From Code to Execution: Multi-Agent GraphRAG for Automated Artifact Generation

Amirhossein Layegh*
amlk@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Amir H. Payberah
payberah@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Mihhail Matskin
misha@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Modern data and Machine Learning (ML) pipelines are developed as code repositories but must be manually transformed into deployable artifacts such as Docker images and Kubernetes workflows, a process prone to dependency and configuration errors. While Large Language Models (LLMs) can assist with code understanding, they often fail on multi-file repositories due to missing structural context and incomplete dependency resolution. We propose a multi-agent graph-based Retrieval-Augmented Generation (GraphRAG) framework that automates the generation of deployment artifacts from code repositories. The framework constructs an Abstract Syntax Tree (AST)-grounded knowledge graph, performs structure-aware retrieval, and coordinates specialized agents to synthesize containerization and workflow specifications. The system further improves reliability through iterative self-healing driven by execution logs. Evaluated on 17 GitHub repositories spanning ETL, ML training, and LLM-based evaluation pipelines, our approach achieves 65% end-to-end execution success on Kubernetes, outperforming a vector-only RAG baseline (24%) and a single-agent GraphRAG variant (47%). These results indicate that combining graph-centric retrieval with agent decomposition improves execution reliability.

CCS Concepts: • **Software and its engineering** → *Software creation and management*.

Keywords: GraphRAG, Multi-agent systems, Deployment automation, Knowledge graphs

ACM Reference Format:
Amirhossein Layegh, Amir H. Payberah, and Mihhail Matskin. 2026. From Code to Execution: Multi-Agent GraphRAG for Automated Artifact Generation. In *The 6th Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3805621.3807637

This work is licensed under a Creative Commons Attribution 4.0 International License.
EuroMLSys '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2605-7/26/04
https://doi.org/10.1145/3805621.3807637

1 Introduction

Modern data-intensive applications rely on increasingly complex pipelines that span data ingestion, transformation, model execution, and distributed orchestration [23]. To manage this complexity and ensure reliability, organizations adopt orchestration frameworks such as Apache Airflow[1], Dagster[2], and the Argo Workflows[3] engine on Kubernetes, which support task scheduling, container-based execution, and fine-grained flow control [19, 20, 33]. Despite substantial progress in workflow engines and container technologies, converting a code repository into deployable artifacts such as container images and Kubernetes-native workflow specifications remains labor- and knowledge-intensive, and remains a persistent source of engineering overhead and deployment failures [6, 31].

This process requires generating deployment artifacts based on a deep, structural understanding of the source code [27]. Engineers must manually perform accurate dependency resolution, synthesize compliant environment artifacts (e.g., Dockerfiles and `requirements.txt`), and craft syntactically and semantically correct orchestration configurations (e.g., Kubernetes YAML) [3, 6, 7]. This manual configuration creates performance and reliability bottlenecks and exposes a persistent gap between experimental code and production-ready deployment specifications, limiting the scalability of data orchestration workflows [6, 7, 27].

Recent advances in Large Language Models (LLMs) have sparked interest in applying them to code understanding, generation, and broader software engineering workflows [15, 32]. LLMs trained on code now achieve strong performance on many programming benchmarks and are increasingly used to assist software development and DevOps processes [12, 13]. However, empirical studies show that LLMs' outputs frequently contain missing dependencies, inconsistent logic, or structural errors in multi-file repositories, where limited context windows prevent models from capturing long-range relationships [2, 14, 28].

[1] `https://airflow.apache.org`
[2] `https://dagster.io`
[3] `https://argoproj.github.io/workflows`

To mitigate these failures, Retrieval-Augmented Generation (RAG) has emerged as a standard approach for grounding model outputs in external artifacts such as documentation, databases, or code repositories [5, 16]. However, standard RAG techniques typically rely on vector similarity over isolated text or code chunks, treating the repository as an unstructured collection of fragments [17, 30]. Consequently, such approaches often fail to capture the relational structure of software systems, including call chains, imports, data flows, and module interactions, which are essential for reasoning over large repositories [5, 22].

