ONE REPOSITORY, A THOUSAND GRAPHS: A REPOSITORY CONTEXT INTERFACE FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large codebases far exceed the context that large language models can effectively process. Encoding repository knowledge in model parameters is costly, lags behind changes, and is difficult to sustain through continual learning. On-demand search can provide uneven coverage or miss relevant information, while maintaining explicit code graphs outside the model for large repositories poses challenges for covering task-relevant relations and keeping them synchronized as the code evolves. We propose ORTG, a repository context interface centered on file-level entity semantics with a bounded set of relation cues. It organizes entity context and a partial set of prebuilt relations by module and provides relevant information on demand through a two-level structure under limited context, while delivering the complete outline when the budget permits. The editing model reuses its understanding of source code and current changes to update entity outlines and relation fields as code is edited. End-to-end evaluation covers SWE-bench Verified, Deep-SWE, and FeatureBench. In the evaluated configurations, complete ORTG achieves higher mean task success rates than the corresponding baselines and ablation configurations. Behavioral statistics on Deep-SWE show fewer interactions and file reads, but higher cumulative token usage including index construction and updates.

1 INTRODUCTION

Modern software repositories contain numerous files, functions, interfaces, configurations, tests, data structures, and engineering constraints, often exceeding the context that a large language model can effectively process in a single pass. More importantly, a repository is not a static document: commits, refactoring, interface migrations, and changing requirements continually alter its state. Repository-level code reasoning therefore faces more than a one-time long-document processing problem: it must determine *how to deliver continuously changing repository knowledge to the current reasoning process in time*. Even as physical context windows grow, positional degradation and information interference can weaken the use of key evidence (Liu et al., 2024a).

One approach encodes repository knowledge through training or fine-tuning, incurring substantial update costs and leaving parameterized knowledge behind the current code version. Another avoids training on the repository and lets an agent search and read local files when a task arises. Although flexible, this approach can overlook relevant information and cannot ensure that all necessary entities scattered across files and components enter the context together. RepoCoder, Repoformer, and Agentless improve code accessibility through iterative retrieval, selective retrieval, and structured localization, respectively (Zhang et al., 2023; Wu et al., 2024; Xia et al., 2025), but the amount of code reached remains constrained by the context budget and exploration path.

Explicit code graphs offer a third option: extracting call, inheritance, usage, and dependency relations outside the model and allowing the agent to query nodes and edges (Liu et al., 2025a; Ouyang et al., 2025; Chen et al., 2025). Task-relevant relations in large repositories include not only calls and inheritance, but also protocol agreements across modules, unit consistency, compatibility constraints, and HTTP/WebSocket communication relations established through dynamic addresses or message routing. As the number of entities grows and requirements evolve, identifying

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

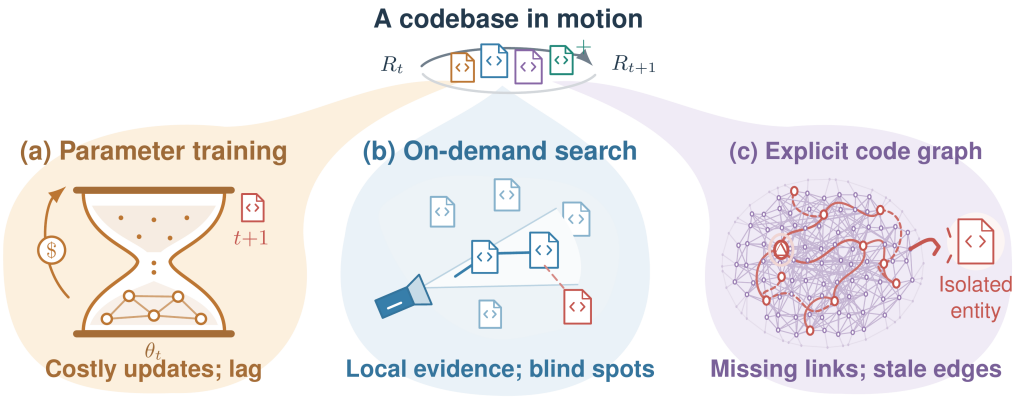

Figure 1: **Three dilemmas in evolving codebases.** The hourglass represents costly parameter updates and stale knowledge; files outside the search beam represent missed evidence; the dense network extending into its periphery illustrates the scale and interdependence of explicit graphs. Irregular red branches illustrate dispersed, intertwined relation checks after one entity changes; the isolated file and the severed connection toward it represent undetected relations, while dashed connections denote relations awaiting synchronization.

and maintaining these relations in advance presents challenges for relation coverage and synchronized updates: edges extracted solely from syntax and call structure may omit relations that require understanding entity semantics, runtime configuration, and task requirements. We investigate how file-level entity context and a post-edit update protocol can help the model access relevant files, inspect their constraints, and coordinate changes within the current task. Appendix E.1 provides an illustrative example involving a data protocol across modules.

Figure 1 summarizes the challenges these three approaches face in large, evolving repositories.

The study is also motivated by everyday repository maintenance: after code changes, repository outlines must be updated promptly, and relevant content must be selected for subsequent steps within the current task. These experiences inform the research question and interface design; controlled experiments below evaluate the integrated interface on independent coding tasks.

ORTG uses entity semantics to support task-time judgments about relationships, with a partial set of prebuilt relations as cues. The model combines file responsibilities, interfaces, constraints, and source code read on demand to inspect connections relevant to the current task; entry updates reuse the source and change context available to the same editing model.

To fit finite context, ORTG uses hierarchical outlines to localize responsibility domains and access entity details (Figure 2). Hierarchical localization has precedents (Xia et al., 2025); our contribution combines entity semantics organized by module, bounded relation cues, two-level on-demand delivery, and entry updates by the editing model into an integrated interface. We evaluate this combination through controlled comparisons on coding tasks.

