CodeStruct: Code Agents over Structured Action Spaces

Myeongsoo Kim, Joe Hsu, Dingmin Wang
Shweta Garg, Varun Kumar, Murali Krishna Ramanathan
AWS AI Labs
{mysoo, hchaochu, wdimmy, shwegarg, kuvrun, mkraman}@amazon.com

Abstract

LLM-based code agents treat repositories as unstructured text, applying edits through brittle string matching that frequently fails due to formatting drift or ambiguous patterns. We propose reframing the codebase as a *structured action space* where agents operate on named AST entities rather than text spans. Our framework, CodeStruct, provides `readCode` for retrieving complete syntactic units and `editCode` for applying syntax-validated transformations to semantic program elements. Evaluated on SWE-Bench Verified across six LLMs, CodeStruct improves Pass@1 accuracy by 1.2–5.0% while reducing token consumption by 12–38% for most models. Models that frequently fail to produce valid patches under text-based interfaces benefit most: GPT-5-nano improves by 20.8% as empty-patch failures drop from 46.6% to 7.2%. On CodeAssistBench, we observe consistent accuracy gains (+0.8–4.4%) with cost reductions up to 33%. Our results show that structure-aware interfaces offer a more reliable foundation for code agents.

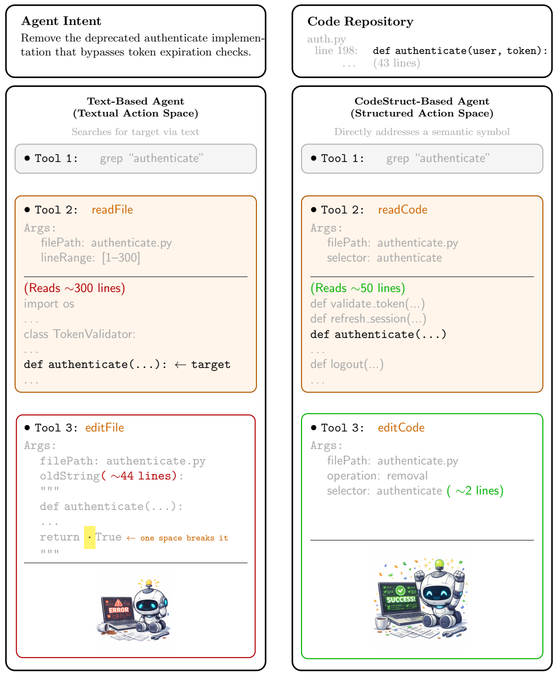

Figure 1: **Contrasting action spaces for code agents.** Text-based agents (left) read ~300 lines to locate a function and regenerate ~44 lines verbatim for removal, making edits brittle to formatting changes. **CodeStruct** (right) reads only the target symbol (~50 lines) and specifies removal in ~2 lines via a symbol-scoped edit.

1 Introduction

Large language models have enabled code agents to solve complex software engineering tasks such as repository-level bug fixing and feature implementation, as demonstrated by benchmarks like SWE-Bench (Jimenez et al., 2024). However, despite their growing capabilities, current agents interact with code repositories through a fundamental abstraction mismatch: they treat programs as flat text rather than structured artifacts. Agents read files as character sequences and apply edits via specifying line numbers or string patterns. This paradigm discards the syntactic and semantic structure inherent in source code, resulting in unstructured representations that are highly error-prone.

This text-centric paradigm introduces critical limitations for both reading and writing code. When retrieving code, agents must choose between reading entire files, which introduces irrelevant context that degrades reasoning (Shi et al., 2023), or selecting line ranges that often truncate functions mid-statement. When modifying code, string-based replacement is particularly wasteful and brittle: even minor edits require the model to regenerate significant amounts of original code verbatim, and such approaches frequently encounter "no occurrence" errors when code formatting drifts, or "multiple occurrence" errors when target patterns repeat across the codebase. These systematic failures force agents into costly trial-and-error cycles.

Recent systems attempt to address these issues by augmenting text-based tools with structural summaries such as repository maps, symbol indices, or dependency graphs (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025; Aider-AI, 2025). However, these mechanisms primarily guide *where* agents should look rather than *how* they interact with code. The underlying read and write actions remain fundamentally text-based, inheriting the same brittleness and inefficiencies.

We introduce **CODESTRUCT**, a framework that grounds agent interactions in AST structure. Source code is defined by precise syntax and organized into named entities, and AST-based transformations are standard in traditional software tools (Fluri et al., 2007; Falleri et al., 2014; van Tonder and Le Goues, 2019). However, these representations have not been adopted as the primary abstraction for LLM-based agents. Rather than operating on text spans, agents in **CODESTRUCT** reference code via **AST nodes** (e.g., `file.py::ClassName::method`) that unambiguously identify program entities regardless of line position. We provide two structure-aware primitives. The `readCode` operation retrieves complete syntactic units such as functions or classes without truncation or excess context, while `editCode` applies transformations directly to AST nodes, eliminating string-matching fragility. For node replacement, agents specify only the signature and new content, avoiding redundant regeneration of unchanged code. As illustrated in Figure 1, operations such as deletion and duplication become atomic actions that require only a node path, yielding an efficient and structure-grounded read/write paradigm for code agents.

We evaluate **CODESTRUCT** on two complementary benchmarks, SWE-Bench Verified (Jimenez et al., 2024) and CodeAssistBench (Kim et al., 2025), across multiple language models. On SWE-Bench Verified, **CODESTRUCT** improves Pass@1 accuracy by 1.2–5.0% for frontier models, with a 20.8 percentage point gain for a smaller model. For most configurations, these accuracy improvements coincide with reduced token consumption (12–38%) and lower inference cost (up to 33%). One notable exception is GPT-5-nano, which achieves a 20.8 percentage point accuracy gain at the cost of increased computation, as structured actions enable sustained exploration that would otherwise terminate in failure. On CodeAssistBench, **CODESTRUCT** consistently improves accuracy by 0.8–4.4% across all evaluated models. These results indicate that exposing codebases as structured action spaces improves agent effectiveness and reliability across diverse code tasks.

Our contributions are: **(1)** A structured action-space interface that bridges AST-based program representations and LLM-based agents, exposing named semantic entities as the primary units of code interaction; **(2)** two structure-aware primitives (`readCode`, `editCode`) designed for LLM usability, featuring human-readable selectors, automatic scope resolution, and syntax-validated edits that support robust agent recovery; and **(3)** extensive empirical evidence across two benchmarks and six language models showing that structure-aware action spaces improve both effectiveness and efficiency, with analysis revealing that gains are largest when text-interface brittleness—rather than reasoning capacity—is the dominant failure mode.[1]

2 Related Work

2.1 LLM-based Code Agents and Tools

Recent work on LLM-based code agents focuses on enabling models to solve repository-level tasks through iterative tool use. Systems such as SWE-Agent (Yang et al., 2024) and Agentless (Xia et al., 2024) equip agents with file-reading and text-editing tools, allowing them to explore repositories and apply patches in a multi-step manner. To improve scalability and navigation, some recent systems augment these textual tool interfaces with repository-level summaries or structure-aware retrieval mechanisms, such as file maps or symbol indices, which expose high-level information about file structure and function signatures (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025).