These limitations have motivated graph-based RAG (GraphRAG) [4], in which documents are represented as Knowledge Graphs (KGs), and context is retrieved through graph-aware operations [18, 22]. By leveraging explicit structural relationships in addition to semantic similarity, GraphRAG provides richer contextual grounding and improves performance over standard vector-based RAG, particularly for tasks that require reasoning over global system state and cross-file dependencies [4, 8].

Our framework extends this graph-centric perspective to automate the transition from experimental code to production-ready orchestration artifacts. We propose a multi-agent GraphRAG approach that combines KG-enhanced retrieval with specialized agents for artifact generation. Concretely, we construct a semantic KG using LLM-based entity extraction and Abstract Syntax Tree (AST) parsing. We then use community structure to enable local (entity-level) and global (community-level) reasoning over repository structure [4].

Building on this KG, we coordinate multiple specialized agents that query the graph to extract domain-specific insights such as dependencies, entry points, and runtime requirements. The agents iteratively generate containerization artifacts (e.g., Dockerfiles), synthesize minimal runtime specifications (RunSpecs), and produce Kubernetes-native workflow specifications in Argo Workflows YAML. The framework also incorporates self-healing. When execution fails (e.g., due to Docker build errors), agents analyze error signals and regenerate improved artifacts by feeding failure context into subsequent LLM prompts, enabling end-to-end deployment with minimal manual intervention.

We study the task of automatically converting Python source code repositories that implement executable data and Machine Learning (ML) workflows into deployable pipeline specifications requiring containerization and Kubernetes orchestration. Our objective is to generate syntactically valid deployment files and to produce artifacts that build successfully and execute end-to-end on a Kubernetes cluster.

The main contribution of this work is an end-to-end multi-agent GraphRAG framework that connects repository understanding to deployable orchestration artifacts. We combine AST-grounded extraction with LLM-based summarization to construct a repository KG, use graph-enhanced retrieval to provide agents with structured context, and incorporate execution feedback for automatic artifact repair. In an evaluation on 17 GitHub repositories, our multi-agent configuration achieves 65% end-to-end execution success on a local Kubernetes (k3d) cluster, compared to 24% for a vector-based RAG baseline and 47% for a single-agent GraphRAG variant, demonstrating that structure-aware retrieval and agent decomposition improve execution reliability. The implementation is available on GitHub [4].

2 Related Work

Data pipelines, orchestration, and automated deployment. Modern data and ML pipelines are commonly operationalized using workflow orchestrators such as Airflow, Dagster, Prefect, Luigi, and Kubernetes-native engines like Argo Workflows [19, 20]. While these systems provide abstractions for Directed Acyclic Graph (DAG)-based scheduling and container execution, they typically assume that developers manually author workflow definitions and manage dependencies, configuration, and failure handling [33]. Prior work has explored partial automation, including learning-based Dockerfile generation [24] and knowledge-guided dependency inference for Python projects [10, 34]. However, Docker build studies show that missing or conflicting dependencies remain common and difficult to repair automatically [9, 21]. End-to-end automation from repository source code to executable, orchestrated pipelines therefore remains limited.

LLMs and retrieval for repository-level reasoning. LLMs have demonstrated strong performance in code generation and software engineering tasks [11, 35], but exhibit systematic errors in multi-file repositories due to incomplete context and incorrect cross-file reasoning [14, 29]. RAG mitigates context limitations by grounding model outputs in external artifacts [16, 26]. In repository-level settings, vector-based retrieval treats code as isolated fragments and often fails to capture structural relationships such as imports, containment, and usage links [17, 25]. Graph-based retrieval approaches address this limitation by modeling repositories as structured graphs and retrieving context via graph traversal [4, 18]. However, existing work focuses primarily on code completion and editing rather than generating deployable artifacts.

Our work extends graph-centric retrieval to the deployment setting by combining structured repository reasoning with multi-agent decomposition to automate containerization, runtime specification synthesis, and Kubernetes-native workflow generation.

