PaveDSL: A Neuro-Symbolic Approach to Plan Specification and Verification for Multi-Agent Code Generation

Wutong Yu, Deyu Zhou*
School of Computer Science and Engineering,
Key Laboratory of Computer Network and Information Integration,
Ministry of Education, Southeast University, China
{230249001, d.zhou}@seu.edu.cn

Abstract

Recent multi-agent code generation frameworks often verify natural-language plans by simulating their behavior on test cases before code generation. However, such verification remains unreliable because model hallucination may cause flawed plans to be incorrectly accepted. Inspired by neuro-symbolic reasoning, we propose PaveDSL, which transforms natural-language plans into structured designs expressed in a domain-specific language. Based on this structured representation, PaveDSL combines symbolic checks with neural inference to examine plans before code generation. Specifically, PaveDSL applies two complementary verifiers in parallel. The deterministic symbolic verifier checks structural constraints without invoking an LLM, while the agent verifier performs graph-guided semantic reasoning over the structured design. If verification fails, a repair agent localizes errors and revises the design before code generation. Experiments on multiple benchmarks with different LLMs show that PaveDSL consistently improves the performance of baseline code generation methods.

1 Introduction

Large language models (LLMs) have significantly advanced code generation (Jiang et al., 2024a; Zhang et al., 2024b). To improve reliability on complex tasks, recent studies adopt multi-agent frameworks where specialized agents collaborate to analyze requirements, produce natural-language plans, and generate code (Dong et al., 2025). In these frameworks, natural-language plans serve as intermediate representations bridging problem understanding and implementation. A common strategy is to let an agent simulate a generated plan on test cases before code generation, aiming to identify flawed solutions early (Lei et al., 2025; Islam et al., 2025).

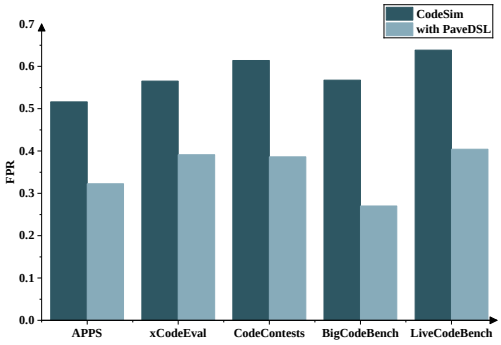


Figure 1: False positive rates of planning-stage verification for CodeSim and CodeSim with PaveDSL using GPT-5-mini across five benchmarks. For each benchmark, both methods are evaluated on the same fixed set of problems under the same protocol. Lower values indicate more reliable plan verification.

However, such verification remains unreliable when performed only on natural-language plans. Since the verifier must simulate plan behavior through language reasoning, an incomplete or flawed plan may still produce a plausible execution trace, causing the verifier to incorrectly accept it. To examine this problem, we use GPT-5-mini as the base model and evaluate CodeSim (Islam et al., 2025) on five benchmarks. We assess plan correctness by examining whether generated implementations follow the plan and pass the tests, with ambiguous cases manually inspected. We define the false positive rate (FPR) as the ratio of accepted plans subsequently determined to be incorrect to all plans determined to be incorrect. As shown in Figure 1, CodeSim produces a high FPR across all benchmarks, indicating that verification based solely on natural-language plans is insufficient as a standalone validation mechanism.

Neuro-symbolic reasoning offers a suitable solution by combining the flexible understanding capabilities of LLMs with the explicit structure of symbolic representations. LLMs can interpret plan semantics and simulate candidate executions, while

* Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 41566–41584
October 24-29, 2026 ©2026 Association for Computational Linguistics

structured checking provides a more systematic basis for verification (Huang et al., 2024b; Dolcetti and Iotti, 2025). Symbolic representations make key design elements explicit, such as functional dependencies, data flow, input constraints, and edge-case handling. By integrating both paradigms, plan verification can use neural models for semantic understanding while relying on explicit structure for deterministic checks and systematic inspection before code generation.

Motivated by this insight, we propose **PaveDSL (Plan as a Verifiable Expression with a Domain-Specific Language)**, a neuro-symbolic approach for pre-code plan verification. Specifically, given a plan produced by an existing code generation method, PaveDSL first uses a design agent to translate the natural-language plan into a DSL-based candidate design under a predefined schema. The candidate design represents the original solution with explicit functional units, input-output relations, dependencies, constraints, and edge cases. PaveDSL then verifies the candidate design in parallel through two complementary mechanisms. The symbolic verifier performs deterministic rule-based checks over the design structure without relying on an LLM, including schema validity, traceability between requirements and functions, and graph consistency. The agent verifier uses an LLM for graph-guided reasoning to examine function responsibilities, input-output consistency, and test-case behavior along the structured design graph. If either verifier reports an error, a repair agent uses the verification feedback to localize the error and revise the design before the entire design is verified again.

Through PaveDSL, plan verification becomes a structured process integrating deterministic symbolic checks with graph-guided agent reasoning. As shown in Figure 1, this combination consistently reduces the FPR of CodeSim across all benchmarks. We integrate PaveDSL with four representative plan-based code generation baselines and evaluate it using three LLMs. Experimental results show that PaveDSL consistently improves baseline performance. Additional evaluations with a broader set of code generation baselines and smaller open-source models further support its general applicability. Our contributions in this work are summarized as follows:

- We formulate pre-code plan verification as a neuro-symbolic design verification problem, transforming natural-language plans into structured and explicitly checkable designs.
- We propose PaveDSL, which verifies each structured design in parallel through deterministic symbolic checks and graph-guided LLM reasoning, and uses the resulting feedback for error-driven repair before code generation.
- We conduct extensive experiments on five benchmarks using three LLMs, showing that PaveDSL consistently improves baseline code generation performance.

2 Related Work

2.1 Neuro-Symbolic Reasoning and Verification

Neuro-symbolic approaches integrate neural networks with symbolic reasoning to enhance the reliability and interpretability of AI systems (Delvecchio et al., 2025). Several studies leverage symbolic solvers to strengthen LLM reasoning. Logic-LM (Pan et al., 2023) translates natural language into logic and invokes solvers with iterative repair, SatLM (Ye et al., 2023) prompts LLMs to produce declarative constraints for SAT solving, LINC (Olausson et al., 2023) offloads reasoning to first-order logic provers (Lyu et al., 2023; Yang et al., 2023), and FLARE (Arakelyan et al., 2025) generates formal reasoning steps to improve faithfulness without external solvers.

A complementary direction focuses on verifying LLM-generated outputs. PAL (Gao et al., 2023) produces executable programs evaluated by interpreters, while Proof of Thought (Ganguly et al., 2024) generates formal thought programs as verifiable representations. Explanation-Refiner (Quan et al., 2024) uses LLM-symbolic theorem proving to validate and refine explanations, and Veri-CoT (Feng et al., 2026) maps chain-of-thought steps into first-order logic for logical consistency checks. Symbolic validators have also been used to check task plans before execution (Sheth et al., 2024; Petruzzellis et al., 2025; Ahn et al., 2025). CEGIS (Jha et al., 2023) couples LLM generation with SMT solving and counterexample-guided repair over solver-compatible representations. Together, these studies show how explicit intermediate representations can connect neural generation with symbolic checking across executable programs, logical arguments, and task plans.