We make three contributions:

- **ORTG: combining entity semantics and relation cues.** We organize file-level semantic outlines and a partial set of prebuilt relations by module, use full-outline delivery or two-level access according to context capacity, and integrate source inspection, code editing, and entry updates into the same model's workflow.
- **Experimental evaluation and main observations.** We evaluate ORTG on code repair and feature development, comparing ways to supply repository information to agents. In these experiments, providing entity descriptions (file functionality, interfaces, and constraints) without additionally listing relations between entities still yields higher mean task performance than using an agent without ORTG. When these relations are scrambled, mean performance is lower than when the additional relations are omitted. Navigation statistics also show fewer interactions and file reads alongside higher cumulative token usage.

- **A reusable ORTG implementation.** We implement file-level outlines and relation fields, repository-wide and module-level access, change detection, MCP/CLI interfaces, and host hooks, with documentation and automated tests. The source code and usage instructions are provided as supplementary material.

2 The Persistent Knowledge Gap in Large Codebases

Let R_t denote a repository at time t, containing an entity set $V(R_t)$. Development continually changes the repository, so $R_t \neq R_{t+1}$. A repository-level task q typically concerns only a subset of these entities, but the entities needed to complete it may be scattered across files, modules, and components. Let B denote the model's effective context budget. When a flat representation of the repository entities exceeds B, providing all information in a single pass is infeasible.

This knowledge gap creates a three-way dilemma (Figure 1):

1. **Encoding knowledge in model parameters:** expensive and prone to lag, with outdated repository states difficult to remove promptly and continual updates potentially causing catastrophic forgetting;
2. **On-demand search:** follows local clues incrementally and can overlook relevant information elsewhere;
3. **External explicit graphs:** require ongoing extraction, storage, updating, and validation of relations; when a repository contains a very large number of entities, updating and maintaining these relations becomes difficult, and new relations or changes to existing ones are easily overlooked.

We study the following question:

> **How can we provide coding agents with effective repository context on demand to support the current repository-level task, without repeatedly training on the entire codebase and while maintaining only a partial set of external relations?**

3 Related Work

Context utilization and repository adaptation. SWE-bench evaluates repository repair using real GitHub issues and executable tests (Jimenez et al., 2024). UTBoost automatically augments tests to improve the reliability of SWE-bench patch evaluation (Yu et al., 2025). Longer context increases code accessibility, but information in the middle can still be underused (Liu et al., 2024a). RepoCoder, Repoformer, and Agentless use iterative retrieval, selective retrieval, and hierarchical localization, respectively (Zhang et al., 2023; Wu et al., 2024; Xia et al., 2025). RLCoder learns snippet retrieval for code completion and does not query a persistent graph at inference, although an import-dependency graph is used to construct its training examples (Wang et al., 2025b). RepoExec studies how dependency context affects functional correctness and dependency utilization (Hai et al., 2025). Semantic-scope customization compares retrieval augmentation and supervised fine-tuning (Finkler et al., 2026), while SWE-Bench-CL evaluates experience accumulation and forgetting across task sequences (Joshi et al., 2025).

Repository semantic summaries and documentation. *Repository-Level Code Understanding* uses project-, directory-, and file-level summaries for top-down code search and bug localization (Oskooei et al., 2026). CodeWiki performs module decomposition and recursive documentation generation based on a code dependency graph (Hoang et al., 2026). RepoAgent combines the project hierarchy, classes/functions, and call-reference relations to generate documentation, and updates the documentation of affected objects through a Git pre-commit hook (Luo et al., 2024).

Explicit repository graphs outside the model. GraphCoder and LocAgent use code-context graphs and heterogeneous navigation graphs, respectively (Liu et al., 2024b; Chen et al., 2025). RepoGraph retrieves local subgraphs through definition, reference, and call relations (Ouyang et al.,

2025), while CodexGraph lets the agent construct graph-database queries to obtain code context (Liu et al., 2025a). RepoHyper searches and expands a repository semantic graph, then refines context through link prediction for code completion (Phan et al., 2025). GRACE, RepoScope, and CoCo organize context through multilayer graph fusion, call-chain prediction, and multigranular graph retrieval, respectively (Wang et al., 2025a; Liu et al., 2025b; Zhao et al., 2025); GRACE also explicitly adds edges through external cross-graph attention and thresholding. We compare SWE-agent and its CodexGraph- and RepoGraph-augmented versions under the same base model, with results in Table 2.

Repository planning and relation maintenance. RPG organizes functional planning and code structure into a unified graph representation, which ZeroRepo uses to generate repositories from scratch (Luo et al., 2026b). RPG-Encoder encodes existing source code into an RPG containing semantic descriptions and structural dependencies, supports node search, source-code reading, and graph traversal, and updates the graph incrementally based on commit differences (Luo et al., 2026a). RIG discusses consistency maintenance for large repository graphs (Cherny-Shahar & Yehudai, 2026); Codebase-Memory maintains graph freshness through file watching, reparsing, and node and edge updates (Vogel et al., 2026).

Post-edit update mechanisms. The authors' public manuscript of *Repository-Level Code Understanding* lists dynamic adaptation of the hierarchy as future work (Oskooei et al., 2026). The published CodeWiki paper identifies version-aware documentation that follows code evolution as a future direction (Hoang et al., 2026). RPG/ZeroRepo focuses on graph-guided generation, editing, and test validation (Luo et al., 2026b); RepoGraph proposes a dynamic version incorporating real-time execution and feedback as future work (Ouyang et al., 2025). CodexGraph reindexes based on version differences (Liu et al., 2025a); RPG-Encoder updates its graph representation incrementally based on commit differences (Luo et al., 2026a). ORTG's update protocol is triggered by file edits within the current task and instructs the editing model to submit the corresponding outlines and relation fields.

