2025 IEEE/ACM 4th International Conference on AI Engineering – Software Engineering for AI (CAIN) | 979-8-3315-0219-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/CAIN66642.2025.00013

Developing Multi-Agent LLM Applications through Continuous Human-LLM Co-Programming

Hui Song, Arda Goknil
SINTEF Digital
Oslo, Norway
firstname.lastname@sintef.no

Xiaojun Jiang, Espen Melum
Oslo University Hospital
Oslo, Norway
jiang.xiaojun, espen.melum@medisin.uio.no

Hyunwhan Joe
Seoul National University
Seoul, Korea
hyunwhanjoe@snu.ac.kr

Caterina Gazzotti
University of Modena
Modena, Italy
270321@studenti.unimore.it

Valerio Frascolla
Intel Deutschland GmbH
Neubiberg, Germany
valerio.frascolla@intel.com

Adela Nedisan Videsjorden, Phu Nguyen
SINTEF Digital
Oslo, Norway
firstname.lastname@sintef.no

***Abstract*—The rapid advancement of Large Language Models (LLMs) has opened new possibilities for intelligent multi-agent systems capable of autonomously performing complex tasks. To build such systems, LLMs can be leveraged for task-solving, tool interaction, and code generation but at the same time their costs and unpredictability have to be properly managed. To do so this paper introduces COPMA, a model-based approach to enabling continuous human-LLM co-programming of multi-agent LLM applications. COPMA uses feature-block models to track application features and their implementations as agents and code blocks. Supported by co-programming patterns, developers are guided in constructing, refining, and refactoring feature implementations via trial-and-errors with LLM agents, leveraging their feedback, suggestions, and code examples. The patterns guide the shift of feature implementations between agents and code to balance flexibility, predictability, and cost. Our experience in developing LLM agents for collecting and reviewing medical research papers demonstrates that human-LLM co-programming can reduce development effort to enable rapid prototyping of multi-agent LLM applications.**

I. Introduction

The widespread use of Large Language Models (LLMs) is reshaping in the society the perception and significance of Artificial Intelligence (AI). While most people use LLMs as conversational assistants, growing attention is focused on multi-agent LLM applications [1]–[5], with multiple LLM-based agents collaborating to accomplish complex tasks, utilizing external data and tools [2]. These agents can independently handle tasks, make decisions, and generate code. However, the inherent unpredictability of LLMs also introduces the risk of unexpected behaviors [6]–[9].

Developing multi-agent LLM applications is complex, requiring the programming of intricate prompts and code to overlook the behaviour of agents and the collaboration between them, and to integrate LLMs with external, legacy tools, libraries, and services. Multi-agent frameworks, e.g., AutoGen [10], OpenAgents [11], and AgentGPT [12], offer foundational components that encapsulate LLM models into collaborative agents and facilitate their interaction with external environments. However, the construction of functioning and robust multi-agent applications still requires heavy programming effort, and effective methods and practices for this new programming paradigm do lack [13], [14].

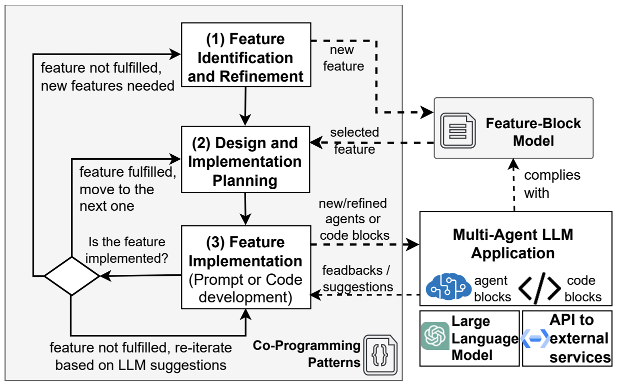
(1) Feature Identification and Refinement
feature not fulfilled, new features needed
new feature
Feature-Block Model
selected feature
(2) Design and Implementation Planning
feature fulfilled, move to the next one
complies with
new/refined agents or code blocks
Is the feature implemented?
(3) Feature Implementation (Prompt or Code development)
Multi-Agent LLM Application
agent blocks
code blocks
feedbacks / suggestions
feature not fulfilled, re-iterate based on LLM suggestions
Co-Programming Patterns
Large Language Model
API to external services

Fig. 1: Overview of COPMA.