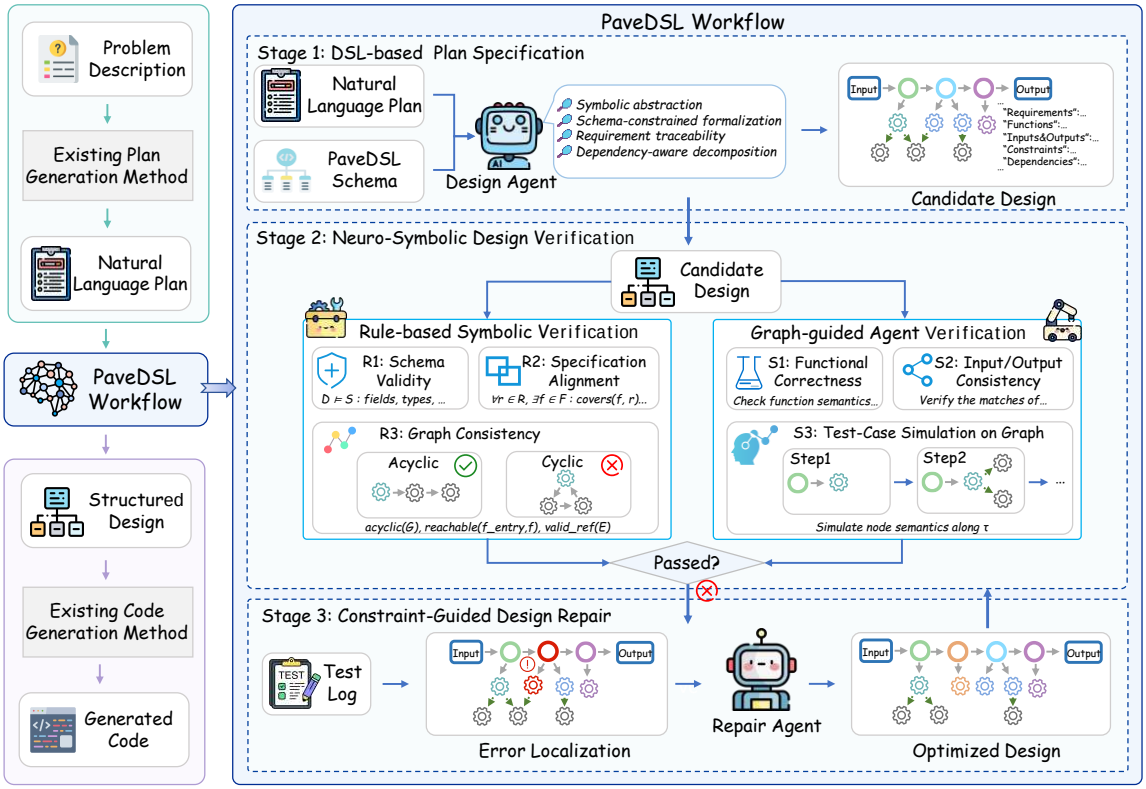


Figure 2: Overview of the PaveDSL workflow. PaveDSL first converts a natural-language plan into a candidate design, then verifies it through symbolic and graph-guided agent verification, and finally repairs failed designs before code generation.

2.2 Planning and Multi-Agent Code Generation

Multi-agent code generation coordinates specialized agents across planning, implementation, testing, and refinement (Dong et al., 2025). Pipeline-based systems emulate software development workflows (Dong et al., 2024; Qian et al., 2024; Hong et al., 2024; Lin et al., 2025), while planning-oriented and specification-aware methods produce or refine intermediate solution representations before implementation (Jiang et al., 2024b; Islam et al., 2024; Lei et al., 2025; Islam et al., 2025; Wang et al., 2025; Tian et al., 2025). Other methods improve solution search and repair through execution feedback, multi-plan exploration, or experience reuse. AgentCoder, CodeCoR, and LDB refine generated code using test results, reflective scoring, or runtime traces (Huang et al., 2024a; Pan et al., 2025; Zhong et al., 2024). PairCoder and CodeTree explore alternative plans or strategies and refine their implementations (Zhang et al., 2024a; Li et al., 2025), while Xolver equips multiple agents with persistent and evolving memory that integrates retrieved experience, tool use, collaboration, and iterative refinement (Hosain et al., 2025). Together, these methods improve planning, solution exploration, and implementation repair. PaveDSL focuses on the reliability of the intermediate plan through neuro-symbolic verification before code generation.

3 Approach

3.1 Overview

We formulate code generation as the task of generating a code snippet C from a problem description Q and a set of public test cases $\mathcal{T} = \{(I_i, O_i)\}_{i=1}^{n}$, where I_i is a test input and O_i is the expected output. Given Q, an existing plan generation method first produces a natural-language plan P. To further verify the plan before code generation, PaveDSL uses a Design Agent to produce a structured design D from Q and P under a predefined schema $\mathcal{S}$. The overall process is formulated as $Q \rightarrow P \rightarrow D \rightarrow C$, where D represents the plan as a graph-structured design with explicit task requirements, function nodes, data interfaces, and dependencies for subsequent verification.

As shown in Figure 2, PaveDSL consists of three stages, including DSL-based plan specifi-

Algorithm 1 PaveDSL

Input: Problem description Q, public test cases $\mathcal{T}$, PaveDSL schema $\mathcal{S}$, repair budget d
Output: Generated code snippet C

// DSL-based Plan Specification
$P \leftarrow \mathcal{A}_{\text{plan}}(Q)$
$D \leftarrow \mathcal{A}_{\text{design}}(Q, P, \mathcal{S})$

// Neuro-Symbolic Design Verification
for $k = 0$ to d **do**
  parallel do
    $(r_{\text{sym}}, F_{\text{sym}}) \leftarrow \mathcal{V}_{\text{sym}}(D, \mathcal{S})$
    $(r_{\text{agent}}, F_{\text{agent}}) \leftarrow \mathcal{V}_{\text{agent}}(Q, D, \mathcal{T})$
  end parallel
  $Pass \leftarrow r_{\text{sym}} \wedge r_{\text{agent}}$
  if $Pass$ **then**
    break
  end if

  // Constraint-Guided Design Repair
  if $k < d$ **then**
    $F \leftarrow F_{\text{sym}} \cup F_{\text{agent}}$
    $D \leftarrow \mathcal{A}_{\text{repair}}(Q, D, F, \mathcal{S})$
  end if
end for

$C \leftarrow \mathcal{A}_{\text{code}}(Q, D)$
return C

3.2 DSL-based Plan Specification

This stage transforms a natural-language plan into a DSL-based structured design for verification before code generation. As shown in Figure 3, PaveDSL defines a lightweight schema $\mathcal{S}$ with seven fields that represent the task goal, functional decomposition, data interfaces, semantic constraints, and inter-function dependencies. Each candidate design is represented as a JSON object conforming to $\mathcal{S}$, allowing the Symbolic Verifier to parse it deterministically and validate its structure.

```
{
  "Task": "Describes the overall problem goal and contextual information, providing the high-level objective of the structured design.",
  "Requirements": "Decomposes the task into functional requirements that specify what capabilities the solution should provide and what behaviors it should satisfy.",
  "Functions": "Defines function nodes that implement the decomposed requirements, with each node responsible for a specific part of the solution logic.",
  "Inputs": "Specifies the data consumed by each function node, including external inputs from the task and intermediate values from previous nodes.",
  "Outputs": "Specifies the data produced by each function node, including intermediate results passed to later nodes and final answers returned by the solution.",
  "Constraints": "Records lightweight semantic conditions, assumptions, boundary cases, and task-specific restrictions used during verification.",
  "Dependencies": "Defines directed data-flow relations among function nodes, showing how produced outputs are passed as required inputs to later functions."
}
```

Figure 3: PaveDSL schema for organizing natural-language plans into verifiable structured designs.

cation, neuro-symbolic design verification, and constraint-guided design repair. In the first stage, the Design Agent takes Q, P, and the PaveDSL schema $\mathcal{S}$ as input and generates a structured design $D = \mathcal{A}_{\text{design}}(Q, P, \mathcal{S})$. The design D captures the functional structure of the original plan, including task requirements, function responsibilities, data interfaces, constraints, and dependencies. In the second stage, the symbolic verifier and the agent verifier examine D in parallel within each iteration. The symbolic verifier performs deterministic rule-based checking without calling an LLM, while the agent verifier uses an LLM for graph-guided semantic checking and test-case simulation. The design passes verification only if both r_{sym} and r_{agent} hold. Otherwise, the combined feedback $F = F_{\text{sym}} \cup F_{\text{agent}}$ guides the Repair Agent to revise D, after which the complete design is examined again by both verifiers. This verification-repair process continues until D passes verification or the repair budget d is reached. If the budget is exhausted before D passes, PaveDSL retains the most recent repaired design to guide code generation without treating it as verified. In either case, PaveDSL uses the resulting structured design to guide code generation as $C = \mathcal{A}_{\text{code}}(Q, D)$. Algorithm 1 summarizes this workflow.

