Harnessing AutoGen: Implementation, Insights, and Prospects in Multi-Agent Systems

Jue Kong

6.7.2025

Abstract

AutoGen is a multi-agent system platform designed for large language models, providing greater flexibility and scalability for complex tasks through modular role assignment, tool invocation, and message orchestration. Adopting an operating-system-level engineering perspective, this paper systematically reviews the core architecture of AutoGen and introduces a generalizable agent taxonomy for multi-agent systems. Based on this framework, we identify and clarify several critical issues present in the current platform and propose targeted optimization strategies. Furthermore, we conduct a preliminary exploration of how knowledge editing and dynamic boundary adjustment mechanisms can enhance the capabilities of expert agents, offering theoretical references for future research.

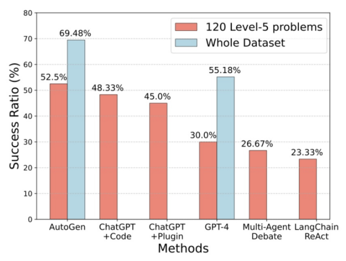
1 Introduction

In recent years, neural network architectures based on self-attention mechanisms, particularly the Transformer [36], have become the mainstream approach in the field of natural language processing (NLP). As model sizes have increased, with parameter counts surpassing hundreds of billions [2, 13], large language models (LLMs) have significantly propelled the development of both NLP and agent systems. Pretrained models represented by the GPT series [31, 2] and BERT [5] are widely applied in downstream tasks, serving as a crucial foundation for agent-based systems.

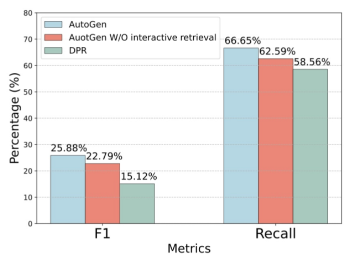
In the study of agent systems and complex automated tasks, traditional solutions have largely relied on single-agent architectures, which are suitable for simple dialogue generation, information retrieval, and tool invocation. However, as task requirements become more complex, the limitations of single-agent architectures in task decomposition, multi-role collaboration, and cross-domain information integration have become increasingly apparent. Benefiting from the enhanced understanding and reasoning capabilities of large language models, both academia and industry have begun to explore multi-agent collaborative frameworks, where role allocation and efficient information flow are leveraged to tackle complex tasks [16, 39].

The AutoGen framework is a response to this trend. By integrating large language model capabilities with multi-agent collaboration mechanisms, AutoGen enables researchers to construct complex dialogue systems and task-solving workflows in a modular and scalable manner. Through abstracting agent roles, tool interfaces, and message-passing mechanisms, AutoGen achieves flexible agent composition and cooperation, thereby improving system generality and practical utility.

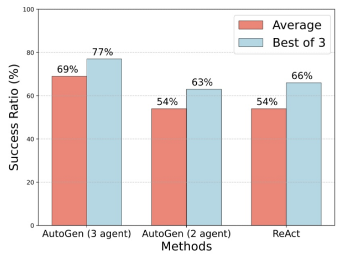
Preliminary experiments show that for small- and medium-scale tasks, the performance gap between single-agent and multi-agent systems is limited, and the additional overhead introduced by multi-agent collaboration—such as token consumption and context passing—must be carefully weighed. However, in large-scale and diversified task environments, multi-agent systems demonstrate stronger adaptability and scalability. AutoGen has exhibited superior performance in multiple benchmark evaluations, especially in complex collaborative task scenarios, where some metrics already match or surpass traditional single-agent solutions.



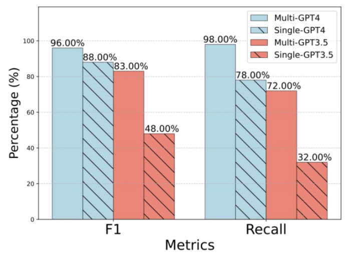
(a) A1: Performance on MATH (w/ GPT-4).



(b) A2: Q&A tasks (w/ GPT-3.5).



(c) A3: Performance on ALFWorld.



(d) A4: Performance on OptiGuide.

Figure 1: multi-agent performance comparison.

In summary, this paper focuses on the multi-agent dialogue and task orchestration mechanisms enabled by the AutoGen architecture. We systematically analyze its advantages and challenges in practical applications and conduct experiments based on the latest open-source framework, aiming to provide insights for the design and implementation of next-generation agent systems.

The main contributions of this work are as follows:

- A systematic exposition of AutoGen's design principles, core components, and practical usage.
- An analysis of agent configuration patterns and typical multi-agent collaboration strategies in AutoGen.
- A comparative study of mainstream platforms, with a novel classification scheme for multi-agent systems that emphasizes the multi-dimensional evaluation of agents, including role knowledge, tool-operational knowledge, and overall knowledge boundaries.
- An analysis of AutoGen's advantages in supporting agent diversity and interactive flexibility, as well as its limitations in logical consistency and cost efficiency.
- A proposal of knowledge type classification criteria based on agent taxonomy, providing references for future research.

2 Related Work

2.1 General Related Work

As previously mentioned, early applications of agent systems typically relied on single-agent architectures, utilizing large language models (LLMs) via prompt chaining and API invocation for dialogue generation, information retrieval, and external tool utilization, thus significantly improving usability for relatively simple automation tasks. Representative examples include the ChatGPT plugin ecosystem and AutoGPT platforms.

However, as single-agent architectures increasingly reveal limitations in information flow, role specialization, and cross-domain collaboration, multi-agent architectures have gained substantial attention within academia and complex application scenarios, becoming one of the most active areas of research aimed at enhancing LLM systems. For instance, the CAMEL framework systematically introduced multi-role collaborative mechanisms based on role-playing to decompose complex reasoning chains [14]. LangChain emphasizes external tool integration to strengthen reasoning and execution capabilities, while projects such as Smallville [28] explore long-term agent collaboration based on social interactions, enriching the evolutionary modes of agent ecosystems.

