Enhancing Contextual Memory in LLMs for Software Engineering via Ontology-based Inference

David Araya[1,*], Catalina Valle[1], Hernán Astudillo[1], Pablo Ormeño[2] and Carla Taramasco[1]

[1]Universidad Andrés Bello, Viña del Mar, Chile
[2]Universidad de Viña del Mar, Viña del Mar, Chile

Abstract

The application of large language models (LLMs) in software engineering has expanded rapidly, supporting tasks such as code generation, system design, error correction, and documentation. However, LLMs face persistent challenges in multi-step development workflows, where accumulated design decisions, dependencies, and constraints require a structured, persistent memory. Existing approaches, including extended context windows and retrieval-augmented generation, rely on statistical similarity between text fragments and lack explicit, formal representations of the evolving system state, limiting their ability to maintain semantic coherence across development stages. We propose an architecture that integrates a dynamically evolving ontology, represented in OWL (Web Ontology Language), as a structured knowledge base, maintained automatically to reflect changes in the codebase and user interactions. OWL is a standard language for formal knowledge representation, allowing the explicit modeling of entities, relationships, and constraints in a machine-readable way. A secondary LLM (LLM2) functions as a context manager, querying the ontology using OWLready2 to retrieve structured information about entities, classes, methods, and associated constraints. This context is provided to the primary LLM (LLM1), which directly interacts with the user to generate code that adheres to both current requirements and historical design decisions. This hybrid approach combines symbolic knowledge representation with generative capabilities, enabling context-aware, semantically coherent, and explainable code generation. We validated the approach through a proof-of-concept implementation in a multi-stage object-oriented development scenario. Comparative evaluation using the same base model, with and without the ontology layer, showed that the ontology-enhanced configuration improved adherence to prior decisions, maintained semantic consistency across stages, and produced outputs that were more interpretable and reliable. These results demonstrate the potential of ontology-supported LLMs to facilitate long-term, semantically grounded software development.

Keywords

Large language models, Code generation, Semantic retrieval, Software engineering, Ontologies

1. Introduction

Large language models (LLMs) have become essential tools in software engineering, supporting tasks such as system design, code development, debugging, and documentation generation from natural language descriptions [1, 2, 3]. Their ability to process unstructured input and produce syntactically correct code has accelerated rapid prototyping and iterative development. Nevertheless, their effectiveness is often limited by the absence of persistent memory mechanisms capable of maintaining contextual coherence across extended development cycles [1, 3, 4].

In typical software projects, development is incremental: high-level architectural definitions are established first and subsequently refined with additional components, constraints, and implementation details. This iterative process produces a growing set of decisions, dependencies, and design rules that must be preserved and referenced to maintain semantic consistency. Current LLMs, however, frequently lose track of evolving context, leading to outputs that contradict or ignore earlier specifications [1]. For example, after defining a class with specific attributes and methods, an LLM may later introduce a

ICAIW 2025: Workshops at the 8th International Conference on Applied Informatics 2025, October 8–11, 2025, Ben Guerir, Morocco
*Corresponding author.
david.araya@unab.cl (D. Araya); cvalleribe@gmail.com (C. Valle); hernan.astudillo@unab.cl (H. Astudillo); pormen@gmail.com (P. Ormeño); carla.taramasco@unab.cl (C. Taramasco)
0009-0006-6254-4339 (D. Araya); 0009-0009-3192-8120 (C. Valle); 0000-0002-6487-5813 (H. Astudillo); 0000-0001-5591-3518 (P. Ormeño); 0000-0001-8318-4201 (C. Taramasco)

function without recognizing its membership in that class, redundantly redefine variables, or overlook established constraints, thereby undermining design integrity and increasing refactoring overhead.