Our distinction. ORTG combines the delivery of entity semantics, bounded relation cues, and updates informed by the editing context through the following design choices. *(1) Repository-wide semantic delivery:* when context capacity permits, the semantic outlines of all repository entities are provided together, presenting file responsibilities and constraints for the model to inspect files whose source has not yet been read and consider potential impacts on them. *(2) Reserving context for entity semantics:* given the same entity outlines, the number of relation targets is bounded to control the context occupied by explicit relation fields and reserve budget for entity outlines. *(3) Module access beyond context capacity:* when the complete outline exceeds capacity, semantic information is organized and accessed by module in separate batches. *(4) Module encapsulation and interface conventions:* code-design requirements favor encapsulation within modules, separation of unrelated functionality, and collaboration through explicit interfaces as guidance for the model when editing code. *(5) Relation checks beyond prebuilt edges:* entity semantics can provide cues about protocols, units, and compatibility constraints, guiding task-specific checks of files that may be affected by an edit even when they are absent from the relation fields. *(6) Reusing the editing model's understanding:* the model performing the edits generates outline and relation-field updates from the source it has read, the current changes, and task context, reusing semantic information already available during editing. *(7) Prompting checks and corrections during source inspection:* in full-outline delivery mode, prompts instruct the model to automatically compare a file's source code with its outline when reading the file and to correct errors or stale descriptions it identifies. For interfaces that deliver only locally retrieved entries, if the corresponding outline is absent from the current context, the model must retrieve it before performing the same check. Together, these choices specify how repository context is delivered, checked, and updated within the current task. Appendix D provides detailed interface comparisons, and Appendix E.1 illustrates constraints across files.

4 ORTG: An Integrated Repository Context Interface

ORTG stores repository context in file-level entity entries. Let W be the model's context window limit and L the token count of the serialized complete outline. The system uses a configurable bud-

get fraction $\alpha \in (0, 1)$: when $L < \alpha W$, the outline is delivered in full ($V_q = V(R)$); otherwise, ORTG uses the two-level module access described below. The value of α can be adjusted according to the model's performance on long contexts and the specifics of the task. We use $\alpha = 0.5$ to reserve context space for task instructions, subsequent source-code reads, tool outputs, and interaction history. The interface requires updates to the corresponding outlines and relation fields to be submitted after code edits.

4.1 Entities and a Partial Set of Prebuilt Edges

The system extracts a set of entities from the codebase R:

$$V(R) = \{v_1, v_2, \ldots, v_n\}.$$

We use files as the unit of entity persistence, addressing, and updating, denoted by $v_i = (\mathrm{id}_i, x_i, m_i)$. x_i is a model-written file outline recording the file's functionality, key functions and classes, public interfaces, and relevant constraints; m_i stores addressing metadata such as path, type, and module or responsibility domain labels. The model combines the outlines with source code read on demand to handle implementation details, and our experiments evaluate this file-level interface. ORTG persistently stores entity outlines and a partial set of prebuilt edges, for example, strong dependencies between entities and relations that are subtle, easy to overlook, or difficult to identify during a task. The post-edit update workflow also covers these prebuilt edges. Each entry's relation field records the linked targets. Let $\mathcal{R}(v_i)$ denote the set of linked file entities; the prebuilt edge set is:

$$E^{\mathrm{rel}}_{\mathrm{input}} = \bigcup_i \{(v_i, u) \mid u \in \mathcal{R}(v_i)\}, \quad |\mathcal{R}(v_i)| \leq 5.$$

Entity outlines and relation fields are generated by the coding model from the global repository outline and source code; targets are ordered by the model's assessment of importance. Pre-task initialization uses repository information without access to the task to be solved. If outlines are missing at the start of an MCP conversation, the interface offers batch generation or deferred completion when the corresponding files are read. Only generated entries can be delivered; a complete outline contains an entry for every represented file. During task execution, the model generates a missing outline and its relation fields after reading the source. Appendix B.2 gives further construction details.

These relation fields are delivered as text within entity entries. The model can use linked targets to access further files without a graph-query or graph-traversal interface. Fields may record relations within a module or across modules; paths, directories, and module labels also provide structural cues. ORTG instructs the model to inspect and, as needed, update related entity outlines and relation fields after edits, without prebuilding a complete call, dependency, or inheritance graph.

4.2 Level 1: Global Module Localization

The first-level global outline compactly presents modules, responsibility domains, entry points, and principal entities to help the model locate the task's relevant modules. Modules organize entity context and delimit access scope; a task may involve multiple modules. Let the first-level outline be $I^{(1)}(R)$. The localization process is

$$\mathcal{D}_q = \mathrm{Locate}\Big(q, I^{(1)}(R)\Big),$$

where $\mathcal{D}_q$ is the set of modules or responsibility domains relevant to the task.

4.3 Level 2: File-Level Entity Access

After global module localization, the second-level outline $I^{(2)}(R)$ provides file-level entries within the relevant scope, including entity outlines, addressing metadata, and relation fields:

$$V_q = \mathrm{Select}\Big(q, \mathcal{D}_q, I^{(2)}(R)\Big) \subseteq V(R).$$

The agent selects modules or file paths through the `module` or `paths` parameters of `ortg_overview`; the program filters and returns the corresponding entries. This two-level structure supplies task-relevant information on demand, reducing the need to load all entity details at once. Context formed by one delivery of entities can be abstracted as

