to 0.25.

ROUGE-L emphasizes recall-oriented evaluation by measuring the longest common subsequence between generated and reference texts, making it suitable for assessing how completely a comment captures the functionality of a command.

METEOR extends beyond exact token matching by incorporating stemming, synonym matching, and word order alignment, thereby better reflecting fluency and semantic correctness.

To ensure consistency with baseline evaluations, BLEU is computed using NLTK, whereas the remaining metrics are computed using the nlg-eval toolkit.

4.3 Baselines

4.3.1 Information retrieval-based approaches

LSI [Haiduc et al.2010] and **VSM** [Haiduc et al.2010] adopt text retrieval techniques. Specifically, LSI selects the most relevant terms in a reduced latent semantic space, while VSM operates in the original vector space; both rely on cosine similarity to rank candidate summaries. **BM25** [Zhang et al.2020] computes document–query relevance based on term frequency and inverse document frequency, while additionally incorporating document length normalization. In contrast, **NNGen** [Liu et al.2018] employs a nearest-neighbor algorithm to generate commit messages by retrieving similar historical instances. **CCGIR** [Yang et al.2022] leverages information retrieval techniques to identify the most similar code fragments in a repository from semantic, lexical, and syntactic perspectives, and then reuses their associated comments.

4.3.2 Deep learning-based approaches

Transformer [Ahmad et al.2020] leverages relative positional representations to model pairwise relationships between source code tokens and employs self-attention to capture contextual dependencies. **CODE-NN** [Iyer et al.2016] encodes code tokens into contextual vectors using LSTM networks with an attention mechanism. **Hybrid-DeepCom** [Hu et al.2020] integrates both source code and AST information for code summarization. **Rencos** [Zhang et al.2020] combines AST representations with serialized tokens to retrieve code snippets that are syntactically and semantically

similar to the input. **SG-Trans** [Gao et al.2023] injects local symbolic information and global syntactic features into the self-attention module of a Transformer to preserve hierarchical code characteristics. **CODESCRIBE** [Guo et al.2022] introduces triplet positional encoding to model hierarchical syntactic structures in source code. **CoSS** [Shi et al.2023] integrates syntactic structures with sequential representations and models structural heterogeneity using eight types of directed edges to capture semantic features better.

4.3.3 Pre-trained model-based approaches

CodeBERT [Feng et al.2020] is a bimodal pre-trained model designed for programming and natural language. It adopts a hybrid training objective and leverages large-scale pre-training followed by task-specific fine-tuning. **PLBART** [Ahmad et al.2021] employs a denoising sequence-to-sequence pre-training strategy, effectively utilizing large amounts of unlabeled code and natural language data. **UniXcoder** [Guo et al.2022] introduces a masked attention matrix with prefix adapters to control model behavior and enhances code representations using cross-modal information such as ASTs and code comments. **CodeT5** [Wang et al.2021] is built upon the T5 architecture and incorporates key token information to enable bidirectional transformation between natural language and programming languages.

4.3.4 Bash comment generation approaches

BASHEXPLAINER [Yu et al.2022] adopts a two-stage strategy for automatic Bash comment generation. First, it fine-tunes a Bash encoder based on CodeBERT. It then retrieves similar code samples, aggregates their vector representations, and generates comments through a decoder. **Bash2Com** [Shen et al.2024] employs a dual-module architecture: the first module enhances stability, while the second leverages CodeBERT to mine semantic information. A Transformer decoder combined with beam search is subsequently used for comment generation. **HBCom** [Zhang et al.2025] proposes a multi-scale heterogeneous feature fusion framework for Bash comment generation. It constructs an AST-based heterogeneous information graph (HIG) with six edge types to integrate syntactic structure and code-sequence information. It employs a multi-scale graph neural network, a Transformer decoder, and a multi-head attention copy mechanism to generate high-quality comments.

4.4 Parameter Settings

We conduct a multi-model comparative study using three code-oriented large language models as backbone architectures: DeepSeek-Coder-6.7B-Instruct, Meta-Llama-3.1-8B-Instruct, and Qwen3-8B. Experiments are performed on a high-performance computing environment equipped with NVIDIA A6000 GPUs, leveraging multi-GPU parallelism to accelerate training and improve computational efficiency. During data preprocessing, each Bash command is concatenated with its corresponding natural language explanation to form a complete instruction-following sequence.