[4]https://github.com/AmirLayegh/code-to-exec

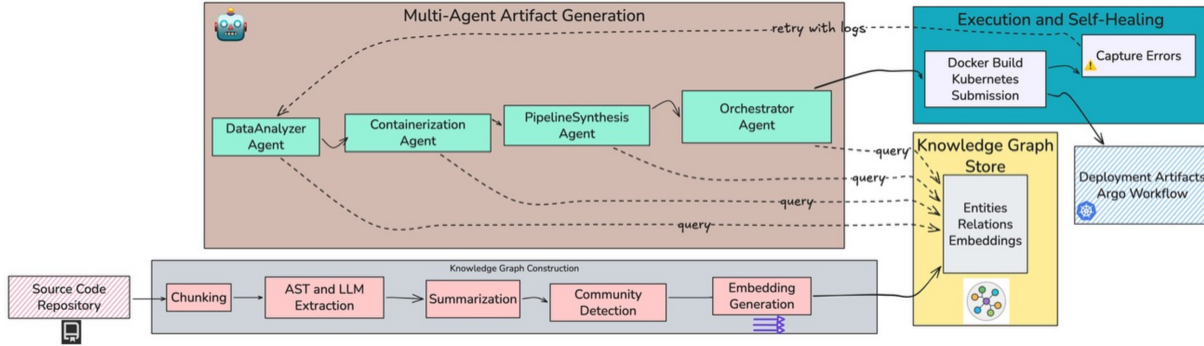

Figure 1. System architecture of the proposed multi-agent GraphRAG framework, illustrating knowledge graph construction, artifact generation, and self-healing execution.

3 Methodology

3.1 System Overview

Our multi-agent GraphRAG framework transforms code repositories into production deployment artifacts through four stages: (1) *KG Construction* that builds a semantic representation of the repository, (2) *Graph-Enhanced Retrieval* that extracts contextual insights through local and global graph queries, (3) *Multi-Agent Artifact Generation* that coordinates specialized agents to synthesize deployment artifacts, and (4) *Self-Healing Execution* that detects and corrects generation errors through iterative refinement.

Figure 1 illustrates the overall system architecture. Given an input repository $\mathcal{R}$, the system constructs a KG $\mathcal{G} = (V, E, M)$, where V denotes code entities, E represents relationships, and M represents hierarchically detected community structures. Artifact generation is performed by four specialized agents, which are described in Section 3.4. All agents query the KG via a graph-enhanced retrieval layer and leverage LLMs to generate artifacts. A self-healing mechanism monitors execution outcomes and triggers artifact regeneration when failures occur.

3.2 KG Construction

The KG construction pipeline transforms raw repository source code into a structured and queryable representation through five stages: (1) *document chunking*, (2) *entity and relationship extraction*, (3) *LLM-based summarization*, (4) *community detection*, and (5) *embedding generation*.

3.2.1 Document Chunking. Given a repository $\mathcal{R}$, we retrieve all Python source files and segment them into overlapping chunks using a language-aware sliding-window strategy. Chunk boundaries respect Python syntactic structure to avoid disrupting code semantics.

3.2.2 Entity and Relationship Extraction. For each code chunk, we extract entities and relationships using a hybrid procedure that combines deterministic AST parsing with LLM-based semantic enrichment. We first parse the chunk using Tree-sitter to obtain its AST. The AST explicitly provides valid entities, such as functions, classes, import statements, and variable assignments, along with their identifiers and source code spans. We aggregate these elements into a structural summary capturing each entity's identifier, type (function, class, import, or variable), and source span. We also derive explicit AST-based relations, including class–method containment, inheritance, and import dependencies. The structural summary, inferred relations, and raw code chunk are provided to the LLM as part of the extraction prompt. Rather than rediscovering syntactic entities, the LLM enriches them with semantic descriptions and predicts additional behavioral relationships not explicitly encoded in the AST.

In a single inference pass, the LLM produces structured outputs for entities and relationships. For each entity, it returns the entity name, its type (Module, Class, Function, or Variable), and a natural language description of its behavior and role. For each relationship, the LLM returns the source and target entities, a relationship type, a natural language description, and a confidence score. Relationship types span structural relations (e.g., HAS_METHOD, INHERITS_FROM), dependency relations (e.g., IMPORTS), and behavioral relations (e.g., CALLS, USES, INITIALIZES). This hybrid design guarantees syntactic validity while enabling the extraction of higher-level semantic and behavioral relations beyond the AST.