Given the emerging role of LLMs in developing software [15], we can explore their strengths in developing multi-agent LLM applications while managing their unpredictability. This leads to our central research question: *How can developers leverage LLM intelligence to efficiently build multi-agent LLM applications while managing their inherent unpredictability?* We propose that developers *co-program* the LLM applications together with LLMs, which provide suggestions for design (e.g., task division, tool selection) and implementation (e.g., generating sample code) to incrementally build the applications. They also co-program to refactor the application, strategically moving feature implementations between the "LLM world" (for efficiency and flexibility) and the "code world" (for predictability and cost control).

In this experience paper, we introduce our initial results on the COPMA approach [provide a link to COPMA] to enabling human-LLM co-programming of multi-agent LLM applications. COPMA employs so called feature-block models with lightweight text-based notation to trace application features and their implementation blocks, for developers and

LLM agents to maintain an overview of the application and the development process. Based on the models, a set of co-programming patterns guide developers in implementing new features, refactoring existing ones, addressing unexpected behaviors, and optimizing efficiency, cost, and predictability, all in a trial and error style, i.e., developers try initial ideas with LLM agents and use their feedbacks to complete and improve the agents themselves.

Figure 1 provides an overview of COPMA, starting with developers defining key features as an initial feature-block model. In each iteration, developers select one feature and make high-level design decisions for its implementation, e.g., creating, reusing, or modifying agents and code blocks. Implementation efforts range from prompt engineering to coding. Developers typically start with simple prompts derived from the feature description, test them with LLM agents, and check the results. If the results are unsatisfactory, they interact with the agents to refine prompts or seek refinement suggestions, such as sample code or clarifications, e.g., wrapping the sample code into implementation blocks. If no immediate solution arises, new features may be added for future iterations. The co-programming patterns guide this process, providing strategies for handling common challenges. The three steps of the proposed flow, shown as white boxes with numbers in Figure 1, are the decision-making points for developers, whereas the arrows in the shadowed area are where patterns guide developers in making these decisions. COPMA is built on AutoGen [10], but its concept supports easy adaptation to other multi-agent frameworks.

To assess our approach, we apply the models and the patterns to develop a multi-agent LLM application for snowballing medical research papers on a specific disease. We observe that co-programming LLM agents can be efficient and engaging, requiring minimal manual coding effort.

II. Background & Related Work

A. LLMs and LLM agents

LLMs, such as OpenAI GPT models, represent a significant advancement in AI. Based on transformer neural networks and trained with extensive datasets through unsupervised learning, LLMs acquire a broad understanding of the world, enabling them not only to complete text but also to solve complex tasks in creative and intelligent ways [16]. LLM agents utilize the reasoning abilities of LLMs to autonomously perform complex tasks, interacting with external environments, tools, and other agents [2], [4], [5]. Once integrated with legacy systems, libraries, and databases, they can retrieve external data or manipulate environments on demand. Their decision-making abilities allow them to plan for complex tasks and adapt to dynamic environments.

Several frameworks simplify LLM agent development and integration into complex systems. AutoGen [10], for example, offers a multi-agent architecture for seamless coordination and interaction with external environments. Similarly, OpenAgents [11] and AgentGPT [12] allow developers to transform LLM models into autonomous agents. While these frameworks enable agent communication and task delegation, there is a lack of structured methodologies for continuous prototyping, feature tracking, and iterative refinement. Built on AutoGen, COPMA addresses these gaps by introducing a model-based, incremental development approach that leverages the capability of LLMs to improve development efficiency.

