DomAgent: Leveraging Knowledge Graphs and Case-Based Reasoning for Domain-Specific Code Generation

Shuai Wang
Chalmers University of Technology
Gothenburg, Sweden
shuaiwa@chalmers.se

Dhasarathy Parthasarathy
Volvo Group
Gothenburg, Sweden
dhasarathy.parthasarathy@volvo.com

Robert Feldt
Chalmers University of Technology
Gothenburg, Sweden
robert.feldt@chalmers.se

Yinan Yu
Chalmers University of Technology
Gothenburg, Sweden
yinan@chalmers.se

ABSTRACT

Large language models (LLMs) have shown impressive capabilities in code generation. However, because most LLMs are trained on public domain corpora, directly applying them to real-world software development often yields low success rates, as these scenarios frequently require domain-specific knowledge. In particular, domain-specific tasks usually demand highly specialized solutions, which are often underrepresented or entirely absent in the training data of generic LLMs. To address this challenge, we propose DomAgent, an autonomous coding agent that bridges this gap by enabling LLMs to generate domain-adapted code through structured reasoning and targeted retrieval. A core component of DomAgent is DomRetriever, a novel retrieval module that emulates how humans learn domain-specific knowledge, by combining conceptual understanding with experiential examples. It dynamically integrates top-down knowledge-graph reasoning with bottom-up case-based reasoning, enabling iterative retrieval and synthesis of structured knowledge and representative cases to ensure contextual relevance and broad task coverage. DomRetriever can operate as part of DomAgent or independently with any LLM for flexible domain adaptation. We evaluate DomAgent on an open benchmark dataset in the data science domain (DS-1000) and further apply it to real-world truck software development tasks. Experimental results show that DomAgent significantly enhances domain-specific code generation, enabling small open-source models to close much of the performance gap with large proprietary LLMs in complex, real-world applications. The code is available at: https://github.com/Wangshuaiia/DomAgent.

KEYWORDS

Large Language Models, Domain-Specific Code Generation, Knowledge Graph, Agent-Based Systems

ACM Reference Format:
Shuai Wang, Dhasarathy Parthasarathy, Robert Feldt, and Yinan Yu. 2026. DomAgent: Leveraging Knowledge Graphs and Case-Based Reasoning for Domain-Specific Code Generation. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 10 pages.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus.

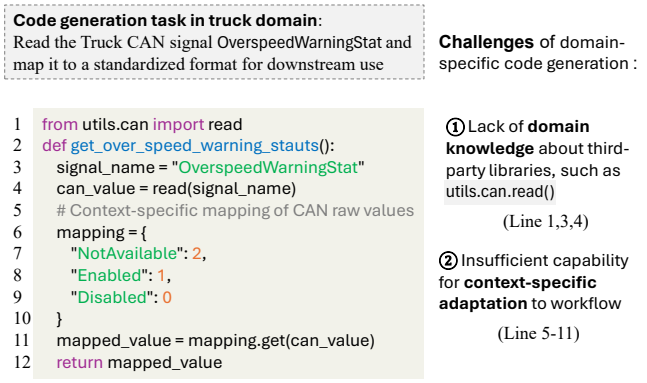

Figure 1: Example of a domain-specific code generation task in truck software development: reading the CAN signal `OverspeedWarningStat` and adapting the function with context-specific mapping to fit the workflow.

1 INTRODUCTION

Recently, large language models (LLMs) have made remarkable progress in the field of code generation. Both closed-source models such as GPT-4o and Claude Sonnet 4.5, and open-source models like CodeLlama [30], Mistral [16], DeepSeek-Coder [11], and Qwen-Coder [15] have demonstrated impressive capabilities, and some of them even surpass human performance in general benchmark tasks. Tools like GitHub Copilot now leverage these models to provide adaptive, real-time coding assistance across various programming environments.

In real-world applications, programming tasks are often highly domain-specific [20, 38]. Software development across different industries, such as truck control systems or database management [14], typically requires code that is tailored to the distinct requirements and constraints of each domain. While existing LLMs

perform well on general-purpose, open-domain tasks, their performance degrades significantly when faced with specialized, domain-specific problems [9]. One key reason is that LLMs often lack the required information to accomplish specialized tasks. Domain-specific programming demands not only *domain knowledge*, such as understanding how to use industry-specific libraries, but also the ability to produce *specialized solutions* aligned with the operational logic of the target system. Figure 1 illustrates an example from truck software development: generating Python code to read the CAN signal `OverspeedWarningStat` and adapting the function with context-specific mapping to fit the workflow. Directly using general LLMs poses two challenges: (1) a lack of knowledge about third-party libraries, and (2) insufficient capability for context-specific adaptation to the workflow.

A straightforward way to address this challenge is to fine-tune LLMs with domain-specific data [1, 8, 40]. However, relying on fine-tuning to equip LLMs with domain-specific knowledge is often impractical, as fine-tuning introduces high development and maintenance costs [5]. The resulting models often struggle to adapt to constantly updated libraries. A more efficient and flexible solution is Retrieval-Augmented Generation (RAG) [18], which augments existing LLMs with external knowledge without modifying model weights. Popular software development frameworks such as AutoGen [44], LangChain and TaskWeaver [27] leverage this strategy to quickly integrate LLMs with domain knowledge at low cost. Moreover, recent studies [26, 32] show that combining lightweight fine-tuning with RAG, using only a small number of domain examples, can substantially enhance model adaptability and accuracy for specialized programming tasks, achieving results comparable to much larger, fully fine-tuned systems.

In practice, RAG-based solutions dominate many real-world applications, accounting for over half (about 51%) of deployed LLM-powered software systems [34]. To provide *domain knowledge* efficiently, recent work has shown that even basic retrieval methods like BM25 can significantly enhance performance by appending top-K results from external knowledge bases to prompts [45]. Using sentence embeddings for retrieval can also improve the retrieval accuracy [19, 46]. However, excessively long retrieved content can negatively impact performance, as extended contexts may obscure key information and mislead the model [3]. Segmenting candidate texts into smaller chunks of 200–800 tokens is a way to solve this issue [41]. *Zhu et al.* [49] improved the segmenting by using structural cues such as abstract syntax trees (ASTs) to produce more meaningful splits [49]. Using another LLMs to filter out irrelevant information and compress the retrieved content also mitigate it [7]. To introduce structured and concise knowledge, knowledge graphs (KGs) have also been explored as retrieval sources for code repair [25]. KGs also contain the dependencies between software modules, supporting repository-level code generation [2]. However, as these methods largely depend on plain textual similarity and neglect explicit links to relevant packages and functions, precisely retrieving relevant knowledge remains a fundamental challenge in complex real-world settings.

Bottom-up retrieval: For domain-specific tasks that require specialized knowledge and domain logic, a common approach is to leverage case-based reasoning (CBR), where a few-shot examples is included in the prompt to guide LLMs toward solving domain-adapted tasks [13, 42]. Studies have shown that a fixed set of examples often fails to cover diverse scenarios. A more effective approach is to dynamically select examples based on the given task, using text embeddings to compute similarity scores [4]. Subsequent work refined the retrieval process by re-ranking candidates through AST analysis to better capture code-level relationships [24]. To broaden the case coverage, *Tan et al.* [33] queried multiple information sources and integrated these retrieved contents through a segmented prompting strategy. Constructing high-quality case libraries from large training datasets is an effective way to improve case coverage [12]. However, in real-world scenarios, creating a large and comprehensive case base is labor-intensive and time-consuming. Achieving high coverage with a small set of examples is still an important and practical challenge.