3.2.3 Summarization. Because entities and relationships may be referenced across multiple chunks, the extraction phase can produce redundant or partially inconsistent descriptions. The summarization stage consolidates these multi-source descriptions into unified representations for both entities and relationships. For each entity with multiple extracted descriptions, we collect its description list, where each description originates from a different chunk. We then invoke the LLM with a summarization prompt that instructs the model to merge the descriptions into a single coherent summary, resolving minor inconsistencies when possible.

The resulting summary is stored as a new property of the entity node and becomes the primary textual representation of the entity for downstream embedding generation and graph-enhanced retrieval. An analogous consolidation is performed for relationships in which multiple edges between the same entity pair are merged into a single summarized edge, while preserving the original edges for traceability.

3.2.4 Community Detection.

To support hierarchical reasoning over repository structure, we partition the entity set into semantically coherent communities using the Louvain algorithm [1]. The algorithm aims to produce clusters of densely connected entities. For each multi-entity community, we generate an LLM-based summary describing its structural composition, along with a complexity score reflecting its architectural significance. These summaries are stored in the graph and linked to member entities via `IN_COMMUNITY` relationships. During retrieval, they provide high-level architectural context for entities identified through vector-based similarity search.

3.2.5 Embedding Generation.

To support semantic similarity search, we generate dense vector embeddings for each entity using OpenAI's `text-embedding-3-small` model. Each embedding is computed from the consolidated entity summary (or concatenated descriptions if no summary exists) and stored as a node property in the Neo4j graph database. A vector index enables efficient approximate nearest-neighbor (ANN) search over the entity set.

3.3 Graph-Enhanced Retrieval

Given a natural language query, retrieval combines vector-based similarity search with structured graph traversal to gather repository-level context. All agents in our framework use this retrieval mechanism, issuing multiple domain-specific queries to gather information for artifact generation. We first embed the query and perform ANN search over entity embeddings to identify the top-10 most semantically similar entities. From these seed entities, we expand context through Cypher graph traversal, collecting associated source code chunks containing the entity mentions, community summaries for the corresponding communities, summarized relationships among retrieved entities, and the consolidated entity summaries. The retrieved context—source is formatted and provided to the LLM, which generates a grounded response that references specific code components and their architectural context.

3.4 Multi-Agent Artifact Generation

The artifact generation layer coordinates four specialized agents, each responsible for a component of the deployment configuration. All agents follows a three-phase workflow: (1) *analysis*, via domain-specific queries to the Graph-Enhanced Retrieval layer to gather structured context; (2) *generation*, in which the agent prompts an LLM using the retrieved graph context to synthesize deployment artifacts; and (3) *execution*, in which the generated artifacts are validated.

DataAnalyzerAgent. Identifies data assets by querying for data-loading functions, file path references, and configuration usage patterns. It augments graph-derived insights with a repository file scan to construct a data catalog required for volume mounting and runtime access.

ContainerizationAgent. Generates containerization artifacts by querying for dependencies, entry points, and runtime requirements. Based on the retrieved context, the LLM synthesizes a `Dockerfile`, a `requirements.txt`, and optionally a `docker-compose.yml`. The agent then executes a Docker build to validate the generated configuration and detect dependency or environment inconsistencies.

PipelineSynthesisAgent. Although the ContainerizationAgent produces a runnable Docker image, it does not define how the application should be invoked within the container. The PipelineSynthesisAgent addresses this limitation by generating a structured RunSpec that formalizes the complete container invocation. It queries the retrieval layer for entry points, CLI arguments, environment variables, and I/O paths. Based on the retrieved context, it synthesizes a minimal RunSpec that specifies the command, arguments, environment variables, volume mounts, and working directory. For example, given a machine learning training script, the agent may produce:

```
{
 "image": "myrepo/ml-trainer:latest",
 "command": "python",
 "args": ["train.py", "--epochs", "10"],
 "env": {"DATA_PATH": "/app/data"},
 "volumes": ["./data:/app/data"],
 "workdir": "/app",
 "timeout_seconds": 3600
}
```