To address these limitations, we propose an architecture that integrates a structured and dynamically evolving knowledge base in the form of an OWL ontology [5]. This ontology provides an explicit and queryable representation of the system at multiple abstraction levels, capturing entities, relationships, and constraints relevant to the target domain. For example, the ontology can capture knowledge about different frameworks, technology stacks, or software architectures, enabling the documentation of design choices, the enforcement of compatibility requirements, and the preservation of decisions made in earlier stages. In the software engineering scenario used for this study, these entities include modules, classes, methods, attributes, and associated constraints. The ontology is continuously and automatically updated to reflect changes in the codebase and user interactions, ensuring alignment with the evolving system state. By maintaining a persistent memory of domain-specific elements and their interconnections, the ontology enables formal reasoning about dependencies, prevents redundant definitions, and preserves semantic integrity.

Our approach combines the generative capabilities of LLMs with a dynamically updated ontology that captures the evolving state of the system. This structured memory provides domain-specific context to guide code generation, ensuring semantic consistency and alignment with prior design decisions.

By combining symbolic knowledge representation with generative capabilities, this architecture supports explainable, context-aware, and semantically coherent code generation. Comparative evaluations using the same base model, with and without the ontology layer, demonstrate that the ontology-enhanced configuration significantly improves adherence to prior design decisions and reduces inconsistencies in generated outputs, effectively bridging the gap between natural language interaction and formal software design rigor.

2. Related Work

Large language models (LLMs) have transformed software engineering, establishing themselves as effective assistants in tasks such as code synthesis, system design, debugging, and documentation generation from natural language [1, 6, 7]. However, their application in real-world environments still faces significant challenges. Several reviews have highlighted that the lack of robust memory mechanisms hinders the maintenance of semantic coherence in multi-step workflows, where decisions, constraints, and dependencies evolve progressively [8]. Models such as Codex have demonstrated strong performance in individual functions on benchmarks such as HumanEval and MBPP [1], but their effectiveness diminishes in more complex settings such as SWE-bench or RepoBench, which require coordination across multiple files and the preservation of invariants [9, 10]. These limitations are associated with phenomena such as positional bias and context loss, documented in studies on extended context windows [11]. Although techniques such as explicit planning or adjustments to attention mechanisms have shown isolated improvements [12, 13], semantic inconsistencies persist, affecting reliability in real-world development scenarios. In automatic debugging, models such as RepairLLaMA outperform traditional approaches on specific benchmarks, but their generalization capability remains limited [14].

Complementarily, the literature has explored the use of ontologies to formally represent software artifacts, relationships, and constraints, providing benefits in contextual reasoning, traceability, and semantic coherence. For example, the OntoTrace V2.0 ontology allows the inference of links between artifacts via SPARQL (SPARQL Protocol and RDF Query Language), yielding empirical improvements in precision and speed for traceability tasks [15]. In model-driven engineering, the ReApp project employs ontological descriptions of ROS components, facilitating their reuse and ensuring semantic consistency in automatic code generation [16]. Similarly, the Knowledge Discovery Metamodel (KDM) of the Object Management Group defines an ontological representation of software architectures, enabling traceability between artifacts, flows, and source code [17].

In parallel, multi-agent architectures have emerged that separate code generation from contextual

knowledge management. Configurations such as AutoGen and MetaGPT assign specific roles to different agents to generate, review, and validate code in sequential stages, integrating external tools and retrieval-augmented methods [18, 19, 20]. Other approaches, such as SWE-agent [21] or RepoCoder [22] combine iterative generation with repository exploration, improving completion rates and issue resolution metrics [21].

Although current approaches have advanced context management through retrieval-augmented generation or multi-agent architectures, a fundamental limitation remains: the absence of a structured memory that explicitly, persistently, and dynamically represents the evolving state of the system. None of the reviewed works integrates a dynamic ontology that is automatically updated during development and serves as a formal knowledge source to guide code generation. Our proposal addresses this gap by integrating an OWL ontology, maintained by a secondary language model, which models system entities, relationships, and constraints, enabling semantic reasoning over past decisions. This explicit symbolic memory complements the generative capabilities of the primary LLM, promoting sustained semantic coherence, traceability, and alignment throughout the entire development cycle, an aspect not previously explored at the intersection of LLMs and software engineering.

