Reducing Domain Engineering Effort with Autonomous Agent Learning

Anonymous ACL submission

Abstract

Building large language model (LLM) systems for specialized knowledge bases (KBs) often requires substantial engineering. Custom components such as ontology parsers, embedding pipelines, and retrieval modules tailored to the target schema are often required to obtain satisfactory system performance. This paper asks whether autonomous LLM agents can substitute for these hand-crafted components, given only a small set of examples, read access to the raw KB files, and a code sandbox for exploration. We compare seven architectures for generating KB configurations. Four **tool-oriented agents** relying on domain-specific infrastructure, namely Cache-Augmented Generation (CaG), CaG with validation, Retrieval-Augmented Generation (RAG), and Graph-enhanced RAG (GraphRAG) and three **autonomous agents** that replace all custom components with a general-purpose Python sandbox. We evaluate on the Drilling and Wells Interoperability Standards (D-WIS) ontology. A simulated human-in-the-loop protocol drives multi-turn refinement, and structural metrics over RDF graphs quantify output quality. The best autonomous agent (Study, 0.931 class-URI F1, 95% confidence interval (CI) [0.900, 0.956]) matches both CaG baselines (0.916–0.921) with overlapping CIs, and approaches RAG (0.954) and GraphRAG (0.955), while all systems achieve 88–93% task completion. These results suggest that autonomous agents can substantially reduce domain-specific engineering, shifting the effort from building custom infrastructure to curating a small example set, while approaching or matching hand-crafted pipeline quality. Code: `https://anonymous.4open.science/r/DWIS-Copilot-CDFE`

1 Introduction

Specialized knowledge bases, such as ontologies and knowledge graphs, encode rich structured vocabularies for domains such as drilling engineering, biomedical curation, and industrial interoperability. Translating natural-language requests into valid expressions over such bases requires familiarity with the KB structure that many domain practitioners lack. LLM provide a natural interface (Brown et al., 2020), but integrating a KB into an LLM system typically requires custom components such as schema parsers, embedding indices, and domain-specific retrieval logic. Maintenance of these components as they evolve usually requires significant engineering investment.

The central question this paper addresses is: *Can a KB copilot be built without domain-specific infrastructure, by supplying an LLM agent with only (i) a small set of examples that demonstrate the desired behavior, (ii) read access to the raw KB files, and (iii) a general-purpose code sandbox?*

These inputs are deliberately simple. The examples are pairs of natural-language requests, and the data access is the unprocessed KB with no schema parser, embedding index, or retrieval module provided. The autonomous agents must discover and exploit the KB structure through their own code. Although the training sheet still requires human curation, it eliminates all domain-specific infrastructure.

To answer this question, we compare seven architectures that span a spectrum from heavily engineered to fully autonomous. Four **tool-oriented agents** that require KB-specific parsers, embeddings, and retrieval logic, and three **autonomous agents** that replace all such infrastructure with a Python sandbox and a training sheet. We evaluated on the D-WIS ontology (D-WIS Industry Group, 2025; Cayeux et al., 2019, 2023), using an LLM-based user simulator (Mosqueira-Rey et al., 2023) for multi-turn feedback. A complete input/output example and conversation trace are provided in Appendix D.

Our contributions are as follows:

1. A framework comparing engineered and au-

tonomous KB copilots across a spectrum of infrastructure requirements.

2. Empirical evidence that the best autonomous agent numerically exceeds CaG baselines (95% CI overlap) and approaches RAG/-GraphRAG quality, while eliminating all custom KB components.

A survey of related work is provided in the Appendix A.

2 Framework

All seven systems are implemented as LangChain (Chase, 2022) Reason-and-Act (ReAct Yao et al., 2023) agents that share the same evaluation protocol. They are organized into two categories, tool-oriented and autonomous, distinguished by their infrastructure requirements (Figure 1).

Tool-Oriented Agents: CaG, CaG+val, RAG+val, GraphRAG+val	
Custom	Ontology parser, embedding index, retrieval pipeline
Optional	Validation module (for +val. variants)
Autonomous Agents: PaL, Study, Study+del	
Generic	Training sheet (examples), raw KB files, code sandbox

Figure 1: Infrastructure requirements by agent category. Tool-oriented agents need domain-specific components, autonomous agents use only generic resources and the unprocessed KB.