B. Sofware engineering for AI, LLMs, and multi-agent systems

AI engineering [17], or software engineering for AI, applies software engineering methodologies, practices, and tools to improve the efficiency of developing AI-integrated software systems [18] and ensure their quality and trustworthiness [19]. While existing approaches primarily focus on applications where AI models function as tools, we explore the development methods with AI driving integration with other tools.

Generative AI for software engineering has become a prominent research area [15], [20]–[22], exploring the application of LLM across traditional software engineering tasks such as requirement elicitation and automated testing. This paper extends this discussion by examining how software engineering practices change when the focus is on developing applications centered around Generative AI agents capable of performing tasks autonomously without being explicit programmed.

Software engineering for multi-agent systems has been a research focus for decades [23], with model-based development approaches widely adopted to ensure predictability and correctness [24], [25], using structured models to design, implement, verify, and simulate agent behaviors [24], [26]–[30] and employ semantic technologies for agent interoperability [31]. While supporting incremental feature integration and modular design, traditional methods are often limited by rigid requirements, making them less suitable for LLM-driven systems with advanced intelligence and inherent unpredictability. COPMA extends model-based development principles with an iterative co-programming approach for multi-agent LLM applications, enabling continuous development with assistance from the LLM agents being developed. This is complementary to Liu et al.'s work on agent pattern catalogue [22] by focusing on how to use patterns in a human-LLM collaborative practice.

III. Motivating Example

We start with the aim of using LLMs to collect medical research papers on Primary Sclerosing Cholangitis (PSC) disease [32]. Beginning with seed papers provided by PSC researchers, LLM agents autonomously perform *backward snowballing* by retrieving references from the seed papers and analyzing their abstracts to assess PSC relevance. References identified as PSC-relevant are used as new seeds for further snowballing. All collected papers are stored in a graph database. We aim to gather 1,000 papers before adding more agents for analysis, recommendation, and question-answering, exploring the use of AI for paper review [33]. The prototype of the snowballing application requires agents that can retrieve paper information from public libraries like PubMed [34] and Crossref [35], review and analyze abstracts, and manage a graph database to store and revisit the papers and reviews.

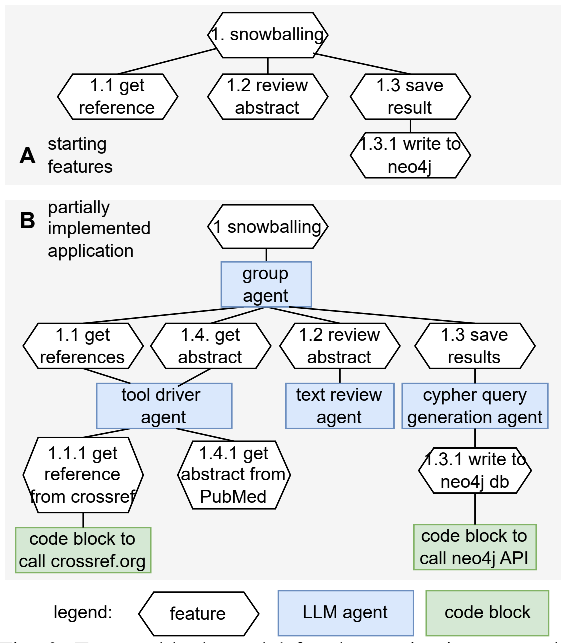

Fig. 2: Feature-block model for the motivating example.

IV. Modeling of Multi-Agent LLM Application

COPMA utilizes lightweight feature-block models to document the required features and their implementations for the application under development. Figure 2 presents the model for our example application in two stages: before implementation (A) and midway through development (B).

Development begins with a partial model outlining key features, such as the ones in Figure 2(A). The main feature is the backward snowballing of medical publications, elaborated into the following sub-features: (1.1) retrieving references, (1.2) reviewing abstracts, and (1.3) saving the review results. Based on experience, we chose Neo4j as the database [36], leading to the sub-feature (1.3.1) for writing to Neo4j. This feature modeling approach is inspired by prior research on feature models [37], [38]. However, we focus solely on essential features without accommodating alternative or optional ones.