$$C_q = \mathrm{Serialize}(q, V_q),$$

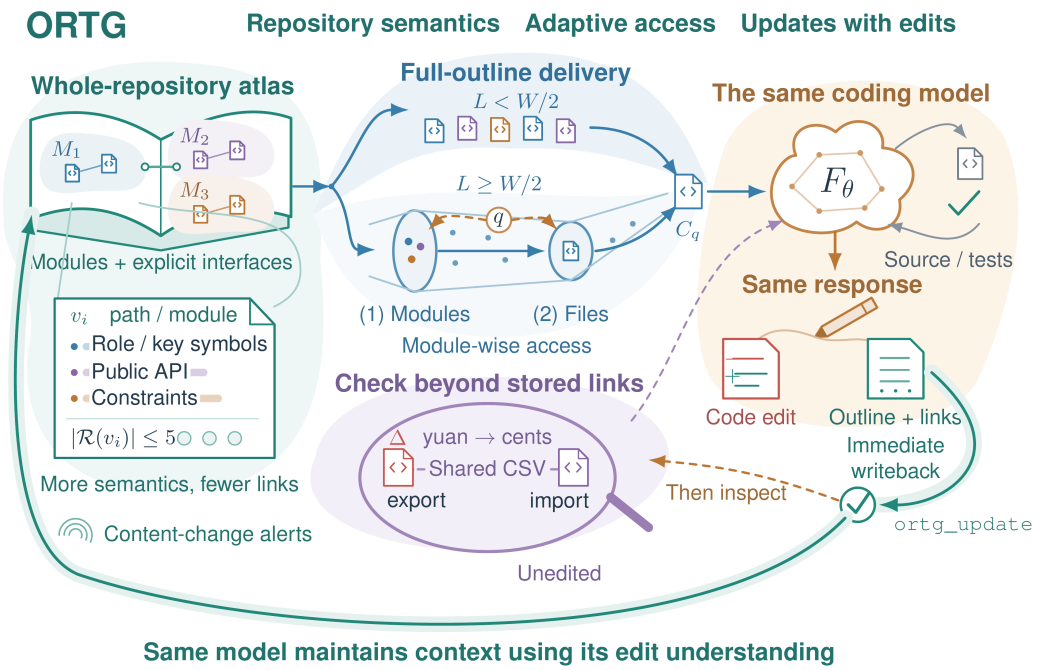

Figure 2: **ORTG: entity semantics, on-demand delivery, and post-edit updates.** The atlas organizes file entries by module. The enlarged entry preserves semantic detail and at most five relation targets, reserving more context space for entity semantics. Module encapsulation and explicit interfaces aim to limit maintenance scope. The figure uses our experimental setting $\alpha = 0.5$: the complete outline is delivered when $L < W/2$; otherwise, global module localization precedes file-entry access by module or path. The current coding model reads source on demand and revisits context using tool and test feedback. One pen depicts the protocol requiring updated outlines and relation fields to be generated in the same response as code edits and submitted immediately afterward. The prompts then ask the model to inspect potentially affected related files and update them as needed. The illustrative CSV unit migration includes an unedited consumer, extending inspection beyond prebuilt relations. Fingerprints flag content changes; semantic checking relies on the model. No external graph is queried at task time.

where Serialize serializes the task and selected entity entries, including relation fields. Locate and Select summarize the agent's module-localization and entity-selection decisions. The model reads source code on demand and repeats selection and access in response to tool and test feedback. C_q describes the context formed by entity delivery; subsequent interactions also include source code, tool outputs, and test feedback.

4.4 Task-Time Workflow

Figure 2 shows the access, editing, and update workflow during task execution. The model obtains the complete outline, or locates relevant modules and obtains file-level entries, then inspects source code on demand. The interface requires it to generate updated entity outlines and relation fields in the same response as a file edit and submit them through `ortg_update` immediately afterward. The prompts then instruct the model to use the global outline, source code, and change semantics to inspect potentially affected related files, modify code or update entries as needed, and continue selecting context and reasoning from tool and test feedback. Such checks may include files whose source is unchanged but whose interface conventions or relation descriptions may be affected.

The update tool validates and writes submitted entries without separately invoking a model to generate their content. File fingerprints detect source-content changes and prompt entry review. Semantic checks, including effects on files whose source is unchanged, rely on the editing model's source inspection and task feedback.

Table 1: State boundaries of prebuilt repository graph methods and our entity interface.

Dimension	Prebuilt repository graph methods	Our approach
State outside the model	Entities, prebuilt relations, and corresponding graph indices	Entity set $V(R)$, two-level outline, and a partial set of prebuilt edges
Source of edges	Static analysis or predefined relation extraction	A partial set of prebuilt edges recorded in relation fields
Task differences	Query task-relevant subgraphs	Deliver the full outline or select modules and file entries for the task
Repository updates	Updates to entities, edges, and graph indices	The editing model generates entry updates; tools write them; fingerprints detect content changes

Table 2: Resolution/success rates (%) ↑ on three benchmarks. Base agents: `mini-swe-agent` for Deep-SWE; `mini_swe_agent` for the qwen3.8-flash setting and Codex for the other FeatureBench setting.

SWE-bench Verified	
SWE-agent + CodexGraph	93.3 ± 2.4
SWE-agent + RepoGraph	94.1 ± 2.6
SWE-agent	90.5 ± 1.9
SWE-agent + single-level ORTG	94.2 ± 1.7
SWE-agent + ORTG	$\mathbf{95.6 \pm 2.1}$

Method	**Deep-SWE**	**FeatureBench**
Baseline	46.1 ± 2.6	19.2 ± 2.7
ORTG	$\mathbf{52.3 \pm 2.2}$	$\mathbf{26.6 \pm 1.8}$
Codex	—	25.2 ± 2.6
Codex + RPG-Encoder	—	29.8 ± 2.2
Codex + ORTG	—	$\mathbf{41.0 \pm 3.1}$

SWE-bench Verified and Deep-SWE use DeepSeek-V4-Flash. The FeatureBench Baseline/ORTG rows use qwen3.8-flash. All Codex experiments use GPT-5.6 Sol with high reasoning effort. Dashes indicate settings not evaluated. SWE-bench Verified reports means over five runs of 500 instances per method ± 95% CI half-widths; Deep-SWE reports means over 15 runs ± 95% CI half-widths; FeatureBench reports means over five runs per condition ± 95% CI half-widths. Single-level ORTG ablates the delivery component. Both graph-augmented comparisons on SWE-bench Verified use SWE-agent and the same base model; FeatureBench additionally includes Codex + RPG-Encoder.

The guidance for code design and modification is to encapsulate related functionality within modules where possible, minimize unnecessary cross-module dependencies, and use explicit interfaces for necessary cross-module collaboration.

Methods based on prebuilt repository graphs maintain entities and their explicit relations (V, E) and use the graph structure to obtain context at task time; our approach maintains V and a sparse, partial set of edges without querying a graph at task time. Table 1 provides a comparison.