While these mechanisms improve code localization and retrieval, they stop short of defining executable, structure-aware action primitives for modification, leaving reads and edits fundamentally text-based. This forces agents to reason about and manipulate structured programs indirectly, motivating the need for action abstractions that operate directly over named program entities rather than unstructured text.

2.2 Code Search and Structural Abstractions

A substantial body of work incorporates program structure to improve code search and understanding. Path-based models such as Code2Vec (Alon et al., 2019) and PSCS (Sun et al., 2020) represent code using sequences of AST paths, enabling more

[1]Code and data are available at https://github.com/amazon-science/CodeStruct.

semantically meaningful retrieval than token-based methods. Similarly, ASTNN (Zhang et al., 2019) and its successors decompose abstract syntax trees into statement-level subtrees for neural representation learning, with later work augmenting these embeddings using static-analysis signals for tasks such as clone detection and code classification. While these approaches demonstrate that structural information is valuable for code representation, they operate exclusively at the encoding level: AST structure is consumed as input features in single-shot prediction settings, but is not exposed as an executable action space that agents can manipulate through tree-level edits in a multi-turn workflow.

2.3 Structure-Aware Program Repair and Code Generation

Structural cues have also been leveraged to improve one-shot program repair and code generation. Classical systems such as DeepFix (Gupta et al., 2017), DrRepair (Yasunaga and Liang, 2020), and BIFI (Yasunaga and Liang, 2021) rely on ASTs or compiler feedback to reduce syntax errors and localize bugs, while grammar-based decoders explicitly enforce syntactic correctness during generation. Similarly, abstract syntax networks (Rabinovich et al., 2017) and retrieval-augmented structural models (Hashimoto et al., 2018) generate or edit code under tree- or grammar-constrained representations to respect programming-language syntax. While these approaches effectively leverage AST structure to constrain or guide prediction, they operate in a single-shot setting and do not define an executable, step-by-step action space over tree edits. As a result, models produce a complete patch or AST in one pass, rather than performing a sequence of explicit, traceable AST transformations suitable for multi-turn agent workflows.

2.4 AST Diffing and Tree Transformations

Work on tree diffing provides the closest analogy to our framework. Algorithms such as GumTree (Falleri et al., 2014) compute fine-grained edit scripts between ASTs using operations such as insert, delete, update, and move, and systems such as PyGGI (An et al., 2019) adopt similar primitives for genetic improvement. However, these operations are primarily used for offline diffing or evolutionary search, rather than as decision-time primitives for an LLM-driven code-editing agent. Beyond diffing, several systems support structural code transformations via AST-aware rules, including Comby (van Tonder and Le Goues, 2019), Piranha (Ramanathan et al., 2020; Ketkar et al., 2024), and Semgrep (Bennett et al., 2024). These tools enable precise, semantics-aware rewrites, but require transformation patterns to be specified *a priori*, making them unsuitable for open-ended problem solving.

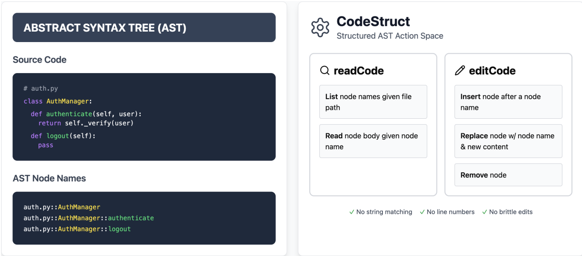

Figure 2: Overview of **CODESTRUCT**. Code agents interact with repositories through a structured AST action space. Source code is parsed into an AST, exposing addressable nodes. The `readCode` and `editCode` tools operate directly on these nodes, enabling structure-aware code navigation and modification without string matching, line numbers, or brittle edits.

Unlike prior AST-based systems that apply transformations offline or through fixed rewrite rules, **CODESTRUCT** exposes semantic program entities as first-class, decision-time actions that an LLM agent can dynamically construct and invoke during multi-step problem solving.

3 CODESTRUCT

We introduce **CODESTRUCT**, a structure-aware interface that exposes a codebase as a structured action space for LLM-based agents. Rather than interacting with repositories through unstructured text spans, agents using **CODESTRUCT** operate over named program entities derived from the abstract syntax tree (AST). This design enables agents to read and modify code via semantically grounded, executable actions with well-defined scope boundaries and structural guarantees.

3.1 Structured AST Action Space

A key observation in **CODESTRUCT** is that source code already defines a rich symbolic interface. Functions, classes, and methods are named program entities with well-defined scope, boundaries, and semantics, and human developers reason about and modify code primarily by referring to these entities by name, rather than by line numbers or character offsets. **CODESTRUCT** exposes this existing structure directly to language-model agents.

As illustrated in Figure 2, **CODESTRUCT** represents a codebase as a structured environment defined by its abstract syntax tree (AST). The AST serves as the environment state over which an agent operates, making named and addressable program entities—such as files, classes, functions, and methods—explicit and executable. In this formulation, the AST is not merely a parsing artifact, but the mechanism that enables unambiguous reference to semantic program elements.

Rather than interacting with repositories through anonymous text spans, **CODESTRUCT** agents act through a small set of structure-aware primitives that operate directly on named AST entities. Concretely, the action space consists of two primitive operations: `readCode` and `editCode`. Each action is parameterized by a selector that identifies a target program entity, allowing the agent to specify *what* it intends to inspect or modify without committing to *how* that change is realized in source text.

A `readCode` action retrieves code for a selected AST node, returning complete syntactic units (e.g., functions or classes) or compact structural summaries. An `editCode` action performs AST-level transformations—such as insertion, replacement, or removal—to a selected entity, producing a modified AST that is syntactically valid by construction.

Each `editCode` action transforms the current AST into a new, syntactically valid AST. Consequently, a multi-step repository-level editing process can be viewed as a trajectory of structured actions over successive AST states. This formulation yields explicit and analyzable action traces, enabling fine-grained analysis of agent behavior beyond final patch correctness.

By grounding all interactions in named program entities, **CODESTRUCT** explicitly separates semantic intent from textual realization. This design avoids brittle dependencies on line numbers or string matching, ensures that edits respect syntactic boundaries, and aligns the agent's action space with the abstractions used by human developers.

3.2 Structure-Aware Tools

We instantiate this action space through two core tools: a structure-aware read operation (`readCode`) and a structure-aware edit operation (`editCode`). Together, these tools define the primitive interactions available to an agent. Algorithms 1 and 2 formalize these operations.

`readCode` This tool provides structure-aware code retrieval by exposing named program entities as the primary units of access (Algorithm 1).