Building upon these explorations, AutoGen was jointly proposed by institutions such as Microsoft Research and Pennsylvania State University [39]. This framework further incorporates system-level abstraction and scheduling mechanisms, officially described as a "multi-agent conversational framework for building AI agents and applications" [39]. In this paper, we regard AutoGen as an "operating system" tailored for large language models (LLMs)—a conceptual alignment similar to definitions given by the AG2 branch—where each agent can be viewed as a heavyweight remote reasoning thread. AutoGen not

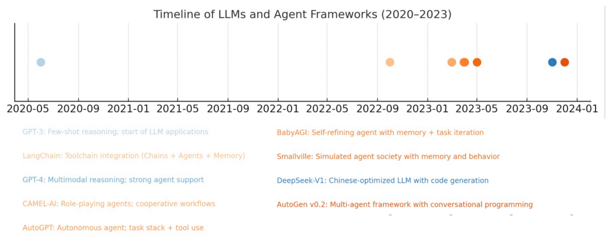


Figure 2: Representative multi-agent frameworks.

only emphasizes flexible agent roles and task definitions but also manages runtime scheduling, inter-thread communication, and system-level operations, thus achieving a unified management capability for parallel tasks analogous to traditional operating systems [23]. The official architecture diagram (see Fig. 3) clearly illustrates this design philosophy.

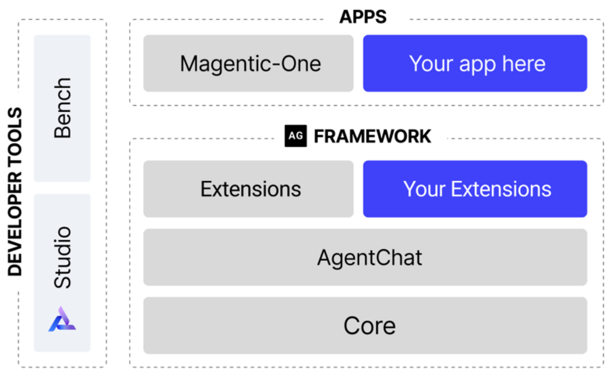


Figure 3: The official AutoGen architecture diagram [23].

Notably, in November 2024, the AutoGen project underwent a community-driven fork initiated by original development team members, known as AG2, emphasizing architectural compatibility and openness. Concurrently, Microsoft's official team continues to evolve the main AutoGen branch and has formally announced deep collaboration between AutoGen and Semantic Kernel, aiming at unifying and interoperating the multi-agent runtime (autogen-core) with Semantic Kernel [22, 10].

Furthermore, recent research efforts around multi-agent collaboration, tool augmentation, and dialogue management continue to flourish. For example, StateFlow introduced state-driven workflows for task-solving, and Multi-Agent Debate explored adversarial reasoning among agents. Collectively, these studies provide important methodological foundations and technical reserves for developing next-generation LLM-based intelligent applications.

Overall, the AutoGen project and its community fork AG2 substantially advance the practical application of multi-agent systems in complex task scenarios through their modular design, multi-role collaboration, and system-level scheduling mechanisms. This paper will further discuss optimization approaches and practical implications of multi-agent collaboration within the AutoGen framework.

2.2 Focused Project: AutoGen Framework and Multi-Agent Mechanisms

The AutoGen project was initially launched by Microsoft Research with the goal of enhancing the effective application of large language models (LLMs) in multi-agent collaboration scenarios. It addresses bottlenecks encountered by traditional single-agent systems in handling complex tasks, including long-context management, tool invocation, role specialization, and task coordination.

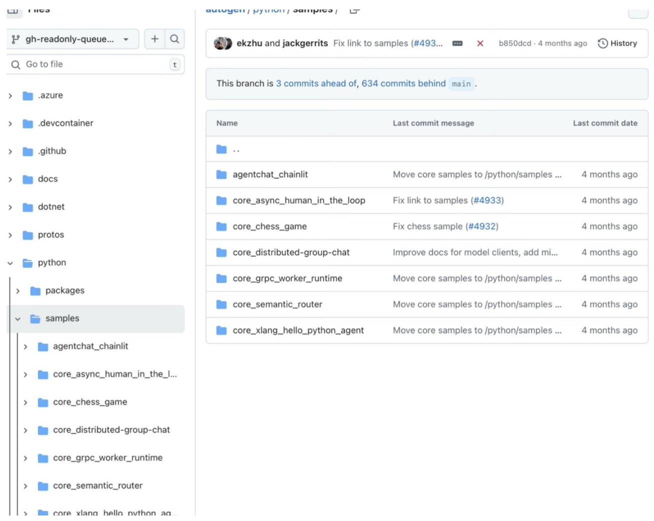
By early 2025, the AutoGen framework had evolved to version v0.4. Compared with v0.2, v0.4 underwent significant architectural and implementation restructuring, notably introducing stricter asynchronous scheduling mechanisms and interface standards, requiring extensive revisions to previous documentation (Appendix B).

In version 0.4, AutoGen maintains its core strengths in modular design and platform openness. Its architecture primarily comprises three modules: *autogen-core*, *autogen-agentchat*, and *autogen-ext*, which respectively handle foundational logic, agent interaction mechanisms, and external model integration. AutoGen also features visualization interfaces and performance benchmarking modules to enhance development efficiency and system reliability (Appendices A, C).

Regarding multi-agent collaboration, AutoGen offers a variety of built-in agent types, all inheriting from a unified abstract base class *BaseChatAgent*, which defines core interfaces and methods for message handling (Appendix D). AutoGen supports flexible group-chat scheduling methods like round robin to automatically manage agent interaction sequences, while each agent can customize message processing via callback functions, balancing ease of use and high customizability (Appendix F).

In the current example directory, the official implementations continue to reflect the multi-agent configuration strategies recommended in the 2023 publication (Appendix E).

Although the official implementation and recommended configurations can handle certain typical application scenarios, there remain unresolved issues in practical tasks such as delayed responses in tool invocation, imprecise knowledge boundary definitions, and challenges in dynamic knowledge updates and interactive flow optimization. Overall, despite AutoGen's strengths in architectural openness and multi-agent configuration strategies, further improvements are necessary in practical aspects such as tool invocation response efficiency, granular knowledge control, and dynamic dialogue flow optimization. This paper will



ekzhu and jackgerrits Fix link to samples (#493... b850dcd · 4 months ago

This branch is 3 commits ahead of, 634 commits behind `main`.

Name	Last commit message	Last commit date
..		
agentchat_chainlit	Move core samples to /python/samples ...	4 months ago
core_async_human_in_the_loop	Fix link to samples (#4933)	4 months ago
core_chess_game	Fix chess sample (#4932)	4 months ago
core_distributed-group-chat	Improve docs for model clients, add mi...	4 months ago
core_grpc_worker_runtime	Move core samples to /python/samples ...	4 months ago
core_semantic_router	Move core samples to /python/samples ...	4 months ago
core_xlang_hello_python_agent	Move core samples to /python/samples ...	4 months ago

Figure 4: Screenshot from the official AutoGen example directory.

address these shortcomings, proposing corresponding optimization approaches and practical recommendations.