Top-down retrieval: Just as human learning also relies on structured understanding, rules, relationships, and conceptual hierarchies, LLMs require top-down knowledge to complement bottom-up example learning. KGs provide such structure by representing packages, functions, and their interrelations as nodes and edges. This structure makes it straightforward to link code examples to KG based on the package calling. For instance, as shown in Figure 1, a code snippet calling `utils.can.read` can be explicitly connected to its corresponding node in the KG. Such links allow retrieving domain knowledge and locating relevant code examples beyond plain text similarity, by directly matching concrete package or function usage. In addition, explicitly evaluating the coverage of KG nodes by the packages and functions used in cases facilitates building a compact, diverse, and coverage-rich candidate case base.

Based on the above insights, we propose an agent system for domain-specific code generation that integrates both bottom-up (case-based) and top-down (knowledge-based) retrieval into a unified reasoning framework. The system is designed to emulate the human learning process: combining experiential reasoning from examples with conceptual understanding from structured knowledge. More specifically, the system leverages KGs as external repositories to provide accurate domain knowledge. To improve task coverage with a small case set, we exploit relational information in the KG to guide case selection, ensuring diversity by covering as many functions in the graph as possible. At the core of our system is a new retrieval module, DomRetriever, which dynamically integrates top-down and bottom-up retrieval to enhance both knowledge and case selection. It enables the model to iteratively identify, refine, and combine relevant knowledge with representative cases. Experiments on a benchmark dataset on data science domain show that our method significantly outperforms similar size LLMs. We also deploy the system in real-world truck software development tasks, demonstrating its robustness in practical settings.

Our contributions are as follows:

- We propose the first agent system (DomAgent) for domain-specific code generation that integrates structured knowledge (top-down) with case-based reasoning (bottom-up), improving both information retrieval and code generation.
- We design a KG-guided case selction method that exploits structural relations in the knowledge graph to achieve broad task coverage with a small set of cases.

- We develop a novel retrieval module (DomRetriever) that unifies knowledge-graph-guided retrieval and case-based reasoning into a dynamic, bidirectional process. DomRetriever can be seamlessly integrated into any LLM pipeline for flexible domain adaptation.
- We validate the agent on a benchmark dataset and in real-world truck software development, demonstrating its effectiveness.

2 RELATED WORK

2.1 Knowledge Retrieval for Code Generation

RAG has become one of the most effective strategies to equip LLMs with external domain knowledge without modifying their parameters [18, 37, 39]. By retrieving relevant information from external repositories, the model can access up-to-date and task-specific context to enhance code generation performance in specialized domains [48]. Yang et al. [45] conducted an empirical study showing that using basic retrieval methods, such as BM25, to fetch relevant content from external knowledge bases and simply appending the top-K results to the prompt can already yield strong performance. Li et al. [19] further enhanced retrieval accuracy by employing sentence embeddings for semantic search, enabling more precise matching between queries and relevant documents. However, long retrieved texts may overwhelm the model's attention and obscure key information, leading to degraded generation quality [3]. To mitigate this issue, Wang et al. [41] proposed splitting candidate texts into smaller chunks of 200–800 tokens, while Zhu et al. [49] leveraged structural cues from ASTs to produce more meaningful segmentations. Gao et al. [7] introduced a secondary LLM to filter irrelevant content and compress retrieved texts, ensuring that only the most salient information is retained. Beyond plain-text retrieval, KGs offer a structured and concise way to represent domain knowledge. KGs can capture the dependencies between entities, which can be used to facilitate repository-level code generation [35, 36]. Ouyang et al. [25] demonstrated that using KGs as retrieval sources can effectively provide relevant information for code repair tasks. Compared to unstructured text, KGs provide explicit relationships among entities such as packages, functions, and classes, making them particularly suitable for retrieving domain-specific programming knowledge.

2.2 In-Context Learning and Case-Based Reasoning

While retrieval-based methods focus on supplementing external knowledge, another effective strategy for domain-specific code generation is to provide task-specific guidance directly through examples. ICL [6] and CBR [13, 42] enable LLMs to learn from a few representative examples embedded in the prompt, helping them infer specialized problem-solving patterns without explicit model updates. A common limitation of fixed few-shot prompts is their inability to generalize across diverse scenarios. Dannenhauer et al. [4] addressed this issue by building a case base and dynamically selecting examples based on semantic similarity, where both the query and cases were encoded into embeddings for retrieval. To further refine the selection, Nashid et al. [24] re-ranked candidate cases using AST analysis, which captures code-level structural relations that pure text similarity may miss. Tan et al. [33] proposed a segmented prompting strategy that integrates retrieved content from multiple sources to enhance contextual coverage. In addition, Guo et al. [12] constructed a large and high-quality case base from extensive training data, and iteratively optimized it through selection and execution feedback to retain the most effective examples. Despite these advances, maintaining large case bases is labor-intensive and costly, making it impractical for fast-evolving industrial environments.

3 PROBLEM DEFINATION

We address the problem of domain-specific code generation by leveraging structured knowledge and reusable examples. Given a domain-specific generation task q, a KG $\mathcal{G}$, and a case base $\mathcal{B}$, the objective is to retrieve the necessary domain knowledge from $\mathcal{G}$ and identify relevant cases from $\mathcal{B}$, and then synthesize the target code $\hat{y}$ by integrating these retrieved resources.

Formally, we consider a KG $\mathcal{G} = \{\langle e, r, e' \rangle \mid e, e' \in \mathcal{E}, r \in \mathcal{R}\}$, where $\mathcal{E}$ and $\mathcal{R}$ represent the sets of entities and relations, respectively. Each triple $\langle e, r, e' \rangle$ encodes a relationship r between entity e and entity e'. In our setting, entities represent domain knowledge such as packages, functions, their descriptions, and parameters, while edges represent direct relationships, e.g., a function belonging to a package.

The case base $\mathcal{B}$ is a collection of reference code cases. Its construction should ensure coverage and diversity while minimizing manual effort, since creating cases is costly and time-consuming. Our goal is to build a small but representative set of high-quality cases that achieves broad coverage.

4 METHODS

Our approach, illustrated in Figure 2, comprises three key components. (1) *Case Base Construction* module leverages the invocation relationships between functions and code cases in the knowledge graph to explicitly and efficiently sample representative cases; (2) *Retrieval-Augmented Code Generation* module unifys knowledge and case retrieval through a reasoning LLM to enhance retrieval effectiveness; (3) For *Agent Training*, we employ reinforcement learning to train the reasoning LLM to autonomously invoke retrieval tools, improving its ability to reason and access relevant knowledge when solving the coding task.

4.1 Hierarchical Case Selection with Knowledge Graph Guidance

To construct a representative case base that covers both diverse packages and their functional usage patterns, we leverage the hierarchical structure of the KG to perform hierarchical case selection.