3. Methods

This section describes the architecture, implementation, and evaluation of the ontology-based contextual code generation system.

3.1. Overall System Design and Rationale

The motivation for the modular architecture stems from two common limitations in LLM-based code generation: first, the models' tendency to produce code that is syntactically valid but semantically inconsistent with the target domain; and second, the challenge of injecting evolving domain knowledge into a conversational flow without retraining the model. By decoupling the generative component (LLM1) from the semantic control component (LLM2), we aimed to mitigate both issues.

The architecture was deliberately designed so that LLM1 never directly queries the ontology. Instead, it always obtains the relevant semantic context from LLM2, which is responsible for maintaining an accurate, up-to-date representation of the domain. This enforces a clean separation of concerns and allows for easy replacement or upgrade of either LLM without disrupting the pipeline.

The ontology itself serves as a persistent, machine-readable memory of the project domain. It actively participates in generation: its constraints influence prompt construction, and its structure adapts dynamically to new user requests. This bidirectional relationship between generation and knowledge representation is one of the key novelties of the method.

To further clarify the interaction flow, Figure 1 shows the step-by-step process from user request to final code generation, including context retrieval, code generation, ontology updates, and optional error handling.

The ontology builder extracts modules, classes, methods, attributes, and relationships directly from the codebase, operating independently of file naming conventions. This makes the approach agnostic to project structures, so different entry points (e.g., main.py or app.py) can be handled without errors.

3.2. LLM1: Code Assistant

LLM1 was implemented in Python using the LangChain [23] orchestration library. Python was selected due to its strong ecosystem for ontology management (e.g., OWLready2), its wide adoption in AI research, and community support for semantic reasoning tools. The underlying model was **gpt-4o**, accessed via the **LangChain ChatOpenAI interface**. Its role was to take the user's request, enrich it with semantic context from LLM2, and produce outputs that were syntactically correct, semantically aligned, and readable.

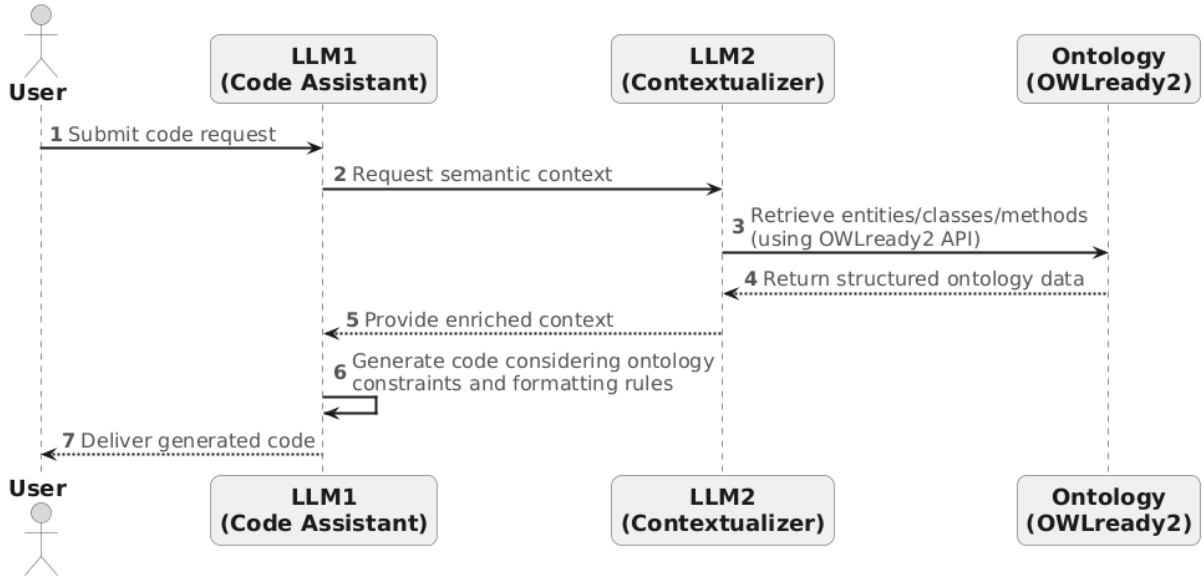

Figure 1: Sequence diagram illustrating the flow of interactions between the user, LLM1 (Code Assistant), LLM2 (Contextualizer), and the ontology.