3 Discussion

As presented in Appendix F, we demonstrate several task-specific examples based on AutoGen. This section begins by comparing AutoGen and the CAMEL platform, then explores the theoretical differences in multi-agent system architectures. We clarify several key definitions, discuss the core strengths of AutoGen, and highlight the challenges it may face in future development.

3.1 Comparison of Built-in Agents in AutoGen and CAMEL

It is worth noting that CAMEL also provides built-in agent classes for constructing LLM-based multi-agent systems (see Appendix D). Both AutoGen and CAMEL adopt a similar three-layer architecture:

- The top layer defines abstract base classes for agent interfaces (e.g., `BaseAgent` in CAMEL and `Agent` or `LLMAgent` in AutoGen);
- The middle layer consists of general agents that implement fundamental dialogue logic and tool invocation mechanisms (e.g., `ChatAgent` in CAMEL and `ConversableAgent` in AutoGen);
- The bottom layer comprises various subclasses for specific, fine-grained task roles.

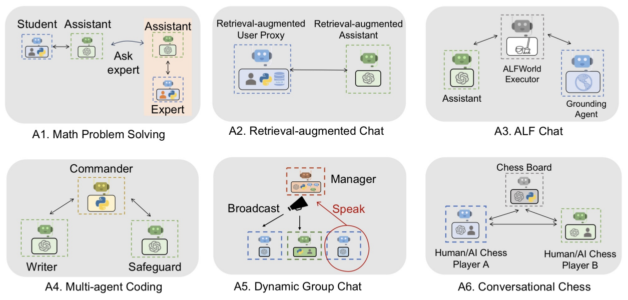


Figure 5: Agent interaction mechanism as illustrated in the AutoGen paper [39].

Although both frameworks share similar inheritance structures and foundational types, their implementation strategies and design philosophies show significant differences:

- **AutoGen** emphasizes function-driven tool invocation and fine-grained control over dialogue processes, making it particularly well-suited for DevOps-style engineering automation workflows (e.g., automated planning, execution, and review).
- **CAMEL**, on the other hand, focuses on emergent behaviors by defining a variety of persona-based agents and cognitive roles (such as `DeductiveReasonerAgent`), aiming to simulate knowledge organization and reasoning as in human collaborative dialogues.

These distinctions reflect two core philosophies: "engineering application-oriented" for AutoGen, and "cognitive role-oriented" for CAMEL. AutoGen is advantageous in strict and explicit business process automation, whereas CAMEL's persona-rich agent modeling is better suited to prototyping, social simulation, or human-computer interaction scenarios. Rather than direct competitors, the two frameworks are largely complementary and their positioning affects user choices depending on the application scenario.

3.2 Agent Taxonomy Based on Functionality and Role

From the above comparison, it is clear that neither AutoGen nor CAMEL has yet established a strictly theoretical agent taxonomy. Most built-in agent types are encapsulated to address practical needs. Nevertheless, the different encapsulation strategies provide valuable insights for constructing an effective taxonomy. Specifically, the main distinction in agent encapsulation between the two platforms can be captured along two dimensions:

- **Functional agents**: Focused on tool invocation, API operations, and task execution;

- **Role-playing agents**: Focused on knowledge expression, semantic roles, and cognitive characteristics.

In practice, neither AutoGen's customized `AssistantAgent` nor CAMEL's role assignment agents can fully satisfy real-world task requirements as single agent types. Often, "one-shot agents"—agents with both distinct persona traits and tool invocation abilities—are needed. Thus, both functional and role-playing agents may be considered specializations of the more general one-shot agent paradigm.

3.3 Reconstruction and Optimization of Agent Taxonomy from a Computational Theory Perspective

Building on these definitions, we introduce a cross-disciplinary perspective to reconstruct the agent taxonomy from a computational theory standpoint.

In computer science, any LLM-driven agent can essentially be viewed as a concrete instantiation of a Turing machine. Regardless of whether these agents are tool-oriented, intelligence-oriented, or human-like, their computational and reasoning limits are governed by the boundaries of Turing machine theory. That is, problems unsolvable by a Turing machine (such as the halting problem) are also unsolvable by these agents. Additionally, when dealing with massive knowledge bases, large language models may encounter difficulties in knowledge retrieval, leading to hallucinations and unreliable outputs. Therefore, while enhancing agent knowledge capabilities, it is crucial to define clear knowledge boundaries to ensure logical consistency and output reliability.

Drawing an analogy between Turing machines and programs, we seek a more practical and actionable expression of agent knowledge boundaries. In computational complexity theory, the class of "P problems" (problems solvable in polynomial time) is often seen as a critical boundary for task manageability and control. By incorporating the "P problem" framework into agent behavior and capability design, we can more effectively delineate which tasks fall within an agent's controllable, predictable domain and which tasks exceed its capabilities, thus requiring human intervention or additional constraints to improve system safety and explainability.

Moreover, during agent knowledge modeling and reasoning, different "language series" (such as natural or formal language) have a direct impact on knowledge expression and inference. In other words, the so-called "knowledge focus" essentially refers to the limitation imposed on agent abilities by the choice of expression and reasoning language. This focus ensures consistency and controllability in knowledge representation and task execution.

Based on these considerations, we propose a comprehensive and practice-oriented agent taxonomy:

$$\text{API tools} + \text{operational knowledge} \rightarrow \text{tool-ness (of agent)} \tag{1}$$

$$\text{API tools} + \text{operational knowledge} = \text{tool agent} \tag{2}$$

$$\text{Knowledge focus (in expression)} \rightarrow \text{role-ness (of agent)} \tag{3}$$

$$\text{Well-defined knowledge boundary} \rightarrow \text{expertness (of agent)} \tag{4}$$

$$\text{Role-ness} - \text{Expertness} = \text{role-playing agent} \tag{5}$$

$$\text{Role-ness} + \text{Expertness} = \text{role-incorporated agent (or expert agent)} \tag{6}$$

$$\text{Tool-ness} + \text{Role-ness} = \text{one-shot agent} \tag{7}$$

$$\text{Role-playing agent} + \text{Tool-ness} = \text{one-shot agent (Type 1)} \tag{8}$$

$$\text{Role-incorporated agent} + \text{Tool-ness} = \text{one-shot agent (Type 2)} \tag{9}$$

This classification offers a theoretically rigorous and practically relevant agent taxonomy, which can both guide real-world development and provide a foundation for future research.

