CODEXGRAPH: Bridging Large Language Models and Code Repositories via Code Graph Databases

Anonymous ACL submission

Abstract

Large Language Models (LLMs) excel in stand-alone code tasks like HumanEval and MBPP, but struggle with handling entire code repositories. This challenge has prompted research on enhancing LLM-codebase interaction at a repository scale. Current solutions rely on similarity-based retrieval or manual tools and APIs, each with notable drawbacks. Similarity-based retrieval often has low recall in complex tasks, while manual tools and APIs are typically task-specific and require expert knowledge, reducing their generalizability across diverse code tasks and realworld applications. To mitigate these limitations, we introduce CODEXGRAPH, a system that integrates LLM agents with graph database interfaces extracted from code repositories. By leveraging the structural properties of graph databases and the flexibility of the graph query language, CODEX-GRAPH enables the LLM agent to construct and execute queries, allowing for precise, code structure-aware context retrieval and code nav-We assess CODEXGRAPH using igation. three benchmarks: CrossCodeEval, SWEbench, and EvoCodeBench. Additionally, we develop five real-world coding applications. With a unified graph database schema, CODEXGRAPH demonstrates competitive performance and potential in both academic and real-world environments, showcasing its versatility and efficacy in software engineering. Our code and demo will be released soon.

1 Introduction

004

011

014

017

027

035

037

045

Large Language Models (LLMs) excel in code tasks, impacting automated software engineering (Chen et al., 2021; Gauthier, 2024; Yang et al., 2024b; Open-Devin Team, 2024). Repository-level tasks (Zhang et al., 2023; Jimenez et al., 2023; Ding et al., 2024) mimic software engineers' work with large codebases (Kovrigin et al., 2024). These tasks require models to handle intricate dependencies and comprehend project structure (Jiang et al., 2024; Sun et al., 2024).

Current LLMs struggle with long-context inputs, limiting their effectiveness with large codebases (Jimenez et al., 2023) and lengthy sequences reasoning (Liu et al., 2024a). Researchers have proposed methods to enhance LLMs by retrieving task-relevant code snippets and structures, improving performance in complex software development (Deng et al., 2024; Arora et al., 2024; Ma et al., 2024). However, these approaches mainly rely on either similarity-based retrieval (Jimenez et al., 2023; Cheng et al., 2024; Liu et al., 2024b) or manual tools and APIs (Zhang et al., 2024b; Örwall, 2024). Similarity-based retrieval methods, common in Retrieval-Augmented Generation (RAG) systems (Lewis et al., 2020), often struggle with complex reasoning for query formulation (Jimenez et al., 2023) and handling intricate code structures (Phan et al., 2024), leading to low recall rates. Meanwhile, existing tool/API-based interfaces that connect codebases and LLMs are typically task-specific and require extensive expert knowledge (Örwall, 2024; Chen et al., 2024). Furthermore, our experimental results in Section 5 indicate that the two selected methods lack flexibility and generalizability for diverse repository-level code tasks.

047

051

052

060

061

062

063

064

065

066

067

068

069

070

071

072

073

074

075

081

087

090

091

093

Recent studies have demonstrated the effectiveness of graph structures in code repositories (Phan et al., 2024; Cheng et al., 2024). Meanwhile, inspired by recent advances in graph-based RAG (Edge et al., 2024; Liu et al., 2024b; He et al., 2024) and the application of executable code (such as SQL, Cypher, and Python) to consolidate LLM agent actions (Wang et al., 2024; Li et al., 2024c; Xue et al., 2023), we present CODEXGRAPH, as shown in Figure 1 (a). CODEXGRAPH alleviates the limitations of existing approaches by bridging code repositories with LLMs through graph databases. CODEXGRAPH utilizes static analysis to extract code graphs from repositories using a task-agnostic schema that defines the nodes and edges within the code graphs. In these graphs, nodes represent source code symbols such as MODULE, CLASS, and FUNCTION, and each node is enriched with relevant meta-information. The edges between nodes represent the relationships among these symbols, such as CONTAINS, INHERITS, and USES (see Figure 2 for an illustrative example). By leveraging the structural properties of graph databases, CODEXGRAPH enhances the LLM agent's comprehension of code structures. CODEXGRAPH leverages repository code information and graph structures for global analysis and multi-hop reasoning, enhancing code task perfor-



Figure 1: (a) Using a unified schema, CODEXGRAPH employs code graph databases as interfaces that allow LLM agents to interact seamlessly with code repositories. (b) CODEXGRAPH supports the management of a wide range of tasks, from academic-level code benchmarks to real-world software engineering applications.

mance. When users provide code-related inputs, the LLM agent analyzes the required information from the code graphs, constructs flexible queries using graph query language, and locates relevant nodes or edges. This enables precise and efficient retrieval, allowing for effective scaling to larger repository tasks.

To evaluate the effectiveness of the CODEXGRAPH, we assess its performance across three challenging and representative repository-level benchmarks: Cross-CodeEval (Ding et al., 2024), SWE-bench (Yang et al., 2024b) and EvoCodeBench (Li et al., 2024b). Our experimental results demonstrate that, by leveraging a unified graph database schema (Section 3.1) and a simple workflow design (Section 3.2), the CODEX-GRAPH achieves competitive performance across all academic benchmarks, especially when equipped with more advanced LLMs. Furthermore, as illustrated in Figure 1 (b), to address real-world software development needs, we extend CODEXGRAPH to the featurerich ModelScope-Agent (Li et al., 2023) framework. Section 6 highlights five real-world application scenarios, including code debugging and writing code comments, showcasing the versatility and efficacy of CODEXGRAPH in practical software engineering tasks.

Our contributions are from three perspectives:

- Pioneering code retrieval system: We introduce CODEXGRAPH, integrating code repositories with LLMs via graph databases for enhanced code navigation and understanding.
- **Benchmark performance:** We demonstrate CODEXGRAPH's competitive performance on three challenging and representative repository-level code benchmarks.
- **Practical applications:** We showcase CODEX-GRAPH's versatility in five real-world software engineering scenarios, proving its value beyond academic settings.

Related Work

2.1 Repository-Level Code Tasks

Repository-level code tasks have garnered significant attention due to their alignment with real-world production environments (Bairi et al., 2023; Luo et al., 2024; Cognition Labs, 2024; Kovrigin et al., Unlike traditional standalone code-related 2024). tasks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which often fail to capture the complexities of real-world software engineering, repository-level tasks necessitate models to understand cross-file code structures and perform intricate reasoning (Liu et al., 2024b; Ma et al., 2024; Sun et al., 2024). These sophisticated tasks can be broadly classified into two lines of work based on their inputs and outputs. The first line of work involves natural language to code repository tasks, exemplified by benchmarks like DevBench (Li et al., 2024a) and SketchEval (Zan et al., 2024), where models generate an entire code repository from scratch based on a natural language description of input requirements. State-of-the-art solutions in this area often employ multi-agent frameworks such as ChatDev (Qian et al., 2023) and MetaGPT (Hong et al., 2023) to handle the complex process of generating a complete codebase. The second line of work, which our research focuses on, includes tasks that integrate both a natural language description and a reference code repository, requiring models to perform tasks like repository-level code completion (Zhang et al., 2023; Shrivastava et al., 2023; Liu et al., 2023; Ding et al., 2024; Su et al., 2024), automatic GitHub issue resolution (Jimenez et al., 2023), and repository-level code generation (Li et al., 2024b). To assess the versatility and effectiveness of our proposed system CODEXGRAPH, we evaluate it on three diverse and representative benchmarks including CrossCodeEval (Ding et al., 2024) for code completion, SWE-bench

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

159

160

161

165

167

123

124

125

126

127

130

276

277

224

(Jimenez et al., 2023) for Github issue resolution, and
EvoCodeBench (Li et al., 2024b) for code generation.