Algorithm 1 READCODE: Structure-Aware Code Retrieval

Require: File path p, file size threshold τ, selector σ (optional), line range (l_s, l_e) (optional)
Ensure: Code content or structural summary

if l_s, l_e specified **then**
    return EXTRACTLINES(p, l_s, l_e)
end if
if $\sigma = \emptyset$ **and** $|p| < \tau$ **then**
    return the full file content of p
end if
$\mathcal{T} \leftarrow$ PARSEAST(p) {Parse file into AST}
$\mathcal{E} \leftarrow$ EXTRACTSIGNATURES$(\mathcal{T})$ {Extract functions, classes}
if $\sigma = \emptyset$ **then**
    return FORMATSIGNATURES$(\mathcal{E})$ {Structural summary}
else
    $\mathcal{M} \leftarrow$ FUZZYMATCH$(\mathcal{E}, \sigma)$ {Selector matching}
    return $\{e.\text{impl} : e \in \mathcal{M}\}$ {e.impl denotes the full source span of entity e (its AST subtree rendered as code)}
end if

The tool supports a coarse-to-fine workflow with three common modes.

(1) Repository browsing (directory input). When the input path p is a directory, `readCode` returns the list of files under p (optionally filtered to source files). This enables agents to navigate the repository layout before selecting a file to inspect.

(2) File summarization (file input, no selector). When p is a file and no selector σ is provided, the tool adaptively returns either (i) the full file content if the file is small (below a size threshold τ, e.g., 10K characters), or (ii) a compact structural summary if the file is large. The summary consists of signatures of top-level entities (e.g., classes and functions) and their scoped names, allowing the agent to identify relevant program entities without loading the entire file.

(3) Entity retrieval (file + selector). When p is a file and a selector σ is provided, `readCode` resolves σ to one or more program entities extracted from the file's AST and returns the complete implementation of each matched entity (i.e., the AST subtree rendered back into source code). Selectors can be *unscoped* (e.g., `load`) or *scoped* (e.g., `User.load`) to restrict matches to methods within a specific class. Selector resolution uses deterministic name-based matching; for example, `guf` can match `get_user_file`, and `User.load` matches method `load` in class `User`. This matching is deterministic and does not involve learned components.

`readCode` encourages agents to first discover relevant entities via directory browsing and structural

Algorithm 2 EDITCODE: Structure-Aware Code Modification

Require: File path p, operation $\omega \in \{\texttt{insert}, \texttt{replace}, \texttt{removal}\}$, selector σ, replacement code r
Ensure: Modified file with syntactic validity guarantee

$\mathcal{T} \leftarrow$ GETORPARSEAST(p) {Reuse cached AST if available}
$n \leftarrow$ FINDNODE($\mathcal{T}, \sigma$) {Locate target AST node}
if $n = \emptyset$ **then**
  return ERROR("Selector not found")
end if
$\delta \leftarrow$ GETINDENTATION(n) {Preserve formatting}
$r' \leftarrow$ APPLYINDENTATION(r, δ)
if $\omega = \texttt{insert}$ **then**
  $\mathcal{T}' \leftarrow$ INSERTAFTER($\mathcal{T}, n, r'$)
else if $\omega = \texttt{replace}$ **then**
  $\mathcal{T}' \leftarrow$ REPLACENODE($\mathcal{T}, n, r'$)
else if $\omega = \texttt{removal}$ **then**
  $\mathcal{T}' \leftarrow$ REMOVENODE($\mathcal{T}, n$)
end if
if HASSYNTAXERROR($\mathcal{T}'$) **then**
  return ERROR("Invalid syntax") {Reject malformed edits}
end if
WRITEFILE(p, $\mathcal{T}'$)
return SUCCESS

summaries, then selectively retrieve only the implementations needed for reasoning. Unlike line-range reading, selector-based retrieval returns complete syntactic units, reducing irrelevant context and avoiding brittle dependence on line numbers.

`editCode` The tool performs structure-aware modification by applying AST-grounded transformations to named program entities (Algorithm 2). Each edit is specified by an operation type ω (insertion, replacement, or removal) and a selector σ that identifies the target entity in the AST.

Given a selector, `editCode` locates its associated AST node and applies the requested transformation within the node's syntactic scope. The tool automatically preserves formatting by computing the local indentation context and validating the modified AST before committing the change. Edits that would introduce syntax errors are rejected, ensuring post-edit syntactic validity via AST validation.

By exposing edits as atomic operations over named entities, `editCode` enables agents to perform targeted and interpretable modifications such as adding a new method, deleting an obsolete function, or replacing the implementation of an existing routine. This design separates semantic intent from textual realization: the agent specifies *what* to change via an entity-level selector, while the tool determines *how* to apply it in the source text.

This syntactic validity guarantee distinguishes CODESTRUCT from text-based editing approaches, where edits are applied via line numbers or string matching and malformed changes can silently corrupt the codebase. Each `editCode` invocation produces an explicit and traceable state transition, enabling fine-grained analysis of agent trajectories beyond final patch correctness.

Agent Interface and Integration `readCode` and `editCode` are exposed through a standardized tool interface, allowing them to be invoked by arbitrary LLM-based agents. In particular, the interface is implemented using the Model Context Protocol (MCP), which is supported by most existing agent frameworks. As a result, CODESTRUCT can be integrated into off-the-shelf agents without modifying their planning or execution logic, enabling the structured action space to be adopted independently of agent-specific infrastructure.

4 Experiments

4.1 Tasks and Datasets

We evaluate CODESTRUCT on repository-level software engineering benchmarks requiring agents to perform multi-step code understanding and modifications across multiple files and program entities.

SWE-Bench Verified. This benchmark consists of 500 real-world Python GitHub issues paired with failing tests that specify the desired behavior (Jimenez et al., 2024) . Solving a task requires locating the relevant code regions, understanding the underlying bug or feature request, and modifying one or more program entities to satisfy the test cases. SWE-Bench-Verified has emerged as a standard benchmark for evaluating repository-level program repair systems, where success is primarily measured by the correctness of applied code edits. These tasks naturally stress an agent's ability to navigate and manipulate codebases through structured interactions rather than ad hoc text edits.

CodeAssistBench Verified. It consists of 135 multi-turn programming assistance tasks across 7 programming languages derived from real-world GitHub issues that involve clarification, code exploration, and iterative refinement (Kim et al., 2025). Tasks in CodeAssistBench frequently require agents to inspect multiple functions or classes, reason about their relationships, and apply localized changes over several interaction steps. Unlike SWE-Bench, CodeAssistBench is not exclusively focused on producing a final patch, but instead evaluates an agent's ability to support interactive and exploratory programming workflows.

4.2 Baselines

We compare **CODESTRUCT** with representative baselines that differ in how agents interact with code repositories.

Baseline Text-Based Agents. We compare against SWE-Agent (Yang et al., 2024) and Open-Hands (Wang et al., 2025), which represent the dominant text-based interaction paradigm. Both systems retrieve code by reading entire files or line ranges and apply modifications via string replacement. To ensure a fair comparison, we enable SWE-Agent's repository map feature, which exposes file structure and function signatures as navigation hints. Despite this structural guidance, all read and edit operations remain text-based: the agent must specify line numbers or match exact strings to retrieve or modify code. This configuration represents the strongest reasonable baseline within the text-based paradigm.