The *Task* field records the overall problem goal. The *Requirements* field decomposes the task into functional behaviors, with `covered_by` referencing the function nodes assigned to each requirement. The *Functions* field defines the function nodes and uses `implements` to associate each node with its declared requirements. The *Inputs* and *Outputs* fields specify the data consumed and produced by each function. The *Constraints* field records semantic conditions and boundary-case restrictions. The *Dependencies* field defines directed data-flow relations among functions. Together, these fields form a graph-structured design with explicit requirement-function traceability, data interfaces, semantic constraints, and execution order.

The Design Agent instantiates these fields to produce a candidate design D. Each function in the *Functions* field forms a node, while each relation in the *Dependencies* field forms a directed edge. The *Inputs* and *Outputs* fields define the data interface of each node. An edge from a source node to a target node indicates that data produced by the source is required by the target. In this form, D explicitly represents how data moves from the task inputs to the final output and provides the structured input for symbolic checking and graph-guided agent verification.

3.3 Neuro-Symbolic Design Verification

After obtaining the structured design, PaveDSL verifies whether it is suitable for guiding code generation. This stage follows a neuro-symbolic verification strategy. The symbolic verifier performs deterministic rule-based checks without calling an LLM, while the agent verifier uses an LLM to reason over the graph structure and simulate its behavior on public test cases.

The rule-based symbolic verifier focuses on the structural validity of the candidate design. It checks three aspects. First, schema validity checks whether the design can be parsed and contains the seven required fields in the predefined format. Second, specification alignment checks the structural traceability between requirements and functions. Each requirement should reference at least one existing function through `covered_by`, and each function should reference at least one existing requirement through `implements`. Third, graph consistency verifies that every dependency connects valid function nodes and applies depth-first search to detect cycles.

The graph-guided agent verifier further checks the semantic consistency of the structured design. Given Q, $\mathcal{T}$, and D, it follows the function nodes and dependency relations represented in D. It examines whether each function is suitable for its assigned requirements and constraints, whether the inputs and outputs of connected nodes are consistent, and whether the design produces the expected outputs when simulated on public test cases. During simulation, the agent follows the dependency order and traces how intermediate results are passed between function nodes.

The two verifiers are complementary. The symbolic verifier detects format errors and structural inconsistencies through deterministic rules, while the agent verifier captures logic errors that are difficult to express with fixed rules. A design passes verification only when both verifiers return a positive result. Otherwise, their feedback is provided to the subsequent repair stage.

3.4 Constraint-Guided Design Repair

When the candidate design fails verification, PaveDSL uses an LLM-based Repair Agent to revise it before code generation. The Repair Agent receives Q, D, $\mathcal{S}$, and the combined verifier feedback F. Based on F, it revises D to address the structural and semantic issues identified during verification.

The two sources of feedback guide different aspects of repair. The symbolic feedback reports violations of deterministic structural rules, while the agent feedback identifies semantic inconsistencies and associates them with the relevant requirements, function nodes, data interfaces, constraints, or dependencies. The Repair Agent limits each revision to the identified parts of the design. When a modification changes a function interface or dependency, the related elements are updated to maintain the consistency of D.

After each repair, the entire updated design is returned to the verification stage and examined again by both verifiers. If it still fails and another repair iteration is available, the combined feedback guides the next revision. This process terminates when D passes verification or the repair budget is reached. If the budget is exhausted before the design passes, the most recent repaired design is retained to guide code generation without being treated as verified.

4 Experimental Settings

4.1 Benchmarks

We evaluate PaveDSL on five code generation benchmarks that cover complementary task settings. APPS (Hendrycks et al., 2021) with 150 problems, xCodeEval (Khan et al., 2024) with 106 problems, and CodeContests (Li et al., 2022) with 165 problems comprise competition-level programming challenges that require complex algorithmic reasoning. BigCodeBench-Instruct (Zhuo et al., 2025) includes 125 problems and evaluates the composition of diverse library API calls under complex instructions. LiveCodeBench (Jain et al., 2025) continuously collects new problems from online platforms to mitigate data contamination, and we use 175 problems from version 6. For each paired comparison, the original method and its PaveDSL-enhanced version receive the same problem descriptions and public test cases and are evaluated using the same hidden test cases.

4.2 Metrics

Following prior work (Chen et al., 2021; Austin et al., 2021; Yu et al., 2024), we adopt Pass@k (Chen et al., 2021) as our primary metric. Given a problem, the model produces k candidate solutions, and the problem is considered solved if any candidate passes all hidden test cases. Pass@k measures the fraction of solved problems across

Table 1: Performance comparison of different code generation approaches with and without PaveDSL across five benchmarks.

LLM	Approach	APPS	xCodeEval	CodeContests	BigCodeBench	LiveCodeBench
GPT-5-mini	CodeSim	79.3%	78.3%	73.3%	71.2%	76.6%
	with PaveDSL	**83.3%**	**83.0%**	**78.8%**	**81.6%**	**85.7%**
	LPW	70.0%	70.8%	69.1%	68.0%	73.7%
	with PaveDSL	**78.7%**	**80.2%**	**78.8%**	**72.0%**	**78.3%**
	MaintainCoder	70.0%	75.5%	72.7%	72.8%	71.4%
	with PaveDSL	**74.7%**	**79.2%**	**78.2%**	**85.6%**	**77.7%**
	μFiX	75.3%	76.4%	74.5%	70.4%	73.1%
	with PaveDSL	**85.3%**	**83.0%**	**80.0%**	**84.0%**	**84.6%**
Claude-4.5-Haiku	CodeSim	76.0%	75.5%	71.5%	69.6%	74.3%
	with PaveDSL	**80.0%**	**82.1%**	**77.6%**	**79.2%**	**85.1%**
	LPW	68.0%	69.8%	68.5%	64.0%	72.6%
	with PaveDSL	**74.7%**	**76.4%**	**75.2%**	**70.4%**	**76.0%**
	MaintainCoder	69.3%	75.5%	70.9%	68.8%	71.4%
	with PaveDSL	**72.0%**	**81.1%**	**76.4%**	**71.2%**	**76.6%**
	μFiX	75.3%	74.5%	68.5%	68.0%	71.4%
	with PaveDSL	**82.0%**	**83.0%**	**78.8%**	**80.8%**	**80.6%**
DeepSeek-V4-Flash	CodeSim	81.3%	83.0%	74.5%	79.2%	74.9%
	with PaveDSL	**84.0%**	**84.9%**	**77.6%**	**84.0%**	**81.1%**
	LPW	73.3%	74.5%	72.7%	72.8%	73.7%
	with PaveDSL	**80.7%**	**81.1%**	**78.8%**	**82.4%**	**77.1%**
	MaintainCoder	68.7%	77.4%	73.3%	72.0%	69.7%
	with PaveDSL	**81.3%**	**80.2%**	**80.0%**	**84.0%**	**73.1%**
	μFiX	78.0%	81.1%	75.8%	76.0%	72.0%
	with PaveDSL	**86.7%**	**85.8%**	**80.6%**	**86.4%**	**82.3%**

the benchmark. Following recent studies (Jiang et al., 2024a; Wang and Chen, 2023; Zhang et al., 2024b), we set $k = 1$ throughout our main experiments to evaluate the first generated solution. Therefore, Pass@1 measures the end-to-end code generation performance of each method under a single generation attempt.

4.3 Baselines

We select four representative code generation methods that incorporate natural-language planning before code generation. Since PaveDSL is designed to improve the intermediate plan, we apply it after each baseline generates its initial natural-language plan. PaveDSL transforms the plan into a structured design, verifies and repairs the design, and passes the resulting design to the code generation stage. This setting isolates the effect of structured plan verification while keeping the remaining pipeline and backbone model unchanged.