2.1 Task Formulation

Given a natural-language request q and a knowledge base $\mathcal{K}$, the system produces a configuration $c = (D, T)$ consisting of:

- *declarations* $D = \{\texttt{Class}_i\texttt{:Instance}_i\}$, each instantiating a class from $\mathcal{K}$.
- *triples* $T = \{(s_j, p_j, o_j)\}$, each asserting a relationship using a predicate p_j defined in $\mathcal{K}$.

The configuration must be *consistent* with $\mathcal{K}$. Every class and predicate URI must exist in the ontology schema.

We instantiate $\mathcal{K}$ as the D-WIS RDF/Turtle ontology (D-WIS Industry Group, 2025), which defines ∼2 200 classes and ∼250 properties for drilling-domain sensor data, signal processing, and equipment interconnection. As a concrete example, a request to "*set up standpipe pressure monitoring with moving-average filtering*" yields declarations such as `SPP:SPP_0` and `PressureDrillingQuantity:SPPQuantity_0`, linked by triples such as `SPP_0 IsOfMeasurableQuantity SPPQuantity_0`. A complete example is provided in Appendix D.

2.2 Tool-Oriented Agents

All four tool-oriented agents require a custom ontology parser that resolves the D-WIS multiple-inheritance class hierarchy. Agents with validation also require a purpose-built evaluator module that checks structural and semantic constraints.

Cache-Augmented Generation (CaG). The entire ontology schema (class hierarchy and property signatures) together with few-shot examples are placed into the system prompt, relying on the model's long context window to attend over all KB information simultaneously (Chan et al., 2025). The agent generates the full configuration (D, T) in a single pass with no retrieval calls.

CaG + val. Extends CaG with a `validation` tool that parses the generated configuration against the ontology, reports class and predicate errors, and enables iterative self-correction within the multi-turn loop.

RAG + val. Replaces the full-context prompt with on-demand semantic retrieval (Lewis et al., 2020). The same validation tool as CaG + val. is available. Retrieval details are provided in Appendix E.

GraphRAG + val. Extends RAG by re-ranking retrieved ontology fragments using graph-structural connectivity scores, favoring coherent ontology subgraphs over isolated class matches (Han et al., 2024).

2.3 Autonomous Agents

The three autonomous agents do not require domain-specific infrastructure. Instead, each receives a *training sheet*, a document with examples in the format of Section 2.1, and has read access to the raw ontology file. All agents share a code sandbox in which they write and execute Python code to explore the KB, parse its structure, and build configurations.

Program-aided Language model (PaL). Following Gao et al. (2023), the agent delegates structured reasoning to the generated Python code. Given a query, it writes scripts to load the ontology file, query class hierarchies, and assemble the output configuration. The training sheet provides in-context examples of the expected format.

Study Agent. Extends PaL with a dedicated *preparation phase* that runs before any test queries. During preparation, the agent studies the training sheet examples, explores the ontology via sandbox code, and produces the following artifacts:

- **Persistent tools**: Python functions that encapsulate common ontology queries and transformations, stored persistently in the sandbox for reuse across test queries.
- **Knowledge summary**: A natural-language document summarizing key domain patterns, class hierarchies, and task conventions discovered during preparation. This summary is included in the prompt at test time to guide reasoning.

The preparation phase allows the agent to amortize the cost of exploration and build reusable tools, while the knowledge summary provides a distilled reference to the KB structure that can be attended to during generation.

Study Agent with Delegation (Study + del.). Replaces the single-agent preparation with an orchestrated multi-agent system. A *Planner* agent creates a study plan, and an *Executor* implements it in code. This decomposition enables specialization, but introduces coordination overhead.

All autonomous agents execute code in an isolated Python environment. The sandbox creates a virtual environment pre-installed with standard data-science packages and has access to the raw data files (ontology, KB). Agents may write *fleeting* (exploratory) or *persistent* code. Persistent code is stored across calls and reusable, effectively allowing the agent to build new tools during the preparation phase. The persistent code is replayed on a sandbox restart to ensure reproducibility.