OrchestratorAgent. Translates the RunSpec into an Argo Workflows specification. It maps RunSpec fields to the corresponding Argo workflow schema, while also synthesizing workflow-level metadata, including entry points, retry strategies, and template definitions. The resulting YAML specification defines a reproducible execution unit for submission to a Kubernetes cluster. For example:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
 name: ml-trainer-workflow
spec:
 entrypoint: main
 templates:
 - name: main
   container:
     image: myrepo/ml-trainer:latest
     command: ["python"]
     args: ["train.py", "--epochs", "10"]
   retryStrategy:
     limit: 2
```

The agents execute sequentially, passing structured outputs between stages. The ContainerizationAgent produces a validated image based on insights from the DataAnalyzerAgent. The PipelineSynthesisAgent incorporates this image into a RunSpec that defines invocation semantics, and the OrchestratorAgent translates the RunSpec into a Kubernetes-native Argo Workflow specification. This staged design maintains artifact consistency while preserving modular specialization.

3.5 Self-Healing Execution

Generated artifacts may fail due to missing dependencies, incorrect configurations, or mismatches in the environment. To reduce manual intervention, we incorporate an iterative self-healing mechanism. When a validation or an execution step fails (e.g., during Docker build or workflow submission), the system captures the error message and calls the LLM again with (1) the original graph-enhanced retrieval context, (2) the previously generated artifacts, and (3) the error message from the failed attempt. The model is instructed to analyze the failure, identify likely root causes, and produce corrected artifacts until success or a predefined retry limit is reached.

4 Evaluation

4.1 Experimental Setup

We compare three system configurations and assess the contributions of graph-based retrieval, multi-agent decomposition, and self-healing execution: (1) *Naive RAG*, a vector-only retrieval baseline that embeds code chunks using the same embedding model as GraphRAG and retrieves the top-5 most similar chunks for each query. Retrieved chunks are concatenated and passed directly to the LLM without structural context, community summaries, or graph traversal; (2) *GraphRAG (Single Agent)*, the full KG construction pipeline with community detection and summarization, using only a single agent that handles all artifact generation (Dockerfile, RunSpec, Argo YAML) in a single LLM call; and (3) *GraphRAG (Multi-Agent)*, the complete framework with KG construction and all four specialized agents (DataAnalyzer, Containerization, PipelineSynthesis, Orchestrator) operating sequentially. Self-healing is enabled with up to 3 retry attempts per agent.

We use OpenAI's GPT-4o for KG construction (entity extraction, summarization) and GPT-4o-mini for agent artifact generation. All generation calls use temperature = 0 and max_tokens = 4096. We use `text-embedding-3-small` for generating the embeddings. The same model configurations are applied across all system variants.

We evaluate on 17 GitHub repositories spanning data processing, ML training, and LLM-based generation pipelines. All repositories are Python-based to ensure consistent AST parsing, contain 5-28 source files to balance realistic complexity against tractable KG construction, and include identifiable execution entry points necessary for automated artifact generation. We focus on repositories requiring containerization and orchestration—data pipelines, training workflows, and evaluation systems—rather than libraries or web services.

The dataset comprises four main categories: (1) ETL pipelines ($n = 5$) focus on batch data processing using PySpark and PostgreSQL, (2) ML training pipelines ($n = 3$) cover supervised learning, including text classification and sensor-based activity recognition, (3) Complex data processing workflows ($n = 4$) include streaming pipelines, MapReduce jobs, and multi-stage processing workflows, and (4) LLM-based generation and evaluation pipelines ($n = 5$) address ontology generation with automated evaluation mechanisms.

4.2 Evaluation Metrics and Failure Mode Taxonomy

Functional correctness is assessed using three metrics: (1) *Docker Build Success Rate* measures the percentage of generated Dockerfiles that complete docker build without errors (exit code 0), (2) *Container Test Success Rate* measures the percentage of successfully built images that pass a dependency-import sanity check; specifically, we execute `docker run -rm <image> python -c "import <main_module>"` with a 60-second timeout, where `<main_module>` is derived from the detected entry point, and (3) *Argo Workflow Validity* measures the percentage of generated Argo Workflow YAML files that pass argo lint schema validation.