CodeSim (Islam et al., 2025) is a multi-agent framework that verifies natural-language plans through test-case simulation and employs a debugging agent to refine generated code.

LPW (Lei et al., 2025) is a two-phase workflow that generates and verifies a solution plan through test-case simulation, then refines generated code through iterative debugging.

MaintainCoder (Wang et al., 2025) integrates the Waterfall model, design patterns, and multi-agent collaboration to decompose requirements into structured designs before code generation.

μFiX (Tian et al., 2025) uses test-case analysis to improve specification understanding and employs feedback-based self-improvement to correct misunderstandings before code generation.

5 Experimental Analysis

To comprehensively evaluate the effectiveness of PaveDSL, we organize our experiments around the following research questions, denoted as **RQs**:

RQ1: Can PaveDSL improve the performance of existing plan-based code generation methods across different benchmarks and LLM backbones?

RQ2: How do the structured representation, symbolic verification, graph-guided agent verification, and design repair contribute to the performance of PaveDSL?

RQ3: How does the verification-repair iteration budget affect the performance and computational cost of PaveDSL?

RQ4: How does PaveDSL identify and repair er-

Table 2: Ablation study of structured representation, verification, and repair on CodeSim and μFiX using GPT-5-mini across five benchmarks.

Baseline	Variant	APPS	xCodeEval	CodeContests	BigCodeBench	LiveCodeBench
CodeSim	w/o Verification	66.0%	68.9%	67.9%	51.2%	68.6%
	w/o Graph-Guided Reasoning	80.7%	78.3%	75.8%	78.4%	81.7%
	w/o Symbolic Verifier	78.7%	78.3%	74.5%	79.2%	80.6%
	w/o Agent Verifier	70.7%	76.4%	71.5%	72.0%	76.0%
	w/o Repair Agent	78.0%	80.2%	72.7%	76.8%	72.0%
	Full PaveDSL	**83.3%**	**83.0%**	**78.8%**	**81.6%**	**85.7%**
μFiX	w/o Verification	73.3%	69.8%	69.7%	64.8%	70.9%
	w/o Graph-Guided Reasoning	82.0%	78.3%	78.2%	83.2%	79.4%
	w/o Symbolic Verifier	80.0%	77.4%	75.2%	81.6%	73.7%
	w/o Agent Verifier	76.7%	74.5%	72.1%	72.8%	68.6%
	w/o Repair Agent	82.0%	75.5%	77.6%	78.4%	76.0%
	Full PaveDSL	**85.3%**	**83.0%**	**80.0%**	**84.0%**	**84.6%**

rors in a natural-language plan through a structured design?

Implementation and evaluation details are provided in Appendix A, and additional experimental results are reported in Appendix B.

5.1 Main Results

To answer **RQ1**, we integrate PaveDSL into four representative plan-based code generation methods and evaluate them across five benchmarks using three LLM backbones. Table 1 reports the Pass@1 results.

Overall Performance Improvement. PaveDSL improves all four baselines across every benchmark and LLM backbone. The largest gain occurs for μFiX on BigCodeBench with GPT-5-mini, where PaveDSL increases Pass@1 by 13.6 percentage points. These results show that verifying and repairing the structured design before code generation consistently improves end-to-end code generation performance.

Benefits Across Planning Methods. The improvements extend to baselines with different planning mechanisms. PaveDSL benefits plan-simulation frameworks such as CodeSim and LPW, the design-oriented MaintainCoder, and the specification-refinement method μFiX. This broad improvement indicates that PaveDSL functions as a complementary verification layer rather than depending on a particular planning workflow.

Consistency Across Settings. All three LLM backbones benefit from PaveDSL across the five benchmarks. The strongest overall gains appear on BigCodeBench, which focuses on composing diverse library APIs under complex instructions. This pattern indicates that the benefits of PaveDSL are not tied to a specific model or benchmark and supports its applicability across the evaluated settings.

5.2 Ablation Study

To answer **RQ2**, we conduct an ablation study on CodeSim and μFiX using GPT-5-mini across all five benchmarks. These two baselines employ different forms of test-case-based reasoning before code generation. CodeSim verifies generated plans through test-case simulation, whereas μFiX improves specification understanding through test-case analysis. We evaluate five ablation variants. The w/o Verification variant directly uses the structured design for code generation without symbolic verification, agent verification, or repair. The w/o Graph-Guided Reasoning variant retains the structured design, Symbolic Verifier, and Repair Agent, but the Agent Verifier reasons over the original natural-language plan rather than the function nodes and dependencies in the design. The remaining three variants remove the Symbolic Verifier, Agent Verifier, or Repair Agent individually. Table 2 reports the results.

Role of Verification and Repair. For both CodeSim and μFiX, Full PaveDSL outperforms w/o Verification by more than 10 percentage points on every benchmark. This result indicates that the structured representation alone does not prevent errors in the original plan or during plan transformation from propagating to code generation. Verification and repair provide substantial benefits beyond the structured representation.

Value of Graph-Guided Agent Verification. Full PaveDSL consistently outperforms w/o Graph-Guided Reasoning across both baselines and all five benchmarks. Because both variants use the same structured design, Symbolic Verifier, and repair process, this comparison isolates the contribution of

graph-guided agent verification. The consistent advantage indicates that graph-guided reasoning provides a more reliable basis for semantic verification than simulation over the original natural-language plan.

Complementary Components. Across both baselines and all five benchmarks, removing any of the three components reduces Pass@1. Removing the Agent Verifier generally causes the largest decline, while removing the Symbolic Verifier or Repair Agent also consistently harms performance. These results show that deterministic structural checking, graph-guided semantic verification, and design repair play different but complementary roles.

5.3 Efficiency and Iteration Trade-off

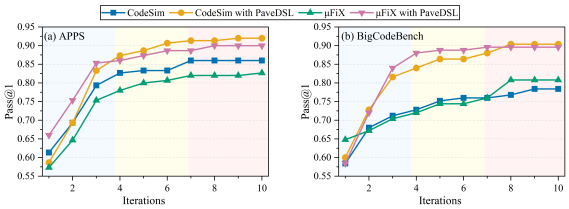


Figure 4: Effect of verification-repair iterations on Pass@1 for CodeSim and μFiX with and without PaveDSL on APPS and BigCodeBench. Performance improves most during the early iterations and gradually stabilizes as the budget increases.

To answer **RQ3**, we vary the maximum number of verification-repair iterations from 1 to 10 on APPS and BigCodeBench. We use GPT-5-mini as the backbone model and apply PaveDSL to CodeSim and μFiX. Figure 4 reports the corresponding Pass@1 results. We further measure the cost of each complete workflow through the average number of API calls and token usage per problem. Detailed results are reported in Table 3 in Appendix B.

Figure 4 shows that the largest improvements occur during the first few iterations. The PaveDSL variants then continue to improve more slowly and gradually stabilize as the budget increases. Under the same iteration budget, PaveDSL generally achieves higher Pass@1 after the early iterations and converges to a higher level than the original pipelines. However, each additional failed verification may invoke another round of agent verification and repair, thereby increasing the potential number of model calls. We therefore set the maximum iteration budget to 3 in the main experiments to capture the early performance gains while limiting repeated model calls.

At the selected iteration budget, the computational cost of PaveDSL varies across benchmarks. PaveDSL introduces moderate overhead on xCodeEval. On BigCodeBench, however, it reduces token usage for most baselines and does not increase the number of API calls for any baseline. Together, these results indicate that early verification can trade planning-stage computation for lower downstream debugging cost, particularly on more complex tasks.

5.4 Case Study

To answer **RQ4**, Figure 5 presents an example from APPS 3554. The task requires finding the maximum score from a dice roll without using the same die in multiple scoring combinations. The natural-language plan lists the scoring rules and proposes exploring non-overlapping combinations, but it does not specify how consumed dice are tracked across recursive choices. The available public tests do not expose this ambiguity, so natural-language simulation accepts the plan.