170

171

172

173

174

175

176

177

178

179

181

189

191

192

193

195

196

197

199

207

210

211

212

213

214

215

216

217

219

2.2 Retrieval-Augmented Code Generation

Retrieval-Augmented Generation (RAG) systems primarily aim to retrieve relevant content from external knowledge bases to address a given question, thereby maintaining context efficiency while reducing hallucinations in private domains (Lewis et al., 2020; Shuster et al., 2021). For repository-level code tasks, which involve retrieving and manipulating code from repositories with complex dependencies, RAG systems-referred to here as Retrieval-Augmented Code Generation (RACG) (Jiang et al., 2024)-are utilized to fetch the necessary code snippets or code structures from the specialized knowledge base of code repositories. Current RACG methodologies can be divided into three main paradigms: the first paradigm involves similarity-based retrieval, which encompasses termbased sparse retrievers (Robertson and Zaragoza, 2009; Jimenez et al., 2023) and embedding-based dense retrievers (Guo et al., 2022; Zhang et al., 2023), with advanced approaches integrating structured information into the retrieval process (Phan et al., 2024; Cheng et al., 2024; Liu et al., 2024b). The second paradigm consists of manually designed code-specific tools or APIs that rely on expert knowledge to create interfaces for LLMs to interact with code repositories for specific tasks (Zhang et al., 2024b; Deshpande et al., 2024; Arora et al., 2024). The third paradigm combines both similarity-based retrieval and code-specific tools or APIs (Örwall, 2024), leveraging the reasoning capabilities of LLMs to enhance context retrieval from code repositories. Apart from the three paradigms, Agentless (Xia et al., 2024) preprocesses the code repository's structure and file skeleton, allowing the LLMs to interact with the source code. Our proposed framework, CODEXGRAPH, aligns most closely with the second paradigm but distinguishes itself by discarding the need for expert knowledge and task-specific designs. By using code graph databases as flexible and universal interfaces, which also structurally store information to facilitate the code structure understanding of LLMs, CODEXGRAPH can navigate the code repositories and manage multiple repository-level code tasks, providing a versatile and powerful solution for RACG.

3 CODEXGRAPH: Enable LLMs to Navigate the Code Repository

CODEXGRAPH is a system that bridges code repositories and large language models (LLMs) through code graph database interfaces. It indexes input code repositories using static analysis, storing code symbols and relationships as nodes and edges in a graph database according to a predefined schema. When presented with a coding question, CODEXGRAPH leverages the LLM agent to generate graph queries, which are executed to retrieve relevant code fragments or code structures from the database. The detailed processes of constructing the code graph database and the LLM agent's interactions with it are explained in sections 3.1 and 3.2, respectively.

3.1 Build Graph Databases from Code Repositories

Schema. We abstract code repositories into code graphs where nodes represent symbols in the source code, and edges represent relationships between these symbols. The schema defines the types of nodes and edges, directly determining how code graphs are stored in the graph database. Different programming languages typically require different schemas based on their characteristics. In our project, we focus on Python and have empirically designed a schema tailored to its features, with node types including MODULE, CLASS, METHOD, FUNCTION, FIELD, and GLOBAL_VARIABLE, and edge types including CONTAINS, INHERITS, HAS_METHOD, HAS_FIELD, and USES.

Each node type has corresponding attributes to represent its meta-information. For instance, METHOD nodes have attributes such as name, file_path, class, code, and signature. For storage efficiency, nodes with a code attribute do not store the code snippet directly in the graph database but rather an index pointing to the corresponding code fragment. Figure 2 illustrates a sample code graph derived from our schema, and Appendix A.1 shows the details of the schema.

Phase 1: Shallow indexing. The code graph database construction process consists of two phases, beginning with the input of the code repository and schema. The first phase employs a shallow indexing method, inspired by Sourcetrail's static analysis process¹, to perform a single-pass scan of the entire repository. During this scan, symbols and relationships are extracted from each Python file, processed only once, and stored as nodes and edges in the graph database. Concurrently, meta-information for these elements is recorded. This approach ensures speed and efficiency, capturing all nodes and their meta-information in one pass. However, the shallow indexing phase has limitations due to its single-pass nature. Some important edges, particularly certain INHERITS and CONTAINS relationships, may be overlooked as they might require context from multiple files.

Phase 2: Complete the edges. The second phase addresses the limitations of shallow indexing by focusing on cross-file relationships. We employ Depth-First Search (DFS) to traverse each code file, using abstract syntax tree parsing to identify modules and classes. This approach is particularly effective in resolving Python's re-export issues. We convert relative imports

https://github.com/CoatiSoftware/ Sourcetrail



Figure 2: Illustration of the process for indexing source code to generate a code graph based on the given graph database schema. Subfigure (3) provides a visualization example of the resultant code graph in Neo4j.

to absolute imports, enabling accurate establishment of cross-file CONTAINS relationships through graph queries. Simultaneously, we record INHERITS relationships for each class. For complex cases like multiple inheritance, DFS is used to establish edges for inherited FIELD and METHOD nodes within the graph database. This comprehensive approach ensures accurate capture of both intra-file and cross-file relationships, providing a complete representation of the codebase structure.

278

281

282

290

291

293

294

295

297

305

307

Summary. Our code graph database design offers four key advantages for subsequent use. *First*, it ensures efficient storage by storing code snippets as indexed references rather than directly in the graph database. *Second*, it enables multi-granularity searches, from module-level to variable-level, accommodating diverse analytical needs. *Third*, it facilitates topological analysis of the codebase, revealing crucial insights into hierarchical and dependency structures. *Last*, this schema design supports multiple tasks without requiring modifications, demonstrating its versatility and general applicability. These features collectively enhance the system's capability to handle complex code analysis tasks effectively across various scenarios.

3.2 Large Language Models Interaction with Code Graph Database

Code structure-aware search. CODEXGRAPH leverages the flexibility of graph query language to construct complex and composite search conditions. By combining this flexibility with the structural properties of graph databases, the LLM agent can effectively navigate through various nodes and edges in the code graph. This capability allows for intricate queries such as: *"Find classes under a certain module that contain a specific method"*, or *"Retrieve the module where a certain class is defined, along with the functions it contains"*. This approach enables code structure-aware searches, providing a level of code retrieval that is difficult to achieve with similarity-based retrieval methods (Robertson and Zaragoza, 2009; Guo et al., 2022) or conventional code-specific tools and APIs (Zhang et al., 2024b; Deshpande et al., 2024).