Prompts to LLM1 were structured in three sections: (1) the user's original request verbatim; (2) a context block summarizing ontology facts, often in tabular or key-value format; and (3) explicit formatting instructions for code syntax, inline comments, and docstring style. This format allowed for deterministic outputs while keeping flexibility for natural language variations.

LLM1 also implements an internal validation step to check semantic consistency against the ontology-enriched context. If inconsistencies are detected, LLM1 generates a warning or requests additional context from LLM2 before finalizing the code. This mechanism ensures that generated outputs adhere to domain constraints.

3.3. LLM2: Contextualizer

LLM2 acts as the intermediary between the ontology and LLM1. It uses gpt-4o to interpret user requests and code analysis results, generating precise OWLready2 update commands. LLM2 retrieves relevant ontology knowledge to inject into LLM1's prompts and updates the ontology when new entities, relationships, or constraints are detected.

Future implementations may incorporate SPARQL-based queries for more advanced retrieval, but the current system relies on direct OWLready2 API calls. Retrieval is optimized to select only the relevant subset of ontology data, reducing prompt size and avoiding overload during inference.

3.4. Ontology Representation and Semantics

The ontology was implemented in OWL using OWLready2, encapsulating core object-oriented programming concepts: *modules*, *classes*, *attributes*, *methods*, *parameters*, and *return types*. Each entity is enriched with datatype and relational constraints, including cardinality restrictions and domain-range specifications, to ensure semantic integrity and enable consistent reasoning over the model. Once stored, information in the ontology is preserved across updates, ensuring persistence of memory throughout the development process.

At initialization, the ontology begins as an empty schema that defines only the fundamental structures mentioned above. These classes serve as predefined containers that are gradually populated as the system analyzes the evolving codebase. In practice, LLM2 generates precise instructions for OWLready2 to insert new entities such as modules, classes, methods, or attributes whenever they are detected, as well as to modify or remove elements when changes occur in the source code. This incremental

population process guarantees that the ontology remains synchronized with the actual state of the project at all times. Furthermore, because the ontology is serialized and stored as an OWL file on disk, it can be reloaded in subsequent sessions, ensuring that accumulated knowledge is never lost across executions and that memory persists independently of runtime constraints.

Incremental updates are continuously orchestrated by LLM2, which monitors the evolving codebase and triggers ontology modifications whenever new entities or relationships appear. Newly instantiated entities are linked through well-defined object properties, maintaining referential integrity, avoiding inconsistencies, and supporting full traceability between code artifacts and their semantic representations.

Although the current implementation relies on OWLready2 for direct API-driven access, future extensions will incorporate SPARQL queries to enable more sophisticated semantic searches and reasoning across entities. This will provide enhanced contextual awareness for code generation, facilitate automated refactoring, and open the door to more advanced reasoning strategies in large-scale projects.

Overall, this ontology-driven approach ensures that the LLM-generated code is not only syntactically correct but also semantically aligned with the underlying conceptual model. By combining persistence, incremental synchronization, and semantic reasoning, the system improves maintainability, traceability, and overall software quality, while laying the foundations for scalable and explainable AI-assisted software engineering.