PaveDSL converts the plan into requirement and function nodes. The Symbolic Verifier accepts the design because its fields, references, and dependencies are structurally valid. The Agent Verifier follows the dependency from `generate_scoring_moves` to `search_best_score` and finds that the generated scoring moves do not carry the state of consumed dice into recursive search. The Repair Agent therefore makes each scoring move record the dice it consumes and updates the search to recurse only over the remaining dice. The repaired design is then rechecked by both verifiers. This example shows how the structured design makes an ambiguous state transition explicit and provides localized repair feedback before code generation.

6 Conclusion

In this paper, we present PaveDSL, a neuro-symbolic approach to pre-code plan verification in multi-agent code generation. PaveDSL addresses unreliable verification of natural-language plans by transforming them into DSL-based structured designs that expose requirements, function responsibilities, and dependency relations before code generation. It combines deterministic and LLM-free symbolic checks of structural properties with LLM-based graph-guided verification of semantic

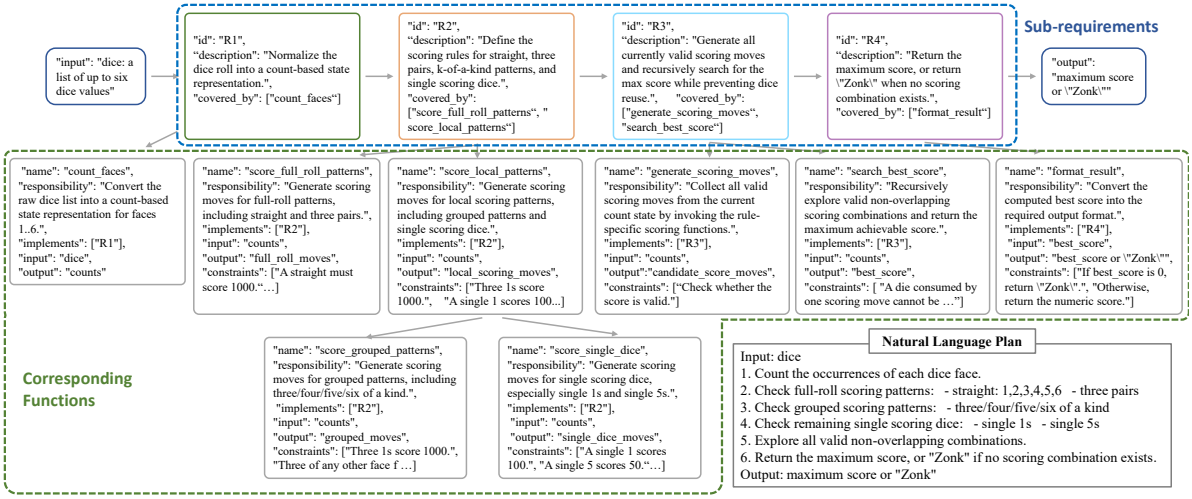


Figure 5: Case study of APPS 3554, comparing the natural-language plan with the structured design after verification and repair.

consistency. Feedback from both verifiers guides design repair before complete re-verification. Experiments across five benchmarks, four plan-based methods, and three LLM backbones show that PaveDSL consistently improves code generation performance. Ablation results further show that the gains do not arise from structured representation alone. The Symbolic Verifier, Agent Verifier, and Repair Agent play distinct but complementary roles in plan verification before code generation.

Limitations

Dependence on LLM-based Plan Specification. PaveDSL relies on an LLM to transform a natural-language plan into a DSL-based structured design. During this transformation, the model may omit, distort, or introduce semantic details even when the resulting design conforms to the schema. Although the subsequent verification process can identify many such errors, it cannot guarantee that every schema-compliant design preserves the intended algorithmic logic.

Limited Coverage of Public Tests. The Agent Verifier evaluates the behavior of a structured design by simulating it on the available public test cases. When these tests do not cover critical interactions, boundary conditions, or state transitions, an incorrect design may remain consistent with the observed test behavior and be accepted. Broader test generation or executable constraints could provide stronger semantic coverage.

Fixed Schema Expressiveness. The current PaveDSL schema uses seven fixed fields and represents plans as function-level data-flow graphs. This representation may be less suitable for tasks that do not decompose naturally into explicit function chains, including highly stateful simulations and mathematical derivations. Supporting task-adaptive schemas could extend PaveDSL to a broader range of solution structures.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (62176053).

References

Sanghyun Ahn, Wonje Choi, Junyong Lee, Jinwoo Park, and Honguk Woo. 2025. Towards reliable Code-as-Policies: A neuro-symbolic framework for embodied task planning. In *Advances in Neural Information Processing Systems*, volume 38, pages 75428–75459. Curran Associates, Inc.

Anthropic. 2025. Introducing Claude Haiku 4.5. https://www.anthropic.com/news/claude-haiku-4-5.

Erik Arakelyan, Pasquale Minervini, Patrick Lewis, Pat Verga, and Isabelle Augenstein. 2025. FLARE: Faithful logic-aided reasoning and exploration. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 23385–23403, Suzhou, China. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

DeepSeek. 2026. DeepSeek V4 preview release. https://api-docs.deepseek.com/news/news260424.

Giovanni Pio Delvecchio, Lorenzo Molfetta, and Gianluca Moro. 2025. Neuro-symbolic artificial intelligence: A task-directed survey in the black-box models era. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10418–10426.

Greta Dolcetti and Eleonora Iotti. 2025. A dual perspective review on large language models and code verification. *Frontiers in Computer Science*, 7:1655469.

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

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with LLM-based agents. *Preprint*, arXiv:2508.00083.

Yu Feng, Nathaniel Weir, Kaj Bostrom, Sam Bayless, Darion Cassel, Sapana Chaudhary, Benjamin Kiesl-Reiter, and Huzefa Rangwala. 2026. VeriCoT: Neuro-symbolic chain-of-thought validation via logical consistency checks. In *The Fourteenth International Conference on Learning Representations*.

Debargha Ganguly, Srinivasan Iyengar, Vipin Chaudhary, and Shivkumar Kalyanaraman. 2024. PROOF OF THOUGHT: Neurosymbolic program synthesis allows robust and interpretable reasoning. In *The First Workshop on System-2 Reasoning at Scale, NeurIPS'24*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pages 10764–10799. PMLR.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Md Tanzib Hosain, Salman Rahman, Md Kishor Morol, and Md Rizwan Parvez. 2025. Xolver: Multi-agent reasoning with holistic experience learning just like an olympiad team. *Preprint*, arXiv:2506.14234.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024a. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Xiaowei Huang, Wenjie Ruan, Wei Huang, Gaojie Jin, Yi Dong, Changshun Wu, Saddek Bensalem, Ronghui Mu, Yi Qi, Xingyu Zhao, Kaiwen Cai, Yanghao Zhang, Sihao Wu, Peipei Xu, Dengyu Wu, Andre Freitas, and Mustafa A. Mustafa. 2024b. A survey of safety and trustworthiness of large language models through the lens of verification and validation. *Artificial Intelligence Review*, 57(7):175.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5128–5154, Albuquerque, New Mexico. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D. Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. 2023. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, pages 944–949.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7).

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. 2025. Planning-driven programming: A large language model programming workflow. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12647–12684, Vienna, Austria. Association for Computational Linguistics.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2025. CodeTree: Agent-guided tree search for code generation with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3711–3726. Association for Computational Linguistics.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Feng Lin, Dong Jae Kim, and Tse-Hsun (Peter) Chen. 2025. Soen-101: Code generation by emulating software process models using large language model agents. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1527–1539. IEEE Press.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics.

Theo X. Olausson, Alex Gu, Benjamin Lipkin, Cedegao E. Zhang, Armando Solar-Lezama, Joshua B. Tenenbaum, and Roger Levy. 2023. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

OpenAI. 2025. GPT-5 system card. `https://cdn.openai.com/gpt-5-system-card.pdf`.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.

Ruwei Pan, Hongyu Zhang, and Chao Liu. 2025. CodeCoR: An LLM-based self-reflective multi-agent framework for code generation. *Preprint*, arXiv:2501.07811.