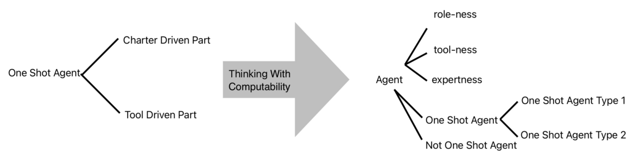


Figure 6: Evolution of agent taxonomy.

3.4 Multi-Agent Configuration Recommendations Based on the Taxonomy

Based on the above agent taxonomy, we propose several recommendations for practical multi-agent configuration:

- Decouple the execution and verification steps from knowledge-intensive agents and assign them to dedicated tool agents, reducing risks of knowledge conflict and logical inconsistency.

- For parts of a task involving multi-dimensional knowledge expression and semantic role use, clarify whether role-playing, role-incorporated, or one-shot agent (Type 1/2) is needed according to the specific task's knowledge focus and boundary requirements.

In summary, we recommend not viewing the agent as a universal entity with all tools and knowledge, but rather as a collaborative ensemble with distinct roles and clear boundaries. Tool invocation should not be seen as a single agent's function, but as explicit collaboration and interaction among different agents.

Such a strategy of clear division of responsibility and functional decoupling is not only suited for intelligent system architecture in engineering, but also provides useful guidance for real-world team management and organizational practice. Clear role boundaries and specialization help to avoid inefficiency and lack of execution that arise from ambiguous role definitions.

4 Future Directions

4.1 Continuous Iteration of the AutoGen Platform to Overcome Technical Bottlenecks

A primary objective for future research is the continuous iteration and optimization of cutting-edge platforms such as AutoGen. Particular attention should be paid to optimizing core technical bottlenecks identified in user experience reports, including resource consumption, context management, and the efficiency of multi-turn interactions. By further refining the underlying architecture, and improving agent-level parallel scheduling and memory management strategies, it is possible to significantly enhance both user interaction experience and overall system performance.

4.2 In-depth Study of Agent Role Allocation to Improve Resource Efficiency

Through more granular analysis of task requirements, we recommend implementing precise agent role allocation and task integration, thereby maximizing platform resource utilization. Specific measures include:

- For tasks of moderate complexity, partially merging agents or stages and employing a hybrid model that combines multi-agent and single-chain reasoning (CoT) can effectively control resource consumption.
- By leveraging the concepts developed above, system prompts (System Messages) should be streamlined and optimized based on knowledge boundaries and focus, retaining only the information essential for core tasks. This substantially reduces context redundancy.

- In multi-agent collaboration workflows, continuous optimization of context summarization and vectorization (embedding) mechanisms is crucial for compressing and refining information transfer, thus lowering overall token consumption. Such strategies are vital for supporting the efficiency and scalability of large-scale multi-agent systems in real-world applications [39, 3].

4.3 Exploring Dynamic Knowledge Boundary Adjustment to Enhance System Reliability

Given diverse task scenarios and evolving knowledge requirements, it is essential to further explore and implement mechanisms for dynamically adjusting agent knowledge boundaries. Such mechanisms are key for mitigating hallucinations and logical contradictions. While expert-agent frameworks remain underdeveloped, promoting the co-evolution of toolchains and knowledge bases is critical—including advances in knowledge visualization, editability, and fine-grained management. These improvements enable more precise and controllable knowledge expression, reasoning, and task execution, ultimately enhancing overall system reliability and practical utility.

For example, although the human-in-the-loop paradigm is already well supported in AutoGen and has shown considerable potential, practical deployment still faces challenges such as reducing the cost of human intervention. In the future, as knowledge boundaries become clearer, it is expected that better delineation between automated and human-in-the-loop processes will help address these issues more effectively.

5 Conclusion

This study envisions the development of the AutoGen platform at an operating-system (OS) level, and, based on its open-source and extensible technological path, systematically proposes a novel taxonomy for agent classification. Using this classification framework, we identify and clarify several critical issues currently present in the platform and suggest corresponding optimization pathways. Furthermore, this research offers a preliminary exploration into leveraging knowledge editing and dynamic adjustment mechanisms to extend the capabilities of expert agents, providing valuable theoretical references for future studies.

It is important to note that as a highly open and extensible engineering project, the potential of AutoGen goes far beyond the aspects discussed in this work. Due to constraints of space and personal research scope, only selected core mechanisms and representative scenarios are analyzed here, and it is not possible to cover all related technical details and application domains. The functional boundaries and ecosystem of the AutoGen platform are still evolving rapidly, and many topics worthy of further investigation remain insufficiently explored. Therefore, this work is intended to serve as a catalyst, providing theoretical guidance and practical insights for subsequent researchers in the field, and to foster the continued development and innovation of AutoGen and its multi-agent systems.

References

[1] Gagan Bansal, Jennifer Wortman Vaughan, Saleema Amershi, Eric Horvitz, Adam Fourney, Hussein Mozannar, Victor Dibia, and Daniel S. Weld. Challenges in human-agent communication. *arXiv preprint arXiv:2107.07646*, 2021.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. 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, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS 2020)*, pages 1877–1901, 2020.

[3] Jiangjie Chen, Xintao Wang, Rui Xu, Siyu Yuan, Yikai Zhang, Wei Shi, Jian Xie, Shuang Li, Ruihan Yang, Tinghui Zhu, Aili Chen, Nianqi Li, Lida Chen, Caiyu Hu, Siye Wu, Scott Ren, Ziquan Fu, and Yanghua Xiao. From persona to personalization: A survey on role-playing language agents. *arXiv preprint arXiv:2404.18231*, 2024.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, 1971.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[6] Victor Dibia, Jingya Chen, Gagan Bansal, Suff Syed, Adam Fourney, Erkang (Eric) Zhu, Chi Wang, and Saleema Amershi. Autogen studio: A no-code developer tool for building and debugging multi agent systems. *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI 2024)*, 2024.

[7] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. Longnet: Scaling transformers to 1,000,000,000 tokens. *arXiv preprint arXiv:2307.02486*, 2023.

[8] Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678*, 2025.

[9] Yingqiang Ge, Yujie Ren, Wenyue Hua, Shuyuan Xu, Juntao Tan, and Yongfeng Zhang. Llm as os, agents as apps: Envisioning aios, agents and the aios-agent ecosystem. *arXiv preprint arXiv:2312.03815*, 2023.

[10] Shawn Henry and Sophia Lagerkrans-Pandey. Semantic kernel roadmap h1 2025: Accelerating agents, processes, and integration. `https://devblogs.microsoft.com/semantic-kernel/semantic-kernel-roadmap-h1-2025-accelerating-agents-processes-and-integration/`, February 2025. Accessed: 2025-07-01.