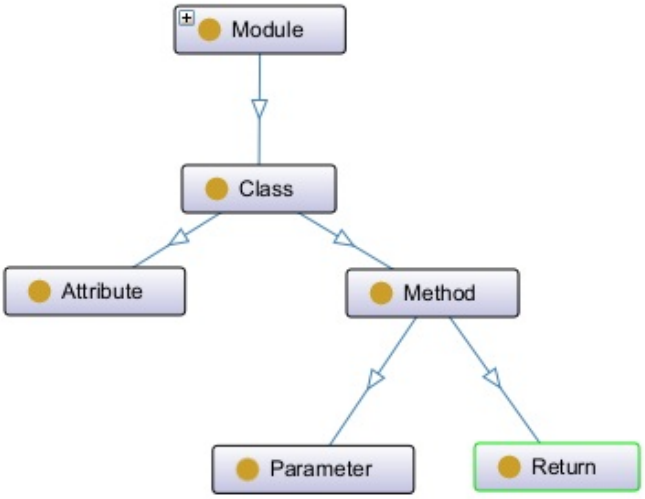

Figure 2: Ontology schema illustrating relationships between modules, classes, attributes, methods, parameters, and return types. Low-level elements may be omitted in visualizations for clarity.

3.5. Experimental Setup

The experiments were orchestrated using LangChain, which enables systematic control over prompt execution and context management across multiple LLM agents. The primary objective of the setup was to evaluate the effect of ontology-enriched context on code generation quality while minimizing variability due to stochastic sampling. Accordingly, the model temperature was set to 0.3 to favor determinism, while other parameters were maintained at their default values, ensuring a controlled baseline for comparison.

Parameter	Value
Model	gpt-4o
Temperature	0.3
Top-p	default
Frequency penalty	default
Presence penalty	default
Max tokens	default
Framework	LangChain
API	OpenAI Chat API
Python version	3.11

Table 1
Parameters and execution environment used in the LLM experiments.

Figure 3 illustrates the ontology generated for Example 3 code. Only modules and classes with their methods are displayed for clarity; attributes, parameters, and return types are omitted to reduce visual complexity. This representation demonstrates the structured semantic mapping that informs LLM code generation.

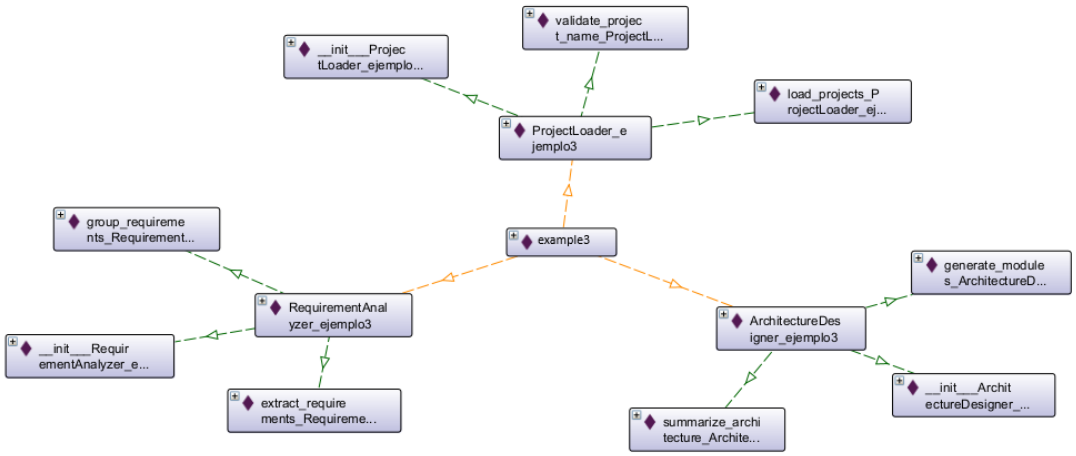

Figure 3: Ontology generated for Example 3 code. Example 3 corresponds to the module `ejemplo3.py`, a simplified object-oriented system used as a running case study in our evaluation. For clarity, only the module and its classes with methods are shown; attributes, parameters, and return types are omitted.

The proof-of-concept implementation and sample datasets are available at https://github.com/daraya78/LLM_Ontology_Mem_WAAI2025

4. Results

To assess the feasibility of the proposed approach, we implemented a proof-of-concept system and evaluated it using a controlled software engineering scenario. The task involved the progressive development of a simplified object-oriented system for managing software project architectures. The development was divided into three sequential stages: project loading, requirement analysis, and architecture design. This progression was chosen to reflect a realistic software engineering workflow while maintaining manageable complexity for controlled experimentation.