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

329

330

331

332

333

334

335

336

339

340

341

342

Write then translate. LLMs power LLM agents and operate based on user-provided prompts to break down tasks, utilize tools, and perform reasoning. This design is effective for handling specific, focused tasks (Gupta and Kembhavi, 2022; Yuan et al., 2024), but when tasks are complex and multifaceted, LLM agents may underperform. This limitation has led to the development of multi-agent systems (Hong et al., 2023; Qian et al., 2023; Guo et al., 2024), where multiple LLM agents independently handle parts of the task. Inspired by this approach, CODEXGRAPH implements a streamlined "write then translate" strategy to optimize LLM-database interactions.

As illustrated in Figure 3, the primary LLM agent focuses on understanding context and generating natural language queries based on the user's question. These queries are then passed to a specialized translation LLM agent, which converts them into formal graph queries. This division of labor allows the primary LLM agent to concentrate on high-level reasoning while ensuring syntactically correct and optimized graph queries. By separating these tasks, CODEX-



Figure 3: The primary LLM agent analyzes the given code question, writting natural language queries. These queries are then processed by the translation LLM agent, which **translates** them into executable graph queries.

GRAPH enhances query success rates and improves the system's ability to accurately retrieve relevant code information.

Iterative pipeline. Instead of completing the code task in a single step, CODEXGRAPH employs an iterative pipeline for interactions between LLM agents and code graph databases, drawing insights from existing agent systems (Yao et al., 2023; Yang et al., 2024b). In each round, LLM agents formulate multiple queries based on the user's question and previously gathered information. Similar to (Madaan et al., 2023), the agent then analyzes the aggregated results to determine whether sufficient context has been acquired or if additional rounds are necessary. This iterative approach fully leverages the reasoning capabilities of the LLM agent, thereby enhancing problem-solving accuracy.

4 Experimental Setting

344

347

351

367

371

376

377

Benchmarks. We employ three diverse repository-level code benchmarks to evaluate CODEXGRAPH: CrossCodeEval (Ding et al., 2024), SWE-bench (Yang et al., 2024b), and EvoCodeBench (Li et al., 2024b). CrossCodeEval is a multilingual scope cross-file completion dataset for Python, Java, TypeScript, and C#. SWE-bench evaluates a model's ability to solve GitHub issues with 2, 294 Issue-Pull Request pairs from 12 Python repositories. EvoCodeBench is an evolutionary code generation benchmark with comprehensive annotations and evaluation metrics.

We report our primary results on the CrossCodeEval Lite (Python) and SWE-bench Lite test sets for Cross-CodeEval and SWE-bench, respectively, and on the full test set for EvoCodeBench. CrossCodeEval Lite (Python) and SWE-bench Lite represent subsets of their respective datasets. CrossCodeEval Lite (Python) consists of 1000 randomly sampled Python instances, while SWE-bench Lite includes 300 instances randomly sampled after filtering out those with poor issue descriptions.

378

379

380

381

382

383

384

387

390

391

392

393

394

396

397

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

Remark: During indexing of 43 Sympy samples from the SWE-bench dataset, we face out-of-memory issues due to numerous files and complex dependencies, leading to their exclusion. Similarly, some EvoCodeBench samples are omitted due to test environment configuration issues. Thus, SWE-bench Lite and EvoCodeBench results are based on 257 and 212 samples, respectively.

Baselines. We evaluate whether CODEXGRAPH is a powerful solution for Retrieval-Augmented Code Generation (RACG) (Jiang et al., 2024). We specifically assess how effectively code graph database interfaces aid LLMs in understanding code repositories, particularly when handling diverse code questions across different benchmarks to test CODEXGRAPH 's general applicability. To achieve this, we select resilient RACG baselines that can be adapted to various tasks. Based on the categories in Section 2.2, we choose BM25 (Robertson and Zaragoza, 2009) and AUTOCODEROVER (Zhang et al., 2024b), which are widely recognized in code tasks (Jimenez et al., 2023; Ding et al., 2024; Kovrigin et al., 2024; Chen et al., 2024), along with a No-RAG method. Besides, since our work focuses on RACG methods and their generalizability, we exclude methods that interact with external websites (OpenDevin Team, 2024; Zhang et al., 2024a) and runtime environments (Yang et al., 2024b), as well as task-specific methods that are not easily adaptable across multiple benchmarks (Cheng et al., 2024; Orwall, 2024). These methods fall outside the scope of our project.

Especially, although (Zhang et al., 2024b) evaluate AUTOCODEROVER exclusively on SWE-bench, we extend its implementation to CrossCodeEval and EvoCodeBench, while retaining its core set of 7 codespecific tools for code retrieval.

Table 1: Performance comparison of CODEXGRAPH and RACG baselines across three benchmarks using various LLMs. The absence of values in SWE-bench Lite for the NO RAG method is due to issues with mismatches between the dataset and the code when running inference scripts². Similarly, the missing values in EvoCodeBench are attributable to task inputs being unsuitable for constructing the required BM25 queries, and the original paper also does not provide the corresponding implementation. Notably, the two agent-based methods, AUTOCODEROVER and CODEXGRAPH, perform poorly when equipped with Qwen2-72b-instruct. Appendix A.4 provides a detailed explanation for this. The best results for each metric are **bolded**.

Model	Method	CrossCodeEval Lite (Python)			SWE-bench Lite EvoCodeBench		deBench	
		EM	ES	ID-EM	ID-F1	Pass@1	Pass@1	Recall@1
	No RAG	8.20	46.16	13.0	36.92	-	19.34	11.34
Owen?	BM25	15.50	51.74	22.60	45.44	0.00	-	-
Qwell2	AUTOCODEROVER	5.21	47.63	10.16	36.54	9.34	16.91	7.86
	CodexGraph	5.00	47.99	9.10	36.44	1.95	14.62	8.60
	No RAG	11.70	60.73	16.90	47.85	-	25.47	11.04
DS Coder	BM25	21.90	67.52	30.60	59.04	1.17	-	-
DS-Couel	AUTOCODEROVER	14.90	59.78	22.30	51.34	15.56	20.28	7.56
	CodexGraph	20.20	63.14	28.10	54.88	12.06	27.62	12.01
	No RAG	10.80	59.36	16.70	48.22	-	27.83	11.79
CDT 4a	BM25	21.20	66.18	30.20	58.71	3.11	-	-
GP1-40	AUTOCODEROVER	21.20	61.92	28.10	54.81	22.96	28.78	11.17
	CodexGraph	27.90	67.98	35.60	61.08	22.96	36.02	11.87

Large Language Models (LLMs). We evaluate CODEXGRAPH on three advanced LLMs with long text processing, tool use, and code generation capabilities: GPT-40, DeepSeek-Coder-V2 (Zhu et al., 2024), and Qwen2-72b-Instruct (Yang et al., 2024a).

- **GPT-40**: Developed by OpenAI³, this model excels in commonsense reasoning, mathematics, and code, and is among the top-performing models as of July 2024⁴.
- DeepSeek-Coder-V2 (DS-Coder): A specialized code-specific LLM by DeepSeek ⁵ with 236B parameters, it retains general capabilities while being highly proficient in code-related tasks.
- Qwen2-72b-Instruct (Qwen2): Developed by Alibaba ⁶, this open-source model has about 72 billion parameters and a 128k long context, making it suitable for evaluating existing methods.

For the hyperparameters of the selected large language models, we empirically set the temperature coefficient to 0.0 for both GPT-40 and Qwen2-72b-Instruct, and to 1.0 for DeepSeek-Coder-V2. All other parameters are kept at their default settings.