The LLM-agent application is built of code blocks (e.g., a Python function) and LLM agents (e.g., a `ConversableAgent` in AutoGen encapsulating an LLM conversation) introduced to realize features. Each block is represented as a box linked to a feature, indicating its implementation. Figure 2(B) shows part of the implementation of the snowballing application. Feature 1.2 (review abstract) is handled by an agent for text review, while Features 1.1 (get references) and 1.4 (get abstract) are managed by a tool driver agent calling the code block under Feature 1.1.1. Feature 1.3.1 is a hanging feature (unlinked to any block) to be addressed in further iterations. The code blocks are the concrete components of the application, while features are annotations to the connections between the blocks.

Developers do not need to create a visual model: Implementation blocks are represented in the codebase as Python functions or objects (in AutoGen, agents are objects of classes like `ConversableAgent`). Features are documented directly in the codebase as comments or markdown cells, with nested numbering for easy navigation of the feature hierarchy.

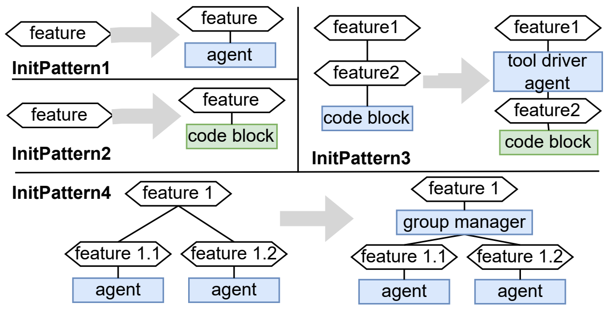

Fig. 3: Initialization patterns.

V. Human-LLM Co-Programming Patterns

COPMA facilitates human-LLM co-programming of LLM agents through patterns that guide developers in adding and refining implementation blocks and identifying missing features, in three categories: ***initialization***, ***refinement***, and ***refactoring***.

A. Initialization patterns: adding blocks to implement features

Initialization patterns link unimplemented features with initial code or agent blocks (Figure 3).

Init-Pattern 1: *Bind a hanging leaf feature to an agent.* This pattern provides a *default choice* for handling *hanging features* by linking them to an LLM-powered agent. If no suitable agent is available, a new one should be created. The initial prompt to the LLM agent can be derived from the feature description. For example, we can implement Feature 1.2 (review abstract) using an LLM agent with the following prompt: *"Given an abstract of a paper, determine if it relates to PSC (Primary Sclerosing Cholangitis). Reply with YES or NO, followed by a justification"* We test this by attaching a sample abstract, and the agent returns a plausible result. Similarly, we can link Feature 1.1 (get references) to an agent with the following prompt: *"Given the DOI of a paper, return all references"* Initially, GPT suggests that access to an online publication database is required. Further interaction with the agent provides a sample code snippet using the `crossref.org` Application Programming Interface (API). The agent does not yet retrieve references independently, which is addressed by refinement patterns. The core of this pattern is that if the implementation of a feature is unclear, try the initial ideas with an LLM agent, which may yield a solution or bring us closer to one.

Init-Pattern 2: *Bind a hanging leaf feature to a code block.* If a clear solution exists, we can directly link a hanging feature to a code block (e.g., a Python function), allowing developers to bypass the trial and error phase with the LLM. For instance, Feature 1.3.1 (write to graph database) can be implemented as a Python function that accepts a Cypher query [39] and executes it using the Neo4j API [36].

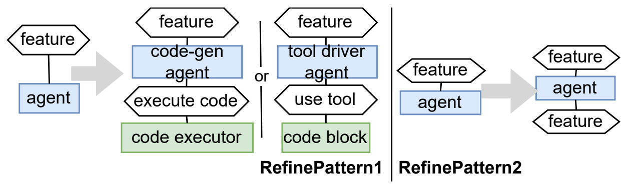

Fig. 4: Refinement patterns.