Package-level anchoring. Suppose there are n software packages $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ represented in the KG. For each package p_i, we take its *root node* in the KG as an *anchor* node a_i. Let $C(a_i) = \{c_{i1}, c_{i2}, \ldots\}$ denote the set of its child nodes (e.g., functions and attributes belonging to package p_i). Each child node c_{ij} is represented by its textual name t_{ij} concatenated with its description s_{ij}, and then encoded into a continuous semantic vector space

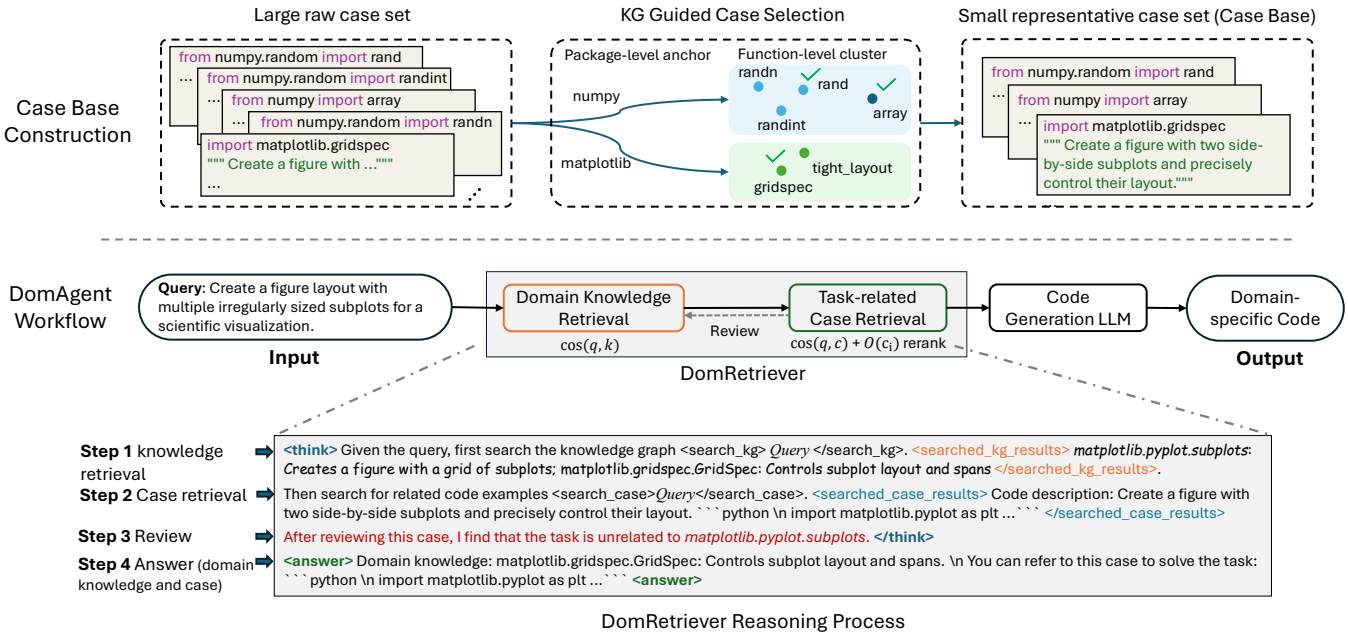

Figure 2: Case base construction (top) and the overall workflow of our DomAgent (bottom). We first perform KG–guided selection to obtain representative cases. Then, our DomAgent retrieves domain knowledge from a KG and cases from the case base, while reviewing the retrieved domain knowledge in the reasoning process based on the selected case. Finally, using the answer of DomAgent, LLM generates the domain-specific code.

through an embedding function:

$$\mathbf{v}_{ij} = f_{\text{embed}}(t_{ij} \,\|\, s_{ij}) \in \mathbb{R}^d \tag{1}$$

where $\|$ denotes string concatenation.

Function-level clustering. For each package p_i, we apply k-means clustering to its embedded child nodes:

$$\{\mathbf{v}_{ij}\} \xrightarrow[\text{k-means}]{} \mathcal{K}_i = \{K_{i1}, K_{i2}, \ldots, K_{im}\} \tag{2}$$

where m denotes the number of clusters per package. Each cluster K_{ik} corresponds to a semantically coherent functional usage pattern.

Coverage-driven case selection. Each candidate code case c imports some packages $p(x)$ and functions $F(x)$. Our goal is to build a case base $\mathcal{B}$ that: 1) covers as many packages as possible, and 2) maximizes the diversity of functional usage within each package.

Let $n' = |\{p(x) \mid x \in \mathcal{B}\}|$, $m' = |\{K_{ik} \mid \exists x \in \mathcal{B},\, F(x) \cap K_{ik} \neq \emptyset\}|$ be the number of distinct packages and the number of covered clusters in the current case base, respectively. The *coverage ratios* are then defined as

$$\alpha = \frac{n'}{n}, \quad \beta = \frac{m'}{m} \tag{3}$$

We iteratively traverse the candidate code cases. For each candidate x, we test whether adding it to $\mathcal{B}$ increases either package coverage or cluster coverage:

$$\Delta\alpha = \frac{n'_{\text{new}}}{n} - \alpha, \quad \Delta\beta = \frac{m'_{\text{new}}}{m} - \beta \tag{4}$$

If $\Delta\alpha > 0$ or $\Delta\beta > 0$, the case is added to $\mathcal{B}$; otherwise, it is discarded as redundant.

Stopping criterion. We define two thresholds τ_1 and τ_2 to control the case selection process. The construction stops when

$$\alpha = \frac{n'}{n} > \tau_1 \quad \text{and} \quad \beta = \frac{m'}{m} > \tau_2 \tag{5}$$

i.e., when both package-level and cluster-level coverage exceed predefined acceptable levels. This strategy ensures that each package is well represented while preserving the diversity of functional usage patterns within packages.

In addition, before storing the code to the case base, we execute it to ensure that it runs correctly. We store these code cases in a vector database, where the key is the natural language description of the corresponding task to support similarity-based task retrieval.

4.2 Domain-Specific Code Generation Agent

4.2.1 ***DomRetriever.*** To generate domain-specific code, we retrieve both relevant domain knowledge and representative task-related cases.

Top-down retrieval. Given a task description q, we first determine which software package should be utilized. Specifically, we use an LLM as the classifier, and the inputs are both q and the textual description t_p of each candidate package. The LLM outputs a binary classification:

$$\hat{p}_i \;=\; LLM_P(q \,||\, t_p) \in \{0, 1\} \tag{6}$$

where $\hat{p}_i = 1$ indicates that p_i is applicable to the task q.

Then, we encode the task q with f_{embed} to obtain $\mathbf{q} = f_{\text{embed}}(q) \in \mathbb{R}^d$. Then we compute the cosine similarity between the encoded

vector $f_{\text{embed}}(q)$ and the embeddings of domain knowledge equations $\mathbf{v}_{ij}$ (as defined in Eq. 1):

$$s_{ij}(q) = \cos(\mathbf{q}, \mathbf{v}_{ij}) = \frac{\mathbf{q}^\top \mathbf{v}_{ij}}{\|\mathbf{q}\|_2 \, \|\mathbf{v}_{ij}\|_2} \tag{7}$$

We select the nodes with the highest T similarity scores as the relevant domain knowledge list $\mathcal{K} = \{k_1, k_2, \ldots, k_T\}$.