Pipeline completion is evaluated using two metrics: (1) *End-to-End Success Rate* measures the percentage of repositories for which the generated Argo workflow executes successfully on a local k3d Kubernetes cluster, verifying that the complete pipeline (Dockerfile, RunSpec, and Argo YAML) produces a runnable workflow rather than merely syntactically valid artifacts, and (2) *Self-Healing Effectiveness* measures, among repositories that failed on the first attempt, the percentage that were successfully recovered through automatic retry (up to three retries per agent), and is computed only for the GraphRAG Multi-Agent configuration.

We also manually categorize failures based on error logs into the following categories: (1) *missing dependencies*, where required packages or libraries are not installed in the container environment; (2) *wrong entrypoint*, where the main execution module is incorrectly identified; (3) *environment variable issues*, where required variables are not set in the container; (4) *path or volume errors*, involving incorrect file paths or missing volume mounts; and (5) *YAML schema errors*, where the Argo Workflow fails `argo lint` validation. For each category, we report frequency and (for the Multi-Agent configuration) self-healing recovery rate.

4.3 Results

4.3.1 Overall Performance.

Table 1 summarizes functional correctness and end-to-end execution on a local k3d cluster across 17 repositories. GraphRAG outperforms the

Table 1. Overall performance across 17 repositories. End-to-end success indicates successful execution on a local k3d Kubernetes cluster.

Metric	Naive RAG	GraphRAG (Single)	GraphRAG (Multi)
Docker Build Success	35% (6/17)	59% (10/17)	76% (13/17)
Container Test Success	29% (5/17)	53% (9/17)	71% (12/17)
Argo Validity	41% (7/17)	65% (11/17)	82% (14/17)
End-to-End Success (k3d)	24% (4/17)	47% (8/17)	65% (11/17)
Self-Healing Effectiveness	—	—	40% (4/10)

Table 2. End-to-end success on k3d by repository category for GraphRAG (Multi-Agent).

Category	Success Rate	Count
ETL Pipelines	80%	4/5
ML Training Pipelines	67%	2/3
Complex Data Processing	50%	2/4
LLM-based Gen. & Eval.	60%	3/5

vector-only baseline, with the multi-agent configuration achieving the highest end-to-end executability.

GraphRAG consistently outperforms Naive RAG on Docker build success, container import sanity checks, and Argo workflow validity (Table 1). This suggests that retrieving context through explicit repository structure (e.g., import relations and module organization) helps the system infer dependencies and execution entry points more reliably than chunk-level similarity alone.

While GraphRAG improves artifact correctness in configurations, the Multi-Agent pipeline yields the best end-to-end success on k3d (65%, 11/17; Table 1). A likely explanation is that decomposing the task into specialized agents reduces compounding errors that arise when generating Dockerfiles, RunSpecs, and orchestration specifications in a single pass, while also enabling targeted retrieval queries at each stage.

Self-healing recovers a subset of initially failing repositories: Multi-Agent succeeds on 7/17 repositories on the first attempt and reaches 11/17 after retries (Table 3). Improvements saturate after two retries, indicating that many recoverable failures correspond to explicit, log-detectable issues (e.g., missing dependencies), whereas remaining failures are more likely driven by implicit configuration requirements that are not directly observable from source code alone.

4.3.2 Performance by Repository Category.

Table 2 reports end-to-end success on k3d for GraphRAG (Multi-Agent) by repository category. ETL pipelines achieve the highest success rate, consistent with their relatively standardized I/O patterns and dependency stacks. In contrast, complex data processing and LLM-based generation pipelines exhibit lower success rates, reflecting greater variability in runtime assumptions and external integration points. One possible explanation is that ETL-style workflows follow common architectural patterns (e.g., data ingestion, transformation, and persistence stages) that may be more frequently represented in LLM training data, making their structure easier for the system to infer.

Table 3. Cumulative end-to-end success on k3d by retry attempt for GraphRAG (Multi-Agent), $n = 17$.

Attempt	Cumulative Successes	Cumulative Rate
0 (first attempt)	7/17	41%
1 (after 1 retry)	10/17	59%
2 (after 2 retries)	11/17	65%
3 (after 3 retries)	11/17	65%

Table 4. Failure mode taxonomy with frequency and self-healing recovery rate (GraphRAG Multi-Agent). Frequency counts all observed failures across attempts; some repositories exhibited multiple failure types.