Init-Pattern 3: *Insert a tool driver agent on top of code blocks*. This pattern handles cases where an unimplemented feature has a sub-feature already implemented by a code block. It suggests using a tool driver agent to control and invoke the code block of the sub-feature. This intermediate agent manages the execution, adding control and flexibility to the implementation. For example, once Feature 1.3.1 (write to Neo4j), a sub-feature of Feature 1.3 (save results), is implemented as a Python function, a tool driver agent (i.e., the "cypher query generation agent") is needed to generate Cypher queries for merging data into the graph, and to use the Python function to execute the query.

Init-Pattern 4: *Insert a group manager agent on top of multiple agents*. It applies when the organization of multiple agents for a complex task is unclear. A group manager—a specialized agent—coordinates other agents by dividing the task into sub-tasks based on the capabilities of each agent and managing their execution order. The feature between the group manager and a group member is refined into an introduction prompt that informs the manager of the abilities of each agent. For instance, once we implement agents for retrieving references, reviewing abstracts, and saving results, we can register them into a group, where the prompt to the group manager is the definition about the snowballing process.

B. Refinement patterns: following the feedbacks of LLM

Refinement patterns complete feature implementations by leveraging the feedbacks and suggestions of LLM (Figure 4).

Refine-Pattern 1: *Transform sample code suggested by an LLM agent into an executable code block.* This pattern applies when an agent cannot directly solve a task but provides example code demonstrating a solution using external tools. There are two refinement options. As the first option, we can introduce (or reuse) a code executor to run the sample code. In AutoGen, a code executor (a Python class) wraps a local or virtual runtime (e.g., Python, shell, or R) to execute agent-generated code. Alternatively, we can wrap the sample code into a code block as an external tool for the original agent, similar to the structure in Init-Pattern 3. In line with the example of Init-Pattern 1, we implement Feature 1.1 with an agent, resulting in a sample code snippet for retrieving references from `Crossref.org`, which we then wrap into a Python function and register as a tool for the agent.

Refine-Pattern 2: *Add a new feature if an agent indicates a lack of external tools for the assigned task.* This pattern applies when an agent cannot suggest specific tools or sample code to address the problem, indicating a need for a new supporting feature implemented in a future iteration. Continuing with the Init-Pattern 4 example, we introduce a group manager to oversee agents responsible for obtaining references, reviewing abstracts, and saving results. Testing show that the data obtained by the reference-retrieval agent lack abstracts of references due to limited data on `crossref.org`, preventing the group manager from fully implementing the snowballing feature. Further dialogue with the agent provides no immediate solution for retrieving abstracts. Therefore, we add a new feature under the group agent, Feature 1.4, as shown in Figure 2. In a later iteration, consultation with a medical colleague reveals PubMed as a reliable source for abstracts of medical publications. We then refine Feature 1.4 into Feature 1.4.1 (retrieve abstract from PubMed). The agent then generates code to access PubMed via the BioPython library, which we encapsulate into a Python function. This completes a functional prototype of the snowballing feature.

Other refinement patterns: Different feedback or behavior of LLM agents may suggest different ways to refine the implementation. For example, if the agent often generates code or Cypher queries with similar mistakes, its prompt should be refined with instructions or counterexamples. If an agent can handle multiple types of tasks but often make mistakes on a particular type, a new agent should be spun off, with focused and specially optimized prompts or tools.

C. Refactoring patterns: balancing between agent and code

Refactoring patterns enhance implementations by reorganizing code and agent structures to improve maintainability, efficiency, scalability, and predictability without changing functionality. The core strategy is to shift implementation between the "LLM world" and the "code world," balancing extensibility, adaptability, and rapid development with predictability, faster execution, and lower costs.

Refactor-Pattern 1: *Switching between code generation and tool invocation*. One option is to have the agent generate code at runtime for each task, supporting adaptability to unexpected situations by adjusting the code as needed. However, regenerating code each time is costly and can introduce unpredictability, such as compilation errors or incorrect data extraction. Alternatively, code snippets can be encapsulated as code blocks in the application for agents to call them. In our example, we eventually wrap all generated Python code as external tools, while Cypher queries for graph reading and writing are generated by the LLM.