Bottom-up retrieval. We first compute the semantic similarity between the task embedding $f_{\text{embed}}(q)$ and the natural language descriptions of candidate code cases. The top-R most similar cases are selected to form an initial case list $C = \{c_1, c_2, \ldots, c_R\}$. To refine this list, we further focus on the similarity between cases and the retrieved packages (or functions, which we refer to as packages for simplicity in the following part). Specifically, for each case c_i, we compute the overlap count between the packages used in c_i and those contained in $\mathcal{K}$:

$$O(c_i) = |\text{Packages}(c_i) \cap \text{Packages}(\mathcal{K})| \tag{8}$$

Then we select the top case c^* according to $O(c_i)$.

LLM-guided refinement with tool-use in the reasoning stage. To further enhance the relevance and completeness of the domain knowledge $\mathcal{K}$, we compare the selected case c^* with $\mathcal{K}$, using a reasoning LLM as an agent to automatically refine $\mathcal{K}$ by removing irrelevant items or retrieving supplementary knowledge based on c^*. As illustrated in the example in Figure 2, during the review process, knowledge that is only superficially semantically related but functionally irrelevant can be filtered out. To achieve this, both the domain knowledge retrieval and case retrieval processes are encapsulated as API tools, denoted as $\text{SearchKG}(q)$ and $\text{SearchCase}(q)$ respectively, which are invoked during the reasoning phase of the LLM.

As shown in Figure 2, we design two special tokens <search_kg> and <search_case> to trigger these searches respectively. During the reasoning stage, similar to DeepSeek-R1 [10], the LLM's internal reasoning is enclosed within the tag <think> $\cdots$ </think>. In the subsequent answering stage (denoted by <answer> $\cdots$ </answer>), the LLM outputs the finalized domain knowledge $\widehat{\mathcal{K}}$ together with the specialized solution $\hat{c}$.

4.2.2 ***Code Generation.*** We concatenate the original task description q, the refined domain knowledge $\widehat{\mathcal{K}}$, and the specialized solution $\hat{c}$ to form the final prompt, which is then fed into another $\text{LLM}_{\text{gen}}(\cdot)$ to generate the target code $\hat{y}$. Formally, the process can be expressed as:

$$\hat{y} = \text{LLM}_{\text{gen}}\big(q \,||\, \widehat{\mathcal{K}} \,||\, \hat{c}\big) \tag{9}$$

In this module, we can either use the same reasoning LLM as DomRetriever for code generation or choose a more powerful LLM to assist with it.

4.3 Agent Training

To enable the LLM to use retrieval tools, that is, to generate the special tokens <search_kg> and <search_case> for triggering retrieval, we constructed fine-tuning examples so that the LLM learns to produce these tokens. Specifically, we manually created several few-shot examples and concatenated them into long chain-of-thought (CoT) sequences. Then, given a query, we leveraged a powerful LLM (e.g., GPT-5) to generate reasoning steps and answers, like the reasoning process in Figure 2. Finally, we fine-tuned our LLM on this synthesized data to teach it to generate the retrieval-triggering tokens.

Based on this, we further apply reinforcement learning (RL) to guide the LLM to review the contents retrieved by the two tools and decide whether to filter out irrelevant knowledge. The final answer should include only the most relevant knowledge and cases. To provide a learning signal, we train a reward model f_r using a powerful external LLM. This model assesses the relevance of a query q to the retrieved knowledge $\widehat{\mathcal{K}}$ and cases $\hat{c}$:

$$\text{reward} = f_r\big(q \,||\, \widehat{\mathcal{K}} \,||\, \hat{c}\big). \tag{10}$$

The obtained reward serves as the feedback signal for training the agent LLM. We employ the classical RL optimization algorithm GRPO [31] to update the model parameters with respect to the reward function:

$$\theta^* = \arg\max_\theta \; \mathbb{E}_{q, \widehat{\mathcal{K}}, \hat{c} \sim \pi_\theta}\Big[f_r(q \,||\, \widehat{\mathcal{K}} \,||\, \hat{c})\Big], \tag{11}$$

where π_θ denotes the policy of the LLM parameterized by θ.

5 EXPERIMENTS

5.1 Experimental Setup

In this paper, we conduct experiments and ablation studies on two datasets: (1) the open benchmark dataset *DS-1000*, and (2) a real-world domain-specific code generation dataset, *Truck CAN Signal*.

Benchmark dataset. We first conducted experiments on the **DS-1000** benchmark dataset [17] in the data science domain. DS-1000 contains 1,000 tasks derived from 451 distinct Stack Overflow questions, covering seven widely used data science packages: *NumPy, Pandas, Matplotlib, SciPy, scikit-learn, TensorFlow*, and *PyTorch*. The original questions are slightly modified to differ from their Stack Overflow sources, preventing LLMs from solving them by simply memorizing pre-training data. Because the tasks require calling and correctly using various functions across these domain libraries, they remain challenging even for state-of-the-art LLMs such as GPT-4o.

We use a knowledge graph **DS-KG** for data science packages which is constructed by Ouyang *et al.* [25]. Each library function is represented as an entity, linking with attributes such as *name* and *description*. They mined the official documentation of major data science libraries and obtained 505,640 triples in total.

Application in real-world truck software development. We further apply our agent for CAN signal reading and writing, which is a fundamental task in truck API development. As shown in Figure 1, working with CAN signals requires understanding third-party packages for CAN message access and applying specific signal transformations to ensure consistency across upstream and downstream systems. We leverage the truck manufacturer's internal CAN signal documentation to provide domain knowledge. In total, we use 776 CAN signals spanning six functional domains: *Driver Productivity, Connected Systems, Energy, Vehicle Systems, Visibility*, and *Dynamics*.

Implementation Details. We conduct experiments using two LLMs as the backbone of our agent: *LLaMA-3.1-8B-Instruct* and *Qwen-2.5-7B*. We construct a set of 500 high-quality examples in open-domain task to teach the models how to invoke these retrieval tools. Once the model learns to use the retrieval tool, we apply it directly to our tasks. We sample only 300 examples (30%) from the DS-1000 dataset to construct the case bases, and use the remaining 700 examples for testing. We use sentence-BERT [28] as the language encoding tool, with the hyperparameters τ_1 and τ_2 both set to 0.9. The training is performed on 8 NVIDIA A100 80G GPUs in total. For each input query, we generate 16 outputs (rollouts). We train for 2 epochs with a batch size of 16 and a learning rate of 1e−6. DS-1000 provides predefined code contexts and testing functions for evaluation, while in the CAN signal reading and writing setting, engineers authored the corresponding test functions. Thus, we executed these test functions and adopted pass@1 as the metric.