Table 1 LoRA Fine-Tuning Hyperparameter Configuration

Parameter	DeepSeek	Llama	Qwen
Target Modules	q,k,v_proj	q,k,v_proj	q,k,v_proj
LoRAγ	8	8	8
LoRAα	32	32	32
LoRA dropout	0.1	0.1	0.1
Batch_size	4	4	4
Epochs	10	8	10
Learning_rate	2e-4	1e-4	2e-4

During training, all models use a unified LoRA configuration to ensure experimental comparability. Gradient checkpointing is enabled to optimize GPU memory usage. The detailed LoRA hyperparameter settings are summarized in Table 1.

4.5 Experimental results

4.5.1 Compare with baselines

To evaluate the effectiveness of the proposed approach, we compare MABash with 19 baseline methods spanning four categories: information retrieval, deep learning, pre-trained models, and Bash-specific generation models. The results are summarized in Table 2 and Fig. 8. For fair comparison, baseline results are obtained following the experimental settings reported in their original studies.

MABash consistently outperforms all baselines across the evaluation metrics. Compared with the strongest information-retrieval baseline CCGIR, MABash improves BLEU-1/2/3/4, BLEU, ROUGE-L, and METEOR by 7.65%, 7.20%, 6.12%, 4.89%, 5.52%, 8.62%, and 4.75%, respectively. These results highlight the limitations of retrieval-based methods in capturing implicit structural compositions and semantic dependencies in Bash commands. In contrast, MABash performs hierarchical lexical, syntactic, and semantic parsing, enabling the generation of structurally consistent and semantically complete comments.

Compared with traditional deep learning models such as Rencos, MABash further improves BLEU-1/2/3/4, ROUGE-L, and METEOR by 13.12%, 12.32%, 11.14%, 9.93%, 11.23%, and 7.09%, respectively. This improvement demonstrates the advantage of the multi-agent collaborative parsing mechanism in modeling complex command structures that are difficult to capture using purely sequence-based neural architectures.

Compared with pre-trained models such as CodeT5, MABash still achieves consistent improvements of 7.64%, 7.39%, 6.55%, 5.62%, 7.93%, and 3.42% on BLEU-1/2/3/4, ROUGE-L, and METEOR, respectively. This result suggests that explicitly modeling structured parsing states provides more effective semantic grounding than relying solely on implicit representations learned from large-scale pre-training.

Even against the current state-of-the-art Bash-specific model HBCom, MABash yields further gains of 2.94%, 3.13%, 2.28%, 1.56%, 2.06%, 2.85%, and 2.73% on

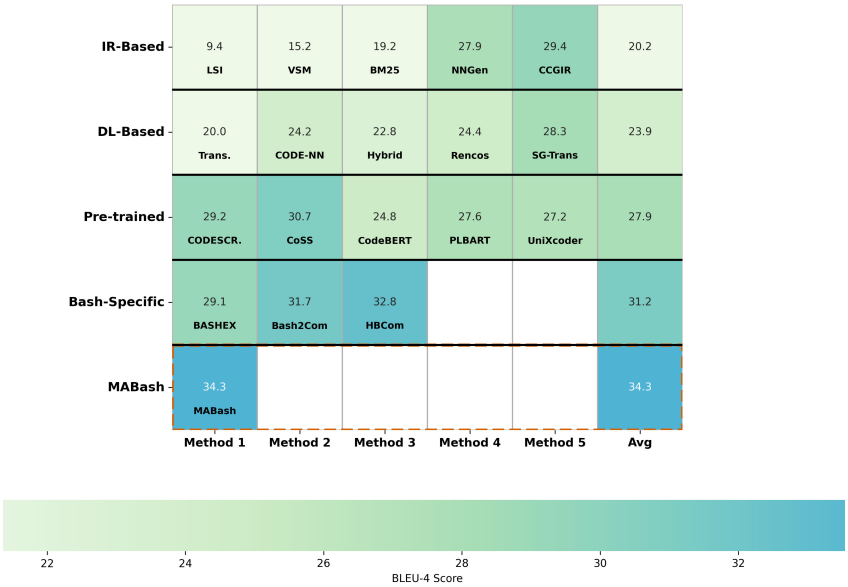


Fig. 8 Comparison of BLEU-4 Scores with Baseline Models

BLEU-1/2/3/4, BLEU, ROUGE-L, and METEOR, respectively. These improvements indicate that the multi-agent collaboration mechanism and the shared blackboard architecture enable more accurate modeling of conditional structures and execution semantics in Bash commands.