3 Experiments

3.1 Evaluation

An LLM-based user simulator (Mosqueira-Rey et al., 2023) interacts with each system for up to $R = 3$ rounds per task, providing targeted natural-language feedback on structural errors.

Interaction-level metrics. We record:

- **Task completion rate**: the fraction of tasks for which the simulator accepts the final configuration as correct.
- **Mean dialog iterations**: the average number of agent responses (initial + corrections) per task.

These capture how often the system succeeds and how much user effort is required.

Structural metrics. Both the generated configuration $c = (D, T)$ and the gold-standard $c^* = (D^*, T^*)$ from the task formulation (Section 2.1) are parsed into RDF graphs for comparison. Instance IDs are normalized via bipartite matching: instances sharing a class are paired by maximizing Jaccard similarity of their predicate signatures, solved as a minimum-weight matching problem. This normalization affects triple comparison but not URI-set metrics.

- **Class-URI F1**: F1 over the multiset of class URIs appearing as `rdf:type` objects in D vs. D^*. Measures whether the correct ontology classes are declared.
- **Pred-URI F1**: F1 over the set of predicate URIs in T vs. T^*. Measures whether the correct relationship types are used.
- **Triple F1**: F1 over the full set of RDF triples after instance-ID normalization. The strictest metric, requiring subject, predicate, *and* object to match.
- **Norm SD** (Normalized Symmetric Difference): $(|E_\Delta| + |V_\Delta|) \,/\, (|E_{\text{exp}}| + |V_{\text{exp}}| + |E_{\text{gen}}| + |V_{\text{gen}}|)$, where E_Δ and V_Δ are symmetric differences of triples and URI nodes, respectively.

3.2 Setup

All experiments use Kimi K2.5 (Team et al., 2026), evaluated on 87 D-WIS configuration tasks with $R = 3$ simulation rounds. The metrics are computed on both the initial response and after the final correction round. The triple-level metrics use the bipartite instance-ID normalization described in Section 3.1.

3.3 Results

Table 1 summarizes the results for all seven systems.

Task completion. All systems achieve 88–93% completion. RAG leads (93.1%), Study and Study + del. both reach 92.0%. Mean iterations range from 1.53 (PaL) to 1.73 (Study + del.). Most tasks are resolved after a single feedback round (see Appendix B).

Structural quality after correction. Study (0.931 class-URI F1) numerically exceeds both CaG variants (0.916–0.921) with overlapping 95%

Table 1: Results on 87 D-WIS test cases. Metrics reported after multi-turn correction, 95% bootstrap confidence intervals in brackets. Best value per column in **bold**.

Approach	Compl. (%)	Iters	Class-URI F1	Pred-URI F1	Triple F1	Norm SD
Tool-Oriented Agents						
CaG	89.7	1.56	0.921 [0.877, 0.957]	0.948 [0.905, 0.982]	0.870 [0.823, 0.911]	0.105 [0.068, 0.150]
CaG + val.	90.8	1.58	0.916 [0.866, 0.957]	0.934 [0.885, 0.972]	0.875 [0.825, 0.918]	0.106 [0.064, 0.155]
RAG + val.	**93.1**	1.62	0.954 [0.932, 0.972]	**0.984** [0.976, 0.990]	0.888 [0.852, 0.920]	0.081 [0.058, 0.108]
GraphRAG + val.	88.5	1.64	**0.955** [0.936, 0.971]	0.962 [0.943, 0.979]	**0.897** [0.867, 0.925]	**0.075** [0.055, 0.098]
Autonomous agents						
PaL	88.5	1.53	0.925 [0.887, 0.956]	0.936 [0.899, 0.964]	0.861 [0.820, 0.897]	0.111 [0.079, 0.150]
Study	92.0	1.64	0.931 [0.900, 0.956]	0.944 [0.914, 0.968]	0.871 [0.832, 0.905]	0.101 [0.073, 0.135]
Study + del.	92.0	1.73	0.915 [0.871, 0.951]	0.922 [0.878, 0.957]	0.849 [0.805, 0.888]	0.122 [0.086, 0.165]