Error Category	Frequency	Recovery Rate
Missing dependencies	8	50%
Wrong entrypoint	3	33%
Environment variables	2	0%
Path/volume errors	2	50%
YAML schema errors	1	100%

4.3.3 Self-Healing Dynamics.

Table 3 shows cumulative end-to-end success for GraphRAG (Multi-Agent) as a function of retry attempt. Most recoveries occur within the first retry, and no additional end-to-end improvements are observed after the second retry. This pattern supports the view that self-healing is most effective when failures provide direct corrective signals (e.g., missing packages, simple configuration defects), but is less effective for failures whose causes are underspecified or require domain knowledge beyond repository artifacts.

4.3.4 Failure Mode Analysis.

Table 4 summarizes observed failure modes, their frequencies, and their recovery rates under self-healing. Missing dependencies are the most frequent failure mode and are partially recoverable, consistent with error logs providing actionable signals. By contrast, environment-variable-related failures are not recovered in this evaluation, suggesting that such requirements are often implicit (e.g., documented externally or assumed in deployment environments) and therefore difficult to infer and correct automatically. The failure distribution aligns with the retry saturation behavior: recoverable issues tend to be those that yield unambiguous error messages, while persistent failures reflect implicit environment and invocation assumptions.

Our failure taxonomy also suggests opportunities for lightweight human guidance. For example, developers could include simple hints such as explicit entry-point files, dependency manifests (e.g., `pyproject.toml`), or configuration templates. Such signals could be incorporated into the retrieval stage to reduce entry-point inference errors and dependency resolution failures.

4.3.5 Summary.

Across 17 repositories, GraphRAG improves artifact correctness over vector-only retrieval, and the Multi-Agent configuration achieves 65% (11/17) end-to-end success on k3d (Table 1). Self-healing increases cumulative success from 41% (7/17) on the first attempt to 65% (11/17) after retries, with most gains achieved within two retries (Table 3). Remaining failures are dominated by missing dependencies, entrypoint inference, and implicit environment configuration requirements (Table 4).

5 Conclusion

In this paper, we examine the challenge of transforming source code repositories into deployable artifacts such as container images and Kubernetes-native workflows. We propose a multi-agent GraphRAG framework that builds an AST-grounded KG, performs structure-aware retrieval, and coordinates specialized agents to generate Dockerfiles, RunSpecs, and Argo Workflows with an iterative self-healing mechanism. Across 17 GitHub repositories, our approach outperformed a vector-based RAG baseline, achieving 65% end-to-end success on a Kubernetes cluster, compared to 24% for vector-only RAG and 47% for a single-agent GraphRAG variant.

Self-healing recovered 40% of repositories that failed initially, with improvements saturating after additional retries. Most failures were due to missing dependencies and incorrect entry-point inference, while implicit environment-variable requirements proved especially difficult to resolve. These findings suggest that graph-centric retrieval combined with agent specialization improves deployability, but also highlight the limits of source-code-only inference when repositories rely on undocumented runtime assumptions or external services. Future work will extend the framework to multi-language settings and incorporate additional signals, such as documentation and logs.

References

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[2] Tuan-Dung Bui, Thanh Trong Vu, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. 2025. Correctness assessment of code generated by Large Language Models using internal representations. *Journal of Systems and Software* 230 (2025), 112570. doi:10.1016/j.jss.2025.112570

[3] Andrei-Alin Corodescu, Nikolay Nikolov, Akif Quddus Khan, Ahmet Soylu, Mihhail Matskin, Amir H Payberah, and Dumitru Roman. 2021. Big data workflows: Locality-aware orchestration using software containers. *Sensors* 21, 24 (2021), 8212.

[4] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[5] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Barcelona, Spain) *(KDD '24)*. Association for Computing Machinery, New York, NY, USA, 6491–6501. doi:10.1145/3637528.3671470

[6] Harald Foidl, Valentina Golendukhina, Rudolf Ramler, and Michael Felderer. 2024. Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers. *Journal of Systems and Software* 207 (2024), 111855.