5 Controlled Experiments

5.1 Experimental Setup

The evaluation covers SWE-bench Verified (Jimenez et al., 2024), Deep-SWE, and FeatureBench. The first two use DeepSeek-V4-Flash; FeatureBench evaluates qwen3.8-flash and Codex. On SWE-bench Verified, we compare SWE-agent, the single-level ORTG ablation, complete ORTG, SWE-agent + CodexGraph, and SWE-agent + RepoGraph. The other two benchmarks compare each setting with and without ORTG; FeatureBench additionally includes Codex + RPG-Encoder. Table 2 gives the models, run counts, and reporting conventions; Appendix B provides detailed experimental settings.

Controlled comparisons. On SWE-bench Verified, all compared methods share evaluation instances, base model, basic-tool and file-reading permissions, issue instructions, sampling settings, tool-call budgets, wall-clock limits, context limits, retry policies, and patch-validation procedures. The FeatureBench Codex ORTG ablations share the initial repository and instances, outline content,

source-code access, update schedule, sampling settings, context and tool budgets, retry policies, and evaluation procedures, except for the specified component changes.

5.2 Main Results and Ablation Analysis

End-to-end performance of the complete method. In Table 2, complete ORTG has a higher observed mean resolution rate than the SWE-agent baseline and its graph-augmented counterparts, supporting the practical value of the repository outline interface in this repair setting.

Ablation protocol. In the FeatureBench Codex experiments, entities only ($k = 0$) removes explicit relation fields while retaining descriptions, addressing metadata, hierarchical access, source access, and updates; descriptions and source may still contain relational cues. Random targets replace links with valid random files while preserving each entry's target count. Caps $k = 1$ and $k = 3$ retain prefixes of the same importance ordering. These transformations apply at every delivery, including after updates. Single-level delivery presents the same initial outlines and relations as a flat list under the same context budget, with fixed ordering and truncation rules. This tests the delivery configuration as a whole. Appendix B.4 gives additional details.

Relation-field and delivery ablations. The single-level delivery comparison examines the system-level effect of context organization (Table 2). The FeatureBench Codex experiments further compare relation fields, target caps, and delivery policies (Table 3): the entities-only condition ($k = 0$, with explicit relation fields removed) yields a higher mean than the Codex baseline, supporting the practical value of entity context without these fields. Random relation targets yield a lower mean than entities only. As the relation-target cap increases under a fixed importance ordering, the mean rises progressively, with $k = 3$ already close to the complete configuration. Complete ORTG also has a higher mean than single-level delivery. Appendix B.4 gives the detailed settings.

Table 3: FeatureBench Codex ablations.

Configuration	Success (%)
Codex	25.2 ± 2.6
Codex + ORTG (complete, $k = 5$)	$\mathbf{41.0 \pm 3.1}$
Entities only (no relation fields, $k = 0$)	36.4 ± 2.4
Random relation targets	33.8 ± 2.1
Edge cap $k = 1$	38.2 ± 2.8
Edge cap $k = 3$	40.6 ± 2.7
Single-level delivery	37.8 ± 2.0

Means over five runs $\pm$ 95% CI half-widths; the first two rows reproduce Table 2. Protocol: Section 5.2 and Appendix B.4.

SWE-bench Verified, Deep-SWE, and FeatureBench. ORTG achieves higher mean task success rates than the corresponding baselines across these benchmarks (Table 2). In the DeepSeek-V4-Flash experiments, ORTG solves some tasks the baseline did not solve, but also fails on some tasks that the baseline solved, with failures to follow task requirements. In the within-Codex comparison on FeatureBench, the base model and agent configuration are held constant, and the observed mean task success rate is higher with ORTG, supporting its effectiveness in this configuration. Future work can examine the relationship between model capability and the benefits of ORTG under a common agent and reasoning configuration.

Navigation efficiency and token usage. Table 4 reports the same Deep-SWE runs as the success-rate results. ORTG reduces interaction steps and file reads, but increases cumulative token usage including index construction and updates. These statistics quantify the trade-off between navigation operations and context costs; they do not directly establish lower wall-clock time or billed cost. Appendix B.5 gives the accounting details.

Table 4: Navigation efficiency and token usage on Deep-SWE with DeepSeek-V4-Flash.

Metric	Baseline	ORTG	Change
Mean steps/instance↓	164 ± 18	$\mathbf{87 \pm 15}$	-47%
File-read commands/instance↓	54 ± 9	$\mathbf{31 \pm 6}$	-43%
Distinct files/instance↓	34 ± 4	$\mathbf{13 \pm 4}$	-62%
Total tokens/instance↓	$(\mathbf{20.9 \pm 3.5})$M	(26.4 ± 5.1)M	$+26\%$

Means over 15 runs $\pm$ 95% CI half-widths; M denotes millions. Changes are rounded from the group means, not confidence intervals for differences. Tokens include index construction, updates after each file modification, and cache hits (approximately 99% hit rate).

5.3 A Behavioral Case of Entity Context Use

The two ORTG conditions on SWE-bench Verified share the same partial set of prebuilt edges and compare overall performance under different delivery configurations. The Sphinx case further illustrates how the model uses entity context for the current task: it reads an outline constructed before the task, follows supplied relations to shared field components, and incorporates build feedback to extend the repair to role-location and node-line utilities. This cross-file repair trajectory provides behavioral support for the use of supplied relation cues and the integration of subsequent source evidence. Appendix B.6 gives the configuration, trajectory, and evaluation outcomes.

6 Conclusion

We propose ORTG as an integrated interface combining entity semantics and bounded relation cues organized by module, two-level on-demand delivery, and updates generated by the editing model. In the evaluated coding tasks, the complete interface achieves higher mean success rates than the corresponding baselines, while component ablations compare relation fields, target counts, and delivery policies. The Sphinx case illustrates how the model uses supplied relations, source code, and feedback. Navigation statistics show that fewer interactions and file reads accompany higher cumulative token usage.

AI USE STATEMENT

No AI was used.

REPRODUCIBILITY STATEMENT

Runnable ORTG source code and usage instructions are provided in the supplementary attachment.

REFERENCES

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.426. URL https://aclanthology.org/2025.acl-long.426/.