We note that structural summaries such as repository maps, symbol indices, and dependency graphs operate at a different and complementary level from **CODESTRUCT**: they primarily support *navigation and planning* (deciding *where* to look), while **CODESTRUCT** targets the *action interface* (redefining *how* agents read and edit code once a target is identified). Since our baseline already includes such structural summaries, the comparison isolates the effect of changing the read/edit action space while holding navigation aids constant.

CODESTRUCT Agents. In contrast, agents using **CODESTRUCT** interact with repositories through structure-aware `readCode` and `editCode` tools. `readCode` is scoped to named functions, classes, or methods, while `editCode` is executed as AST-level transformations that preserve syntactic validity. This interface enables agents to directly operate over program structure, rather than reasoning indirectly through unstructured text.

4.3 Experimental Setup

Models. We evaluate all methods using a diverse set of large language models that span both proprietary and open-weight families, including GPT-5 (OpenAI, 2025), GPT-5-mini, GPT-5-nano, Qwen3-480B-A30B-Coder (Yang et al., 2025), Qwen3-32B, and Qwen3-8B. Unless otherwise specified, all models are used with default decoding parameters. No method is provided with model-specific fine-tuning or task-specific adaptations.

Prompts and Budgets. To isolate the effect of the action interface, we do not modify the system or task prompts between the baseline and CODESTRUCT, and use the default prompts provided by the underlying agent frameworks. We fix a maximum interaction budget for each task to \$5 for large models (GPT-5 and Qwen3-480B-A30B-Coder), \$3 for mid-size models (GPT-5-mini and Qwen3-32B), and \$1 for small-size models (GPT-5-nano and Qwen3-8B).

4.4 Results

4.4.1 SWE-Bench Verified Results

Table 1 reports the main results of integrating **CODESTRUCT** into SWE-Agent-style workflows on SWE-Bench Verified. Beyond accuracy, **CODESTRUCT** substantially improves interaction efficiency. Across most models, token consumption is reduced by 12–38%, reflecting fewer large file reads and more targeted access to relevant program elements. Output token usage also decreases consistently for most models, indicating that structure-aware actions reduce redundant rewrite attempts and patch churn, rather than merely shifting cost from reads to edits. These improvements are accompanied by reductions in the number of API calls, suggesting that agents converge to correct solutions in fewer interaction steps.

Reductions in token usage directly translate into lower inference cost, where **CODESTRUCT** reduces the total inference cost by 19.5% for GPT-5, 32.6% for GPT-5-mini, and 17.4% for Qwen3-32B, while simultaneously improving Pass@1.

Addressing Motivating Limitations. Aggregate improvements in Table 1 reflect **CODESTRUCT**'s impact each motivating limitation in the Introduction. We explicitly map each to its empirical proxy:

Irrelevant context → Input tokens. Selector-based retrieval returns only the targeted syntactic unit rather than entire files, reducing input tokens by 12–38% across most models (Table 1); removing `readCode` reverses this trend, increasing input tokens by +41% for Qwen3-32B (Table 4). On `django__django-11211`, the text-based agent reads ~300 lines while our **CODESTRUCT** agent reads ~50 (Figure 1).

Wasteful exploration → Interaction steps

Model	Interface	Pass@1 (%) ↑	Input Tokens	Output Tokens	LLM Calls ↓	Cost ($) ↓
GPT-5	Baseline	66.0	452.7M	0.81M	16,436	574.0
	CodeStruct	**67.2** (+1.2)	**366.3M** (-19.1%)	**0.44M** (-45.7%)	**16,307** (-0.8%)	**462.2** (-19.5%)
GPT-5-mini	Baseline	60.4	593.7M	1.27M	18,560	151.0
	CodeStruct	**62.0** (+1.6)	**404.5M** (-31.9%)	**0.35M** (-72.4%)	**14,811** (-20.2%)	**101.8** (-32.6%)
GPT-5-nano	Baseline	19.6	808.0M	0.86M	24,037	**40.7**
	CodeStruct	**40.4** **(+20.8)**	1,137.4M (+40.8%)	0.95M (+10.5%)	27,278 (+13.5%)	57.3 (+40.8%)
Qwen3-Coder	Baseline	61.2	805.8M	1.50M	26,961	365.3
	CodeStruct	**66.2** (+5.0)	**705.3M** (-12.5%)	**2.17M** (+44.6%)	**32,346** (+20.0%)	**321.3** (-12.1%)
Qwen3-32B	Baseline	14.8	366.0M	1.23M	25,543	55.6
	CodeStruct	**16.0** (+1.2)	**302.1M** (-17.5%)	**1.03M** (-16.3%)	**24,653** (-3.5%)	**45.9** (-17.4%)
Qwen3-8B	Baseline	13.2	84.0M	0.11M	8,833	2.36
	CodeStruct	13.0 (-0.2)	**51.8M** (-38.3%)	**0.08M** (-27.3%)	**8,313** (-5.9%)	**1.46** (-38.1%)

Table 1: Main results on SWE-Bench Verified. Agents follow SWE-Agent-style workflows—iterative loops that alternate between reading repository files and applying code edits via tool calls. We compare text-based interaction (*Baseline*) against **CODESTRUCT**. Inline percentages indicate relative change compared to *Baseline* (percentage points for Pass@1; relative % for tokens, calls, and cost). Green denotes improvement; red denotes regression.

Model	Interface	Accuracy (%) ↑	Input Tokens	Output Tokens	LLM Calls ↓	Cost ($) ↓
GPT-5	Baseline	53.3	143.9M	1.27M	566	$19.57
	CodeStruct	**54.1** (+0.8)	**122.1M** (-15.1%)	**1.18M** (-7.0%)	**550** (-2.8%)	**$16.74** (-14.5%)
GPT-5-mini	Baseline	51.1	125.3M	1.26M	648	$3.45
	CodeStruct	**51.9** (+0.8)	**83.1M** (-33.7%)	**0.89M** (-28.7%)	**630** (-2.8%)	**$2.30** (-33.3%)
GPT-5-nano	Baseline	46.7	56.0M	1.25M	742	$0.34
	CodeStruct	**48.1** (+1.4)	**52.8M** (-5.7%)	**1.05M** (-15.6%)	**652** (-12.1%)	**$0.32** (-5.9%)
Qwen3-Coder	Baseline	31.1	829.1M	6.95M	796	$385.61
	CodeStruct	**31.9** (+0.8)	**779.2M** (-6.0%)	7.00M (+0.7%)	826 (+3.8%)	**$363.24** (-5.8%)
Qwen3-32B	Baseline	15.6	110.5M	2.18M	858	$29.01
	CodeStruct	**20.0** (+4.4)	142.4M (+28.9%)	**1.25M** (-42.7%)	**776** (-9.6%)	$38.71 (+23.7%)
Qwen3-8B	Baseline	13.3	0.63M	0.21M	984	$0.05
	CodeStruct	14.1 (+0.8)	**0.55M** (-12.7%)	**0.17M** (-21.7%)	**944** (-4.1%)	**$0.04** (-17.6%)

Table 2: CodeAssistBench results comparing text-based interaction (*Baseline*) against CODESTRUCT. Inline values show relative change over *Baseline* (percentage points for Accuracy; relative % for tokens, calls, and cost).

and LLM calls. On the same instance, localization drops from 21 steps to 2 (Table 3; Appendix A). Across all instances, LLM calls decrease by up to 20.2% (GPT-5-mini). Without `readCode`, `str_replace` calls increase 7.8× on regressed instances (Appendix B).