Metrics. In metrics selection, we follow the original papers' settings (Jimenez et al., 2023; Ding et al., 2024; Li et al., 2024b). Specifically, for CrossCodeEval, we measure performance with code match and identifier match metrics, assessing accuracy with exact match (EM), edit similarity (ES), and F1 scores. SWE-bench utilizes % Resolved (Pass@1) to gauge the effective-

ness of model-generated patches based on provided unit tests. EvoCodeBench employs Pass@k, where krepresents the number of generated programs, for functional correctness and Recall@k to assess the recall of reference dependencies in generated programs. We set k to 1 in our main experiments.

Implementation details. Before indexing, we filter the Python repositories for each benchmark to retain only Python files. For the SWE-bench dataset, we also exclude test files to avoid slowing down the creation of the code graph database. Following the process outlined in Section 3.1, we construct code graph databases for the indexed repositories, storing the corresponding nodes and edges. We select Neo4j as the graph database and Cypher as the query language.

5 Results

5.1 Analysis of Repository-Level Code Tasks

RACG is crucial for repository-level code tasks. In Table 1, RACG-based methods—BM25, AU-TOCODEROVER, and CODEXGRAPH—basically outperform the NO-RAG method across all benchmarks and evaluation metrics. For instance, on the Cross-CodeEval Lite (Python) dataset, using GPT-40 as the backbone LLM, RACG methods improve performance by 10.4% to 17.1% on the EM metric compared to NO-RAG. This demonstrates that the NO-RAG approach, which relies solely on in-file context and lacks interaction with the code repository, significantly limits performance.

Existing RACG methods struggle to adapt to various repo-level code tasks. Experimental results in Table 1 reveal the shortcomings of existing RACG-based methods like BM25 and AUTOCODEROVER. While these methods perform well in specific tasks, they often underperform when applied to other repository-level

440

441

442

443

415 416

> 448 449 450

> > 451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

444

445

446

²https://github.com/princeton-nlp/ SWE-bench/issues/2

³We use the gpt-4o-2024-05-13 version, https: //openai.com/api

⁴https://huggingface.co/spaces/lmsys/ chatbot-arena-leaderboard

⁵https://chat.deepseek.com/coder

⁶https://dashscope.console.aliyun.com/ model

Table 2: Average token cost comparison across three benchmarks (GPT-40 as the backbone LLM). CCEval* refers to CrossCodeEval Lite (Python) and SWE-bench[†] refers to SWE-bench Lite in this table.

Method	CCEval*	$SWE\text{-}bench^\dagger$	EvoCodeBench
BM25	1.47k	14.76k	-
AUTOCODEROVER	10.74k	76.01k	21.41k
CODEXGRAPH	22.16k	102.25k	24.49k

code tasks. This discrepancy typically arises from their inherent characteristics or task-specific optimizations.

479

480

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

499

503

507

508

510

511

512

513

514

515

516

518

519

524

525

Specifically, AUTOCODEROVER is designed with code tools tailored for SWE-bench tasks, leveraging expert knowledge and the unique features of SWEbench to optimize tool selection and design. This optimization refines the LLM agent's action spaces, enabling it to gather valuable information more efficiently and boosting its performance on SWE-bench tasks (22.96%). However, these task-specific optimizations limit its flexibility and effectiveness in other coding tasks, as evidenced by its subpar results on Cross-CodeEval Lite (Python) and EvoCodeBench compared to other methods.

Similarly, BM25 faces the same issues. In Cross-CodeEval Lite (Python), its similarity-based retrieval aligns well with code completion tasks, enabling it to retrieve relevant usage references or direct answers easily. This results in strong performance, particularly in the ES metric. However, BM25 lacks the reasoning capabilities of LLMs during query construction, making its retrieval process less intelligent. Consequently, when confronted with reasoning-heavy tasks like those in SWE-bench, BM25 often fails to retrieve appropriate code snippets, leading to poor performance.

CODEXGRAPH shows versatility and efficacy across diverse benchmarks. Table 1 shows that CODEXGRAPH achieves competitive results across various repository-level code tasks with general code graph database interfaces. Specifically, with GPT-40 as the LLM backbone, CODEXGRAPH outperforms other RACG baselines on CrossCodeEval Lite (Python) and EvoCodeBench, while also achieving results comparable to AUTOCODEROVER on SWE-bench Lite. This demonstrates the generality and effectiveness of the code graph database interface design.

CODEXGRAPH increases token consumption. CODEXGRAPH uses code graph databases as interfaces and retrieves information from the code repository by writing graph queries. While benefiting from larger and more flexible action spaces, it also incurs increased token costs. The primary reason for this is that the length of the query outcomes is not controllable. Moreover, CODEXGRAPH sometimes encounters loops where it fails to generate executable graph queries. As demonstrated in Table 2, this leads to a higher token usage compared to existing RACG methods.



Figure 4: Performance comparison of two querying strategies on CrossCodeEval Lite (Python) and SWE-bench Lite.

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

559

560

561

562

563

564

565

568

569

570

571

5.2 Deeper Analysis of CODEXGRAPH

Optimal querying strategies vary across different benchmarks. There are two strategies for formulating queries in each round within CODEXGRAPH: either generating a single query or producing multiple queries for code retrieval. Opting for a single query per round can enhance precision in retrieving relevant content but may compromise the recall rate. Conversely, generating multiple queries per round can improve recall but may reduce precision. Experimental results, as illustrated in Figure 4, reveal that for CrossCodeEval Lite (Python), which involves lower reasoning difficulty (26.43 vs. 27.90 in the EM metric), the "multiple queries" strategy is more effective. In contrast, for SWE-bench Lite, which presents higher reasoning difficulty, the "single query" strategy yields better outcomes (22.96 vs. 17.90 in the Pass@1 metric). These findings provide valuable guidance for researchers in selecting the most appropriate querying strategy.

"Write then translate" eases reasoning load. When the assistance of the translation LLM agent is removed, the primary LLM agent must independently analyze the coding question and directly formulate the graph query for code retrieval. This increases the reasoning load on the primary LLM agent, leading to a decline in the syntactic accuracy of the graph queries. Experimental results in Table 3 highlight the significant negative impact of the removal of the translation LLM agent on CODEXGRAPH's performance across all selected LLMs in the CrossCodeEval Lite (Python) benchmark. Even when GPT-40 is used as the backbone model, performance metrics exhibit a significant drop (e.g., the EM metric drops from 27.90% to 8.30%), underscoring the critical role of the translation LLM agent in alleviating the primary LLM agent's reasoning burden.

Edges in code graphs matter. Table 3 illustrates that removing edges from code graphs significantly degrades the performance of CODEXGRAPH across various backbone LLMs on CrossCodeEval Lite (Python). For instance, when GPT-40 is used as the LLM, the Exact Match (EM) metric drops from 27.90% to 14.50%. This decline can be attributed to incomplete code graphs, which increase the failure rate of graph queries and hinder deep searches that rely on the combination of complex conditions.