5.2 Main Results

5.2.1 DS-1000. Table 3 presents a comprehensive comparison across multiple models and code-generation frameworks on the DS-1000 benchmark dataset. Among the vanilla LLMs, GPT-4o achieves the highest overall pass@1 accuracy (51.0%), demonstrating strong general-purpose reasoning ability but revealing limited domain adaptation for specialized deep learning libraries, such as Pytorch and Tensorflow. Vanilla open-source models such as Qwen2.5-7B and LLaMA3.1-8B perform moderately, highlighting the performance disparity between proprietary and open models in zero-shot code generation settings.

Within the coding-agent family, models like WizardCoder and Magicoder show clear improvements over standard LLMs through specialized fine-tuning on code-related data. However, even the strongest variant, MagicoderS-CL, achieves a total score of only 37.5%, indicating that static code fine-tuning alone cannot fully generalize across diverse data science tasks. It is worth noting that MagicoderS-CL employs a two-stage large-scale fine-tuning strategy: it is initially trained on 75K carefully curated synthesized programming instruction data and subsequently fine-tuned on the 110K open-source complex instruction dataset Evol-Instruct. Such an approach requires large amounts of data and computational resources, making it impractical in real-world scenarios.

Our proposed DomAgent framework substantially improves the performance of small open-source models. DomAgent (Qwen2.5-7B) and DomAgent (LLaMA3.1-8B) reach 39.2% and 40.5%, respectively, outperforming all previous coding agents by approximately +2.0 to +3.0 percentage points. When paired with large external models for code generation, DomRetriever further enhances performance. For instance, coupling LLaMA3.1-8B (DomRetriever) with GPT-4o as the code generation model yields the best result at 58.6%, representing a +7.6% gain over GPT-4o alone.

5.2.2 Truck CAN Signal. Table 2 compares the performance of large proprietary GPT-4o and small open-source models Qwen2.5-7B across six highly specialized domains. Despite its strong general capability, the proprietary GPT-4o performs unsatisfactorily on these highly domain-specific code generation tasks, reaching an overall score of 71.22%. When enhanced with DomRetriever, its performance increases to 98.04%, highlighting the effectiveness of DomRetriever. This also ensures that our agent can operate effectively within real-world truck software development workflows. In addition, the DomAgent, built on the small open-source Qwen2.5-7B model, achieves 96.64%, nearly matching GPT-4o with DomRetriever and vastly outperforming its vanilla baseline (39.62%). These results demonstrate that domain-adaptive architectures such as DomAgent can effectively bridge the performance gap between proprietary large models and small open-source models in complex, real-world industrial tasks.

5.3 Ablation Study

We formulate our ablation design to understand how each component in our proposed framework contributes to domain-specific code generation across diverse data science libraries and real-world expert domains.

5.3.1 DS-1000. We first compare the configurations with knowledge grounding (*+KG*) and case-based reasoning (*+CBR*), and their combination with a standard RAG mechanism *+KG+CBR (RAG pipeline)* to quantify how each component contributes individually and jointly. The results show that while knowledge grounding improves baseline performance, case-based reasoning provides larger gains. The combination of *+KG+CBR* in a standard RAG pipeline further enhances overall performance, with improvements roughly additive to the individual contributions. Further, comparing *+KG+CBR (RAG pipeline)* and *+KG+CBR (DomAgent)* reveals that DomAgent's retrieval mechanism (cf. Section 4.2.1) consistently yield improvements across both Qwen2.5-7B and LLaMA3.1-8B, achieving competitive results with *LLaMA3.3-70B*, particularly in specialized tasks requiring deeper logical composition and multi-step data manipulation, such as operations in Pandas and Tensorflow. Moreover, we study the synergy between *DomRetriever* and large external code-generation models. Combining small retrieval models (e.g., LLaMA3.1-8B or Qwen2.5-7B) with strong generators (e.g., GPT-4o or LLaMA3.3-70B) consistently improves performance across libraries.

Case-based reasoning is an important step in DomAgent. Since constructing a high-quality case base is both expensive and time-consuming, an efficient strategy is crucial. To evaluate the effectiveness of our Hierarchical Case Selection with Knowledge Graph Guidance introduced in Section 4.1, we conduct an ablation study by gradually increasing the proportion of sampled cases. As shown in Figure 3, performance (pass@1) steadily improves with larger sampling proportions for both strategies, but the KG-guided case selection consistently outperforms random sampling across all proportions. The improvement is most pronounced in the low-sampling regime (achieving near-full performance with only 30% of the data), demonstrating that knowledge-guided selection effectively identifies more representative and contextually diverse examples.

5.3.2 Truck CAN Signal. We conducted experiments on six types of truck CAN signals using our proposed DomAgent, which employs Qwen2.5-7B as its backbone. We also tested an alternative configuration that leverages GPT-4o as an external LLM for code generation. The results are summarized in Table 4.

Table 1: Performance comparison (pass@1) across different libraries and methods on Data Science dataset. All the results marked with an asterisk (*) are copied from the Magicoder paper [43].

Package Count	Size	Matplotlib 155	Numpy 220	Pytorch 291	Pandas 68	Scipy 106	Sklearn 115	Tensorflow 45	Total 1000
Baseline (Vanilla LLMs)									
Qwen2.5-7B	7B	54.2	34.9	25.3	16.8	23.1	21.4	29.7	29.3
LLaMA3.1-8b	8B	55.1	36.2	26.4	18.3	24.2	22.5	30.1	30.4
LLaMA3.3-70B	70B	60.2	40.0	36.5	37.8	40.1	42.0	43.5	40.3
GPT-3.5-Turbo	-	65.8	32.7	30.2	36.8	39.6	40.0	42.2	39.4
GPT-4o	-	65.2	56.8	41.9	47.1	48.1	50.4	46.7	51.0
Coding Agents									
StarCoder [21]	15B	51.7*	29.7*	21.4*	11.4*	20.2*	29.5*	24.5*	26.0*
WizardCoder [23]	15B	55.2*	33.6*	26.2*	16.7*	22.4*	24.9*	26.7*	29.2*
CodeLLaMA-Python [29]	7B	55.3*	34.5*	19.9*	16.4*	22.3*	17.6*	28.5*	28.0*
WizardCoder-CL [43]	7B	53.5*	34.4*	25.7*	15.2*	21.0*	24.5*	28.9*	28.4*
Magicoder-CL [43]	7B	54.6*	34.8*	24.7*	19.0*	25.0*	22.6*	28.9*	29.9*
MagicoderS-CL (large-scale fine-tuning) [43]	7B	55.9*	40.6*	**40.4***	28.4*	28.8*	**35.8***	**37.6***	37.5*
DomAgent (Qwen2.5-7B; **ours**)	7B	**62.5**	37.9	32.7	**33.5**	30.6	29.1	36.9	39.2
DomAgent (LaMA3.1-8B; **ours**)	8B	**62.5**	**43.1**	33.1	29.2	**31.5**	29.3	37.1	**40.5**
External LLM (Code Gen) Enhanced with DomRetriever									
LLaMA3.1-8B (DomRetriever) + LLaMA3.3-70B (Code Gen)	70B	63.5 (+3.3)	49.3 (+9.3)	44.6 (+8.1)	41.4 (+3.6)	44.0 (+3.9)	46.3 (+4.3)	49.2 (+5.7)	47.9 (+7.6)
LLaMA3.1-8B (DomRetriever) + GPT-4o (Code Gen)	-	68.6 (+3.4)	64.2 (+7.4)	50.8 (+8.9)	49.1 (+2.0)	50.7 (+2.6)	56.4 (+6.0)	53.3 (+6.6)	58.6 (+7.6)

Table 2: Comparison of small open models-based DomAgent (Qwen2.5-7B-based) against GPT-4o-based models for domain-specific code generation tasks: truck CAN signal reading/writing across six domains.