Brittle string-matching edits → Tool-level errors and empty patches. For capable models, edit errors per instance decrease by 76–88% (Table 5). Empty-patch failures drop from 233 to 36 for GPT-5-nano (-84.5%; Appendix D).

Redundant code regeneration → Output tokens. Structure-aware edits specify only the entity name and new content, avoiding verbatim reproduction of surrounding code. Output tokens decrease by 45.7% for GPT-5 and 72.4% for GPT-5-mini.

Two models exhibit increased token usage under CODESTRUCT. For GPT-5-nano, CODESTRUCT achieves substantially higher accuracy at the expense of increased computation. This reflects a natural tradeoff under limited model capacity: structure-aware actions encourage deeper exploration and sustained interaction rather than early termination, increasing compute while improving solution quality. For Qwen3-Coder (480B), although input tokens decrease by 12.5%, output tokens increase by 45% and LLM calls rise by 20%. We therefore do not claim efficiency improvements for this model. Instead, the benefit is accuracy: +5.0pp in Pass@1 at a 12.1% lower total cost, since input tokens dominate the cost. This suggests that CODESTRUCT leads the model to spend more effort on intermediate reasoning and problem localization rather than on repeated text-based editing attempts. This indicates that CODESTRUCT's benefits are not limited to uniform efficiency gains, but can also arise from improving the quality of agent exploration under a fixed interaction budget.

Code Editing Error Analysis. To understand how structured interfaces affect operational reliability, we analyze tool-level error patterns—failed edit operations—across all agent trajectories (see Appendix C for error analysis and Appendix D for empty-patch analysis). Table 5 reveals a clear capability-dependent effect. For higher-capacity models (GPT-5, GPT-5-mini, Qwen3-Coder), **CODESTRUCT** reduces errors per instance by 76–88%, indicating that AST-based operations are substantially more reliable than text-based string matching.

For Qwen3-8B and Qwen3-32B, **CODESTRUCT** also substantially reduces tool-level errors, confirming that structured actions mitigate interface brittleness even for weaker models; however, error counts remain high in absolute terms (10–14 per instance), and accuracy gains are limited.

In contrast, GPT-5-nano exhibits a 20% increase in tool-level errors despite a 20.8pp accuracy gain, reflecting a redistribution rather than a reduction of failures: structured navigation enables correct localization and more edit attempts while dramatically reducing early agent terminations (empty patches drop from 233 to 36).

Ablation Study. To isolate the contribution of each component, we evaluate **CODESTRUCT** with either `readCode` or `editCode` removed (Table 4; detailed analysis in Appendix B). Both primitives contribute to effectiveness, but in complementary ways. Removing `readCode` causes the largest accuracy degradation (-7.8 Pass@1 for Qwen3-32B, -5.2 for GPT-5-mini), accompanied by substantially higher token usage and more LLM calls, indicating that without structured navigation, agents resort to inefficient trial-and-error exploration. Notably, Qwen3-32B without `readCode` underperforms even the text-based baseline (8.2% vs. 14.8%). This occurs because the hybrid configuration creates a mismatch: the agent's prompts and planning still expect structured navigation (e.g., selector-based retrieval), but without `readCode` it must fall back to text-based exploration, leading to less coherent search strategies than the fully text-based baseline where all tools are mutually consistent. Analysis of regressed instances confirms this: `str_replace` calls increase by $7.8\times$ and the dominant failure mode shifts from incorrect patches to budget exhaustion (Appendix B). Removing `editCode` yields smaller accuracy drops but disproportionate cost penalties: GPT-5-mini incurs a 38.7% cost increase for only 1.4pp less accuracy, as agents fall back to brittle string-based edits requiring more validation cycles. These results confirm that `readCode` and `editCode` serve complementary roles: structured navigation minimizes exploration cost while structured editing minimizes transformation cost.

Qualitative Analysis. To illustrate these efficiency gains concretely, we compare AST-based and text-based agent trajectories on representative SWE-Bench instances (Appendix A). On `django__django-11211`, the text-based agent spends 21 steps navigating via `grep`, `sed`, and line-range reads before locating the target method, while **CODESTRUCT** achieves the same localization in 2 steps using selector-based access. This reduces total steps from 54 to 24—a 55% reduction—demonstrating how structure-aware navigation eliminates the trial-and-error exploration common in text-based approaches.

Overall, these results demonstrate that exposing the codebase as a structured action space improves both the effectiveness and efficiency of SWE-Agent on SWE-Bench, while also reducing interaction-level failure modes, without requiring model-specific tuning or changes to the agent's decision-making logic.

4.4.2 Results on CodeAssistBench

We further evaluate **CODESTRUCT** on **CodeAssistBench (CAB)**, a benchmark designed to assess multi-turn, interactive code-assistance scenarios. Unlike SWE-Bench, CAB emphasizes conversational problem solving and incremental tool use rather than single-shot patch generation. We use OpenHands, the default agent framework provided by CodeAssistBench, and added read and edit operations with CODESTRUCT.

Table 2 compares the baseline text-based interaction interface against CODESTRUCT with structure-aware AST read and edit actions across six models. CODESTRUCT consistently improves accuracy while reducing token usage and cost for most models. Among GPT-family models, CODESTRUCT achieves accuracy gains of $+0.8$ to $+1.4$ percentage points while reducing resource consumption. GPT-5-mini shows the largest efficiency improvement, with input tokens reduced by 33.7% and cost reduced by 33.3%. GPT-5-nano achieves the highest accuracy gain ($+1.4$) alongside a 12.1% reduction in LLM calls. For Qwen models, results are more nuanced. Qwen3-32B shows the largest accuracy improvement (+4.4), though at the cost of increased input tokens (+28.9%). This shows that for weaker models, CODESTRUCT's structured actions enable more thorough exploration that improves solution quality. In contrast, Qwen3-Coder and Qwen3-8B show modest accuracy gains with improved efficiency.

Overall, these results demonstrate that structured program interactions generalize beyond patch-based benchmarks to interactive code-assistance workflows, where efficient exploration and precise context selection are critical.

5 Conclusion

We introduced **CODESTRUCT**, a structure-aware interface that exposes a codebase as a programmable action space for LLM-based agents. Rather than interacting with repositories through unstructured text spans, agents operate over named program entities derived from the AST, enabling semantically grounded reads and syntax-preserving edits. This design directly addresses the abstraction mismatch in existing code agents, which treat programs as flat text despite their inherently structured nature, and provides a principled alternative for repository-level code reasoning and modification. Across SWE-Bench Verified and CodeAssistBench Verified, we showed that replacing text-based read and edit operations with structure-aware actions improves both effectiveness and efficiency. **CODESTRUCT** reduces unnecessary context retrieval, lowers inference cost, and mitigates brittle string-based failures, with particularly large gains for models whose failures stem from text-interface brittleness rather than reasoning limitations.