Flavio Petruzzellis, Cristina Cornelio, and Pietro Lio. 2025. Hierarchical planning for complex tasks with knowledge graph-RAG and symbolic verification. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267. PMLR.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Xin Quan, Marco Valentino, Louise A. Dennis, and Andre Freitas. 2024. Verification and refinement of natural language explanations through LLM-symbolic theorem proving. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 2933–2958. Association for Computational Linguistics.

Amit Sheth, Vishal Pallagani, and Kaushik Roy. 2024. Neurosymbolic AI for enhancing instructability in generative AI. *IEEE Intelligent Systems*, 39(5):5–11.

Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2025. Fixing large language models' specification misunderstanding for better code generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pages 1514–1526. IEEE Press.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with LLMs: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289.

Zhengren Wang, Rui ling, Chufan Wang, Yongan Yu, Sizhe Wang, Zhiyu li, Feiyu Xiong, and Wentao Zhang. 2025. Maintaincoder: Maintainable code generation under dynamic requirements. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Zhun Yang, Adam Ishay, and Joohyung Lee. 2023. Coupling large language models with logic programming for robust and general reasoning from text. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5186–5219, Toronto, Canada. Association for Computational Linguistics.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. Satisfiability-aided language models using declarative prompting. In *Advances in Neural Information Processing Systems*, volume 36.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024a. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1319–1331.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024b. Unifying the perspectives of NLP and software engineering: A survey on language models for code. *Preprint*, arXiv:2311.07989.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

A Implementation and Evaluation Details

A.1 Implementation Details

We implement PaveDSL in Python 3.11 and conduct the main experiments with GPT-5-mini (OpenAI, 2025), Claude-4.5-Haiku (Anthropic, 2025), and DeepSeek-V4-Flash (DeepSeek, 2026). We set the sampling temperature to 0.7 during planning and allow at most three verification-repair iterations. In each iteration, the Symbolic Verifier and Agent Verifier examine the same structured design in parallel. If either verifier rejects the design, the Repair Agent receives the combined feedback and revises the identified parts. Both verifiers then examine the complete repaired design again. During code generation, we set the temperature to 0 and allow at most five debugging iterations. All remaining hyperparameters follow the default settings of each baseline method.

For each paired comparison, the original method and its PaveDSL-enhanced version use the same backbone model and receive the same problem description and public test cases. We evaluate their generated code on the same hidden tests. These hidden tests are not exposed during planning, verification, repair, code generation, or debugging. Unless otherwise specified, the supplementary experiments use GPT-5-mini and follow the same generation and evaluation settings.

A.2 FPR Evaluation Protocol

Figure 1 evaluates planning-stage verification for CodeSim and CodeSim with PaveDSL using GPT-5-mini. For each benchmark problem, we separately run the two methods and record whether the planning-stage verifier accepts the final plan or structured design. We also retain the final planning artifact passed to code generation. We then determine the correctness of this artifact through a post-hoc procedure that combines repeated code generation, test execution, plan-code consistency assessment, and manual inspection.

For a plan accepted by the planning-stage verifier, we independently sample five code implementations under the same generation settings. Each implementation is executed against the benchmark test suite. Claude-Opus-4.6 then assesses whether each generated implementation faithfully follows its source plan, including whether it omits, modifies, or introduces algorithmic steps. The auxiliary model does not directly judge whether the plan correctly solves the original problem. Instead, it assesses whether the execution outcome can reasonably be attributed to the plan rather than to a deviation introduced during code generation.

When all five implementations are faithful to the plan and pass the tests, we label the plan as correct. When all five implementations are faithful to the plan and fail the tests, we label it as incorrect. If the five executions produce mixed outcomes or Claude-Opus-4.6 detects a discrepancy between the plan and any generated implementation, we manually inspect the plan, the implementations, and their execution results to determine whether the failure originates from the plan or the code-generation stage.

We also manually inspect cases in which the planning stage reaches the maximum verification-

repair budget without accepting the plan but the subsequently generated code passes the tests. A successful implementation in this setting can indicate either that the verifier incorrectly rejected a valid plan or that the code generator deviated from or implicitly corrected the plan. We therefore check whether the successful implementation faithfully follows the plan before assigning the final correctness label.

After obtaining the post-hoc correctness labels, we count a false positive when the planning-stage verifier accepts a plan that is subsequently determined to be incorrect. For each method and benchmark, we calculate

$$\mathrm{FPR} = \frac{N_{\text{accepted and incorrect}}}{N_{\text{incorrect}}}. \quad (1)$$

Here, $N_{\text{accepted and incorrect}}$ denotes the number of accepted plans subsequently determined to be incorrect, while $N_{\text{incorrect}}$ denotes the total number of plans subsequently determined to be incorrect. Across the five benchmarks, each method configuration contributes 721 final planning artifacts. We assign a post-hoc correctness label to each artifact based on the outcomes of five independently generated implementations.

B Supplementary Experiments

B.1 Cost Analysis

Table 3: Average API calls and token usage per problem for complete workflows using GPT-5-mini. Token counts are reported in thousands.

Approach	xCodeEval		CodeContests		BigCodeBench	
	API	Tokens	API	Tokens	API	Tokens
CodeSim	8.4	31.2	14.3	59.6	15.6	70.1
with PaveDSL	9.3	35.0	14.4	61.1	14.8	63.7
LPW	7.6	29.4	14.7	61.4	16.3	71.0
with PaveDSL	8.8	31.5	14.1	60.5	14.1	62.2
MaintainCoder	12.1	33.7	15.9	64.6	15.9	73.5
with PaveDSL	12.9	37.6	14.0	60.3	13.8	61.4
μFiX	8.1	30.8	13.5	54.2	15.2	66.9
with PaveDSL	9.3	33.0	13.6	55.9	15.2	69.1

Table 3 reports the average API calls and token usage per problem for each complete workflow using GPT-5-mini. The measurement covers planning, PaveDSL-based design generation and verification when present, code generation, and debugging. The Symbolic Verifier does not contribute an API call because it is a deterministic program. Comparing the complete workflows captures both the added planning-stage cost and any downstream debugging cost avoided after early verification.

On xCodeEval, PaveDSL increases API calls and token usage for all four methods, with CodeSim increasing from 8.4 to 9.3 API calls per problem. This overhead reflects the additional design generation, agent verification, and repair required when the original workflow involves relatively limited downstream debugging. The pattern differs on CodeContests and BigCodeBench, where PaveDSL reduces or preserves the measured costs in most settings. The overall cost therefore depends on the baseline and benchmark. From an efficiency perspective, PaveDSL is particularly well suited to more difficult tasks, where early plan verification can offset the greater cost of downstream debugging.

B.2 Additional Code Generation Methods

Table 4: Pass@1 results for four additional code generation methods with and without PaveDSL on three benchmarks using GPT-5-mini.

Approach	APPS	CodeContests	LiveCodeBench
LDB	67.3%	66.7%	68.6%
LDB with PaveDSL	**86.7%**	**79.4%**	**84.0%**
MapCoder	72.0%	70.9%	70.3%
MapCoder with PaveDSL	**82.0%**	**77.6%**	**77.1%**
PairCoder	77.3%	69.7%	69.1%
PairCoder with PaveDSL	**81.3%**	**77.6%**	**78.9%**
CodeTree	82.0%	75.8%	80.0%
CodeTree with PaveDSL	**86.7%**	**81.2%**	**86.9%**

A plan verification method should remain effective across pipelines that organize planning, strategy selection, and debugging differently. We assess this generality by integrating PaveDSL with LDB (Zhong et al., 2024), MapCoder (Islam et al., 2024), PairCoder (Zhang et al., 2024a), and CodeTree (Li et al., 2025). Table 4 reports their results on APPS, CodeContests, and LiveCodeBench using GPT-5-mini. For MapCoder, PaveDSL verifies the plan produced by the Planning Agent before the Code Generation Agent receives it. For PairCoder, PaveDSL verifies and repairs the plan selected by the Navigator before the Driver implements it. For CodeTree, PaveDSL verifies each strategy produced by the Thinker before the Solver generates the corresponding code. Their subsequent exploration, execution feedback, refinement, and candidate selection procedures remain unchanged.