Refactor-Pattern 2: *Switching between smart and fixed workflows*. A group manager agent can autonomously plan and coordinate workflows among agents, but this can lead to task misassignment or missing context. To reduce unpredictability, consider these three options: (1) introduce a dedicated "moderator" agent to plan and oversee workflows, with fine-grained instructions in its prompt; (2) fix the execution order by adding transition rules to the group or replace the group with a sequence of agents; (3) substitute the group manager with a code block to orchestrate agents programmatically. Increasing workflow rigidity enhances stability but restricts the group to a specific high-level task.

TABLE I: Statistics of the snowballing use case.

Development process					
features:	38	agents:	6	code blocks:	21
prompts:	27	sentences:	78	tokens:	∼1340
code:	552	(lines, of which generated by LLM:			∼350)
iterations:	80	days:	8	hours:	18
initiation:	30%	refinement:	50%	refactoring:	20%
cost :	$4.2	**API cost (USD) via OpenAI subscription*			
Operational result					
papers:	936	valid:	861	PSC:	255
verified:	72	with review:	63	accurate:	62
cost:	$18.5	throughput	3.2	*papers / min in average*	

Refactor-Pattern 3: *Breaking long-lasting and recurrent tasks into small ones.* Tasks often involve multiple, sometimes repetitive, sub-tasks. For example, snowballing from a source paper requires obtaining references, reviewing each one, and saving results. As agents handle each sub-task, the growing context can lead to unpredictable behavior and higher costs. A solution is to break the task into smaller steps: a first one to get references and save them as papers in the graph, a second one to retrieve and save abstracts, and a third one to review each paper with its abstract. This approach mirrors the technique of recording intermediate results to aid human thinking.

VI. Case Study

We applied COPMA to develop a prototype of an LLM-based snowballing application for PSC research papers, serving as both a proof of concept and a foundation for future collaboration on using LLMs for medical research. Table I provides key statistics on the development process and operational results, built with Python and Jupyter Notebook, using AutoGen 0.2.35 and OpenAI GPT-4o (version 2024-08-06).

The snowballing feature is refined into 38 features across four layers, implemented as 6 agents and 21 code blocks (Python functions), with 27 prompts and 552 lines of Python code—of which approximately 350 lines were LLM-generated with minor edits. Development involves 80 major iterations, including adding, removing, or merging agents and code blocks. About 50% of the iterations follow refinement patterns, with 30% for initiation and 20% for refactoring.

The prototype has thus far snowballed 1231 papers from one seed paper [40]. A PSC researcher manually reviewed 63 papers (within 72 direct references of the seed, 9 were not successfully reviewed by the agents due to missing DOIs or abstracts, mostly non-medical or from non-mainstream sources), only one showed a discrepancy: the expert identified it as PSC-related, while the agents did not. She confirmed that this accuracy is sufficient to continue snowballing. The development and trial run incurred costs of $4.20 and $18.50. The agents process 3.2 papers per minute in average.

Development efficiency. Co-programming with LLM has significantly reduced development effort by leveraging LLM-generated code and employing agents to plan workflows. The snowballing process, for example, utilizes ten distinct Cypher queries to Neo4j database—such as finding PSC-related papers as new seeds for snowballing. All these queries are generated on-site by the LLM agents, eliminating the need for manual programming and tuning. LLM agents also streamline interoperability, as they can autonomously extract key information from API responses and communicate seamlessly without strict formatting or semantic annotation. With these benefits, a single developer completed the prototype as a side project over eight days, totaling 18 effective hours. In contrast, the same developer estimated that a full week (40+ effective hours) would have been needed to develop it traditionally.