4.5.2 Fine-Tuned Generative LLM

To determine the most suitable generative backbone for Bash command annotation, we conduct unified LoRA fine-tuning and evaluation on three representative large language models: Qwen3-8B, Llama-3.1-8B, and DeepSeek-Coder-6.7B. These models are selected to cover both general-purpose and code-specialized architectures. All models are trained and evaluated under identical dataset splits and hyperparameter settings, taking raw Bash commands as input to generate natural-language explanations. The quantitative results are summarized in Table 3.

Fig. 9 presents the training dynamics over 10 epochs. Fig. 9(a) illustrates the training loss, while Fig. 9(b) shows the validation loss. All models exhibit stable convergence behavior, with both training and validation losses steadily decreasing during optimization. DeepSeek-Coder-6.7B achieves the fastest convergence and the lowest final loss, consistent with its code-oriented pretraining background. The small gap between training and validation losses indicates good generalization without evident overfitting.

Overall, DeepSeek-Coder-6.7B outperforms the other models across all evaluation metrics, including BLEU, ROUGE-L, and METEOR. Based on these results, the fine-tuned DeepSeek-Coder-6.7B is selected as the final generative model for explanation generation.

Table 2 Comparison between MABash and baseline models

Method name	BLEU-1(%)	BLEU-2(%)	BLEU-3(%)	BLEU-4(%)	BLEU(%)	ROUGE-L(%)	METEOR(%)
LSI[Haiduc et al.2010]	30.18	18.07	12.48	9.40	-	28.82	18.30
VSM[Haiduc et al.2010]	36.16	24.47	18.62	15.25	-	34.58	22.04
BM25[Zhang et al.2020]	42.08	30.41	19.24	19.24	-	38.49	26.35
NNGen[Liu et al.2018]	50.62	38.75	32.11	27.85	-	45.88	27.69
CCGIR[Yang et al.2022]	51.74	40.23	33.68	29.43	32.90	47.67	28.16
Transformer[Ahmad et al.2020]	46.39	33.37	25.42	19.97	-	44.01	25.22
CODE-NN[Iyer et al.2016]	49.60	37.18	29.53	24.17	-	47.21	26.85
Hybrid-DeepCom[Hu et al.2020]	47.78	35.45	27.91	22.75	-	45.36	26.27
Rencos[Zhang et al.2020]	46.27	35.11	28.66	24.39	-	45.06	25.82
SG-Trans[Gao et al.2023]	49.51	39.63	31.91	28.27	31.62	47.53	27.69
CODESCRIBE[Guo et al.2022]	51.82	40.79	33.37	29.19	33.27	48.39	28.94
CoSS[Shi et al.2023]	53.65	41.91	35.74	30.74	34.41	49.16	29.52
CodeBERT[Feng et al.2020]	48.65	37.02	29.84	24.83	-	47.36	27.16
PLBART[Ahmad et al.2021]	50.75	39.10	32.21	27.55	-	47.91	28.82
UniXcoder[Guo et al.2022]	49.99	38.52	31.80	27.25	-	48.24	29.03
CodeT5[Wang et al.2021]	51.75	40.04	33.25	28.70	-	48.36	29.49
BASHEXPLAINER[Yu et al.2022]	51.74	40.41	33.73	29.13	33.76	48.81	28.78
Bash2Com[Shen et al.2024]	54.21	42.96	36.28	31.68	34.87	51.35	30.12
HBCom[Zhang et al.2025]	56.45	44.30	37.52	32.76	36.36	53.44	30.18
MABash	**59.39**	**47.43**	**39.80**	**34.32**	**38.42**	**56.29**	**32.91**

The best performance metrics for each method are in bold.

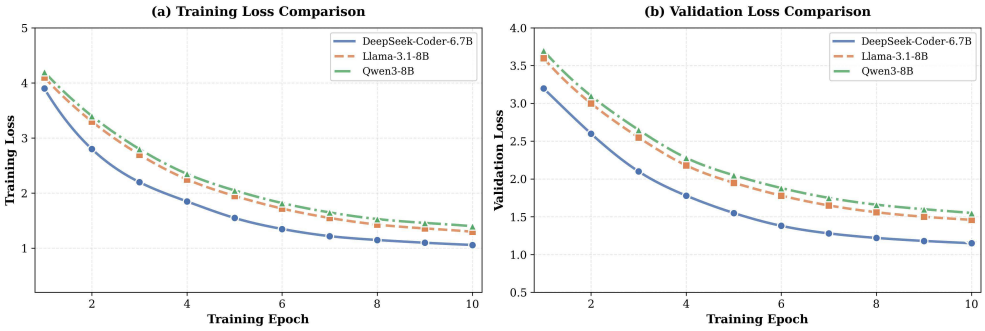


Fig. 9 Comparison of Training and Validation Loss

Table 3 Performance of the Fine-Tuned LLM on Bash Command Annotation

Model	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	BLEU (%)	ROUGE-L (%)	METEOR (%)
Qwen3-8B	51.94	38.26	29.98	24.32	30.30	48.01	28.11
Llama-3.1-8B	52.19	39.06	31.37	26.14	31.95	49.25	28.46
Deepseek-coder-6.7b	**54.85**	**41.81**	**33.68**	**27.89**	**33.01**	**51.43**	**29.75**

The best performance metrics for each model are in bold.