Domain	GPT-4o-based		Qwen2.5-7B-based (open-source)	
	Vanilla	+DomRetriever (Qwen2.5-7B)	Vanilla	DomAgent
Driver productivity	65.1	100	32.6	100
Connected systems	73.4	97.4	46.8	97.8
Energy	58.8	100	36.2	100
Vehicle system	71.1	97.1	35.1	94.3
Visibility	82.2	100	44.3	100
Dynamics	75.5	96.4	39.8	91.4
Total	**71.22**	**98.04**	**39.62**	**96.64**

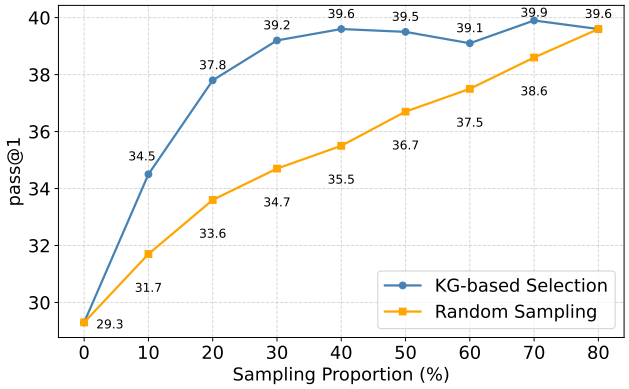

Figure 3: Effect of KG-based case selection versus random sampling with different sampling proportions on case base construction, using the Qwen2.5-7B-based DomAgent on the *Data Science* dataset.

The results clearly demonstrate that DomAgent significantly outperforms the vanilla models in both settings. Specifically, DomAgent with Qwen2.5-7B achieves a 57.02% improvement over the vanilla Qwen2.5-7B model, while DomAgent with GPT-4o achieves a 26.82% improvement. Overall, both models achieve strong final performance: Qwen2.5-7B reaches a pass@1 of 96.64, and GPT-4o reaches 98.04. Notably, DomAgent performs consistently well across all six CAN signal types. Both configurations even achieve 100% pass@1 on the *Driver Productivity*, *Energy*, and *Visibility* signal types, confirming the reliability of our method for industrial deployment.

By contrast, the vanilla models perform poorly, even though we manually curated representative cases and carefully tuned prompts. The vanilla Qwen2.5-7B only achieves 39.62 pass@1, while the stronger GPT-4o reaches 71.22 pass@1. This finding highlights the limitations of relying solely on fixed prompt engineering or static few-shot examples. Without injecting external domain knowledge and customizing the reasoning process, existing LLMs struggle to handle domain-specific code generation tasks effectively.

When we add domain knowledge (*+KG*), Qwen2.5-7B improves by 31.27 points, and GPT-4o improves by 16.58 points, both showing substantial gains. This improvement is mainly because our knowledge graph is constructed from internal CAN signal documentation, which contains descriptions of signal attributes and natural-language explanations of how to use essential third-party libraries. With this domain-specific knowledge, the models can accurately call internal CAN signal read/write tools. Interestingly, GPT-4o achieves 87.80 pass@1 using only *+KG*, reflecting its strong document understanding and code generation capabilities, allowing it to leverage documentation effectively even without explicit few-shot examples.

Adding case-based reasoning (*+CBR*) leads to an even larger improvement: Qwen2.5-7B +KG improves by an additional 44.95 points, and GPT-4o improves by 19.88 points. This is because retrieving highly relevant cases enables more effective few-shot learning, where the retrieved cases often include domain knowledge, such as concrete examples of how to invoke third-party libraries or convert CAN signals.

Table 3: Ablation study of different modules on the Data Science dataset.

Count	Size	Matplotlib 155	Numpy 220	Pytorch 291	Pandas 68	Scipy 106	Sklearn 115	Tensorflow 45	Total 1000
Ablation on End-to-end DomAgent									
Qwen2.5-7B	7B	54.2	34.9	25.3	16.8	23.1	21.4	29.7	29.3
Qwen2.5-7B+KG	7B	56.1 (+1.9)	37.5 (+2.6)	27.8 (+2.5)	18.4 (+1.6)	25.2 (+2.1)	23.5 (+2.1)	31.2 (+1.5)	31.4 (+2.1)
Qwen2.5-7B+CBR	7B	57.3 (+3.1)	32.9 (-2.0)	32.4 (+7.1)	24.6 (+7.8)	27.5 (+4.4)	28.5 (+7.1)	37.9 (+8.2)	34.8 (+5.5)
Qwen2.5-7B+KG+CBR (RAG pipeline)	7B	62.3 (+8.1)	40.9 (+6.0)	31.6 (+6.3)	23.8 (+7.0)	30.4 (+7.3)	27.3 (+5.9)	36.3 (+6.6)	37.4 (+8.1)
Qwen2.5-7B+KG+CBR (DomAgent)	7B	62.5 (+8.3)	37.9 (+3.0)	32.7 (+7.4)	33.5 (+16.7)	30.6 (+7.5)	29.1 (+7.7)	36.9 (+7.2)	39.2 (+9.9)
LLaMA3.1-8b	8B	55.1	36.2	26.4	18.3	24.2	22.5	30.1	30.4
LLaMA3.1-8b+KG	8B	57.2 (+2.1)	38.1 (+1.9)	28.6 (+2.2)	19.1 (+0.8)	26.0 (+1.8)	24.0 (+1.5)	32.5 (+2.4)	32.2 (+1.8)
LLaMA3.1-8b+CBR	8B	61.6 (+6.5)	40.1 (+3.9)	30.8 (+4.4)	26.6 (+8.3)	30.4 (+6.2)	28.7 (+6.2)	37.4 (+7.3)	37.6 (+7.2)
LLaMA3.1-8b+KG+CBR (RAG pipeline)	8B	60.4 (+5.3)	41.3 (+5.1)	32.6 (+6.2)	24.1 (+5.8)	30.6 (+6.4)	29.1 (+6.6)	36.7 (+6.6)	39.2 (+8.8)
LLaMA3.1-8b+KG+CBR (DomAgent)	8B	62.5 (+7.4)	43.1 (+6.9)	33.1 (+6.7)	29.2 (+10.9)	31.5 (+7.3)	29.3 (+6.8)	37.1 (+7.0)	40.5 (+10.1)
Ablation on External LLM (Code Gen) Enhanced with DomRetriever									
LLaMA3.3-70B	70B	60.2	40.0	36.5	37.8	40.1	42.0	43.5	40.3
LLaMA3.3-70B+KG	70B	61.0 (+0.8)	42.2 (+2.2)	37.0 (+0.5)	38.4 (+0.6)	41.0 (+0.9)	42.7 (+0.7)	44.2 (+0.7)	40.4 (+0.1)
LLaMA3.3-70B+CBR	70B	62.5 (+2.3)	47.1 (+7.1)	38.5 (+2.0)	40.0 (+2.2)	42.5 (+2.4)	44.1 (+2.1)	45.5 (+2.0)	46.0 (+5.7)
LLaMA3.3-70B+KG+CBR (RAG pipeline)	70B	63.1 (+2.9)	47.8 (+7.8)	39.1 (+2.6)	40.6 (+2.8)	43.2 (+3.1)	44.9 (+2.9)	46.2 (+2.7)	46.5 (+6.2)
Qwen2.5-7B (DomRetriever) + LLaMA3.3-70B (Code Gen)	70B	63.9 (+3.7)	48.2 (+8.2)	43.8 (+7.3)	41.8 (+4.0)	44.3 (+4.2)	46.0 (+4.0)	47.3 (+3.8)	47.4 (+7.1)
LLaMA3.1-8B (DomRetriever) + LLaMA3.3-70B (Code Gen)	70B	63.5 (+3.3)	49.3 (+9.3)	44.6 (+8.1)	41.4 (+3.6)	44.0 (+3.9)	46.3 (+4.3)	49.2 (+5.7)	47.9 (+7.6)
GPT-4o	-	65.2	56.8	41.9	47.1	48.1	50.4	46.7	51.0
GPT-4o+KG	-	65.5 (+0.3)	56.9 (+0.1)	41.3 (-0.6)	46.2 (-0.9)	50.1 (+2.0)	51.8 (+1.4)	52.0 (+5.3)	51.2 (+0.2)
GPT-4o+CBR	-	69.0 (+3.8)	60.8 (+4.0)	49.1 (+7.2)	52.0 (+4.9)	53.2 (+5.1)	53.7 (+3.3)	53.5 (+6.8)	56.1 (+5.1)
GPT-4o+KG+CBR (RAG pipeline)	-	68.5 (+3.3)	61.2 (+4.4)	50.0 (+8.1)	50.6 (+3.5)	51.0 (+2.9)	56.4 (+6.0)	53.1 (+6.4)	56.4 (+5.4)
Qwen2.5-7B (DomRetriever) + GPT-4o (Code Gen)	-	67.8 (+2.6)	62.3 (+5.5)	49.3 (+7.4)	49.6 (+2.5)	50.3 (+2.2)	55.6 (+5.2)	52.2 (+5.5)	57.8 (+6.8)
LLaMA3.1-8B (DomRetriever) + GPT-4o (Code Gen)	-	68.6 (+3.4)	64.2 (+7.4)	50.8 (+8.9)	49.1 (+2.0)	50.7 (+2.6)	56.4 (+6.0)	53.3 (+6.6)	58.6 (+7.6)