[11] Zhenwei Ji, Nayeon Lee, Rita Frieske, Tengyu Yu, Dan Su, Yanliang Li, Xinyan Xiao, Yichong Xu, Zhengbao Jiang, Qipeng Guo, Linlin Liu, Zhisong Zhang, Zhijing Jin, Boya Xing, Wei Wu, Xiaoyan Zhu, Minlie Huang, Xiaodan Zhu, Eric Cambria, Ji Zhang, Pascale Fung, Helen Meng, Xipeng Qiu, Yang Liu, Xuanjing Huang, Lingpeng Kong, Yuxuan Lai, Yansong Feng, Chaojun Xiao, and Xiang Ren. Survey of hallucination in natural language generation. *arXiv preprint arXiv:2302.03629*, 2023.

[12] Weiqiang Jin, Hongyang Du, Biao Zhao, Xingwu Tian, Bohang Shi, and Guang Yang. A comprehensive survey on multi-agent cooperative decision-making: Scenarios, approaches, challenges and perspectives. *arXiv preprint arXiv:2503.13415*, 2025.

[13] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, Zhifeng Chen, Thang Luong, James Qin, Laizhong Zheng, Ailin Li, Hongkun Yu, Yuanyuan Zhou, Xiaobing Liu, Alexander Passos, Jean-Baptiste Cordonnier, David Mazières, and Yonghui Wu. Gshard: Scaling giant models with conditional computation and automatic sharding. *International Conference on Learning Representations (ICLR 2021)*, 2021.

[14] Haotian Li, Yizhou Wang, Qingxiu Dong, Jianhao Yan, Xiaoyu Wang, Weihao Cheng, Yingjun Du, Jiacheng Ye, Chong Zhang, Xiaoqing Jin, Weixin Cai, Binbin Hu, Guangyu Sun, Xianglong Liu, Hang Su, Xueyan Wang, Xinyang Zhang, Yu Qiao, Juncheng Wei, Yizhou Sun, and Xueqian Wang. Camel: Communicative agents for "mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*, 2023.

[15] Xinzhe Li. A review of prominent paradigms for llm-based agents: Tool use (including rag), planning, and feedback learning. *arXiv preprint arXiv:2406.05804*, 2024.

[16] Yuxuan Li, Ziqing Yang, Zeyang Lei, Zhifang Sui, and Yue Zhang. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[17] Huajun Liu, Fuqiang Liu, Xinyi Fan, and Dong Huang. Polarized self-attention: Towards high-quality pixel-wise regression. *arXiv preprint arXiv:2107.00782*, 2021.

[18] Ben Lorica. Building the operating system for ai agents. The Data Exchange Podcast (Mar. 2025), 2025. Interview with Chi Wang on AG2 (Agent OS) by Google DeepMind.

[19] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint arXiv:2404.11584*, 2024.

[20] Microsoft AutoGen Team. Autogen agentchat python reference. https://microsoft.github.io/autogen/stable//reference/python/autogen_agentchat.agents.html, 2024. Accessed: 2025-07-01.

[21] Microsoft AutoGen Team. Autogen v0.4 release introduction. https://devblogs.microsoft.com/autogen/autogen-reimagined-launchingautogen-0-4/, 2024. Accessed: 2025-07-01.

[22] Microsoft AutoGen Team. Integrating autogen with semantic kernel. https://devblogs.microsoft.com/autogen/microsofts-agentic-frameworks-autogenand-semantic-kernel/, 2024. Accessed: 2025-07-01.

[23] Microsoft AutoGen Team. Autogen architecture: Core design concepts. https://microsoft.github.io/autogen/stable/user-guide/core-user-guide/coreconcepts/architecture.html, 2025. Accessed: 2025-07-01.

[24] Microsoft AutoGen Team. Autogen official documentation. https://microsoft.github.io/autogen/stable/index.html, 2025. Accessed: 2025-07-01.

[25] Microsoft AutoGen Team. Autogen v0.4 migration guide. https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/migration-guide.html, 2025. Accessed: 2025-07-01.

[26] Microsoft AutoGen Team. Github issue #4886: Can not get consistent hand off to user and have agent respond with message prior to tool call. https://github.com/microsoft/autogen/issues/4886, 2025. Accessed: 2025-07-01.

[27] Microsoft AutoGen Team. Github issue #4895: Assistantagent first "think" before calling a tool. https://github.com/microsoft/autogen/issues/4895, 2025. Accessed: 2025-07-01.

[28] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *ACM Symposium on User Interface Software and Technology (UIST) 2023*, pages 1–20, 2023. Also as arXiv:2304.03442.

[29] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*, 2021.

[30] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. *arXiv preprint arXiv:1901.03429*, 2019.

[31] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI preprint*, 2018.

[32] Reddit user: Various. Local model gets much slower after multiple turns? `https://www.reddit.com/r/AutoGenAI/comments/1gbnt36/local_model_gets_much_slower_after_multiple_turns/`, 2024. Accessed: 2025-07-01.

[33] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems (NeurIPS 2015)*, pages 2503–2511, 2015.

[34] Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*, 2025.

[35] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS 2017)*, pages 5998–6008, 2017.

[37] Jiří Wiedermann and Jan van Leeuwen. Large language models and the extended church-turing thesis. *arXiv preprint arXiv:2409.06978*, 2024.

[38] Michael Wooldridge. *An Introduction to Multi-Agent Systems.* John Wiley & Sons, 2nd edition, 2009.

[39] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[40] Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. *arXiv preprint arXiv:2203.08913*, 2022.

[41] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao

Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[42] Zhuo Zhao, Rongzhen Li, Kai Liu, Huhai Zou, KaiMao Li, Jie Yu, Tianhao Sun, and Qingbo Wu. Kaos: Large model multi-agent operating system. *arXiv preprint arXiv:2406.11342*, 2024.

Experience Report A: Analysis of Existing Issues in AutoGen

1. The Pseudo One-shot Problem (See Appendix H for Detailed Analysis)

The current design of AutoGen does not allow a single Agent to simultaneously perform tool invocation and knowledge expression within a single turn, thus failing to achieve true "One-shot" (single-turn response) capability.