4.5.3 Ablation Study

To evaluate the contribution of individual components in the proposed multi-agent collaborative framework, we conduct a series of ablation experiments by progressively removing specific analysis modules and measuring the resulting performance changes in Bash command annotation. All configurations share identical dataset splits, training strategies, and inference settings. The explanation generator is consistently implemented using the fine-tuned DeepSeek-Coder-6.7B model to ensure fair comparison across variants.

We evaluate five configurations:

- Single-LLM: the fine-tuned LLM directly generates annotations without any analysis modules;
- w/o Lexical-Agent: removal of the lexical analysis agent;
- w/o Syntax-Agent: removal of the syntactic analysis agent;
- w/o Semantic-Agent: removal of the semantic analysis agent;
- MABash (Full): the complete multi-agent collaborative framework.

A visual comparison across four key metrics is presented in Fig. 10, while the complete evaluation results are reported in Table 4.

As shown in Table 4, removing the lexical analysis agent mainly reduces lower-order BLEU scores, suggesting that lexical parsing helps accurately ground command

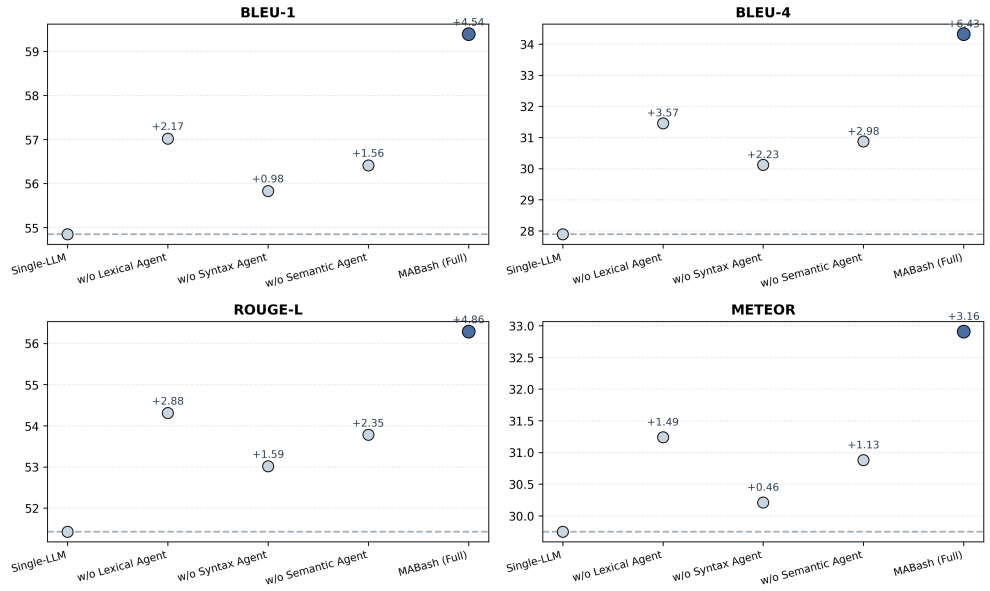


Fig. 10 Ablation Performance Comparison

Table 4 Ablation study results of MABash components

Method namel	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	BLEU (%)	ROUGE-L (%)	METEOR (%)
Single-LLM	54.85	41.81	33.68	27.89	33.01	51.43	29.75
w/o Lexical Agent	57.02	44.95	32.71	31.46	36.15	54.31	31.24
w/o Syntax Agent	55.83	43.27	35.94	30.12	34.98	53.02	30.21
w/o Semantic Agent	56.41	44.12	36.58	30.87	35.62	53.78	30.88
MABash (Full)	**59.39**	**47.43**	**39.80**	**34.32**	**38.42**	**56.29**	**32.91**

The best performance metrics for each method are in bold.

arguments and options at the token level. In contrast, removing syntactic or semantic agents leads to larger declines in higher-order BLEU metrics, underscoring their critical role in capturing structural dependencies and semantic relations in complex Bash commands. Overall, these results demonstrate that explicitly modeling intermediate parsing states provides effective structural guidance for comment generation.