Model	Method	CrossCodeEval Lite (Python)				
		EM	ES	ID-EM	ID-F1	
Qwen2	CODEXGRAPH	5.00	47.99	9.10	36.44	
	w/o translation LLM Agent	0.50 (-4.50)	10.45 (-37.54)	0.60 (-8.50)	2.62 (-33.82)	
	w/o edges	4.80 (-0.20)	48.74 (+0.75)	9.10 (-0.00)	36.90 (+0.46)	
DS-Coder	CODEXGRAPH	20.20	63.14	28.10	54.88	
	w/o translation LLM Agent	5.50 (-14.70)	53.56 (-9.58)	11.20 (-16.90)	39.75 (-15.13)	
	w/o edges	14.50 (-13.40)	56.64 (-11.34)	21.00 (-14.60)	47.18 (-13.90)	
GPT-40	CODEXGRAPH	27.90	67.98	35.60	61.08	
	w/o translation LLM Agent	8.30 (-19.60)	56.36 (-11.62)	14.40 (-21.20)	44.08 (-17.00)	
	w/o edges	16.40 (-11.50)	57.14 (-10.84)	22.70 (-12.90)	48.27 (-12.81)	

Table 3: Ablation study about the translation LLM agent and the edges of code graphs on CrossCodeEval Lite (Python).



Figure 5: WebUI for Code Chat, used for answering any questions related to code repositories.

CODEXGRAPH is enhanced when equipped with advanced LLMs. Code graph databases provide CODEXGRAPH with a flexible interface, expanding its capabilities beyond existing methods. However, this approach demands strong reasoning and coding abilities from the underlying LLM to formulate effective queries. As shown in Table 1, CODEXGRAPH's performance improves with more advanced LLMs, progressing from Qwen2-72b-Instruct to DeepSeek-Coder-v2 to GPT-40. This trend indicates that as LLMs advance in coding, reasoning, and text comprehension, they become better equipped to leverage code graph databases within CODEXGRAPH, overcoming potential retrieval failures and enhancing overall performance across various benchmarks and metrics.

572

573

574

575

576

577

578

579

580

584

586

6 Real-World Application Scenario

588To showcase CODEXGRAPH's practical value, we de-589veloped five code agents using the ModelScope-Agent590framework (Li et al., 2023). These agents address591common coding challenges that involve understanding592and navigating complex inter-file dependencies: Code593Chat (repository inquiry), Code Debugger (bug diag-

nosis and resolution), Code Unittestor (test generation), Code Generator (new feature implementation), and Code Commentor (documentation enhancement). Each agent integrates key CODEXGRAPH concepts to solve specific production environment issues. Examples and details are provided in Appendix A.3, with Figure 5 illustrating Code Chat's WebUI.

7 Conclusion

CODEXGRAPH addresses the limitations of existing RACG methods, which often struggle with flexibility and generalization across different code tasks. By integrating LLMs with code graph database interfaces, CODEXGRAPH facilitates effective, code structureaware retrieval for diverse repository-level code tasks. Our evaluations highlight its competitive performance and broad applicability on academic benchmarks. Additionally, we provide several code applications in ModelScope-Agent, demonstrating CODEXGRAPH 's capability to enhance the accuracy and usability of automated software development. The qualitative analysis and the schema explanation have been postponed to Appendix A.1 and A.2, respectively.

611

612

613

614

615

594

595

596

631

632

646

667

8 Limitations

617CODEXGRAPH has only been evaluated on Python. In618the future, we plan to extend CODEXGRAPH to more619programming languages, such as Java and C++. Sec-620ondly, there is room for improvement in the construc-621tion efficiency and schema completeness of the code622graph database. Faster database indexing and a more623comprehensive schema (e.g., adding edges related to624function calls) will enhance the broader applicability625of CODEXGRAPH. Finally, the design of CODEX-626GRAPH's workflow can further integrate with existing627advanced agent techniques, such as finer-grained multi-628agent collaboration.

9 Potential Risks

Given that CODEXGRAPH requires scanning the entire code repository, any sensitive information not adequately sanitized by users could lead to data breaches and privacy risks. Furthermore, the implementation of CODEXGRAPH may partially supplant human labor, potentially leading to job displacement, though it also has the potential to create new opportunities in the field.

10 Ethical Considerations

The introduction of CODEXGRAPH aims to aid code professionals in addressing repository-level coding tasks and to assist practitioners in comprehending and familiarizing themselves with complex code repositories. However, the quality and accuracy of CODEX-GRAPH's outputs remain questionable. Overreliance on CODEXGRAPH by novice coders, who may lack the ability to discern the veracity of its results, could lead to misuse of the tool. Additionally, CODEXGRAPH's operation incurs a computational overhead, and the environmental impact of these computational resources warrants consideration.

References

- Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. Codeplan: Repository-level coding using llms and planning. *Preprint*, arXiv:2309.12499.
 - Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun,

Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. Coder: Issue resolving with multi-agent and task graphs. *Preprint*, arXiv:2406.01304.

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflowguided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782*.
- Cognition Labs. 2024. Devin, AI software engineer. https://www.cognition-labs. com/introducing-devin.
- Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, et al. 2024. R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models. *arXiv preprint arXiv:2406.01359*.
- Ajinkya Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. 2024. Class-level code generation from natural language using iterative, toolenhanced reasoning over repository. *Preprint*, arXiv:2405.01573.
- Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.
- Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*.
- Paul Gauthier. 2024. Aider: Ai pair programming in your terminal. https://aider.chat/. Accessed: 2024-08-15.
- Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified crossmodal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.
- Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *Preprint*, arXiv:2402.01680.
- Tanmay Gupta and Aniruddha Kembhavi. 2022. Visual programming: Compositional visual reasoning without training. *Preprint*, arXiv:2211.11559.

693

694

695

697

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

717

719

720

721

722

723

724

669

670

671

672

673

- 725 731 732 734 735 736 737 738 739 740 741 749 743 744 745 747 748 749 750 751 752 753 754 755 756 757 758 759 761
- 763 764 765 766 767
- 771 774 775
- 776 777 778 779

- Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-retriever: Retrievalaugmented generation for textual graph understanding and question answering. arXiv preprint arXiv:2402.07630.
- Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Metagpt: Meta programming for et al. 2023. multi-agent collaborative framework. arXiv preprint arXiv:2308.00352.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. arXiv preprint arXiv:2406.00515.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:2310.06770.
- Alexander Kovrigin, Aleksandra Eliseeva, Yaroslav Zharov, and Timofey Bryksin. 2024. On the importance of reasoning for context retrieval in repository-level code editing. arXiv preprint arXiv:2406.04464.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems, 33:9459-9474.
- Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. 2024a. Devbench: A comprehensive benchmark for software development. arXiv preprint arXiv:2403.08604.
- Chenliang Li, Hehong Chen, Ming Yan, Weizhou Shen, Haiyang Xu, Zhikai Wu, Zhicheng Zhang, Wenmeng Zhou, Yingda Chen, Chen Cheng, Hongzhu Shi, Ji Zhang, Fei Huang, and Jingren Zhou. 2023. Modelscope-agent: Building your customizable agent system with open-source large language models. Preprint, arXiv:2309.00986.
- Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024b. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. arXiv preprint arXiv:2404.00599.
- Zhuoyang Li, Liran Deng, Hui Liu, Oiaoqiao Liu, and Junzhao Du. 2024c. Unioqa: A unified framework for knowledge graph question answering with large language models. arXiv preprint arXiv:2406.02110.
- Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics, 12:157–173.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. arXiv preprint arXiv:2306.03091.