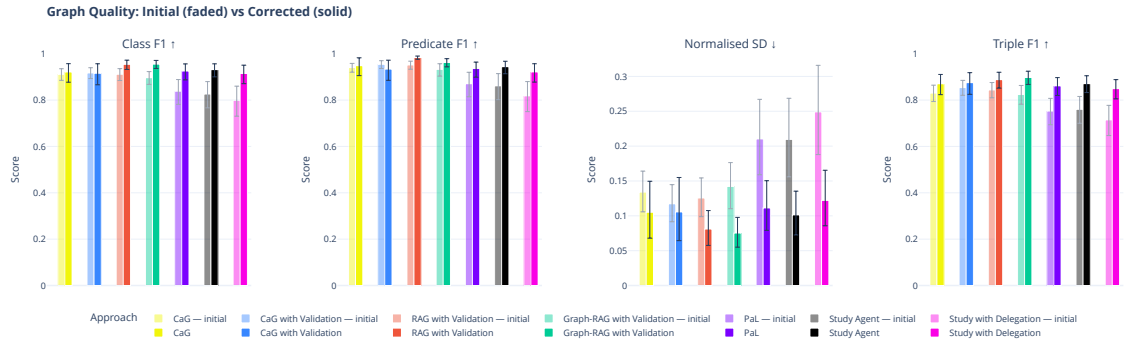

Figure 2: Initial (faded) vs. corrected (solid) structural metrics. Autonomous agents gain 8.7–11.6% class-URI F1 through correction, tool-oriented agents gain at most 5.8%.

CIs, approaching RAG (0.954) and GraphRAG (0.955). PaL (0.925) matches CaG + val. in point estimate. GraphRAG achieves the lowest Norm SD (0.075). All autonomous agents fall at 0.101–0.122.

Asymmetric improvement from feedback. The key finding (Figure 2) is that autonomous agents benefit from multi-turn correction: +8.7–11.6% vs. $\leq$5.8% for engineered engines. Single-shot evaluation would underestimate the practical utility.

4 Discussion

Autonomous agents are viable KB copilot builders. Study matches or numerically exceeds both CaG baselines without any KB-specific tooling, confirming that an LLM agent can learn domain structure from examples alone (Brown et al., 2020; Gao et al., 2023). Among autonomous variants, preparation clearly helps (Study 0.931 vs. PaL 0.925 class-URI F1), while Study + del. (0.915) slightly underperforms, suggesting coordination overhead can outweigh the benefits of multi-agent decomposition (see Appendix C).

Engineering–quality trade-off. RAG and GraphRAG remain strongest in structural quality at similar cost. RAG requires 10.8M tokens vs. 10.0M for Study. Autonomous agents therefore represent a compelling alternative with similar quality.

Limitations

Experiments cover a single model (Kimi K2.5) and domain (D-WIS), so generalization requires broader validation. The simulator reveals target class names in 55.5% of feedback turns despite the zero-leakage rule, so corrected-round metrics should be interpreted as upper bounds. Current metrics also do not capture all semantic constraints (e.g., cardinality and range), motivating more constraint-aware evaluation and human-audited studies in future work.

References

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

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), pages 169–176. IEEE.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Eric Cayeux, Benoît Daireaux, Nejm Saadallah, and Sergey Alyaev. 2019. Toward seamless interoperability between real-time drilling management and control applications. In *SPE/IADC Drilling Conf. and Exh.*, page D031S017R002.

Eric Cayeux, Carlos Damski, J Macpherson, Moray Laing, Pradeep Annaiyappa, Philip Harbidge, M Edwards, and Jonathan Carney. 2023. Connecting multilayer semantic networks to data lakes: The representation of data uncertainty and quality. *SPE Drilling & Completion*, 38(01):18–33.

Brian J Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen Huang. 2025. Don't do rag: When cache-augmented generation is all you need for knowledge tasks. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 893–897.

Harrison Chase. 2022. LangChain. GitHub.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Gretchen Krueger, and 1 others. 2021. Evaluating large language models trained on code. Technical report, OpenAI.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

D-WIS Industry Group. 2025. Drilling and wells interoperability standards (d-wis). Accessed: 2025-05-31.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International conference on machine learning*, pages 10764–10799. PMLR.

Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. 2021. Beyond I.I.D.: Three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, pages 3477–3488.

Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, and 1 others. 2024. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309*.

Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. 2012. OWL 2 Web Ontology Language primer (second edition). W3C Recommendation.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 6769–6781.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks. In *International Conference on Learning Representations*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. 2023. Human-in-the-loop machine learning: a state of the art. *Artificial Intelligence Review*, 56(4):3005–3054.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*.

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*, pages 3982–3992.

Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. 2025. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

In *Advances in Neural Information Processing Systems*, volume 36.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, and 1 others. 2026. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 2609–2634.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

A Related Work

A.1 Prompting Strategies for Complex Reasoning

Chain-of-thought prompting (Wei et al., 2022) demonstrated that eliciting intermediate reasoning steps markedly improves LLM performance on multi-step tasks. Plan-and-Solve (Wang et al., 2023) extends this by having the model first devise a plan and then execute each step, closely related to the preparation phase used by the autonomous agents. Program-aided language models (PAL) (Gao et al., 2023) and Program-of-Thought (Chen et al., 2022) offload computation to generated code, the core strategy of the autonomous agents here. Decomposed prompting (Khot et al., 2023) decomposes tasks into subproblems handled by specialized modules, paralleling the Orchest agent's delegation to Planner and Executor sub-agents.

A.2 Retrieval Augmented and Cache Augmented Generation

Retrieval Augmented Generation (Lewis et al., 2020) grounds LLM outputs in retrieved external knowledge. Dense passage retrieval (Karpukhin et al., 2020) and learned embeddings (Reimers and Gurevych, 2019) are standard components. Self-RAG (Asai et al., 2023) adds self-reflection to decide when retrieval is needed. GraphRAG (Han et al., 2024) augments retrieval with structural graph information, improving coherence for knowledge-graph tasks. Cache Augmented Generation (Chan et al., 2025) takes the opposite approach: the entire knowledge base is placed in-context, trading retrieval complexity for prompt length.

A.3 Tool-Augmented and Agentic LLMs

Toolformer (Schick et al., 2023) and ToolLLM (Qin et al., 2024) showed that LLMs can learn to invoke external tools at scale. ReAct (Yao et al., 2023) interleaves reasoning and tool use. Wang et al. (2024) survey LLM-based autonomous agents, identifying planning, memory, and tool use as core capabilities. Sapkota et al. (2025) distinguish vibe coding from *agentic coding*. The autonomous agents here instantiate the latter, while the copilot engines occupy a middle ground where tools are hand-crafted rather than agent-discovered. Code generation agents (Chen et al., 2021) provide conceptual foundations.

A.4 Knowledge-Base Question Answering and Ontologies

Knowledge-base question answering (KBQA) has been studied extensively in both structured and graph settings (Gu et al., 2021; Gao et al., 2010). Knowledge-graph-augmented code generation has been explored for repository-level tasks (Athale and Vaddina, 2025). The Web Ontology Language (OWL) (Hitzler et al., 2012) provides formal ontology semantics. The D-WIS initiative (D-WIS Industry Group, 2025; Cayeux et al., 2019, 2023) defines an RDF-based ontology for drilling-domain interoperability and is used here as the evaluation domain.

B Additional Figures

C Extended Discussion

Preparation helps, but orchestration may not generalize. Among autonomous variants, Study consistently outperforms PaL (0.931 vs. 0.925

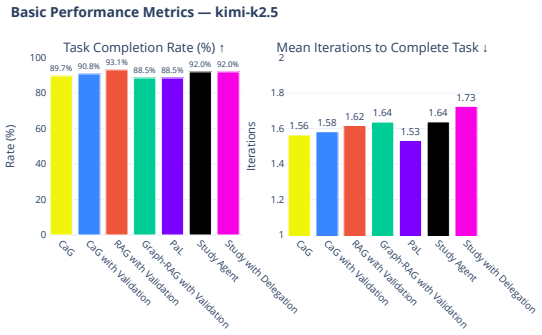

Figure 3: Task completion rate and mean iterations across all approaches. Tool-oriented agent variants in blue, autonomous agents in orange.