Predictability and hallucination. After extensive refactoring—such as rewriting prompts, shifting implementations from LLM to code, and breaking large tasks, the snowballing application now demonstrates highly stable behavior and is able to autonomously collect 900 papers with accurate review results. The agents reliably handle large data blocks, such as using a list of 100 DOIs to compose Cypher queries without transcription errors. Only one instance of hallucination was observed when the reference-retrieval agent received an empty list from the online library and created three fake DOIs. Occasionally, agents generate Cypher queries with syntax errors but can autonomously correct them based on the error feedback of Neo4j. We observe some issues with conversation termination. For instance, agents sometimes continue tasks beyond their scope, requiring manual interruption. On the other hand, the LLM agents demonstrates robustness in handling edge cases from external tools, e.g., they filter out `unknown DOI`s from the library responses without being explicitly prompted to do so.

VII. Conclusion & Future Work

In this paper, we present our approach and experience in developing multi-agent LLM applications through continuous human-LLM co-programming, with potential to reduce manual coding effort, enable rapid prototyping of LLM-based applications with external systems, and achieve reliable behavior.

Our next step will be focused on defining and evaluating a more systematic, general-purpose, and automated human-LLM co-programming approach. Future work will extend COPMA with patterns on more complex multi-agent systems, incorporating advanced tasks such as automated reasoning, large-scale data analysis, and real-time tool interactions. We will explore further automation by developing meta-level agents that analyze the development context and make automatic decisions for developers. Additionally, we will improve error detection and correction by developing automated mechanisms that allow agents to identify, diagnose, and resolve syntax errors, logical issues, and workflow inconsistencies in real time. This includes implementing advanced debugging capabilities and guardrails [41] that enable LLM agents to handle unexpected situations autonomously, improving system reliability.

Acknowledgment

The work is partially funded by the European Commission's Horizon Europe framework programme under grant agreement No. 101135576 (INTEND), 101070455 (DYNABIC), and the International Research Development Program of the National Research Foundation of Korea under RS-2023-00268071.

References

[1] Z. Liu, H. Hu, S. Zhang, H. Guo, S. Ke, B. Liu, and Z. Wang, "Reason for future, act for now: A principled architecture for autonomous llm agents," in *ICML'24*, 2024.

[2] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[3] Y. Talebirad and A. Nadiri, "Multi-agent collaboration: Harnessing the power of intelligent llm agents," *arXiv preprint arXiv:2306.03314*, 2023.

[4] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang *et al.*, "Agentbench: Evaluating llms as agents," *arXiv preprint arXiv:2308.03688*, 2023.

[5] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

[6] D. Ganguli, D. Hernandez, L. Lovitt, A. Askell, Y. Bai, A. Chen, T. Conerly, N. Dassarma, D. Drain, N. Elhage *et al.*, "Predictability and surprise in large generative models," in *FAccT'22*, 2022, pp. 1747–1764.

[7] T. A. Chang and B. K. Bergen, "Language model behavior: A comprehensive survey," *Computational Linguistics*, vol. 50, no. 1, pp. 293–350, 2024.

[8] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[9] R. Schaeffer, B. Miranda, and S. Koyejo, "Are emergent abilities of large language models a mirage?" *NeurIPS'23*, vol. 36, 2024.

[10] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.

[11] T. Xie, F. Zhou, Z. Cheng, P. Shi, L. Weng, Y. Liu, T. J. Hua, J. Zhao, Q. Liu, C. Liu *et al.*, "Openagents: An open platform for language agents in the wild," *arXiv preprint arXiv:2310.10634*, 2023.

[12] AgentGPT, "Agentgpt: a tool to configure and deploy autonomous ai agents," https://agentgpt.reworkd.ai/, Visited in 2024.

[13] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, 2024.

[14] V. Dibia, "Multi-agent llm applications — a review of current research, tools, and challenges," *designingwithml.com*, 2024.

[15] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[16] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[17] I. Ozkaya, "What is really different in engineering ai-enabled systems?" *IEEE software*, vol. 37, no. 4, pp. 3–6, 2020.

[18] Y. Cheng, J. Chen, Q. Huang, Z. Xing, X. Xu, and Q. Lu, "Prompt sapper: a llm-empowered production tool for building ai chains," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–24, 2024.

[19] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, and S. Wagner, "Software engineering for ai-based systems: a survey," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, pp. 1–59, 2022.