Complexity Level	Segmentation Criteria	Statistics (Total N=1064)
Simple	• num_pipes == 0 • num_redirections == 0 • num_exec <= 1	Count:598(56.2%)
Medium	• num_pipes == 1 OR • num_redirections == 1	Count:242(22.7%)
Complex	• num_pipes >= 2 OR • (num_pipes >= 1 AND num_redirections >= 1) OR • num_exec >= 2 OR • contains subshell / logical operators	Count:224(21.1%)

Fig. 11 Dataset Stratification Based on Command Structural Complexity

5 Discussion

5.1 Command Structural Complexity

To evaluate the stability and scalability of the proposed method across Bash commands with varying structural complexity, we perform a stratified analysis of the test set based on parsable syntactic features. Specifically, commands are categorized into three levels, namely Simple, Medium, and Complex, according to the number and nesting of structural elements such as pipelines, redirections, subcommands, and logical operators. The classification is derived entirely from automated syntactic parsing without manual intervention, ensuring objectivity and reproducibility.

Among the 1,064 Bash commands in the test set, 598, 242, and 224 samples belong to the Simple, Medium, and Complex categories, respectively. The distribution of these categories is illustrated in Fig. 11.

Table 5 Performance Comparison of Different Models Across Command Complexity Levels

Model	Complexity	BLEU(%)	BLEU-4(%)	ROUGE-L(%)	METEOR(%)
Bash2Com	Simple	27.20	21.34	29.79	44.15
HBCom	Simple	34.80	27.32	29.45	53.92
MABash	**Simple**	**37.51**	**32.50**	**32.77**	**56.40**
Bash2Com	Medium	28.41	25.07	30.59	44.40
HBCom	Medium	37.29	36.70	30.64	56.44
MABash	**Medium**	**38.12**	**34.98**	**32.52**	**54.86**
Bash2Com	Complex	27.69	22.87	30.44	44.55
HBCom	Complex	38.19	39.66	31.68	58.51
MABash	**Complex**	**41.59**	**37.06**	**33.72**	**57.54**

The best performance metrics for each method are in bold.

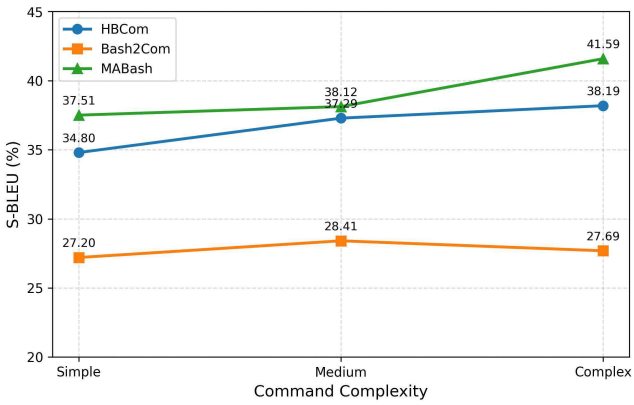


Fig. 12 BLEU Trend Across Command Complexity Levels for Different Models

According to the results in Table 5 and Fig. 12, command structural complexity significantly affects Bash comment generation performance. As complexity increases, Bash2Com shows only minor variation across metrics and limited improvement on the Complex subset, indicating insufficient capability to model semantic relationships in commands with multi-level pipelines, redirections, and nested structures. HBCom performs better on complex commands; however, its improvements are not consistent across subsets. For instance, it achieves a BLEU score of 37.29% on the Medium subset. In contrast, the gain on the Complex subset remains moderate, suggesting continued reliance on implicit representations within a single model.

In contrast, MABash maintains strong performance across all complexity levels and shows a clear advantage on the Complex subset, achieving a BLEU score of 41.59%, significantly outperforming Bash2Com (27.69%) and HBCom (38.19%). These results indicate that the proposed multi-agent collaborative parsing mechanism more effectively captures compositional semantics across structural units in complex commands, thereby improving the semantic completeness and structural consistency of generated annotations.

5.2 Limitations

In the current framework, the lexical, syntactic, and semantic agents primarily rely on static grammar rules and structural patterns. Consequently, parsing accuracy may decline for Bash commands involving dynamic command generation, interactive scripts, or environment variable expansion. For example, when processing commands such as `for file in $(ls *.txt); do mv "$file" "${file%.txt}.bak"; done`, which contain loops and string substitution, the semantic agent may fail to fully capture the dynamic execution logic, resulting in annotations with incomplete semantics.