783

784

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

837

- Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024b. Graphcoder: Enhancing repositorylevel code completion via code context graphbased retrieval and language model. arXiv preprint arXiv:2406.07003.
- 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. 2024. Repoagent: An Ilmpowered open-source framework for repositorylevel code documentation generation. Preprint, arXiv:2402.16667.
- Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? arXiv preprint arXiv:2406.01422.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. Preprint, arXiv:2303.17651.
- OpenDevin Team. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. https://github.com/OpenDevin/ OpenDevin. Accessed: ENTER THE DATE YOU ACCESSED THE PROJECT.
- Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. Repohyper: Better context retrieval is all you need for repository-level code completion. arXiv preprint arXiv:2403.06095.
- Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. arXiv preprint arXiv:2307.07924.
- Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. Found. Trends Inf. Retr., 3(4):333-389.
- Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. arXiv preprint arXiv:2306.10998.
- Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. Preprint, arXiv:2104.07567.
- Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Arks: Active retrieval in knowledge soup for code generation. Preprint, arXiv:2402.12317.

897

898

899

- Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, et al. A survey of neural code intelligence: Paradigms, advances and beyond. arXiv preprint arXiv:2403.14734. Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. arXiv preprint arXiv:2402.01030.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. arXiv preprint arXiv:2407.01489.

839

847

849

850

851

858

859

865

866

871

874

875

877

878

884

885

896

2024.

- Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, et al. 2023. Db-gpt: Empowering database interactions with private large language models. arXiv preprint arXiv:2312.17449.
- An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024a. Qwen2 technical report. Preprint, arXiv:2407.10671.
 - John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. arXiv preprint arXiv:2405.15793.
 - Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. Preprint, arXiv:2210.03629.
 - Zhengqing Yuan, Ruoxi Chen, Zhaoxu Li, Haolong Jia, Lifang He, Chi Wang, and Lichao Sun. 2024. Mora: Enabling generalist video generation via a multiagent framework. Preprint, arXiv:2403.13248.
 - Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, et al. 2024. Codes: Natural language to code repository via multi-layer sketch. arXiv preprint arXiv:2403.16443.
 - Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and

Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570.

- Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. Codeagent: Enhancing code generation with tool-integrated agent systems for realworld repo-level coding challenges. Preprint. arXiv:2401.07339.
- Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024b. Autocoderover: Autonomous program improvement. arXiv preprint arXiv:2404.05427.
- Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv preprint arXiv:2406.11931.
- Albert Örwall. 2024. Moatless tools. https:// github.com/aorwall/moatless-tools.

Appendix Α

A.1 Details of the Graph Database Schema

This schema is designed to abstract code repositories into code graphs for Python, where nodes represent symbols in the source code, and edges represent relationships between these symbols.

A.1.1 Node Types

In the code graph, each node represents a distinct element of Python code, with each node type characterized by a specific set of attributes that capture its metadata. These node types and their associated attributes are comprehensively outlined in the Nodes Schema.

A.1.2 Edge Types

Edges in the code graph define the relationships between nodes, illustrating how various elements within Python code are interconnected. Each edge type represents a specific kind of relationship, which helps to clarify the overall structure and flow of the code. The defined edge types, along with the relationships they represent, are detailed in the Edges Schema below:

A.2 Qualitative Analysis

CODEXGRAPH demonstrates robustness and adaptability across various benchmarks. In this section, we illustrate how CODEXGRAPH effectively addresses a GitHub issue through a bug fix task example. The process involves collecting code context and generating a patch based on the issue description and the corresponding codebase. The workflow is depicted in Figure 6. The specific issue, labeled as "django-11848" and included in the SWE-bench lite dataset, involves a flaw in the Django project related to date parsing logic.

The issue centers on the 'parse_http_date' function, which parses dates according to the HTTP RFC7231

Graph Database Schema: Nodes

```
## Nodes
MODULE:
  Attributes:
    - name (String): Name of the module (dotted name)
    - file_path (String): File path of the module
CLASS:
  Attributes:
    - name (String): Name of the class
    - file_path (String): File path of the class
    - signature (String): The signature of the class
    - code (String): Full code of the class
FUNCTION:
  Attributes:
    - name (String): Name of the function
    - file_path (String): File path of the function
    - code (String): Full code of the function
    - signature (String): The signature of the function
FIELD:
  Attributes:
    - name (String): Name of the field
    - file_path (String): File path of the field
    - class (String): Name of the class the field belongs to
METHOD:
  Attributes:
    - name (String): Name of the method
    - file_path (String): File path of the method
    - class (String): Name of the class the method belongs to
    - code (String): Full code of the method
    - signature (String): The signature of the method
GLOBAL_VARIABLE:
  Attributes:
    - name (String): Name of the global variable
    - file_path (String): File path of the global variable
    - code (String): The code segment in which the global variable is defined
```

Graph Database Schema: Edges

```
## Edges
CONTAINS:
 Source: MODULE
 Target: CLASS or FUNCTION or GLOBAL_VARIABLE
HAS_METHOD:
 Source: CLASS
  Target: METHOD
HAS_FIELD:
 Source: CLASS
 Target: FIELD
INHERITS:
 Source: CLASS
 Target: CLASS (base class)
USES:
  Source: FUNCTION or METHOD
  Target: GLOBAL_VARIABLE or FIELD
  Attributes:
   - source_association_type (String): FUNCTION, METHOD
    - target_association_type (String): GLOBAL_VARIABLE, FIELD
```



Figure 6: Workflow for Resolving Issue #11848 in Django's 'parse_http_date' Function

section 7.1.1.1. The function supports three date formats: RFC1123, RFC850, and ASCTIME. However, the problem arises due to the hardcoded logic for interpreting two-digit years, which does not dynamically adjust based on the current year, leading to noncompliance with the RFC 7231 standard.

Given this issue description, CODEXGRAPH begins by analyzing the potential cause, identifying that the core of the issue lies in the 'parse_http_date function'. To address this, it is essential to retrieve the code of the 'parse_http_date' function for further analysis. Here, CODEXGRAPH employs a combination of the "generating a single query" and "Write then translate" strategies. Specifically, the primary LLM agent first generates a natural language query, which is then translated into a Cypher query by the translation LM agent.

By executing this Cypher query, CODEXGRAPH retrieves the relevant data from the graph database and returns it to the primary LLM agent for further analysis. Upon analyzing the results, the primary LLM agent concludes that to accurately locate the problematic function, it is necessary to identify the file path of the module containing the 'parse_http_date function'. After another iteration, the primary LLM agent successfully identifies the bug's location and generates the required patch to fix it.

973

974

975

976

977

978

979

980

981

982

983

984

985

986

987

988

989

990

991

992

993

994

The CODEXGRAPH demonstrates the ability to iterate and refine its analysis, effectively handling complex code issues. By identifying the exact location of the bug and proposing a patch, the CODEXGRAPH resolves the problem, showcasing its utility in automated code analysis and bug fixing.

A.3 Real-World Application

In this section, we present the WebUI interface for CODEXGRAPH, showcasing its five practical applications: Code Chat, Code Debugger, Code Unittestor, Code Generator, and Code Commentor. The interface is designed to facilitate user interaction, providing a streamlined and intuitive environment for various code-related tasks. We built the WebUI interface using Streamlit⁷, a powerful and user-friendly framework that allows for the rapid development of interactive web applications.