LDB performs code-level debugging and does not originally generate an explicit plan. Its PaveDSL variant therefore adds a planning stage and PaveDSL before the initial code generation

Table 5: Pass@1 results for CodeSim and μFiX with and without PaveDSL using smaller open-weight backbone models.

Backbone	Approach	APPS	xCodeEval	CodeContests	BigCodeBench	LiveCodeBench
Qwen3-8B	CodeSim	74.7%	67.0%	64.2%	64.8%	65.1%
	CodeSim with PaveDSL	**76.7%**	**77.4%**	**70.3%**	**72.0%**	**70.3%**
	μFiX	72.0%	68.9%	63.6%	64.8%	64.0%
	μFiX with PaveDSL	**77.3%**	**75.5%**	**71.5%**	**71.2%**	**69.1%**
Gemma-3-12B	CodeSim	62.0%	61.3%	60.0%	57.6%	60.0%
	CodeSim with PaveDSL	**67.3%**	**72.6%**	**65.5%**	**67.2%**	**68.0%**
	μFiX	60.7%	61.3%	59.4%	54.4%	59.4%
	μFiX with PaveDSL	**64.0%**	**73.6%**	**64.8%**	**64.0%**	**63.4%**

while retaining the original runtime debugging process. This comparison changes the original pipeline more substantially than the integrations with MapCoder, PairCoder, and CodeTree, so we use it to examine the complementarity of plan-level verification and code-level debugging.

PaveDSL improves all four methods on each of the three benchmarks. The gains for MapCoder, PairCoder, and CodeTree show that selected candidate plans still benefit from verification before implementation. The gains for LDB further show that early plan verification can complement execution-level debugging performed after code generation.

B.3 Smaller Open-Weight Models

Robustness to backbone choice helps distinguish the contribution of PaveDSL from the capability of the underlying model. Accordingly, we evaluate CodeSim and μFiX with the smaller open-weight Qwen3-8B and Gemma-3-12B models rather than the larger backbones used in the main experiments. Table 5 reports results across all five benchmarks. For each backbone, the original baseline and its PaveDSL variant use the same model and evaluation instances. PaveDSL improves both baselines in all twenty paired settings, with gains ranging from 2.0 to 12.3 percentage points. Averaged across the two baselines and five benchmarks, the gain is 6.2 percentage points for Qwen3-8B and 7.4 percentage points for Gemma-3-12B. These results show that the improvements extend to smaller open-weight backbones.

B.4 Extended Function-Level Benchmarks

Simple function-level tasks provide a complementary setting for evaluating PaveDSL when planning errors are less frequent and baseline performance is already high, while the extended suites examine the effect of broader test coverage. Specifically, we evaluate HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and their extended variants. HumanEval and MBPP contain relatively simple tasks in which each problem requires implementing an individual function from a natural-language specification. Table 6 reports results using GPT-5-mini. Both baselines already achieve 100.0% on HumanEval, which leaves no room for further improvement. On MBPP, where the original methods also perform strongly, PaveDSL improves CodeSim by 3.1 percentage points and μFiX by 0.8 percentage points.

Table 6: Pass@1 results for CodeSim and μFiX with and without PaveDSL on standard and extended function-level benchmarks using GPT-5-mini.

Approach	HumanEval	HumanEval-ET	MBPP	MBPP-ET
CodeSim	100.0%	90.2%	91.9%	77.6%
CodeSim with PaveDSL	100.0%	**97.0%**	**95.0%**	**80.9%**
μFiX	100.0%	89.0%	92.9%	74.3%
μFiX with PaveDSL	100.0%	**94.5%**	**93.7%**	**78.1%**

HumanEval-ET and MBPP-ET retain the same function-level tasks but use more extensive test suites that expose additional boundary conditions and less common behaviors. PaveDSL improves CodeSim by 6.8 percentage points on HumanEval-ET and 3.3 percentage points on MBPP-ET. It improves μFiX by 5.5 and 3.8 percentage points on the same benchmarks. The larger gains on the extended suites, particularly relative to HumanEval and for μFiX on MBPP, are consistent with explicit constraints and data-flow relations helping the generated implementation cover behaviors omitted by simpler tests.

B.5 Serialization Format Comparison

Separating the effect of structured verification from that of a particular serialization syntax is necessary to identify the source of the gains. We isolate this factor by comparing JSON, Markdown, and YAML using GPT-5-mini with CodeSim and

Table 7: Pass@1 comparison of serialization formats for CodeSim with PaveDSL using GPT-5-mini.

Format	APPS	xCodeEval	CodeContests	BigCodeBench	LiveCodeBench
JSON	83.3%	**83.0%**	78.8%	**81.6%**	85.7%
Markdown	83.3%	82.1%	**79.4%**	**81.6%**	**86.3%**
YAML	83.3%	**83.0%**	**79.4%**	80.0%	85.7%

PaveDSL. As shown in Table 7, the experiment changes only the serialization format. All three variants preserve the same requirements, functions, inputs, outputs, constraints, and dependencies and use the same downstream verification and code generation pipeline. The maximum Pass@1 difference within any benchmark is 1.6 percentage points, and no format consistently outperforms the others. These results suggest that the explicit schema and verification process matter more to end-to-end performance than the surface syntax used to serialize the design. We retain JSON because its restricted grammar supports deterministic parsing and schema validation.

B.6 Performance across Difficulty Levels

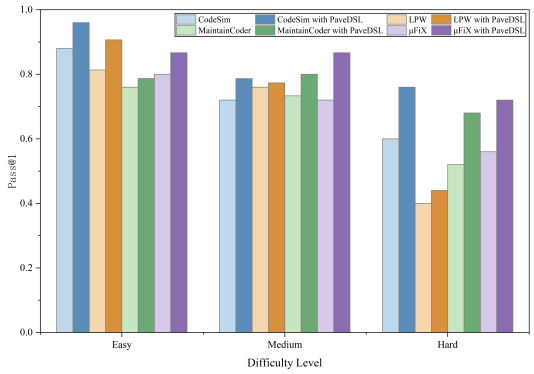


Figure 6: Pass@1 for four baselines with and without PaveDSL across easy, medium, and hard LiveCodeBench problems using GPT-5-mini.

Performance across difficulty levels characterizes how the benefit of PaveDSL changes with problem complexity. For this analysis, we compare CodeSim, LPW, MaintainCoder, and μFiX on easy, medium, and hard LiveCodeBench problems using GPT-5-mini. Figure 6 shows that PaveDSL improves every baseline at each difficulty level. The largest absolute gains occur on hard problems for CodeSim, MaintainCoder, and μFiX, whereas LPW achieves its largest gain on easy problems. Overall, PaveDSL improves all four baselines at every difficulty level, with three of them achieving their largest gains on hard problems.

B.7 Multilingual Robustness

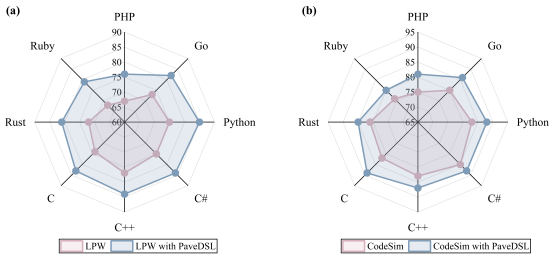


Figure 7: Number of correct solutions on xCodeEval across eight target languages for LPW in panel (a) and CodeSim in panel (b), with and without PaveDSL using GPT-5-mini.

Cross-language performance provides evidence about whether the structured verification process is tied to a particular target language. We evaluate this transfer on xCodeEval using GPT-5-mini. The experiment covers PHP, Go, Python, C#, C++, C, Ruby, and Rust and compares LPW and CodeSim with their PaveDSL-enhanced variants under the same language-specific evaluation settings.

Figure 7 shows that PaveDSL increases the number of correct solutions for both LPW and CodeSim in each of the eight evaluated languages. The consistent gains across both pipelines and all eight evaluated languages support the use of a common structured verification interface in multilingual code generation.