Limitations

File-Level AST Scope. **CODESTRUCT** currently operates on per-file ASTs and does not explicitly model cross-file dependencies such as inheritance hierarchies or inter-module call graphs. We note that text-based baselines also operate at the file level with no cross-file semantic modeling, so this is a shared limitation of current agent interfaces rather than one specific to **CODESTRUCT**. Our strongest baseline already includes SWE-Agent's repository map, which provides file-level structural summaries and symbol indices as navigation hints; **CODESTRUCT** is complementary to such mechanisms, changing *how* agents read and edit code once a target file is identified, rather than *where* they look. Incorporating cross-file structure (e.g., inheritance hierarchies, call graphs) is a promising direction, though it involves a practical trade-off: file-level AST parsing is stateless and completes in milliseconds, while cross-file analysis requires whole-repository indexing that must be updated after every edit.

Language Coverage. Our bug-fixing evaluation on SWE-Bench Verified focuses on Python, though CodeAssistBench provides additional coverage across seven languages. Extending the evaluation to other languages with different syntactic characteristics (e.g., statically-typed languages with complex generics) is a natural direction for future work.

AST Parsing Overhead. AST construction introduces additional tool execution time relative to raw text operations. However, this overhead is negligible compared to LLM inference latency. Using tree-sitter for local parsing, median execution times are 146–171ms for `readCode` and 189–212ms for `editCode`, while median LLM call latency ranges from 4–12 seconds. Parsed ASTs are cached via GETORPARSEAST (Algorithm 2, line 1) to avoid redundant parsing. End-to-end, all AST operations consumed 35.7 minutes across 500 SWE-Bench runs with GPT-5 (75.95 compute hours total), accounting for less than 0.8% of total runtime—substantially outweighed by the 12–38% reduction in LLM tokens and inference calls.

Robustness to Syntax Errors. **CODESTRUCT** requires syntactically valid source files for AST parsing. While our `editCode` tool includes syntax validation to reject malformed edits and preserve repository integrity (Algorithm 2), files that are already syntactically invalid prior to agent interaction cannot benefit from structure-aware operations. In practice, committed code in software repositories is overwhelmingly syntactically valid, limiting the impact of this constraint.

Task Coverage. We evaluate on two core repository-level coding tasks: bug fixing (SWE-Bench Verified) and interactive code assistance (CodeAssistBench). Evaluating structure-aware action spaces on additional tasks such as code review and test generation remains future work.

References

Aider-AI. 2025. Aider: Ai pair programming in your terminal. `https://github.com/Aider-AI/aider`. Accessed: 2025-12.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL).

Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. Pyggi 2.0: Language independent genetic improvement framework. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1100–1104.

Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. 2024. Semgrep*: Improving the limited performance of static application security testing (sast) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 614–623, New York, NY, USA. Association for Computing Machinery.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA. Association for Computing Machinery.

Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, pages 1345–1351.

Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *Advances in Neural Information Processing Systems*, 31.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*.

Ameya Ketkar, Daniel Ramos, Lazaro Clapp, Raj Barik, and Murali Krishna Ramanathan. 2024. A lightweight polyglot code transformation language. *Proc. ACM Program. Lang.*, 8(PLDI).

Myeongsoo Kim, Shweta Garg, Baishakhi Ray, Varun Kumar, and Anoop Deoras. 2025. Codeassistbench (CAB): Dataset & benchmarking for multi-turn chat-based code assistance. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

OpenAI. 2025. Gpt-5 system card. Technical report.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1139–1149.

Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha: Reducing feature flag debt at uber. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 221–230.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR.

Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. 2020. Pscs: A path-based neural model for semantic code search. *arXiv preprint arXiv:2008.03042*.

Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 363–378, New York, NY, USA. Association for Computing Machinery.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 1(FSE).

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir

Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 10799–10808. PMLR.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pages 11941–11952. PMLR.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE Press.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

A Appendix: Detailed Tool Comparison Example

We illustrate the difference between AST-based and text-based code navigation tools using the SWE-bench instance `django__django-11211`. The task requires fixing Django's `GenericForeignKey.get_prefetch_queryset` method to correctly handle primary key type conversion.

A.1 Problem Description

When prefetching `GenericForeignKey` relations, the lookup fails because the stored `object_id` (a string) is compared against the model's primary key (an integer) without type conversion. The fix requires modifying the `gfk_key` function inside `get_prefetch_queryset` to use `to_python()` instead of `get_prep_value()`.

A.2 Comparison Summary

In Figure 3, the AST-based `-selector` flag of **CODESTRUCT** allows the agent to directly retrieve method bodies by their qualified name (e.g., `QuerySet.delete`), eliminating the need for manual `grep`/`sed` searches and line-range guessing. The `replace_node` operation targets methods semantically by name (`SQLDeleteCompiler.as_sql`), making edits robust to formatting variations and reducing the total steps by more than 50%. In contrast, the agent with text-based action space uses multiple inefficient grep and read loops for function location, leading to additional 19 steps for target location as shown in Table 3.

Metric	**CodeStruct**	**Text-Based**
Total Steps	24	54
Steps to Locate Target	2	21
Edit Success on First Try	Yes	Yes
Final Outcome	Success	Success

Table 3: Comparison on django__django-11211

B Ablation Study on SWE-Bench Verified

To isolate the contribution of individual components in CODESTRUCT, we perform an ablation study removing either the structure-aware read (`readCode`) or write (`editCode`) actions while keeping all other agent components fixed. This analysis reveals how each primitive contributes to both task accuracy and computational efficiency.

B.1 Complementary Roles of `readCode` and `editCode`.

Table 4 reveals that both structure-aware primitives contribute to CODESTRUCT's effectiveness, but in different ways.

Removing `readCode` causes the largest performance degradation: -7.8 Pass@1 for Qwen3-32B and -5.2 for GPT-5-mini. This degradation is accompanied by substantially higher input token usage (+41% for Qwen3-32B, +7.6% for GPT-5-mini) and more LLM calls (+44% for Qwen3-32B, +6.1% for GPT-5-mini), indicating inefficient exploration driven by repeated full-file reads and imprecise localization of relevant program elements. Without structured navigation, agents cannot efficiently narrow down the search space and instead resort to exhaustive file reading and trial-and-error execution.

Removing `editCode` yields smaller but consistent accuracy drops (-3.2 Pass@1 for Qwen3-32B, -1.4 for GPT-5-mini) while incurring cost penalties disproportionate to the performance loss. For GPT-5-mini, the w/o `editCode` configuration achieves only 60.6% Pass@1 (vs. 62.0% full) but requires 141.22$ in cost (vs. 101.83$ full)—a 38.7% cost increase for 1.4pp less accuracy. This suggests that without precise, structure-aware modifications, agents fall back to brittle string-based edits that require more iterations and validation cycles to achieve correct transformations. The small accuracy gap indicates that string edits can eventually succeed, but at significantly higher computational cost.