Tsvi Cherny-Shahar and Amiram Yehudai. Repository intelligence graph: Deterministic architectural map for llm code assistants, 2026. URL https://arxiv.org/abs/2601.10112.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan,

Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Ulrich Finkler, Irene Manotas, Wei Zhang, Geert Janssen, Octavian Popescu, and Shyam Ramji. Automated customization of llms for enterprise code repositories using semantic scopes, 2026. URL https://arxiv.org/abs/2602.05780.

Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. On the impacts of contexts on repository-level code generation. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1496–1524, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.82. URL https://aclanthology.org/2025.findings-naacl.82/.

Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi D. Q. Bui. CodeWiki: Evaluating AI's ability to generate holistic documentation for large-scale codebases. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 5812–5827, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.288. URL https://aclanthology.org/2026.findings-acl.288/.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. Swe-bench-cl: Continual learning for coding agents, 2025. URL https://arxiv.org/abs/2507.00014.

Jia Li, Hongyi Deng, Yiran Zhang, Kechi Zhang, Tianqi Shao, Tiankuo Zhao, Weinan Wang, Zhi Jin, Ge Li, Yang Liu, Yingtao Fang, and Yihong Dong. Realbench: A repo-level code generation benchmark aligned with real-world software development practices, 2026. URL https://arxiv.org/abs/2604.22659.

Jingjing Liu, Silin Li, Zeming Liu, Zihao Cheng, Yuhang Guo, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. Towards multi-language repository-level code generation: From-scratch to guided tasks. *Neurocomputing*, 679:133204, May 2026a. ISSN 0925-2312. doi: 10.1016/j.neucom.2026.133204. URL http://dx.doi.org/10.1016/j.neucom.2026.133204.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024a. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Shuhan Liu, Zhiyi Zhao, Xing Hu, Kui Liu, Xiaohu Yang, and Xin Xia. Race-bench: A reasoning-augmented benchmark for repository-level code agents on feature addition, 2026b. URL https://arxiv.org/abs/2603.26337.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 570–581. ACM, October 2024b. doi: 10.1145/3691620.3695054. URL http://dx.doi.org/10.1145/3691620.3695054.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *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)*, pp. 142–160, Albuquerque, New Mexico, April 2025a. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.7. URL https://aclanthology.org/2025.naacl-long.7/.

Yang Liu, Li Zhang, Fang Liu, Zhuohang Wang, Donglin Wei, Zhishuo Yang, Kechi Zhang, Jia Li, and Lin Shi. Reposcope: Leveraging call chain-aware multi-view context for repository-level code generation, 2025b. URL https://arxiv.org/abs/2507.14791.

Jane Luo, Chengyu Yin, Xin Zhang, Qingtao Li, Steven Liu, Yiming Huang, Jie Wu, Hao Liu, Yangyu Huang, Yu Kang, Fangkai Yang, Ying Xin, and Scarlett Li. Closing the loop: Universal repository representation with rpg-encoder, 2026a. URL https://arxiv.org/abs/2602.02084.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. Rpg: A repository planning graph for unified and scalable codebase generation. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 91932–91985, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/9482f45fdd89aba9130bb04c44f788a9-Paper-Conference.pdf.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In Delia Irazu Hernandez Farias, Tom Hope, and Manling Li (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 436–464, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-demo.46. URL https://aclanthology.org/2024.emnlp-demo.46/.

Amirkia Rafiei Oskooei, Selcan Yukcu, Mehmet Cevheri Bozoglan, and Mehmet S. Aktas. Repository-level code understanding by llms via hierarchical summarization: Improving code search and bug localization. In Osvaldo Gervasi, Beniamino Murgante, Chiara Garau, Yeliz Karaca, Maria Noelia Faginas Lago, Francesco Scorza, and Ana Cristina Braga (eds.), *Computational Science and Its Applications – ICCSA 2025 Workshops*, pp. 88–105, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-031-97576-9.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 30098–30121, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4a4a3c197deac042461c677219efd36c-Paper-Conference.pdf.

Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. Repohyper: Search-expand-refine on semantic graphs for repository-level code completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pp. 14–25. IEEE, April 2025. doi: 10.1109/forge66646.2025.00009. URL http://dx.doi.org/10.1109/Forge66646.2025.00009.

Zhengyuan Shi, Jingxin Wang, Tairan Cheng, Changran Xu, Weikang Qian, and Qiang Xu. Cktevo: Repository-level rtl code benchmark for design evolution, 2026. URL https://arxiv.org/abs/2603.08718.

Martin Vogel, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. Codebase-memory: Tree-sitter-based knowledge graphs for llm code exploration via mcp, 2026. URL https://arxiv.org/abs/2603.27277.

Xingliang Wang, Baoyi Wang, Chen Zhi, Junxiao Han, Xinkui Zhao, Jianwei Yin, and Shuiguang Deng. Grace: Graph-guided repository-aware code completion through hierarchical code fusion, 2025a. URL https://arxiv.org/abs/2509.05980.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1140–1152. IEEE, April 2025b. doi: 10.1109/icse55347.2025.00014. URL http://dx.doi.org/10.1109/ICSE55347.2025.00014.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 53270–53290. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/wu24a.html.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, June 2025. ISSN 2994-970X. doi: 10.1145/3715754. URL http://dx.doi.org/10.1145/3715754.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-bench. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.189. URL https://aclanthology.org/2025.acl-long.189/.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

YuBo Zhang, Kaichun Yao, LiBo Zhang, and Chen Zhao. Repocbench: A benchmark for c-oriented repository-level code generation with large language models and agents. *Expert Systems with Applications*, 331:133242, December 2026. ISSN 0957-4174. doi: 10.1016/j.eswa.2026.133242. URL http://dx.doi.org/10.1016/j.eswa.2026.133242.

Xinkui Zhao, Rongkai Liu, Yifan Zhang, Chen Zhi, Lufei Zhang, Guanjie Cheng, Yueshen Xu, Shuiguang Deng, and Jianwei Yin. Completion by comprehension: Guiding code generation with multi-granularity understanding, 2025. URL https://arxiv.org/abs/2512.04538.

A Notation and State Boundaries

Table 5: Main notation used in this paper.