[7] Tomáš Golis, Pavle Dakić, and Valentino Vranić. 2023. Automatic deployment to kubernetes cluster by applying a new learning tool and learning processes. *SQAMIA 2023 Software Quality Analysis, Monitoring, Improvement, and Applications* 1613 (2023), 0073.

[8] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. 2024. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309* (2024).

[9] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas Reps. 2021. Shipwright: A human-in-the-loop system for dockerfile repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1148–1160.

[10] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 328–338.

[11] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology* (2024).

[12] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3747588 Just Accepted.

[13] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3747588 Just Accepted.

[14] 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* (2023).

[15] Ehsan Azizi Khadem and Ali Movaghar. 2025. From challenges to metrics: An LLM-driven DevOps recommendation system grounded in evidence-based mappings. *Array* 28 (2025), 100547. doi:10.1016/j.array.2025.100547

[16] 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 (2020), 9459–9474.

[17] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003* (2024).

[18] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025. Codexgraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 142–160.

[19] Anthony Mbata, Yaji Sripada, and Mingjun Zhong. 2024. A survey of pipeline tools for data engineering. *arXiv preprint arXiv:2406.08335* (2024).

[20] JC Ogeawuchi, AC Uzoka, CE Alozie, OA Agboola, TP Gbenle, and S Owoade. 2022. Systematic review of data orchestration and workflow

automation in modern data engineering for scalable business intelligence. *International Journal of Social Science Exceptional Research* 1, 1 (2022), 283–290.

[21] Yun Peng, Ruida Hu, Ruoke Wang, Cuiyun Gao, Shuqing Li, and Michael R Lyu. 2024. Less is more? an empirical study on configuration issues in python pypi ecosystem. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–12.

[22] Huy Nhat Phan, Hoang Nhat Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. Repohyper: Better context retrieval is all you need for repository-level code completion. *CoRR* (2024).

[23] Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Tian J Wang. 2020. Modelling data pipelines. In *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 13–20.

[24] G Rosa, A Mastropaolo, S Scalabrino, G Bavota, R Oliveto, et al. 2023. Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises. In *2023 IEEE/ACM International Conference on Software and System Processes (ICSSP)*. 1–12.

[25] Pratik Shah, Rajat Ghosh, Aryan Singhal, and Debojyoti Dutta. 2025. RANGER–Repository-Level Agent for Graph-Enhanced Retrieval. *arXiv preprint arXiv:2509.25257* (2025).

[26] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 3784–3803.

[27] Shirin Tahmasebi, Amirhossein Layegh, Nikolay Nikolov, Amir H. Payberah, Khoa Dinh, Vlado Mitrovic, Dumitru Roman, and Mihhail Matskin. 2022. Dataclouddsl: Textual and Visual Presentation of Big Data Pipelines. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. 1165–1171. doi:10.1109/COMPSAC54236.2022.00183

[28] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards understanding the characteristics of code generation errors made by large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 717–717.

[29] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models. arXiv:2406.08731 [cs.SE] https://arxiv.org/abs/2406.08731

[30] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-Bench: Can Retrieval Augment Code Generation?. In *Findings of the Association for Computational Linguistics: NAACL 2025*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 3199–3214. doi:10.18653/v1/2025.findings-naacl.176

[31] Hongtao Yang, Zhichen Xu, Sergey Yudin, and Andrew Davidson. 2025. Unlocking the Power of CI/CD for Data Pipelines in Distributed Data Warehouses. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4887–4895.

[32] Xinwei Yang, Zhaofeng Liu, Chen Huang, Jiashuai Zhang, Tong Zhang, Yifan Zhang, and Wenqiang Lei. 2025. ELABORATION: A Comprehensive Benchmark on Human-LLM Competitive Programming. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 59–104. doi:10.18653/v1/2025.acl-long.4

[33] Jerin Yasmin, Jiale Amber Wang, Yuan Tian, and Bram Adams. 2025. An empirical study of developers' challenges in implementing Workflows as Code: A case study on Apache Airflow. *Journal of Systems and Software* 219 (2025), 112248.

[34] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-based environment dependency inference for Python programs. In *Proceedings of the 44th International Conference on Software Engineering*. 1245–1256.

[35] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).