Table 4: Ablation study on the truck CAN signal code generation task.

Domain	Driver productivity	Connected systems	Energy	Vehicle system	Visibility	Dynamics	Total
Count	**135**	**212**	**72**	**136**	**56**	**155**	**776**
Ablation on End-to-end DomAgent							
Qwen2.5-7B	32.6	46.8	36.2	35.1	44.3	39.8	39.62
Qwen2.5-7B + KG	62.1 (+29.5)	76.4 (+29.6)	78.7 (+42.5)	67.3 (+32.2)	65.4 (+21.1)	72.5 (+32.7)	70.89 (+31.27)
Qwen2.5-7B + CBR	74.4 (+41.8)	88.1 (+41.3)	86.3 (+50.1)	84.8 (+49.7)	89.3 (+45.0)	85.9 (+46.1)	84.57 (+44.95)
Qwen2.5-7B+KG+CBR (RAG pipeline)	92.2 (+59.6)	91.1 (+44.3)	94.2 (+58.0)	87.1 (+52.0)	95.2 (+50.9)	90.4 (+50.6)	91.03 (+51.41)
Qwen2.5-7B+KG+CBR (DomAgent)	100 (+67.4)	97.8 (+51.0)	100 (+63.8)	94.3 (+59.2)	100 (+55.7)	91.4 (+51.6)	96.64 (+57.02)
Ablation on External LLM (Code Gen) Enhanced with DomRetriever							
GPT-4o	65.1	73.4	58.8	71.1	82.2	75.5	71.22
GPT-4o + KG	85.2 (+20.1)	85.3 (+11.9)	84.3 (+25.5)	90.7 (+19.6)	94.9 (+12.7)	90.0 (+14.5)	87.80 (+16.58)
GPT-4o + CBR	93.5 (+28.4)	92.1 (+18.7)	90.2 (+31.4)	88.3 (+17.2)	93.4 (+11.2)	89.7 (+14.2)	91.10 (+19.88)
GPT-4o+KG+CBR (RAG pipeline)	97.6 (+32.5)	96.2 (+22.8)	96.5 (+37.7)	95.1 (+24.0)	98.0 (+15.8)	96.6 (+21.1)	96.49 (+25.27)
Qwen2.5-7B (DomRetriever) + GPT-4o (Code Gen)	100 (+34.9)	97.4 (+24.0)	100 (+41.2)	97.1 (+26.0)	100 (+17.8)	96.4 (+20.9)	98.04 (+26.82)
LLaMA3.1-8B (DomRetriever) + GPT-4o (Code Gen)	100 (+34.9)	98.3 (+24.9)	100 (+41.2)	98.6 (+27.5)	100 (+17.8)	97.2 (+21.7)	98.71 (+27.49)

Comparing the *RAG pipeline* with our DomAgent shows that review and filtering of irrelevant information further enhance performance. Moreover, reinforcement learning within DomAgent strengthens its ability to autonomously explore and reason, contributing to its superior results.

Overall, each module of our approach brings substantial improvements to domain-specific code generation. The integration of domain knowledge, case-based reasoning, information filtering, and reinforcement learning enables DomAgent to achieve state-of-the-art results and makes it feasible for deployment in real industrial environments.

6 CONCLUSION AND FUTURE WORK

In this work, we introduced DomAgent, an autonomous agent for domain-specific code generation that leverages KG–based retrieval to accurately acquire structured domain knowledge. During case retrieval, DomAgent exploits the relationships between packages and cases, re-ranking candidates based on package overlap. By integrating the reasoning capabilities of LLMs, DomAgent can invoke retrieval tools during the reasoning process, review the retrieved domain knowledge against the target case, and filter out irrelevant information to improve retrieval precision. We trained the LLM to use retrieval tools through supervised fine-tuning and employed reinforcement learning to encourage the model to explore the relevance between cases and domain knowledge. In addition, we

proposed a KG-guided case selection method for constructing the case base, achieving comparable performance by selecting only 30% of the cases compared to random sampling with 80%. DomAgent can serve as a retriever for essential domain knowledge and cases, making it easily integrable with other external LLMs for code generation. Experiments on both a benchmark dataset and a real-world truck domain dataset demonstrated that our approach outperforms LLMs of similar size. Moreover, we successfully deployed DomAgent in a real factory to generate code for accessing truck CAN signals. These results collectively validate the effectiveness and practical value of our method.