Symbol	Meaning
R_t	Repository state at time t
$V(R_t)$	Set of repository file entities stored outside the model
$v_i = (\mathrm{id}_i, x_i, m_i)$	File entity: identifier, outline, and addressing metadata
$I^{(1)}, I^{(2)}$	Global outline and file-level entity outlines
q	Current repository-level task
V_q	Subset of entities included in the context for the current task
m_i	Per-entity addressing metadata for entity v_i, which may carry weak structural cues
$\mathcal{R}(v_i), E^{\mathrm{rel}}_{\mathrm{input}}$	Linked targets (at most 5) and the prebuilt edge set

B Detailed Experimental Configuration and Efficiency Statistics

B.1 Benchmarks, Models, and Statistical Conventions

SWE-bench Verified and Deep-SWE use DeepSeek-V4-Flash (DeepSeek-AI et al., 2026). FeatureBench evaluates qwen3.8-flash and Codex.

SWE-bench Verified. The comparison includes SWE-agent, its single-level and complete ORTG variants, and its CodexGraph- and RepoGraph-augmented variants (Table 2). Each method uses five runs of 500 instances, with resolution rates reported as the arithmetic mean $\pm$ the 95% confidence interval half-width. Shared experimental conditions are specified in Section 5.1.

Deep-SWE and FeatureBench. Deep-SWE uses `mini-swe-agent` to compare the baseline with its ORTG-augmented variant over 15 runs, reporting the experimental mean and the half-width of the 95% confidence interval; R3 baseline and v3 in the efficiency records correspond to Baseline and ORTG, respectively. FeatureBench compares `mini_swe_agent` with qwen3.8-flash and Codex, each with and without ORTG, and additionally includes an RPG-Encoder comparison under the Codex setting, using five runs per condition. Pass rates are reported as means $\pm$ 95% confidence interval half-widths in Table 2. The instance lists, tool configurations, execution budgets, and evaluator versions follow the official evaluation settings of the respective benchmarks.

B.2 Additional ORTG Construction and Update Details

Generation and access. Sections 4.1 and 4.3 describe initialization, on-demand completion, and tool access. During construction, the model uses the global repository outline to interpret module responsibilities and file relationships before producing file descriptions from source; this context helps place each file within the repository.

Relation discovery and ordering. The model identifies candidate relations while interpreting the global outline and source code. Stored relations aim to reduce repeated searches for connections between files and highlight links that may be difficult to discover or easy to overlook. The model assesses target importance and records selected targets in importance order, with at most five targets per entity in the complete configuration. Target-cap ablations retain this ordering (Section 5.2).

Update checks and implementation. Section 4.4 specifies post-edit submission, fingerprint checks, and the model's semantic inspection. When code tests expose implementation problems, the prompts instruct the model to trace the relevant source and outlines and correct any description or relation errors it identifies. Generation and update costs are included in Table 4. Tool parameters, prompting rules, and response handling are detailed in the supplementary ORTG source code and usage instructions.

B.3 Controlled Differences on SWE-bench Verified

Table 6: The SWE-agent baseline and both ORTG conditions on SWE-bench Verified do not query an external graph. The two ORTG configurations share the same partial set of prebuilt edges and compare single-level with hierarchical delivery.

Condition	Entry point for repository information	Prebuilt entity relation edges
SWE-agent	Original search, read, and test workflow	None
SWE-agent + single-level ORTG (ablation)	Single-level, flat entity entry point	Partial
SWE-agent + ORTG (complete method)	Access to file-level outlines after global localization	Partial

B.4 FeatureBench Codex Component Ablations

Sections 5.1 and 5.2 specify the shared conditions and ablation definitions; Table 3 reports the results. If an entry has fewer than k targets, all available targets are retained. Random targets exclude self-links and duplicates, with the random seed fixed for each run. Single-level comparisons vary delivery configurations, not the budget fraction α. Results are means over five runs $\pm$ the 95% confidence interval half-width. Appendix B.2 describes relation generation and ordering.

B.5 Deep-SWE Navigation Efficiency and Token Accounting

Table 4 in the main text reports the navigation and token statistics from the same runs as the Deep-SWE success rates. R3 baseline and v3 in the efficiency records correspond to Baseline and ORTG, respectively. Total tokens per instance cover the entire task process, including index construction, index updates after every file modification, and cache hits; the table notes give the statistical conventions.

B.6 Sphinx: A Behavioral Case of Using Repository Relation Cues

We analyze a Sphinx C-domain instance from FeatureBench (`test_domain_c`). Both conditions use Codex, repository commit `e347e59c`, and the same container image, with Web search enabled and one run each. The ORTG condition uses version 0.3.35 and reads an outline generated from repository source analysis before the task. This case is reported separately; the main FeatureBench Codex experiments use five runs per condition.

The input outline already supplies cues connecting the C-domain to shared document-field components, including the calls among `CObject`, `DocFieldTransformer`, and `TypedField.make_xrefs`. The raw tool events confirm that the model read the outline, and its later repair explanation explicitly refers to this cue. The baseline treats the missing shared field method as outside the task scope. In the ORTG condition, the model combines source inspection with build feedback, implements `make_xrefs` in the shared base class `Field` for inheritance by `TypedField`, and subsequently restores `SphinxRole.get_location` and the shared utility `get_node_line`. The latter two helpers have no corresponding description in the pre-task outline; they are located through runtime errors and source inspection.

Table 7: A single paired Sphinx C-domain case. F2P and P2P report passed/total FAIL_TO_PASS and PASS_TO_PASS tests, respectively; resolution follows the instance-level evaluation.

Condition	Modified source files	F2P	P2P	Resolved
Codex	4	28/35	81/81	No
Codex + ORTG	8	34/35	81/81	No

Table 7 shows improved build-test outcomes while retaining the remaining macro-parsing failure. The record illustrates use of supplied relations and expansion of the cross-file implementation scope. It is a behavioral case from one pair of runs. The supplementary attachment provides this case's pre-task outline, original model responses, complete tool events, code patches, and per-test evaluation results in `featurebench/02-sphinx-domain-c/` for verification of the run and its results.