Even when `reflect_on_tool_use=True` helps to conceal the intermediate output process of tool results, from the system execution perspective, two model calls and one function execution are still required. This implies:

- **Unavoidable performance jitter:** The response latency t_F for tool calls can fluctuate significantly (e.g., in the case of network requests or database queries), resulting in brief periods where the user interface appears unresponsive and lacks progress feedback. This extends the perceived wait time and makes the experience feel sluggish.
- **Complex exception handling:** If a tool call fails or times out, the model may attempt re-reasoning, but the output may remain incomplete or not meet user expectations, as the framework does not automatically retry failed calls. This logic split can manifest at the UI layer as "the API seems stuck but the model is still thinking", weakening user trust.

Additionally, in complex tasks, the model may need to invoke multiple tools—for example, making API requests and database queries simultaneously. While AutoGen's default mechanism supports both parallel and sequential tool calls (`parallel_tool_calls`), it lacks explicit dependency and order management: it cannot guarantee execution order or require one tool's success before another is called.

2. Issues with Execution Efficiency and Parallel Optimization

Although AutoGen is conceptually designed as an "operating system" for multi-Agent parallel interactions, there remain significant practical limitations, particularly in parallel execution and context management:

- **Insufficient parallel execution efficiency:** While AutoGen claims support for "parallel" Agents via mechanisms like GraphFlow, empirical tests indicate that, regardless of whether multithreading or asynchronous coroutines are used, the underlying implementation often remains sequential. Genuine performance gains from parallel responses are limited. Thus, AutoGen currently relies more on context chaining between heterogeneous Agents than on decomposing tasks into parallel sub-tasks executed by homogeneous Agents, which is typical of modern OS designs.

- **Lack of effective historical context management:** Currently, AutoGen lacks built-in context compression or efficient history management mechanisms. As the number of dialogue turns increases, historical messages accumulate, causing locally deployed or small-scale models to experience significant inference slowdown in long conversations. According to feedback from developers on Reddit, inference speed drops noticeably after many turns, and users must manually trim conversation history, severely impacting practical usability and convenience.

Due to these two limitations, we believe that, although AutoGen has established the rudimentary architecture of an "Agent Operating System", it still falls short of matching the precise context separation and high-degree parallel control achieved by modern computer operating systems in terms of actual functionality and performance.

In summary, at this stage, the AutoGen framework is more akin to a feature preview or partial development version, suitable for basic function demonstrations and small-scale applications. For future development, if AutoGen is to become a true productivity tool, it must draw on the engineering practices of modern operating system design by further strengthening its parallel response mechanisms and context management. This will be essential to improving AutoGen's practical performance and user experience in complex task environments.

Experience Report B: Challenges in the Field

1. Resource and Cost Bottlenecks

Currently, multi-agent architectures represented by AutoGen face significant challenges in terms of resource consumption and cost control. Compared to approaches based on a single LLM output or traditional Chain-of-Thought (CoT) reasoning, multi-agent systems incur markedly higher token consumption and API call frequency. The primary reasons for this are as follows:

Repeated system prompts: Each Agent is required to carry its specific role-defining system message in every dialogue turn, which significantly increases the length of the prompt per turn and thus directly drives up overall token usage.

Redundant accumulation of conversational history: To ensure continuity and accuracy of dialogue, the system typically transmits the entire history as part of each interaction. As the number of turns grows, token consumption increases exponentially.

Multiple network API calls: In typical multi-agent collaboration workflows, two to three or more Agents are often involved, with each response necessitating a separate model inference and network call. In contrast, conventional CoT methods generally complete the entire task with a single end-to-end inference.

Context transfer overhead due to frequent role switching: Frequent switching of roles among Agents requires the system to repeatedly embed each Agent's latest reply and key information into the next input, further exacerbating the context inflation problem.

The above factors collectively lead to a substantial increase in overall reasoning costs, which severely restricts the scalability of multi-agent systems in resource-constrained or cost-sensitive application scenarios.

2. Challenges of Hallucination and Logical Consistency

Although multi-agent architectures, in theory, can leverage role-based mutual constraints to reduce the "hallucination" phenomenon of LLMs compared to single-chain CoT reasoning, this mechanism can only mitigate, rather than fundamentally eliminate, hallucinations at a probabilistic level. The specific challenges manifest as follows:

Echoing bias: Multiple Agents in the system often rely on identical or highly similar system prompts and shared information sources for reasoning and judgment. If the initial input contains errors, these Agents tend to reinforce each other's outputs, making error correction difficult and even amplifying the original mistake. Therefore, the design of multi-agent systems should emphasize the independence and heterogeneity of information sources to avoid the chain propagation and database contamination resulting from initial misinformation.

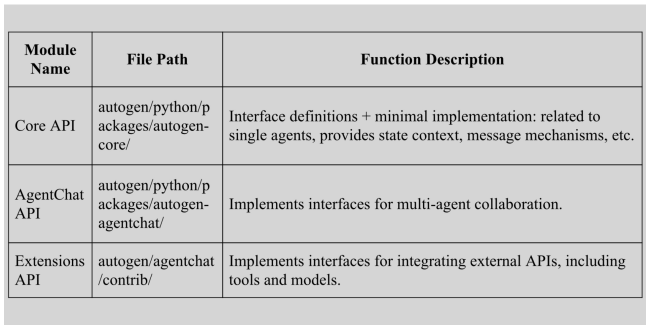
Lack of critical capacity due to role specialization: In current multi-agent collaboration paradigms, execution-oriented or tool-oriented Agents primarily focus on the concrete execution of low-level tasks and typically lack high-level semantic understanding or logical consistency checking of the overall task. As a result, potential logical errors or reasoning biases are difficult to detect and correct in a timely manner. To address this, it is advisable to introduce dedicated Critic or Validator Agents to independently assess the logical soundness and consistency of system outputs, thereby significantly enhancing the robustness and reliability of the overall system.

Appendix A: Detailed Architecture and Module Overview of the AutoGen Framework

The AutoGen framework consists of the core Python modules `autogen-core`, `autogen-agentchat`, and `autogen-ext`, along with auxiliary tool modules such as `autogen-studio` and `agbench`. Specifically:

- `autogen-core` provides fundamental communication interfaces and asynchronous scheduling logic;
- `autogen-agentchat` implements message exchange and task flow definitions between Agents;
- `autogen-ext` is used for extending external models and tools.

Among the auxiliary tools, `autogen-studio` offers a web-based visual interaction interface running locally, while `agbench` provides performance evaluation features, including average dialogue turns (*average_turns*), success rate (*success_rate*), average response time (*avg_response_time*), and accuracy (*accuracy*).