To standardize input formats and improve generalization, specific file paths and variable names are normalized during preprocessing (e.g., replacing `/home/user/file.txt` with `<PATH>`). Although this normalization helps the model

focus on structural and semantic patterns, it may also introduce generic placeholders in the generated annotations, such as `"find files in <PATH>"` instead of `"find files in /home/user/docs"`, which reduces readability and practical usability.

Finally, the experiments are conducted on a dataset containing 10,592 Bash command–annotation pairs. Although relatively large for this task, it remains substantially smaller than code summarization datasets for mainstream programming languages such as Java or Python. This limitation may restrict the model's ability to generalize to rare commands and highly complex structures.

5.3 Validity threats

Internal threats mainly stem from implementation details and randomness. To ensure a fair comparison, we strictly followed the publicly available code and parameter settings of the baseline models and conducted all experiments in a unified hardware and software environment. We adopted the official dataset split provided by [Yu et al.2022] (8,474/1,059/1,059) without re-partitioning to maintain comparability. To reduce stochastic effects, we fixed random seeds and reported the average results over three independent runs.

External threats concern generalizability. Although the Bash annotation dataset used in this study is relatively large for this domain (10,592 pairs), it remains smaller than code summarization datasets for mainstream languages such as Java and Python. Moreover, the data are primarily collected from public communities like Stack Overflow, which may not fully represent enterprise-level or domain-specific Bash usage.

Construct threats concern whether the evaluation metrics accurately reflect the model's true performance. We employed BLEU, ROUGE-L, and METEOR, which measure textual similarity but are limited in their ability to capture semantic completeness and structural correctness. Therefore, the automatic evaluation results may not fully represent the practical quality of the generated annotations.

6 Conclusion

This paper presents MABash, a multi-agent collaborative framework for generating Bash command comments. The framework decomposes command understanding into lexical, syntactic, and semantic parsing stages, and coordinates them through a shared blackboard mechanism. By explicitly constructing intermediate representations, MABash enables the modeling of structural dependencies and condition scopes in Bash commands. Experimental results show that MABash consistently outperforms existing approaches across multiple evaluation metrics, particularly for commands with more complex structures. These results indicate that incorporating structured intermediate states can help improve the quality of generated comments.

Although the current framework focuses on static command structures, it can be extended to handle more complex Bash scenarios, including dynamic command generation and variable expansion. In addition, the idea of combining structured parsing with neural generation may apply to other code understanding tasks.

Funding. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant U24A201643; in part by the Science and Technology

Program of Guizhou Province under Grant Qian Ke He Jichu-ZK [2023] Zhongdian 011; and in part by the Big Data Security and Network Security Innovation Team of Guizhou Provincial High Education Institution under Grant [2023] 052.

Data availability. We have made the experimental materials and data publicly available at https://github.com/chenyang-co/MABash

Declarations

Conflict of interest. The authors declare no conflict of interest. We ensure that accepted principles of ethical and professional conduct have been followed.

Ethical approval. Not applicable.

References

[Ahmad et al.2020] Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W.: A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 4998–5007 (2020). https://doi.org/10.18653/v1/2020.acl-main.449

[Ahmad et al.2021] Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W.: Unified pre-training for program understanding and generation. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 2655–2668 (2021). https://doi.org/10.18653/v1/2021.naacl-main.211

[Bahdanau et al.2015] Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: International Conference on Learning Representations (ICLR) (2015). https://doi.org/10.48550/arXiv.1409.0473

[Brown et al.2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., *et al.*: Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing Systems, pp. 1877–1901 (2020). https://doi.org/10.48550/arXiv.2005.14165

[Choi et al.2021] Choi, Y., Bak, J., Na, C., Lee, J.-H.: Learning sequential and structural information for source code summarization. In: Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pp. 2842–2851 (2021). https://doi.org/10.18653/v1/2021.findings-acl.251

[Fernandes et al.2018] Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: International Conference on Learning Representations (2018). https://doi.org/10.48550/arXiv.1811.01824

[Feng et al.2020] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D.: Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP, pp. 1536–1547 (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139

[Ghaemi et al.2024] Ghaemi, H., Alizadehsani, Z., Shahraki, A., Corchado, J.M.: Transformers in source code generation: A comprehensive survey. Journal of

Systems Architecture **153**, 103193 (2024) https://doi.org/10.1016/j.sysarc.2024.103193

[Gao et al.2023] Gao, S., Gao, C., He, Y., Zeng, J., Nie, L., Xia, X., Lyu, M.R.: Code structure-guided transformer for source code summarization. ACM Transactions on Software Engineering and Methodology **32**(1), 1–32 (2023) https://doi.org/10.1145/3522674