Notably, configurations without `readCode` underperform even the baseline text-based interface for Qwen3-32B (8.2% vs. 14.8%), highlighting that structured navigation—not just structured editing—is critical for scalable repository-level reasoning, especially for weaker models.

Configuration	Qwen3-32B					GPT-5-mini				
	Pass@1 (%)	Input (M)	Output (M)	Calls	Cost ($)	Pass@1 (%)	Input (M)	Output (M)	Calls	Cost ($)
Baseline	14.8	366.0	1.23	25,543	55.64	60.4	593.7	1.27	18,560	150.97
CODESTRUCT	**16.0**	**302.1**	**1.03**	**24,653**	**45.93**	**62.0**	**404.5**	**0.35**	**14,811**	**101.83**
w/o `editCode`	12.8	291.6	0.85	24,535	44.25	60.6	562.1	0.35	15,757	141.22
w/o `readCode`	8.2	423.4	1.78	35,670	64.58	56.8	435.2	0.33	15,715	109.46

Table 4: Ablation results with cost analysis on SWE-Bench Verified. Removing either `readCode` or `editCode` degrades performance, with `readCode` causing the largest accuracy drop while `editCode` removal causes the largest cost penalty relative to performance. Input/Output tokens shown in millions. Cost is computed using per-model API pricing, consistent with Table 1.

B.2 Why `readCode` causes the largest accuracy drop?

To understand why removing `readCode` causes the largest Pass@1 degradation, we analyze the subset of instances that are solved by the full system but fail without `readCode` (*regressed instances*).

For Qwen3-32B, removing `readCode` regresses 60/500 instances and induces a large behavioral shift: on regressed instances, the agent requires 2.52× more steps (29.3→74.0), 3.34× more tokens (260K→870K), and 2.51× more LLM calls (29.2→73.2). Moreover, the dominant failure mode becomes budget exhaustion: the clean-submit rate drops from 85% (51/60) to 22% (13/60), while context-exhaustion rises from 15% (9/60) to 75% (45/60).

This inefficiency is explained by an action-pattern collapse: without `readCode`, structured navigation actions disappear and the agent falls back to trial-and-error text manipulation, with `str_replace` increasing by 7.8× (314→2456) and `bash` by 1.9× (609→1143) over the same regressed set. The agent cannot efficiently localize relevant code and instead performs blind exploration through repeated full-file reads, string searches, and execution cycles.

For GPT-5-mini, removing `readCode` regresses 54 instances and increases token usage by 1.64× (817K→1338K), primarily through more trial-and-error execution (`bash` +1010), although budget exhaustion is rare (2%).

B.3 Why `editCode` Matters for Efficiency.

While the accuracy drop from removing `editCode` is smaller, the cost analysis reveals its importance for efficient code transformation. Without `editCode`, agents must rely on string-based `str_replace` operations that: (1) require exact string matching, leading to frequent failures from whitespace or formatting mismatches; (2) cannot verify syntactic correctness before application, resulting in broken code that requires additional repair cycles; and (3) lack scope awareness, making it difficult to perform precise transformations like replacing a specific function parameter or adding an import statement without affecting unrelated code.

These limitations manifest as higher iteration counts and validation cycles. Even when agents eventually succeed using string edits, they require more attempts, more execution traces to verify correctness, and more LLM calls to diagnose and repair malformed transformations. This explains why the cost penalty (38.7% for GPT-5-mini) substantially exceeds the accuracy loss (1.4pp).

B.4 Case Study: Complementary Tool Benefits (django__django-15368).

With both `readCode` and `editCode`, the agent efficiently solves the task in 37 steps. It first uses `readCode -selector bulk_update` to precisely localize the target function within the QuerySet class hierarchy, then applies a single scoped transformation via `editCode replace_node` that modifies only the relevant code block while preserving surrounding context.

Without `readCode`, the agent cannot efficiently localize the relevant code. It performs 145 steps of exhaustive file reading and string searches, eventually submitting a working patch only after reaching context limits—a 3.9× increase in exploration cost that demonstrates how structured navigation prevents expensive brute-force exploration even when such exploration can ultimately succeed.

Without `editCode`, the agent can still localize the correct function using structured navigation, but struggles with precise modification. It performs 62 steps (1.7× the baseline) with iterative `str_replace` attempts, encountering multiple failures from whitespace mismatches and scope errors before producing a working transformation through trial-and-error.

This case study demonstrates that `readCode` and `editCode` serve complementary roles: structured navigation enables efficient localization of relevant program elements, while structured editing enables precise, scope-aware transformations. Together, they minimize both exploration cost (fewer navigation steps) and transformation cost (fewer edit-validate cycles).

C Code Editing Error Analysis

To understand how **CODESTRUCT**'s structured interface affects operational reliability across different model capabilities, we analyzed error patterns in agent trajectories by counting occurrences of string replacement failures, edit operation errors, and malformed tool invocations in execution logs.

C.1 Methodology

We analyzed trajectory logs from all model configurations on SWE-Bench Verified, searching for error patterns indicative of failed edit operations:

- **String replacement errors**: Patterns matching `str_replace.*(error|fail)` indicating failed text-based edits, typically caused by (i) the target string occurring multiple times in the file (ambiguous replacement) or (ii) the target string not being found due to formatting or whitespace mismatches.
- **Edit operation errors**: Patterns matching `editCode.*(error|fail)` indicating failed AST-based edits, most commonly arising when the specified AST node selector cannot be resolved (e.g., no matching node name found).

Error counts were aggregated across all instances and normalized per instance to enable cross-model comparison. These patterns capture *recoverable, tool-level execution failures*, complementing the empty patch analysis which captures complete trajectory failures.

C.2 Results

Table 5 presents error rates across all evaluated models. The results reveal a clear capability threshold for effective AST-based editing.

Model	Approach	Total Errors	Errors/Instance	Reduction
GPT-5	Text-based	426	0.845	-
GPT-5	CODESTRUCT	52	0.103	**-87.8%**
GPT-5-mini	Text-based	568	1.125	-
GPT-5-mini	CODESTRUCT	127	0.252	**-77.6%**
GPT-5-nano	Text-based	459	0.911	-
GPT-5-nano	CODESTRUCT	551	1.093	**+20.0%**
Qwen3-Coder-480B	Text-based	458	0.909	-
Qwen3-Coder-480B	CODESTRUCT	109	0.216	**-76.2%**
Qwen3-32B	Text-based	6,556	13.008	-
Qwen3-32B	CODESTRUCT	5,155	10.228	-21.4%
Qwen3-8B	Text-based	7,179	14.247	-
Qwen3-8B	CODESTRUCT	6,031	11.966	-16.0%

Table 5: Error rates across models and interfaces. **CODESTRUCT** dramatically reduces errors for capable models but increases them for GPT-5-nano, identifying a capability threshold.