Module Name	File Path	Function Description
Core API	autogen/python/packages/autogen-core/	Interface definitions + minimal implementation: related to single agents, provides state context, message mechanisms, etc.
AgentChat API	autogen/python/packages/autogen-agentchat/	Implements interfaces for multi-agent collaboration.
Extensions API	autogen/agentchat/contrib/	Implements interfaces for integrating external APIs, including tools and models.

Figure 7: AutoGen module and component overview.

The platform also supports cross-language integration based on Protocol Buffers (`proto`), allowing deployment and integration with languages such as C#/.NET and enabling seamless multi-language system architectures.

Appendix B: Version Evolution from AutoGen v0.2 to v0.4

The main release dates for AutoGen versions are: v0.2 (November 2023) and v0.4 (January 2025). Compared to v0.2, v0.4 underwent major architectural refactoring, with a particular focus on asynchronous scheduling mechanisms and strict interface specifications. Through layered design and enhanced developer tool support, v0.4 enables deep customization, while the independent `pyautogen` package encapsulates v0.4 functionality with a simplified interface, providing a smooth transition for beginners and rapid prototyping with backward compatibility for v0.2.

Version v0.2 focused on achieving "conversational programming" via natural language prompt orchestration, whereas v0.4 emphasizes standardized interface definitions and code-level dynamic control. This shift marks a deep evolution from prompt-based to code-based control, greatly enhancing the framework's flexibility and scalability and making it more suitable for industrial-grade applications—albeit at the cost of a steeper learning curve for new users. To address this, the former Microsoft AutoGen team developed the standalone `pyautogen` package, which offers a classic interface to help beginners and prototype developers migrate smoothly to the new architecture.

Appendix C: Details of AutoGen's Compatibility with External Platform Ecosystems

AutoGen exhibits a high degree of ecosystem openness, enabling direct compatibility with multiple platforms such as AutoGPT, the LangChain toolchain, and LCEL expressions, while also supporting seamless integration with resources from CAMEL and Microsoft Semantic Kernel. The open interface design not only consolidates resources and avoids redundant feature stacking, but also greatly reduces the barrier for developers to integrate external tools, thereby significantly improving rapid deployment capabilities in complex business scenarios. This flexibility allows developers to implement highly customized application requirements without the need to deeply modify the core framework code. The main interface invocation modes are as follows:

Platform	Example Invocation Method
CAMEL	Role Assignment Agent based on role configuration (see Appendix G)
LangChain	Direct invocation of predefined toolchains (e.g., retrieval, translation tools)
LangChain LCEL	Defining toolchain processes with LCEL expressions for complex tool composition

Table 1: Examples of AutoGen compatibility and invocation methods with external platforms.

Appendix D: Built-in Practical Agent Types in AutoGen (with Comparison to CAMEL)

Table 2: AutoGen Built-in Agent Types and Application Scenarios

AutoGen Agent Class	**Functionality and Application Scenarios**
AssistantAgent	Basic LLM Agent for content generation and tool invocation.
UserProxyAgent	Simulates a human user, supporting manual input or automated process control.
CodeExecutorAgent	Multi-language code execution; supports automated code running and feedback.
MessageFilterAgent	Message filtering to block inappropriate content, ensuring safety and efficiency in multi-agent collaboration.
SocietyOfMindAgent	Multi-sub-agent collaboration; simulates a "Society of Mind" structure for complex decision making.

Table 3: CAMEL Built-in Agent Types and Application Scenarios

CAMEL Agent Class	**Functionality and Application Scenarios**
ChatAgent	General-purpose conversational agent; supports role-playing, memory, multi-model backends, and asynchronous operation.
RolePlayingAgent	Multi-role playing agent, commonly used for collaboration or game-theoretic simulation.
CriticAgent	Evaluates outputs of other agents, providing feedback and quality control.
TaskAgent	Task decomposition and subtask scheduling, supports automatic breakdown of complex tasks.
KnowledgeGraphAgent	Structured knowledge reasoning and retrieval, suitable for tasks requiring knowledge graph reasoning.
DeductiveReasonerAgent	Complex logical reasoning and stepwise problem-solving.
MultiHopGeneratorAgent	Multi-hop reasoning, generating complex conclusions and intermediate reasoning steps.
SearchAgent	Information retrieval across multiple data sources.
EmbodiedAgent	Designed for tasks in the physical/embodied world, suitable for simulation and context-aware applications.

Appendix E: Officially Recommended Multi-Agent Configuration Strategies in AutoGen

Table 4: Typical Multi-Agent Configuration Strategies in AutoGen

Application Scenario	**Typical Agent Configuration**	**Description**
Mathematical reasoning tasks	AssistantAgent (reasoning) + UserProxyAgent (human-in-the-loop)	Humans can intervene at any time; the LLM handles reasoning, improving accuracy.
Long-text retrieval and QA	UserProxyAgent (initiator) + AssistantAgent (search/-tool invocation)	The user asks questions; the agent automatically searches resources and generates answers.
Action decision tasks	Planner + Controller	Planner is responsible for goal decomposition and planning; Controller executes actions.
Code generation and repair	AssistantAgent (code generation) + UserProxyAgent (execution/feedback)	The agent generates code; the UserProxyAgent executes the code and provides feedback.
Free-form dialogue simulation	RoundRobinGroupChat (multi-role dynamic discussion)	Multiple roles take turns speaking, simulating free dialogue and multi-perspective debates.
Decision execution tasks	Decision Agent + Execution Agent	One agent is responsible for decision-making, another for concrete execution.

Appendix F: Macro Scheduling and Micro Execution Mechanisms in AutoGen

AutoGen provides a decoupled mechanism for macro-level scheduling and micro-level execution. At the macro level, the framework uses the `GroupChat` container to centrally manage the speaking order and termination conditions for multiple agents. `GroupChat` supports common scheduling modes such as round robin, where each agent takes turns to speak in a fixed sequence until a termination condition is met. You can also customize the speaking order or termination logic to accommodate more complex business needs. Below is a simple example of round robin scheduling:

Listing 1: Key steps for round robin scheduling in AutoGen

```
from autogen.agentchat import GroupChat,
    AssistantAgent, UserProxyAgent, GroupChatManager

assistant = AssistantAgent(name="assistant",
    llm_config=False)
user = UserProxyAgent(name="user")

groupchat = GroupChat(
    agents=[user, assistant],
    max_round=3
)
manager = GroupChatManager(groupchat=groupchat)
reply = manager.run(message="...", sender=user)
print(reply)
```

In this example, `GroupChatManager` automatically uses the round robin strategy to let the user and assistant take turns speaking, so the developer does not need to manually manage the speaking order.