[Gong et al.2022] Gong, Z., Gao, C., Wang, Y., Gu, W., Peng, Y., Xu, Z.: Source code summarization with structural relative position guided transformer. In: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 13–24 (2022). https://doi.org/10.1109/SANER53432.2022.00013

[Guo et al.2022] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J.: Unixcoder: Unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, pp. 7212–7225 (2022). https://doi.org/10.18653/v1/2022.acl-long.499

[Guo et al.2022] Guo, J., Liu, J., Wan, Y., Li, L., Zhou, P.: Modeling hierarchical syntax structure with triplet position for source code summarization. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, pp. 486–500 (2022). https://doi.org/10.18653/v1/2022.acl-long.37

[Geng et al.2024] Geng, M., Wang, S., Dong, D., Wang, H., Li, G., Jin, Z., Mao, X., Liao, X.: Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In: Proceedings of the IEEE/ACM International Conference on Software Engineering, pp. 1–13 (2024). https://doi.org/10.1145/3597503.3608134

[Haiduc et al.2010] Haiduc, S., Aponte, J., Marcus, A.: Supporting program comprehension with source code summarization. In: International Conference on Software Engineering (ICSE), pp. 223–226 (2010). https://doi.org/10.1145/1810295.1810335

[Haiduc et al.2010] Haiduc, S., Aponte, J., Moreno, L., Marcus, A.: On the use of automated text summarization techniques for summarizing source code. In: Working Conference on Reverse Engineering, pp. 35–44 (2010). https://doi.org/10.1109/WCRE.2010.13

[Hu et al.2018] Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210 (2018). https://doi.org/10.1145/3196321.3196334

[Hu et al.2020] Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering **25**, 2179–2217 (2020) https://doi.org/10.1007/s10664-019-09730-9

[Hellendoorn et al.2019] Hellendoorn, V.J., Sutton, C., Singh, R., Maniatis, P., Bieber, D.: Global relational models of source code. In: International Conference on Learning Representations (2019). https://api.semanticscholar.org/CorpusID:213352113

[Hu et al.2022] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. In: International Conference on Learning Representations (ICLR) (2022). https://doi.org/10.48550/arXiv.2106.09685

[Hong et al.2023] Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., *et al.*: Metagpt: Meta programming for multi-agent collaborative framework. arXiv preprint arXiv:2308.00352 (2023) https://doi.org/10.48550/arXiv.2308.00352

[Iyer et al.2016] Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pp. 2073–2083 (2016). https://doi.org/10.18653/v1/P16-1195

[Kidwai et al.2021] Kidwai, A., Arya, C., Singh, P., Diwakar, M., Singh, S., Sharma, K., Kumar, N.: A comparative study on shells in linux: A review. Materials Today: Proceedings **37**, 2612–2616 (2021) https://doi.org/10.1016/j.matpr.2020.08.508

[Kuhn et al.2007] Kuhn, A., Ducasse, S., Girba, T.: Semantic clustering: Identifying topics in source code. Information and Software Technology **49**(3), 230–243 (2007) https://doi.org/10.1016/j.infsof.2006.10.017

[LeClair et al.2020] LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020). https://doi.org/10.1145/3387904.3389268

[Liu et al.2018] Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z., Wang, X.: Neural-machine-translation-based commit message generation: How far are we? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 373–384 (2018). https://doi.org/10.1145/3238147.3238190

[Milligan and Baker2014] Milligan, I., Baker, J.: Introduction to the bash command line. Technical report, The Editorial Board of the Programming Historian (2014). https://doi.org/10.46430/phen0037

[Majumdar et al.2026] Majumdar, S., Deshpande, A., Das, P.P., Chakrabarti, P.P.: Comprehending c codes with llms: Effective comment generation through retrieval and reasoning. Pattern Recognition Letters **199**, 295–302 (2026) https://doi.org/10.1016/j.patrec.2025.10.007

[Mazurak and Zdancewic2007] Mazurak, K., Zdancewic, S.: Abash: Finding bugs in bash scripts. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, pp. 105–114 (2007). https://doi.org/10.1145/1255329.1255347

[Oli et al.2023] Oli, P., Banjade, R., Chapagain, J., Rus, V.: The behavior of large language models when prompted to generate code explanations. arXiv preprint arXiv:2311.01490 (2023) https://doi.org/10.48550/arXiv.2311.01490