To experience our application firsthand, you can visit ModelScope-Agent and navigate to the CODEXGRAPH This repository provides a detailed guide on how to set

Streamlit: https://streamlit.io/

>		Deploy :			Deploy :
🛠 Code Debugger			🔍 Code Unit Tester		
Settings		*	Settings		
Bug Issue			Code needing unittest		
File Path (optional)			File Path (optional)		
Enterne patrinere					
Please copy and paste the code snippet and describe the bug or issue you are facing. Include any error messages if available.			Please copy and paste the code snippet for which you need unit tests, include any specific scenarios or edge cases you want to test.		
3103			Sind		
	(a) Code Debugger		(b) Code Unittestor	
>		Deploy i >		,	Deploy
🤸 Code Generator			🍃 Code Commenter		
Settings		*	Settings		•
New Requirements			Code needing comments		
File Path (optional)			File Path (optional)		
new feature or module you need.			commonts here.		
Sind			Sered		
Send					
Sed					

(c) Code Generator

(d) Code Commentor

Figure 7: WebUI for Code Debugger, Code Unittestor, Code Generator, and Code Commentor.

995 996	up and interact with the various applications we have described.	Summarize the CodexGraphAgentChat class, what has method, and what for
		analysis
		The user's question asks for a summary of the CodexGraphAgentChat. Class, specifically what methods it has and their purposes. To answer this, we need to:
		1. Identify the CodexGraphAgentChat class and its file path.
		2. List all methods within this class.
		 Optionally, if method signatures or code snippets are available, they could provide insights into the purpose of each method.
		Given the schema, we can start by querying for the CodexGraphAgentChat. class to ensure it exists and to gather its methods.
		code_search
007	A 2.1 Example of Code Chot	Text Query 1
997	A.J.I Example of Code Chat	Retrieve the names and file paths of all methods within the CodexGraphAgentChat class.

Figure 8: Using Cypher queries to retrieve information about the 'CodexGraphAgentChat' class, from the code repository.

998Code Chat allows users to inquire about a code reposi-
tory, providing insights into code structure and function1000usage. This functionality is particularly useful for un-
derstanding complex codebases, identifying dependen-
cies, and exploring the usage of specific classes, meth-
10031003ods, and functions.

1004Here is an example of Code Chat. The user's ques-1005tion is "Summarize the 'CodexGraphAgentChat' class,1006what has method, and what for".

Summarize the CodexGraphAgentChat class, what has method, and what for

answer:

The CodexGraphAgentChat class is designed for managing chat-based interactions involving an agent. It includes the following methods:

- 1. __init__ : Initializes the agent.
- 2. llm_call(self, nsg) : Makes a Language Model (LLM) call with a given message.
- update_agent_message(self, msg) : Updates the agent's message.
- update_user_message(self, msg) : Updates the user's message
 get_chat_history(self) : Retrieves the chat history.
- generate(self, messages) : Generates responses based on the messages.
- _run(self, user_query: str, file_path: str = '', **kwargs) -> str: Runs specific tasks with a user auery and optional file path.
- 8. set_action_type_and_message(self) : Sets the action type and message.

Each method plays a crucial role in facilitating chat interactions, leveraging an LLM for generating responses, and maintaining a history of interactions.

analysis:

The CodexGraphAgentChat class is pivotal for handling chat-based tasks, integrating with a Language Model for generating responses, and managing the state of chai interactions. The methods are designed to initialize the agent, handle LLM calls, update messages, retrieve chat history, generate responses, run specific tasks, and set action types and messages. This class is essential for the functionality of a chat-based agent system.

Figure 9: Once the necessary information is gathered, Code Chat constructs a comprehensive response to the user's question. This response includes a summary of the 'CodexGraphAgentChat' class, a list of its methods, and a description of what each method does.

A.3.2 Example of Code Debugger

The Code Debugger diagnoses and resolves bugs by applying iterative reasoning and information retrieval to suggest targeted fixes. It utilizes Cypher queries to analyze the code repository, identify the cause of the issue, and recommend precise modifications.

Here is an example of Code Debugger. The user's input is a real issue⁸ where the outcome does not match the expected behavior. The Code Debugger first analyzes the problem, then uses Cypher queries to retrieve relevant information and infer the cause of the bug. Finally, it provides an explanation of the bug and suggests the location for the modification.

Problem Description
When calling the OpenAi class's_chat_no_stream() method, the usage_info result is empty. It seems does not execute stat_last_call_token_info inchat_no_stream
Specific Issue
<pre>from modelscope_agent.llm import OpenA1 msg = (['role': 'uuer', 'content': 'hello']] llm = OpenA('ulm_config) resp = llm.chat(messages=msg,</pre>
Actual Output
>>> usage_info = ()
Expected Output
<pre>>>> usage_info = {'prompt_tokens': 5, 'completion_tokens': 10, 'total_tokens': 15}</pre>

Figure 10: The issue describes a problem where the outcome does not match the expected behavior.

analysis

The base pertains to the weakingse_spect.lls module.ppedically the ignest class. The problem is that when calling the _ubst_w_stream() method of the issues. Class, the usage_info result is empty, indicating that the ust_it_list_list_method might not be executed as expected. This is exidenced by the problematic area are: I. The class to locate the problematic area are: I. The specific uses with the medicatespe_specific list of the specific area area. I. The dust_it_list_method within the issues class. I. The dust_it_list_method within the issues class. I. The dust_it_list_method within the issue class. I. The dust_it_list_method with method within the operate class. I. The dust_it_list_method within method within the issue class. I. The dust_it_list_method within method with the dust are are class to updating the usage_info as expected. I. The dust_it_list_method within the issue was also the method is ensure it correctly calls stat_list_cill_time_info. I. Class the get_usage method to ensure it returns the correct scalage_info.

code_search

Text Query 1

Figure 11: Analyzing the problem and retrieving information using Cypher queries.

Extracted Cypher query 0:
NATCH (mod:HODULE (name: 'modelscope_agent.llm')) NATCH (mod:=[:CONTAINS]=-(cls:CLASS (name: 'OpenA'')) NATCH (cls]=[:RETON]=->(method:HETIMOD (name: '_chat_no_stream')) RETURN method.code
Response for Cypher query 0:
<pre>def _chat_no_stream(self,</pre>
messages: List[Dict],
<pre>stop: Optional[List[str]] = None,</pre>
<pre>**kwargs) -> str:</pre>
<pre>stop = selfupdate_stop_word(stop)</pre>
logger.info(
f'call openai api, model: {self.model}, messages: {str(messages)}, '
f'stop: (str(stop)), stream: False, args: (str(kwargs))')
response = self.client.chat.completions.create(
model-self.model,
message s -messages,
stop-stop,
stream-False,
**kwargs)
setr.stat_last_call_token_into(response)
Logger . Into
i cart openar aprisuccess, output: (response.cn010es[0].message.content);
/ TOPOL arran handling
return reproprie children (1) marriage content

Figure 12: Executing Cypher queries to search the code for relevant information.