The initial codebase, contained in a module named `ejemplo3.py`, was parsed and analyzed by the ontology builder. The parser successfully extracted structural elements and mapped them to ontological entities, ensuring that the representation was both machine-interpretable and aligned with the domain semantics. The system instantiated the following classes within the ontology: `ProjectLoader_ejemplo3`, `RequirementAnalyzer_ejemplo3`, and

`ArchitectureDesigner_ejemplo3`. Each class was populated with its corresponding methods, along with metadata such as visibility, parameter types, return types, and textual descriptions derived from docstrings or inferred from naming conventions.

For example, the class `ProjectLoader_ejemplo3` was associated with three public methods: `__init__(source_path)`, `load_projects()`, and `validate_project_name(name)`. The method `load_projects()` included a descriptive annotation indicating its purpose: *"Loads projects from the specified source path"*. Similarly, `RequirementAnalyzer_ejemplo3` defined methods for extracting and grouping requirements, while `ArchitectureDesigner_ejemplo3` encapsulated logic for generating and summarizing architectural modules.

The populated ontology was serialized and stored in an OWL file (`ontologia.owl`). This persisted representation served as the authoritative source of truth for subsequent development stages. Leveraging this internal state, the system could generate structured queries, which in turn enabled the retrieval of highly relevant development information. This structured context was then automatically integrated into prompts sent to LLM1, ensuring that the generated code was consistent with previously defined structures.

A preliminary qualitative evaluation was conducted by Hernan Astudillo (software architect) who examined whether the ontology-supported configuration preserved a coherent internal representation of the system across stages. The evaluation confirmed that the approach enabled the identification of previously defined elements, supported traceable and explainable code generation, and facilitated a structured interface for visualizing system components. As part of the visualization process, a high-level system diagram was generated directly from the ontology, highlighting the interaction between components and their associated methods (see Figure 1).

These initial results confirm the viability of using dynamic ontologies to maintain context and support structured reasoning in code generation tasks involving LLMs. While quantitative performance measures were not computed at this stage, the qualitative findings provide strong preliminary evidence of the approach's promise.

5. Discussion

The results obtained from the proof-of-concept system confirm the feasibility of using a dynamically evolving ontology to support code generation tasks involving large language models (LLMs). The system demonstrated the ability to maintain semantic consistency across multiple development stages, validating the core hypothesis of this work: that a structured, external knowledge base can guide generative models in a traceable and coherent manner. This observation is consistent with evidence that coupling LLMs with external structured knowledge via retrieval, augmented generation, or knowledge graphs improves factuality, controllability, and provenance [24].

One of the main strengths of the approach lies in the use of a formal OWL ontology to capture and persist the evolving design context. By storing classes, methods, attributes, and their associated constraints, the ontology acts as an explicit long-term memory that complements the inherently limited context window of current LLMs. This enables the system to recall previously defined elements and enforce design consistency without relying solely on prompt engineering or repeated manual input.

Another notable advantage is the interpretability of the generated outputs. In contrast to typical LLM-based code assistants, whose internal reasoning is opaque, our architecture makes each decision step explainable through direct references to ontological entities. The inclusion of semantic annotations, method signatures, and preconditions allows users to trace the rationale behind generated code, aligning the approach with explainable AI principles and making it more suitable for critical or regulated domains.

Nonetheless, there are limitations to consider. The evaluation conducted was qualitative and restricted to a simplified scenario, meaning that results may not generalize directly to large-scale, real-world projects. Additionally, the LLM parameters used were temperature = 0.3, top-p = default, frequency penalty = default, presence penalty = default, max tokens = default, which may have influenced output variability. While much of the workflow is automated, some tasks, particularly prompt validation and

semantic accuracy checks, still require human oversight. This limits full automation and could be a bottleneck in larger deployments.