[Park et al.2023] Park, J.S., O'Brien, J.C., Cai, C.J., Morris, M.R., Liang, P., Bernstein, M.S.: Generative agents: Interactive simulacra of human behavior. In: Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, pp. 1–22 (2023). https://doi.org/10.1145/3586183.3606763

[Su et al.2026] Su, C.-Y., Bansal, A., Huang, Y., Li, T.J.-J., McMillan, C.: Context-aware code summary generation. Journal of Systems and Software **231**, 112580 (2026) https://doi.org/10.1016/j.jss.2025.112580

[Shi et al.2023] Shi, C., Cai, B., Zhao, Y., Gao, L., Sood, K., Xiang, Y.: Coss: Leveraging statement semantics for code summarization. IEEE Transactions on Software

Engineering (2023) https://doi.org/10.1109/TSE.2023.3256362

[Shen et al.2024] Shen, Y., Ju, X., Chen, X., Yang, G.: Bash comment generation via data augmentation and semantic-aware codebert. Automated Software Engineering **31**(1), 30 (2024) https://doi.org/10.1007/s10515-024-00431-2

[Shido et al.2019] Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., Matsumura, T.: Automatic source code summarization with extended tree-lstm. In: International Joint Conference on Neural Networks, pp. 1–8 (2019). https://doi.org/10.1109/IJCNN.2019.8851751

[Shoham and Leyton-Brown2008] Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations, (2008)

[Su and McMillan2024] Su, C.-Y., McMillan, C.: Distilled gpt for source code summarization. Automated Software Engineering **31**(1), 22 (2024) https://doi.org/10.1007/s10515-024-00421-4

[Shen et al.2025] Shen, Y., Qiu, J., Wang, J., Zhou, Y.: Bashagent: Automatic bash comment generation via large language models and chain-of-thought. In: Proceedings of the 3rd International Conference on Educational Knowledge and Informatization, pp. 74–78 (2025). https://doi.org/10.1145/3765325.3765339

[Sutskever et al.2014] Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014). https://doi.org/10.48550/arXiv.1409.3215

[Vaswani et al.2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017). https://doi.org/10.48550/arXiv.1706.03762

[Wei et al.2020] Wei, B., Li, Y., Li, G., Xia, X., Jin, Z.: Retrieve and refine: Exemplar-based neural comment generation. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 349–360 (2020). https://doi.org/10.1145/3324884.3416578

[Wooldridge2009] Wooldridge, M.: An Introduction to Multiagent Systems, 2nd edn. (2009)

[Wang et al.2021] Wang, Y., Wang, W., Joty, S., Hoi, S.C.H.: Codet5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 8696–8708 (2021). https://doi.org/10.18653/v1/2021.emnlp-main.685

[Wu et al.2021] Wu, H., Zhao, H., Zhang, M.: Code summarization with structure-induced transformer. In: Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pp. 1078–1090 (2021). https://doi.org/10.18653/v1/2021.findings-acl.93

[Xi et al.2023] Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., *et al.*: The rise and potential of large language model-based agents: A survey. arXiv preprint arXiv:2309.07864 (2023) https://doi.org/10.48550/arXiv.2309.07864

[Yang et al.2021] Yang, G., Chen, X., Cao, J., Xu, S., Cui, Z., Yu, C., Liu, K.: Comformer: Code comment generation via transformer and fusion method-based

hybrid code representation. In: Proceedings of the 8th International Conference on Dependable Systems and Their Applications (DSA), pp. 30–41 (2021). https://doi.org/10.1109/DSA52907.2021.00013

[Yang et al.2022] Yang, G., Liu, K., Chen, X., Zhou, Y., Yu, C., Lin, H.: Ccgir: Information retrieval-based code comment generation method for smart contracts. Knowledge-Based Systems **237**, 107858 (2022) https://doi.org/10.1016/j.knosys.2021.107858

[Yu et al.2022] Yu, C., Yang, G., Chen, X., Liu, K., Zhou, Y.: Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 82–93 (2022). https://doi.org/10.1109/ICSME55016.2022.00016

[Yao et al.2023] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., *et al.*: React: Synergizing reasoning and acting in language models. In: International Conference on Learning Representations (2023). https://doi.org/10.48550/arXiv.2210.03629

[Zhang et al.2020] Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X.: Retrieval-based neural source code summarization. In: Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE), pp. 1385–1397 (2020). https://doi.org/10.1145/3377811.3380383

[Zhang et al.2025] Zhang, J., Zhu, Y., Lu, A., Yan, Y., Wan, Y.: Bash command comment generation via multi-scale heterogeneous feature fusion. Automated Software Engineering **32**(1), 28 (2025) https://doi.org/10.1007/s10515-025-00494-9