	Let's analyze collected context first.
۲	analysis
	The analysis of the _chat_mo_stream method implementation from the OpenA1 class in the modelscope_agent.llm module reveals the following:
	 The methoddat_we_stress is correctly calling self_stat_last_call_take_lefs(response) after receiving the response from the OpenA API. This indicates that the suum angle not be directly valued to thedat_we_stress method thealPut rather how the stat_last_call_take_info processes the response how the user_info is managed and retrieved.
	 The task last gold gold important for method is expected to supplies the suspection become from the OpenALAPI. The fact that usage information empty suggests that either the stark_last_soll_time_informethod is not functioning as intended or the get_usage method is not returning the updated suspective.
	Next steps to resolve the issue:
	Verify the implementation of the stat_last_call_takes_info method to ensure it correctly updates the usage_info. Check the get_usage method to ensure it returns the correct_usage_info.
	Text Query 2
	$Retrieve the code of the method named {\tt stat_last_call_token_info} from the class {\tt OpenAi} in the module {\tt modelscope_agent.llm}.$
۲	code_search
	Text Query 1
	Retrieve the code of the method named stat_last_call_token_info from the class OpenAi in the module modelscope_agent.llm.
	Text Query 2
	Retrieve the code of the method named get_usage from the class OpenA1 in the module modelscope_agent.llm.

Figure 13: Analyzing the retrieved information to identify potential causes of the bug.

⁸https://github.com/modelscope/ modelscope-agent/pull/549

Extracted Cypher query 0:	
<pre>MATGH (md:HODME (name: 'modelscope_agent.llm')) MATGH (mdd:[:COMIADS]><(LscLASS (name: 'dpend')) MATGH (cls:]:KM_METROD[->(methid:HETROD (name: 'stat_lsst_call_token_info')) RETURN method.code</pre>	
Response for Cypher query 0:	
<pre>def stal_last_call_taken_info:self, response): try: self.last_call_upage_info = response.usage.dtct() response.usage.dtct() eccupt AttributeFror: for chunk in response: for sastrr(chunk, 'usage') and chunk.usage is not None: self.last_call_usage_info = chunk.usage dict() yleid chunk</pre>	
Extracted Cypher query 1:	
<pre>MATCH (mod:MODULE (name: 'modelscope_agent.llm')) MATCH (mod:[CONTANS]=/C(LstCLASS (name: 'openAt')) MATCH (cl-c):MAS_UETENDD)=/nmethod:METHOD (name: 'get_usage')) RETURN method.code</pre>	
Response for Cypher query 1:	
<pre>def get_usage(self) → Dict: return self.last_call_usage_info</pre>	

Figure 14: Performing additional Cypher code searches to gather more information.



Figure 15: Inferring the cause of the bug based on the analysis of the retrieved information.

<section-header><section-header><section-header><section-header><section-header><section-header><section-header><section-header><list-item><list-item><list-item><section-header>

Explanation:
1. Modification: The star_last_cill_takes_tells method is modified to ensure that isst/last_cill_takes_tells in case of an attributivers. This provests the get_mage method from netwring lases, which would cause takes when trying to access to properties.
2. Modification: The set_mage method is modified to ensure any dictionary if scill_tats_cill_takes_tells. This ensures that the method always returns a dictionary, even if the star_last_scill_takes_tells method failed to update the same_tells.

Figure 17: Providing a detailed explanation of the issue and the underlying cause of the bug.



Figure 18: Suggesting the first modification to resolve the bug.

Figure 19: Suggesting the second modification to ensure the bug is resolved.

A.3.3 Example of Code Unittestor

Here is an example of Code Unittestor. The user's input is: "Generate test cases for TaskManager." The Codex-Graph agent will first retrieve all methods and inheritance relationships in 'TaskManager', and then generate detailed test case code.



Figure 20: Generated detailed unit test code for the 'TaskManager' class, covering its methods and inheritance relationships.

1020
1021
1022
1023
1024
1025

1031

1032 1033

1034

1035

1036

1037

A.3.4 Example of Code Generator

The user has requested a function to retrieve the number of input and output tokens of 'CypherAgent'. However, the challenge is identifying the corresponding fields within 'CypherAgent' as this information is not provided in the user's input.

۲	thought
	To fulfill the user's requirement of generating a function to get the number of input tokens and output tokens of a CypherAgent, I need to first identify CypherAgent class and its methods or fields that are related to taken counting. This involves querying the graph database to retrieve the methods are fields of the CypherAgent class or any base classe in kinets from, an the Autoniouxian graph is behave to refere the methods are fields of the CypherAgent class or any base classe in kinets from, any the Autoniouxian graph is behave to refere a characteristic of the class or any base classes in kinets from, any the Autoniouxian graph is behaved or defined in a base class. Additionally, I sho check if there are any methods or fields specifically related to taken counting within the class or its hierarchy.
	action
	TEXT_QUERIES
	action_input ∞
	[start_of_text_queries]
	Text Query 1
	Retrieve all methods and fields of the class 'CypherAgent'.
	Text Query 2
	Retrieve all base classes that 'CypherAgent' inherits from.
	Text Query 3
	Retrieve methods and fields from the base classes of 'CypherAgent' that contain 'token' in their names. [end_of_text_queries]

Figure 21: The thought process in determining how to identify the relevant fields.

۲	thought
	Based on the retrieved information, the CypherAgent class does not directly contain methods or fields related to token counting. However, it does have fields injunct, token, we and surgert, token, we that tore the number of input and output tokens, respectively. These fields are initialized in the surgert in the surgert of the surgert requirement.
	generate code
	<pre>def get_token_counts(self): return self.input_token_num, self.output_token_num</pre>

Figure 22: By using Cypher queries, it was discovered that the corresponding fields are 'input_token_num' and 'output_token_num', which enables the generation of the correct code.

A.3.5 Example of Code Commentor

The Code Commentor analyzes code to provide detailed comments, enhancing code readability and maintainability. It leverages the code graph database to understand the code's structure and behavior, ensuring accurate and informative comments.



Figure 23: The thought process: Understand the 'Task' class and 'add_item' method.

۲	# file path: task_manager.py
	from user import User
	from task import Task
	from base_manager import BaseManager
	class TaskManager(BaseManager):
	<pre>definit(self): """</pre>
	Initializes a new instance of the TaskManager class.
	<pre>super()init()</pre>
	<pre>def add_task(self, task_id, title, description, user_id): """</pre>
	Adds a new task to the task manager.
	Parameters:
	task_id: Unique identifier for the task.
	title: Title of the task.
	description: Description of the task.
	user_id: Identifier of the user associated with the task.
	Returns:
	bool: True if the task was successfully added, False if the task already exists.
	<pre>task = Task(task_id, title, description, user_id)</pre>
	<pre>return self.add_item(task_id, task)</pre>

Figure 24: By using Cypher queries, the specific implementation of the return function was obtained, and the return type was clarified.

1038

1039

1040

1041

1042

1043

1044

1045

1046

1047

1048

1049

1051

1052

1053

1054

A.4 Analysis of the Poor Performance of Agent-Based Methods

Table 1 shows that both AUTOCODEROVER and CODEXGRAPH, which are agent-based RACG methods, unexpectedly perform poorly across all benchmarks when using Qwen2-72b-instruct, even falling behind BM25. We believe this is due to the complexity and fragility of the agent workflow. In particular, when handling repository-level code tasks, the agent system must simultaneously manage long-context understanding, code reasoning, tool or API invocation, and formatted output. This multi-faceted process can easily lead to error accumulation from the very beginning, as every step in the workflow is critical. We argue that this issue is a general weakness of agent systems equipped with relatively "small" LLMs, rather than a problem specific to our method.