C Prompt Templates

This section presents the prompt templates for the three LLM-based components in PaveDSL. The Design Agent converts a natural-language plan into a structured design under the PaveDSL schema. The Agent Verifier performs graph-guided semantic checking and public-test simulation over the design. The Repair Agent uses localized feedback from both verifiers to revise the identified parts of the design.

Design Agent Prompt

```
You are the Design Agent.

Your task is to convert a natural-language plan into a structured design under the PaveDSL
    schema. Preserve the algorithmic logic of the plan while making the task goal, functional
     requirements, function nodes, data interfaces, constraints, and dependencies explicit.

Input:
Problem description: {problem_description}

Natural-language plan: {natural_language_plan}

PaveDSL schema: {pavedsl_schema}

Generate a JSON design with exactly the following seven top-level fields:

- Task: Describes the overall problem goal and contextual information, providing the high-
    level objective of the structured design.
- Requirements: Decomposes the task into functional requirements that specify what
    capabilities the solution should provide and what behaviors it should satisfy.
- Functions: Defines function nodes that implement the decomposed requirements, with each node
     responsible for a specific part of the solution logic.
- Inputs: Specifies the data consumed by each function node, including external inputs from
    the task and intermediate values from previous nodes.
- Outputs: Specifies the data produced by each function node, including intermediate results
    passed to later nodes and final answers returned by the solution.
- Constraints: Records lightweight semantic conditions, assumptions, boundary cases, and task-
    specific restrictions used during verification.
- Dependencies: Defines directed data-flow relations among function nodes, showing how
    produced outputs are passed as required inputs to later functions.

Generation requirements:
1. The design must be consistent with both the problem description and the given natural-
    language plan.
2. Use exactly the seven top-level fields defined by the schema and do not introduce
    unsupported solution logic.
3. Give every requirement a unique identifier and cover it with at least one function through
    covered_by.
4. Give every function a unique name and associate it with at least one requirement through
    implements.
5. Specify the inputs and outputs of every function and distinguish task data, intermediate
    data, and final outputs.
6. Record task-specific semantic conditions and boundary cases in Constraints.
7. Every dependency must connect existing functions and name data produced by the source and
    consumed by the target.
8. The dependency graph should provide a valid data-flow path from task inputs to final
    outputs without cycles.
9. Return valid JSON only. Do not include explanations outside the JSON.

Return format:
{
  "Task": "...",
  "Requirements": [
    {
      "id": "R1",
      "description": "...",
      "covered_by": ["function_name"]
    }
  ],
  "Functions": [
    {
      "name": "function_name",
      "description": "...",
      "responsibility": "...",
      "implements": ["R1"],
      "inputs": ["..."],
      "outputs": ["..."]
```

```
    }
  ],
  "Inputs": {
    "task_inputs": ["..."],
    "intermediate_inputs": {
      "function_name": ["..."]
    }
  },
  "Outputs": {
    "final_outputs": ["..."],
    "intermediate_outputs": {
      "function_name": ["..."]
    }
  },
  "Constraints": [
    {
      "target": "function_name or global",
      "description": "..."
    }
  ],
  "Dependencies": [
    {
      "from": "source_function",
      "to": "target_function",
      "data": "transferred_output"
    }
  ]
}
```

Agent Verifier Prompt

```
You are the Agent Verifier.

Your task is to assess the semantic consistency of a structured design before code generation.
     Use the function nodes and Dependencies field in the design to perform graph-guided
     checking and public-test simulation. This assessment does not constitute a formal proof
     of correctness.

Input:
Problem description: {problem_description}

Public test cases: {public_test_cases}

Structured design in JSON format: {structured_design}

Verification goals:
1. Check whether each function is suitable for its assigned requirements and constraints.
2. Check input and output consistency.
3. Simulate the behavior of the design on public test cases along the dependency graph.

Verification instructions:

First, check function responsibilities.
For each function node, examine whether its responsibility is clear and whether its described
     behavior is suitable for the requirements and constraints assigned to it. Check the
     behavior against the problem description without introducing assumptions that are absent
     from the task.

Second, check input and output consistency.
For each relation in Dependencies, check whether the output of the source function provides
     the input required by the target function. Check whether the name, meaning, and type of
     the transferred data remain consistent across the connected nodes.

Third, perform graph-guided test-case simulation.
For each public test case, derive an execution order from Dependencies. Simulate how each
     function processes its inputs, produces intermediate outputs, and passes those outputs to
```

```
     later functions. Compare the final result with the expected output. If a mismatch occurs,
     use the simulation trace to identify the responsible requirement, function, input,
    output, constraint, or dependency.

Important notes:
1. Do not rewrite the design.
2. Do not generate code.
3. Use only the public test cases provided in the input and do not assume access to hidden
    tests.
4. Do not claim formal correctness or perform formal theorem proving.
5. Mark the design as passed only if the function, interface, constraint, dependency, and
    public-test checks reveal no semantic inconsistency.
6. If the design fails, localize each issue and provide evidence that the Repair Agent can use
     directly.

Return format for a failed design:
{
  "passed": false,
  "feedback": [
    {
      "location_type": "function",
      "location": "function_name",
      "issue": "localized semantic inconsistency",
      "evidence": "reasoning or failed simulation step",
      "suggested_repair": "targeted correction"
    }
  ],
  "simulation_traces": [
    {
      "test_case": "public test identifier",
      "status": "fail",
      "failed_location": "function_name",
      "summary": "concise trace summary"
    }
  ]
}

Return valid JSON only. Use an empty feedback list when the design passes.
```

Repair Agent Prompt

```
You are the Repair Agent.

Your task is to repair a structured design using the localized feedback produced by the
    Symbolic Verifier and the Agent Verifier. The feedback identifies structural or semantic
    problems in requirements, function nodes, data interfaces, constraints, or dependencies.
    Perform targeted local repair rather than rewriting the complete design from scratch.

Input:
Problem description: {problem_description}

Current structured design in JSON format: {current_structured_design}

PaveDSL schema: {pavedsl_schema}

Localized feedback from the Symbolic Verifier: {symbolic_feedback}

Localized feedback from the Agent Verifier: {agent_feedback}

Repair principles:
1. Inspect both feedback sources and address every reported issue.
2. Modify only the problematic requirement, function, input, output, constraint, or dependency
     identified by the feedback.
3. Preserve all unaffected content unless a direct consistency update is required.
4. If a repair changes a function interface or dependency, update the directly connected
    fields and nodes to maintain consistent data flow.
```

```
5. Preserve the covered_by and implements links between requirements and functions.
6. The repaired design must follow the supplied schema and retain exactly its seven top-level
     fields.
7. Do not generate code or claim that the repaired design has passed verification.

Repair instructions:

First, read the two feedback lists.
Use each reported location as a repair target. A location may refer to a requirement, function
      node, data interface, constraint, dependency relation, or intermediate result in a
     simulation trace.

Second, perform local repair.
Revise only the fields that are necessary to fix the reported issue. For example:
- If the feedback indicates that a requirement is not covered, add or adjust the related
     function coverage.
- If the feedback indicates that a function behavior is incomplete, refine its responsibility
     or logic description.
- If the feedback indicates that an input-output relation is inconsistent, adjust the data
     interface.
- If the feedback indicates that a dependency creates a cycle, modify the dependency relation
     to recover a valid execution order.
- If the feedback indicates that a previous simulation trace failed at a specific node or
     intermediate result, revise the corresponding function, constraint, or data interface
     according to that feedback.

Third, check the impact of the local repair.
Determine whether the repair changes an input, output, constraint, or dependency used by
     another part of the design. If so, update the directly affected nodes or fields. Do not
     leave broken references, inconsistent data flow, or missing requirement-function links.

Fourth, return the complete repaired design using exactly the seven top-level fields in the
     supplied PaveDSL schema. PaveDSL will send this complete design back to both verifiers.
     Return valid JSON only and do not include explanations outside the JSON.
```