Overall, Table 5 shows that the impact of structured editing on tool-level error rates is strongly model-dependent. For capable models (GPT-5, GPT-5-mini, and Qwen3-Coder-480B), **CODESTRUCT** reduces errors per instance by 76–88%, indicating that AST-based operations are substantially more reliable than text-based string matching in this regime. In contrast, smaller models (GPT-5-nano, Qwen3-32B, and Qwen3-8B) exhibit higher absolute error rates, with GPT-5-nano showing a 20% increase in errors per instance despite improved task accuracy. This divergence highlights a capability threshold: while structured interfaces eliminate brittle text-level failures for sufficiently capable models, they impose additional syntactic and specification demands that smaller models struggle to satisfy consistently.

GPT-5-nano Analysis. GPT-5-nano exhibits a counterintuitive pattern: although **CODESTRUCT** increases operational error rates by 20%, it substantially improves Pass@1 accuracy (+20.8pp) and reduces empty patches (233→36, -84.5%). This is not a contradiction, but a redistribution of failure modes.

Concretely, structured navigation enables GPT-5-nano to reliably *localize* the correct code regions, increasing the number of attempted edits per trajectory. As a result, the model incurs more *recoverable tool-level errors* when expressing AST-based edits, but far fewer trajectories terminate early without producing any valid patch.

This shift manifests as:

- **More operational errors**: Increased failed edit attempts (551 vs. 459) due to difficulty producing well-formed AST operations.
- **Fewer early agent terminations**: A dramatic reduction in empty patches (-84.5%), indicating that the agent continues attempting fixes instead of abandoning the task.
- **Higher compute cost**: Additional retries explain increased token usage, rather than deeper or unnecessary exploration.
- **Higher final accuracy**: When a valid edit is eventually produced, it more often targets the correct location.

Overall, for GPT-5-nano, structured primitives improve high-level reasoning about *what* to change, while degrading low-level execution of *how* to express edits. The net effect favors accuracy over efficiency.

Qwen Model Scaling. The Qwen family demonstrates non-linear scaling effects:

- **Qwen3-8B**: Maintains extremely high error rates (11.966/instance) despite 16% reduction, explaining why empty patch improvements don't yield accuracy gains
- **Qwen3-32B**: Shows similar behavior (10.228/instance), suggesting fundamental reasoning limitations persist at this scale
- **Qwen3-Coder-480B**: Achieves near-GPT-5-mini performance (0.216 vs 0.252 errors/instance), validating that sufficient scale enables effective structured editing

Error Type Distribution. Analysis of error categories reveals different failure modes:

- **Text-based interface**: Errors dominated by string replacement failures (exact text matching issues, whitespace sensitivity)
- **CODESTRUCT**: Errors split between AST operation syntax issues and AST node name not found issues
- **No occurrence**: Neither "duplicated text" nor "no text found" patterns appear in logs, suggesting these specific failure modes are rare in SWE-Bench tasks

D Empty Patch Analysis

Empty Patch Analysis. To better understand agent behavior, we analyze *empty patches*, where SWE-Agent terminates without producing a valid code diff. Such terminations are typically triggered after repeated invalid edits, parse failures, or cyclic tool-use patterns, reflecting a failure to externalize an intended edit rather than incorrect high-level reasoning. Across models, **CODESTRUCT** substantially reduces this failure mode: GPT-5-mini drops from 35 to 6 empty patches, and GPT-5-nano from 233 to 36. Notably, GPT-5-nano's reduction correlates directly with its 20.8pp accuracy gain, suggesting these were instances where the model had correct intent but could not express valid edits through the text-based interface. In contrast, Qwen3-8B also reduces empty patches (179 to 138) but without corresponding accuracy improvement, indicating that its failures stem from reasoning limitations rather than interface brittleness. This divergence illustrates that **CODESTRUCT**'s benefits depend on the model's dominant failure mode: when failures arise from text-interface brittleness, structured actions unlock new solutions; when failures arise from limited reasoning capacity, structured actions improve efficiency without changing outcomes.

✓ AST-Based Approach

24 steps • Succeeded

Step 1: Read Method with Selector

```
> readCode django/db/models/query.py \
    --selector QuerySet.delete

Found 2 match(es) for 'QuerySet.delete':

Match 1: function delete in QuerySet (Line
     724)

def delete(self):
    """Delete␣the␣records␣in␣the␣current
␣␣␣␣QuerySet."""
    self._not_support_combined_queries('delete
         ')
    ...
```

Direct method retrieval using `--selector` flag.

Step 2: Read Another Method

```
> readCode django/db/models/sql/query.py \
    --selector get_initial_alias

Found 1 match(es) for 'get_initial_alias':

function: get_initial_alias in Query (Line
     916)

def get_initial_alias(self):
    """Return␣the␣first␣alias␣for␣this␣query""
         "
    ...
```

Step 3: Apply Fix with Semantic Edit

```
> editCode django/db/models/sql/compiler.py \
    replace_node SQLDeleteCompiler.as_sql \
    --replacement 'def␣as_sql(self):
␣␣␣␣␣␣␣␣"""Create␣the␣SQL␣for␣this␣query."""
␣␣␣␣␣␣␣␣if␣not␣self.query.where.children:
␣␣␣␣␣␣␣␣␣␣␣␣return␣self._as_sql(self.query)
␣␣␣␣␣␣␣␣if␣self.single_alias:
␣␣␣␣␣␣␣␣␣␣␣␣return␣self._as_sql(self.query)
␣␣␣␣␣␣␣␣...'

Successfully executed replace_node on
'SQLDeleteCompiler.as_sql'
```

✗ Text-Based Approach

54 steps • Succeeded

Steps 1–21: Navigate and Locate Code

```
> grep -R "def␣delete(" -n /testbed/django

> str_replace_editor view \
    /testbed/django/db/models/query.py \
    --view_range 1 400

> sed -n '700,940p' \
    /testbed/django/db/models/query.py

> grep -R "class␣DeleteQuery" -n /testbed

> grep -R "SQLDeleteCompiler" -n /testbed

> str_replace_editor view \
    /testbed/django/db/models/sql/compiler.py
         \
    --view_range 1360 1520
...
```

21 steps of manual navigation using grep, sed, and view commands.

Step 22: str_replace Attempt

```
> str_replace_editor str_replace \
    /testbed/django/.../compiler.py \
    --old_str 'class␣SQLDeleteCompiler(...):
␣␣␣␣@cached_property
␣␣␣␣def␣single_alias(self):
␣␣␣␣␣␣␣␣...'
```

Steps 23–54: Additional Navigation

The agent continues with additional `grep` searches, Python introspection commands, and multiple edit attempts to complete the fix.

Total additional steps: 32 more operations including:

- Multiple `grep` searches
- Python introspection
- Several edit attempts

Figure 3: Comparison of **CodeStruct** (AST-based) vs. text-based code editing approaches. **CodeStruct** completes in 24 steps vs. 54 steps for text-based—a 55.6% reduction.