Future work should address scalability, as larger ontologies will require efficient reasoning strategies and possibly modular decomposition to maintain performance. Another promising direction is the integration of automated ontology validation tools to reduce human intervention. Furthermore, deeper coupling between the LLM and the ontology, whether through specialized prompting strategies, fine-tuning, or symbolic-neural hybrid methods, could enhance robustness and reduce ambiguity in context interpretation. Similar directions have been highlighted by recent studies that couple LLMs with structured knowledge sources, suggesting that hybrid symbolic–neural methods are a promising avenue for long-term consistency (Edge et al., 2025; [25, 26]. Another important direction for future work is the evaluation of the approach in a real-world business software project. Such a scenario would allow us to examine whether the ontology-based memory can persist across multiple iterations of software updates, capturing design decisions over time and validating its usefulness beyond controlled proof-of-concept settings. An additional direction for future work is the integration of software quality evaluation into the proposed framework. While our current focus was on maintaining semantic consistency and design coherence, incorporating metrics such as PEP8 compliance, code readability, or maintainability indices would allow us to assess whether the ontology layer also contributes to enforcing good coding practices. This extension would strengthen the evidence that ontology-based memory not only preserves design decisions but also improves the overall quality of the generated software.

In conclusion, the proposed architecture demonstrates the potential of combining LLMs with dynamic ontologies to enable structured, explainable, and context-aware code generation. These findings lay the groundwork for developing AI-assisted software engineering tools capable of maintaining long-term semantic consistency, supporting human oversight, and adapting to evolving project requirements.

Acknowledgments

The authors acknowledge the support provided by the National Fund for Scientific and Technological Development (FONDECYT), Chile, through project No. 1251519 “Continual learning for recognizing abnormal activity patterns related to dementia in the elderly,” led by Prof. Carla Andrea Taramasco Toro (Universidad Andrés Bello).

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edmonson, N. Reiman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[2] W. U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, arXiv preprint arXiv:2103.06333 (2021).

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, et al., Language models are few-shot learners, in: Advances in Neural Information Processing Systems, volume 33, 2020, pp. 1877–1901.

[4] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, R. Salakhutdinov, Transformer-xl: Attentive language models beyond a fixed-length context, in: Proceedings of ACL, 2019, pp. 2978–2988.

[5] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, P. Patel-Schneider, Oil: An ontology infrastructure for the semantic web, IEEE Intelligent Systems 16 (2001) 38–45.

[6] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d’Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Rob-

son, P. Kohli, N. de Freitas, K. Kavukcuoglu, O. Vinyals, Competition-level code generation with alphacode, Science 378 (2022) 1092–1097. URL: http://dx.doi.org/10.1126/science.abq1158. doi:10.1126/science.abq1158.

[7] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, Y. Zhang, Sparks of artificial general intelligence: Early experiments with gpt-4, 2023. URL: https://arxiv.org/abs/2303.12712. arXiv:2303.12712.

[8] S. Wheeler, O. Jeunen, Procedural memory is not all you need: Bridging cognitive gaps in llm-based agents, in: Adjunct Proceedings of the 33rd ACM Conference on User Modeling, Adaptation and Personalization, UMAP Adjunct '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 360–364. URL: https://doi.org/10.1145/3708319.3734172. doi:10.1145/3708319.3734172.

[9] T. Liu, C. Xu, J. McAuley, Repobench: Benchmarking repository-level code auto-completion systems, 2023. URL: https://arxiv.org/abs/2306.03091. arXiv:2306.03091.

[10] X. Liu, B. Lan, Z. Hu, Y. Liu, Z. Zhang, F. Wang, M. Shieh, W. Zhou, Codexgraph: Bridging large language models and code repositories via code graph databases, 2024. URL: https://arxiv.org/abs/2408.03910. arXiv:2408.03910.

[11] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, P. Liang, Lost in the middle: How language models use long contexts, 2023. URL: https://arxiv.org/abs/2307.03172. arXiv:2307.03172.