[20] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative ai," *Automated Software Engineering*, vol. 31, no. 1, p. 26, 2024.

[21] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *ICSE-FoSE'23*, 2023, pp. 31–53.

[22] Y. Liu, S. K. Lo, Q. Lu, L. Zhu, D. Zhao, X. Xu, S. Harrer, and J. Whittle, "Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents," *Journal of Systems and Software*, vol. 220, p. 112278, 2025.

[23] A. G. R. Choren, C. L. P. Giorgini, and T. H. A. Romanovsky, "Software engineering for multi-agent systems iv," 2004.

[24] G. Kardas, "Model-driven development of multiagent systems: a survey and evaluation," *The Knowledge Engineering Review*, vol. 28, no. 4, pp. 479–503, 2013.

[25] J. Pavón, J. Gómez-Sanz, and R. Fuentes, "Model driven development of multi-agent systems," in *ECMDA-FA'06*, 2006, pp. 284–298.

[26] G. Kardas, A. Goknil, O. Dikenelli, and N. Y. Topaloglu, "Modeling the interaction between semantic agents and semantic web services using mda approach," in *ESAW'06*, 2007, pp. 209–228.

[27] ——, "Model transformation for model driven development of semantic web enabled multi-agent systems," *MATES'07*, pp. 13–24, 2007.

[28] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "Agent-oriented model-driven development for jade with the jadel programming language," *Computer Languages, Systems & Structures*, vol. 50, pp. 142–158, 2017.

[29] F. Santos, I. Nunes, and A. L. Bazzan, "Model-driven agent-based simulation development: A modeling language and empirical evaluation in the adaptive traffic signal control domain," *Simulation Modelling Practice and Theory*, vol. 83, pp. 162–187, 2018.

[30] G. Kardas, A. Goknil, O. Dikenelli, and N. Y. Topaloglu, "Metamodeling of semantic web enabled multiagent systems," *Multiagent Systems and Software Architecture*, vol. 2006, pp. 79–86, 2006.

[31] ——, "Model driven development of semantic web enabled multi-agent systems," *International Journal of Cooperative Information Systems*, vol. 18, no. 02, pp. 261–308, 2009.

[32] T. H. Karlsen, T. Folseraas, D. Thorburn, and M. Vesterhus, "Primary sclerosing cholangitis–a comprehensive review," *Journal of hepatology*, vol. 67, no. 6, pp. 1298–1323, 2017.

[33] H. Pearson, "Can ai review the scientific literature — and figure out what it all means?" *Nature*, vol. 635, pp. 276–278, 2024.

[34] K. Canese and S. Weis, "Pubmed: the bibliographic database," *The NCBI handbook*, vol. 2, no. 1, 2013.

[35] G. Hendricks, D. Tkaczyk, J. Lin, and P. Feeney, "Crossref: The sustainable source of community-owned scholarly metadata," *Quantitative Science Studies*, vol. 1, no. 1, pp. 414–427, 2020.

[36] Neo4j, "Neo4j: Graph database and analytics," https://neo4j.com/, Visited in 2024.

[37] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information systems*, vol. 35, no. 6, pp. 615–636, 2010.

[38] K. C. Kang, J. Lee, and P. Donohoe, "Feature-oriented product line engineering," *IEEE software*, vol. 19, no. 4, pp. 58–65, 2002.

[39] Cyper, "Cyper query language," https://opencypher.org/, Visited in 2024.

[40] C. L. Zimmer, E. von Seth, M. Buggert, O. Strauss, L. Hertwig, S. Nguyen, A. Y. Wong, C. Zotter, L. Berglin, J. Michaëlsson *et al.*, "A biliary immune landscape map of primary sclerosing cholangitis reveals a dominant network of neutrophils and tissue-resident t cells," *Science Translational Medicine*, vol. 13, no. 599, p. eabb3107, 2021.

[41] A. Biswas and W. Talukdar, "Guardrails for trust, safety, and ethical development and deployment of large language models (llm)," *Journal of Science & Technology*, vol. 4, no. 6, pp. 55–82, 2023.