C Limitations and Future Work

The current results cover SWE-bench Verified and Deep-SWE with DeepSeek-V4-Flash, and FeatureBench with qwen3.8-flash and Codex. The additional FeatureBench setting broadens the evaluation, but model and agent configurations change together, preventing attribution to the base model alone; applicability across languages and additional task types remains to be tested.

Our evaluation unit is an independent coding task. We have not evaluated reuse of updated entries across sequences of commits or tasks, nor independently quantified outline semantic accuracy or relation-field precision and recall. The current token and navigation statistics do not constitute a full

lifecycle cost assessment; initialization, individual updates, and amortized costs across tasks require separate measurement.

Design-guided whole-repository generation in RealBench and multilingual repository generation in ReCode-bench offer evaluation settings beyond repair (Li et al., 2026; Liu et al., 2026a). Process evaluation for feature addition in RACE-Bench, C-oriented generation in RepoCBench, and RTL optimization in CktEvo could further broaden process and domain coverage (Liu et al., 2026b; Zhang et al., 2026; Shi et al., 2026).

D Comparison of Interface Designs with the Closest Approaches

Table 8 compares repository representations and task-time information processing in related work. Hierarchical summarization, RepoAgent, and RPG-Encoder already address hierarchical semantics, documentation updates, and incremental graph maintenance, respectively (Oskooei et al., 2026; Luo et al., 2024; 2026a). The comparison distinguishes repository-wide coverage in an external representation from simultaneously supplying the task model with the semantics of all files. For example, RPG-Encoder aggregates file-level semantics while constructing the representation, but uses node access at task time. The table compares the interface workflows described in these works; inferring relations that are not explicitly stored is not treated as a capability unique to ORTG.

Table 8: Repository representations and access mechanisms of the closest approaches. Because tasks and metrics differ, this table compares interface designs and does not rank performance.

Approach	Persistent state outside the model	Task-time information processing	ORTG interface design
Hierarchical summaries	Project-, directory-, and file-level semantic summaries	Select directories first, then read file summaries within the selected scope	Deliver complete file outlines when the budget permits; otherwise, access them by module
RepoAgent	Project hierarchy, class/function documentation, and reference relations	The Q&A prototype combines retrieved documentation, source code, and a directory tree; documentation is updated through pre-commit	The editing model checks files using change semantics and updates entries during the task
RPG-Encoder	An RPG combining semantic hierarchy and structural dependencies	Node search, reading, and graph traversal; incremental updates based on commit differences	Deliver file outlines directly and update the relevant outlines and relation fields based on inspection results
RLCoder	Import dependency graphs used to construct training samples; code snippets retrieved at inference time	Snippets selected for completion tasks after learning a retrieval policy	The current coding model generates and maintains file-level entity outlines and bounded relation fields
CodexGraph, RepoGraph, LocAgent	Explicit nodes, call/dependency edges, and corresponding graph stores or subgraphs	Queries, traversal, or task-subgraph selection on a prebuilt graph	Entries and relation fields are returned by module or file path, without querying a graph index at task time
GRACE	Prebuilt multilayer code graphs	External cross-graph attention fusion, with edges explicitly added through thresholding	Entity outlines and partial relation targets are serialized as text and delivered to the coding model by module or path
RepoScope	Prebuilt repository semantic structure graph (RSSG)	Task-relevant call chains predicted from the RSSG and serialized as context	Global module localization and file-level entity access provide repository context on demand
CoCo	Multigranular graphs comprising ASTs, CFGs, and cross-file dependencies	Graph-based ranking and serialization of structural context	File-level entries are selected by module or path; affected outlines and relation fields are updated after code changes
This work	**File-level entity outlines and partial relation fields (at most 5 target entities per entity)**	**Complete outlines when the budget permits; otherwise, access by module, with source code read on demand**	**The editing model generates outline and relation updates; tools validate and write them; file fingerprints prompt review**

E Relation Storage and Update Procedures

Explicit graph systems maintain relations as persistent repository state, involving schema extension, edge freshness, incremental updates, and graph consistency. ORTG stores entity outlines and partial relation fields as entries, accesses them on demand by module or path, and accepts entry updates submitted by the model after edits. The comparison here concerns the organization of persistent

state and its update interface:

Explicit graph:	E_t persists with the repository and is continuously maintained;
ORTG:	$V(R_t), E^{\text{rel}}_{\text{input}}$ are stored as entity entries; accessed by module or path, with updates submitted after edits.

E.1 An Illustrative Example: Semantic Relations Across a Data Protocol

Consider an illustrative scenario of related-file inspection: `export_orders.py` and `import_orders.py` exchange order data through CSV files, run from independent entry points, and do not call each other directly. Their format specifies that the `amount` field is measured in yuan. The task requires the new format to use fen instead while retaining support for reading historical files. Completing this task requires coordinating export conversion, import interpretation, format-version identification, and the corresponding test data and documentation.

The files share a data protocol between producer and consumer, together with constraints linking the unit change to historical compatibility. An ordinary abstract syntax tree can represent syntax within each file, but call and inheritance edges alone do not directly yield these semantic constraints across modules. An explicit graph can express these connections by adding protocol, unit, and compatibility relations. Relation coverage depends on identifying the relevant entities and their shared conventions, and synchronizing updates after version changes.

In ORTG's design, entity outlines can store component responsibilities and format conventions, and identified non-obvious connections can also be stored as a partial set of prebuilt edges. For the unit-migration task, the model can combine this information with source code to inspect affected files and coordinate the required changes. For example, after the exporter switches to fen, the importer should be included in the inspection because of the shared amount protocol, even if it has not yet been modified, with its unit interpretation, historical compatibility handling, and corresponding outline adjusted as needed; this connection need not be represented in advance by a direct call edge between the two files. If the current context contains only the exporter, the model may not yet have considered the consumer's convention for interpreting units; entity outlines provide cues for selecting files to inspect. This example illustrates the requirement to inspect unmodified files when interface conventions change.