[12] J. Wen, J. Guan, H. Wang, W. Wu, M. Huang, Codeplan: Unlocking reasoning potential in large language models by scaling code-form planning, in: The Thirteenth International Conference on Learning Representations, 2025. URL: https://openreview.net/forum?id=dCPF1wlqj8.

[13] I. Beltagy, M. E. Peters, A. Cohan, Longformer: The long-document transformer, CoRR abs/2004.05150 (2020). URL: https://arxiv.org/abs/2004.05150. arXiv:2004.05150.

[14] A. Silva, S. Fang, M. Monperrus, RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair, IEEE Transactions on Software Engineering (2024) 1–16. URL: https://doi.ieeecomputersociety.org/10.1109/TSE.2025.3581062. doi:10.1109/TSE.2025.3581062.

[15] D. Mosquera, M. Ruiz, O. Pastor, J. Spielberger, Ontology-based automatic reasoning and nlp for tracing software requirements into models with the ontotrace tool, in: A. Ferrari, B. Penzenstadler (Eds.), Requirements Engineering: Foundation for Software Quality, Springer Nature Switzerland, Cham, 2023, pp. 140–158.

[16] S. Zander, G. Heppner, G. Neugschwandtner, R. Awad, M. Essinger, N. Ahmed, A model-driven engineering approach for ros using ontological semantics, in: 6th International Workshop on Domain-Specific Languages and models for Robotic systems (DSLRob-15), 2015.

[17] R. Pérez-Castillo, I. G.-R. de Guzmán, M. Piattini, Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems, Comput. Stand. Interfaces 33 (2011) 519–532. URL: https://doi.org/10.1016/j.csi.2011.02.007. doi:10.1016/j.csi.2011.02.007.

[18] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, C. Wang, Autogen: Enabling next-gen LLM applications via multi-agent conversations, in: First Conference on Language Modeling, 2024. URL: https://openreview.net/forum?id=BAakY1hNKS.

[19] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, J. Schmidhuber, MetaGPT: Meta programming for a multi-agent collaborative framework, in: The Twelfth International Conference on Learning Representations, 2024. URL: https://openreview.net/forum?id=VtmBAGCN7o.

[20] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, M. Sun, Chatdev: Communicative agents for software development, 2024, pp. 15174–15186. doi:10.18653/v1/2024.acl-long.810.

[21] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, O. Press, Swe-agent: agent-computer interfaces enable automated software engineering, in: Proceedings of the 38th International Conference on Neural Information Processing Systems, NIPS '24, Curran Associates Inc., Red Hook, NY, USA, 2025.

[22] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, W. Chen, RepoCoder: Repository-level code completion through iterative retrieval and generation, in: H. Bouamor, J. Pino, K. Bali (Eds.), Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Singapore, 2023, pp. 2471–2484. URL: https://aclanthology.org/2023.emnlp-main.151/. doi:10.18653/v1/2023.emnlp-main.151.

[23] O. Topsakal, T. C. Akinci, Creating large language model applications utilizing langchain: A primer on developing llm apps fast, in: Proceedings of the International Conference on Applied Engineering and Natural Sciences, volume 1, 2023, pp. 1050–1056. doi:10.59287/icaens.1127.

[24] X. Zhao, H. Li, Y. Zhang, G. Cheng, Y. Xu, Trail: Joint inference and refinement of knowledge graphs with large language models, 2025. URL: https://arxiv.org/abs/2508.04474. arXiv:2508.04474.

[25] T. Guo, Q. Yang, C. Wang, Y. Liu, P. Li, J. Tang, D. Li, Y. Wen, Knowledgenavigator: Leveraging large language models for enhanced reasoning over knowledge graph, Complex & Intelligent Systems 10 (2024). doi:10.1007/s40747-024-01527-8.

[26] Y.-H. Lin, Q.-H. Chen, Y.-J. Cheng, J.-R. Zhang, Y.-H. Liu, L.-Y. Hsia, Y.-N. Chen, Llm inference enhanced by external knowledge: A survey, 2025. URL: https://arxiv.org/abs/2505.24377. arXiv:2505.24377.