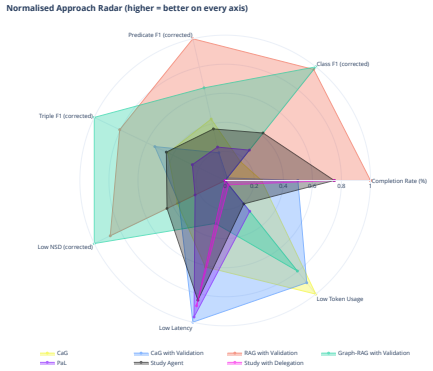

Figure 4: Radar chart comparing all approaches across normalised metrics. Larger area indicates better overall performance.

class-URI F1, 0.101 vs. 0.111 Norm SD), confirming that a dedicated preparation phase yields lasting benefits. However, Study + del. (0.915 class-URI F1) performs slightly worse than Study despite its more sophisticated Planner–Executor decomposition. One explanation is that the additional coordination overhead introduces noise, or that the sequential planning–execution split is suboptimal for the D-WIS domain. Whether orchestrated preparation generalizes better to larger or more complex ontologies remains an open question.

Simulator leakage. The zero-leakage audit found that the simulator reveals target class names in 55.5% of feedback turns. This means corrected-round metrics are likely upper bounds on real-world performance, particularly for autonomous agents that benefit most from multi-turn correction. Future work should improve the simulator's adherence to leakage constraints or use human evaluators for a subset of tasks.

Work-shift risk A practical concern is that the proposed simplification may primarily relocate effort rather than reduce it. Building parsers and retrieval stacks is replaced by curating representative training sheets, maintaining coverage as schemas evolve, and auditing examples for hidden leakage or bias. In domains with sparse historical configurations or frequent ontology changes, this curation burden can become a bottleneck and narrow the apparent engineering advantage. A fair comparison should therefore include not only runtime and output quality, but also the human cost of creating and maintaining high-quality example sets over time.

D Task Example

Table 2 shows a representative D-WIS task. The user describes a drilling sensor configuration in natural language. The system must produce declarations D (class instantiations) and triples T (relationships) consistent with the D-WIS ontology. Each declaration has the form `Class:InstanceID`, where `Class` is a URI from the ontology and `InstanceID` is an agent-chosen identifier. Each triple has the form `Subject Predicate Object`, where the predicate is a property URI from the ontology.

Table 2: Example D-WIS task: standpipe pressure monitoring.

User request q

"Set up standpipe pressure monitoring with moving-average filtering, transmitted via top-side telemetry and provided by a data flow node."

Expected configuration $c^* = (D^*, T^*)$

Declarations D^**:*

```
SPP:SPP_0
MudStandpipe:StandPipe_0
PressureDrillingQuantity:SPPQuantity_0
TimeWindowMovingAverage:TWAvg_0
TopSideTelemetry:TopSideTelemetry_0
DataFlowNode:DataFlowNode_0
```

Triples T^**:*

```
SPP_0 IsHydraulicallyLocatedAt StandPipe_0
SPP_0 IsOfMeasurableQuantity SPPQuantity_0
SPP_0 IsProcessedBy TWAvg_0
SPP_0 IsTransmittedBy TopSideTelemetry_0
SPP_0 IsProvidedBy DataFlowNode_0
```

In the multi-turn protocol, the user simulator compares the generated configuration against c^* and provides natural-language feedback, e.g.: "*The pressure quantity should have drilling component. Also, the standpipe component is missing.*" The agent then corrects its output. Most tasks are re-

solved within 1–2 feedback rounds (Figure 3).

E RAG Retrieval

The RAG retrieval used by the RAG + val. and GraphRAG + val. operate as `retrieve` tool. This allowes the agent to retrieve at its own discretion, rather than being forced to retrieve for every query and also the ability to iteratively refine retrieval queries based on feedback.

GraphRAG extends the retrieval by re-ranking retrieved ontology fragments based on graph-structural connectivity scores. This favors coherent subgraphs of the ontology that are more likely to contain relevant classes and properties, rather than isolated matches.