In future work, we plan to enhance DomAgent along three main directions. First, we will incorporate AST–based fine-grained similarity analysis [47], to improve the performance of retrieving relevant code cases at the structural level. Second, we will explore the use of soft prompts [22], a lightweight form of knowledge injection, to enable the model to efficiently adapt to new domains with minimal retraining. Finally, we intend to evaluate DomAgent in a wider range of real-world industrial scenarios to further validate its practicality and generalizability.

ACKNOWLEDGMENT

This work was partially funded by the Autonomous Systems and Software Program (WASP), supported by the Knut and Alice Wallenberg Foundation, and the Chalmers Artificial Intelligence Research Centre (CHAIR).

REFERENCES

[1] DM Anisuzzaman, Jeffrey G Malins, Paul A Friedman, and Zachi I Attia. 2025. Fine-tuning large language models for specialized use cases. *Mayo Clinic Proceedings: Digital Health* 3, 1 (2025), 100184.

[2] Mihir Athale and Vishal Vaddina. 2025. Knowledge Graph Based Repository-Level Code Generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 169–176.

[3] Chia-Yuan Chang, Zhimeng Jiang, Vineeth Rakesh, Menghai Pan, Chin-Chia Michael Yeh, Guanchu Wang, Mingzhi Hu, Zhichao Xu, Yan Zheng, Mahashweta Das, et al. 2025. Main-rag: Multi-agent filtering retrieval-augmented generation. *The 63rd Annual Meeting of the Association for Computational Linguistics (ACL)* (2025).

[4] Dustin Dannenhauer, Zohreh Dannenhauer, Despina Christou, and Kostas Hatalis. 2024. A case-based reasoning approach to dynamic few-shot prompting for code generation. In *ICML 2024 Workshop on LLMs and Cognition*.

[5] Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2024. How Abilities in Large Language Models are Affected by Supervised Fine-tuning Data Composition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 177–198.

[6] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. 2024. A Survey on In-context Learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 1107–1128.

[7] Xinyu Gao, Yun Xiong, Deze Wang, Zhenhan Guan, Zejian Shi, Haofen Wang, and Shanshan Li. 2024. Preference-guided refactored tuning for retrieval augmented code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 65–77.

[8] Yubin Ge, Devamanyu Hazarika, Yang Liu, and Mahdi Namazifar. [n.d.]. Supervised Fine-Tuning of Large Language Models on Human Demonstrations Through the Lens of Memorization. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

[9] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–22.

[10] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[11] Daya Guo, Qihao Zhu, Dejian Yang, and et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[12] Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024. DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning. In *ICML*.

[13] Kostas Hatalis, Despina Christou, and Vyshnavi Kondapalli. 2025. Review of case-based reasoning for LLM agents: theoretical foundations, architectural components, and cognitive integration. *arXiv preprint arXiv:2504.06943* (2025).

[14] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-generation database interfaces: A survey of llm-based text-to-sql. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[15] Bohan Hui et al. 2024. Qwen2.5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[16] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, and et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[17] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.

[18] 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.

[19] Chengwei Li, Zhenyu Xu, Bowen Wu, Xiang Li, Wenqiang Zhang, Ningyu Zhang, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2025. Retrieval-Augmented Code Generation for Universal Information Extraction. *arXiv preprint arXiv:2501.04702* (2025). https://arxiv.org/abs/2501.04702

[20] Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems* 37 (2024), 57619–57641.

[21] Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, et al. [n.d.]. StarCoder: may the source be with you! *Transactions on Machine Learning Research* ([n. d.]).

[22] Zongqian Li, Yinhong Liu, Yixuan Su, and Nigel Collier. 2025. Prompt Compression for Large Language Models: A Survey. 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)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 7182–7195. https://doi.org/10.18653/v1/2025.naacl-long.368

[23] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *ICLR*.

[24] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.

[25] Shuyin Ouyang, Jie Zhang, Zeyu Sun, and Albert Merono Penuela. 2025. Knowledge-Enhanced Program Repair for Data Science Code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 782–782.

[26] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. 2024. Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 237–250.

[27] Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. 2023. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541* (2023).

[28] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 3982–3992.

[29] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[30] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, and et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[31] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[32] Heydar Soudani, Evangelos Kanoulas, and Faegheh Hasibi. 2024. Fine tuning vs. retrieval augmented generation for less popular knowledge. In *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in*

Information Retrieval in the Asia Pacific Region. 12–22.

[33] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *ACM Transactions on Software Engineering and Methodology* (2024).

[34] Tim Tully, Joff Redfern, and Derek Xiao. [n.d.]. *2024: The State of Generative AI in the Enterprise*. https://menlovc.com/2024-the-state-of-generative-ai-in-the-enterprise/

[35] Shuai Wang, Wenji Mao, Penghui Wei, and Daniel D Zeng. 2022. Knowledge structure driven prototype learning and verification for fact checking. *Knowledge-Based Systems* 238 (2022), 107910.

[36] Shuai Wang, Penghui Wei, Qingchao Kong, and Wenji Mao. 2024. A knowledge enhanced learning and semantic composition model for multi-claim fact checking. *Knowledge-Based Systems* 304 (2024), 112439.

[37] Shuai Wang and Yinan Yu. 2025. iQUEST: An Iterative Question-Guided Framework for Knowledge Base Question Answering. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15616–15628. https://doi.org/10.18653/v1/2025.acl-long.760

[38] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 373–384.

[39] Xixi Wang, Miguel Costa, Jordanka Kovaceva, Shuai Wang, and Francisco C Pereira. 2025. Plugging Schema Graph into Multi-Table QA: A Human-Guided Framework for Reducing LLM Reliance. *arXiv preprint arXiv:2506.04427* (2025).

[40] Xixi Wang, Jordanka Kovaceva, Miguel Costa, Shuai Wang, Francisco Camara Pereira, and Robert Thomson. 2025. Domain-Adapted Pre-trained Language Models for Implicit Information Extraction in Crash Narratives. *arXiv preprint arXiv:2510.09434* (2025).

[41] 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.). 3199–3214.

[42] Ian Watson and Farhi Marir. 1994. Case-based reasoning: A review. *The knowledge engineering review* 9, 4 (1994), 327–354.

[43] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. In *International Conference on Machine Learning*. PMLR, 52632–52657.

[44] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. [n.d.]. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

[45] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology* (2025).

[46] Zhixiong Zeng, Shuai Wang, Nan Xu, and Wenji Mao. 2021. Pan: Prototype-based adaptive network for robust cross-modal retrieval. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*. 1125–1134.

[47] Yilin Zhang, Xinran Zhao, Zora Zhiruo Wang, Chenyang Yang, Jiayi Wei, and Tongshuang Wu. 2025. cAST: Enhancing Code Retrieval-Augmented Generation with Structural Chunking via Abstract Syntax Tree. *arXiv preprint arXiv:2506.15655* (2025).

[48] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. 2024. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473* (2024).

[49] Hao Zhu, Yiming Zhang, Shikun Feng, Zhiqiang Chen, Pengjun Qian, Min Zhang, and Jie Zhou. 2023. AceCoder: Utilizing Existing Code to Enhance Code Generation. *arXiv preprint arXiv:2312.03618* (2023). https://arxiv.org/abs/2312.03618