At the micro level, each agent can flexibly process message content and invoke tools by registering callback functions (e.g., `register_for_llm_message`). For example:

Listing 2: Custom callback registration (simplified)

```
@assistant.register_for_llm_message
def custom_on_message(message, sender, config):
    if "Hello" in message["content"]:
        return "Hello!~..."
```

This design allows novice users to quickly start multi-agent collaboration processes with simple configuration, while advanced users can implement deeper customization through custom callbacks and scheduling strategies, balancing ease of use and flexibility.

Appendix G: Implementation and Application of Different Agent Types in AutoGen

Both AutoGen and CAMEL support one-shot definitions (see the later One-shot section), but there are significant differences in platform design focus. CAMEL places greater emphasis on knowledge and context setup, making tool integration relatively complex; in contrast, AutoGen features native API tool integration, greatly lowering the barrier for extension. AutoGen's highlight is its "conversational programming" paradigm, enabling efficient control of complex tasks through dialogue flows.

This appendix provides a practical overview of the three core types of Agents supported by AutoGen, including their configuration and mechanisms.

—

1. Tool-driven Agent

AutoGen provides very convenient support for tool-driven agents, especially for scenarios involving third-party APIs or custom functions.

Implementation process:

1. Encapsulate the tool interface as a Python function;
2. Wrap it with `FunctionTool` and add a description;
3. Instantiate an `AssistantAgent`, passing the tool list via the `tools` parameter.

Example: Integrating the quotable.io API to fetch quotes

```
def random_quote(tag: str = "technology") -> str:
    ...
quote_tool = FunctionTool(
    random_quote,
    description="Fetch an English quote for a specified topic"
)
assistant = AssistantAgent(
    name="Quoter",
    tools=[quote_tool]
)
```

In the current version, `AssistantAgent` *is the only built-in agent type with native tool-calling support.*

Advantages of tool registration:

- Implementation is straightforward and highly extensible;

—

2. Role-driven Agent

In AutoGen, the core idea behind role-driven agents is to dynamically constrain the agent's behavioral scope and knowledge performance—without directly modifying the model's internal knowledge—through prompt engineering and conversational programming. AutoGen primarily employs the following mechanisms:

System message (prompt): By crafting a detailed system message (agent prompt), the agent's knowledge domain and behavioral patterns are constrained, and processes or rules can be embedded within the prompt itself.

Conversation flow and control logic (conversational programming): Using the `on_messages()` [1]callback interface, developers can flexibly control message handling, event triggering, tool invocation, and multi-turn dialogue logic, thereby dynamically adjusting the agent's operational boundaries.

Core advantages:

- Clear separation of concerns: "what to do when" is defined by the dialogue flow, "how to do it" is handled by explicit code or external tools;
- More intelligent than traditional regex-based triggers, but still supports regex/keyword triggers (e.g., `TERMINATE`);
- Agent autonomy and context passing improves the engineering and reusability of reasoning flows;
- Supports dynamic next-speaker selection, context trimming and passing, and precise domain definition (e.g., legal, medical), approaching expert-level agent performance.

  Comparison with traditional approaches:
  Traditional frameworks such as LangChain can only specify "who does what and when," whereas AutoGen supports models that can autonomously pose questions and dynamically pass topics, making it more general, robust, and engineer-friendly.

[1]**Version Note (0.2 $\rightarrow$ 0.4):** In version 0.2, agent message responses mainly relied on the `generate_reply()` method, with each call triggering a model inference or function execution. Developers would choose between auto-reply (auto-generating a reply for each message) and manual process control via the **`human_input_mode`** parameter.

Starting from version 0.4, this mechanism has been replaced by dynamic message processing via **`on_messages()`**, where developers can flexibly set conditions to skip, invoke, or perform multiple rounds of model processing within the same method. Meanwhile, the role of the system message has reverted to traditional prompt engineering, serving mainly to specify the model's identity, style, and knowledge domain as static context. The main carrier of conversational programming has shifted from prompt-based flows (as stressed in early papers) to the programmable **`on_messages()`** interface, enabling more sophisticated process customization and interaction choreography.

While simple dialog control logic can still be guided via system message/prompt in AutoGen, since version 0.4, the officially recommended and widely supported method is code-based control flow via **`on_messages()`** callbacks. This approach—using programming techniques to manage conditions, branches, loops, etc.—greatly improves the scalability and reliability of multi-agent systems. As a result, "controlling via natural language dialogue" is no longer the mainstream paradigm, but rather an initial constraint or supplementary guideline.

Limitations:

- Still limited by the model's own knowledge blind spots and hallucinations;
- Difficult to achieve truly fine-grained and precise knowledge control; currently remains a "soft constraint."

—

3. Role-playing Agent

If high precision on specific knowledge is not required and only CAMEL-style role-play and interaction are needed, you can directly use CAMEL's `RolePlaying` API to generate role definitions, and use the generated role names and system messages for AutoGen instantiation:

```
from camel.societies import RolePlaying

task_prompt = "Design an engaging three-day itinerary for first-time
    visitors to Berlin."
role_play = RolePlaying(
    assistant_role_name="Travel Planner",
    user_role_name="Traveler",
    task_prompt=task_prompt,
    with_task_specify=False,
)

assistant = AssistantAgent(
    name="travel_planner",
    system_message=role_play.assistant_sys_msg,
)

traveler = AssistantAgent(
    name="traveler",
    system_message=role_play.user_sys_msg,
)
```

Appendix H: The "Pseudo One-shot" Mode of AutoGen

In practical applications, it has been observed that even with guidance via system message, tool-driven agents generally require at least two rounds of dialogue to link tool invocation and natural language generation:

1. **First round:** The agent plans (Planning) based on the system message and issues a `FunctionCall` (e.g., calls `random_quote`).
2. **Tool execution:** The framework executes the corresponding function, returning a `FunctionExecutionResult` (Observation).
3. **Second round:** The agent reasons over the tool result and generates a natural language response (Responding).

Official documentation states:

- The `AssistantAgent` by default performs at most one tool-call iteration; if `reflect_on_tool_use=False`, it directly returns the tool result (`ToolCallSummaryMessage`), i.e., the tool result is not further processed by the model for reflection or NLG.
- If `reflect_on_tool_use=True`, the framework automatically triggers a second inference, allowing the model to generate a natural language output based on the tool result, returning the final answer.

Overall, AutoGen's tool-calling tasks strictly follow a "Planning $\rightarrow$ Action $\rightarrow$ Observation $\rightarrow$ Response" multi-step process and require at least two rounds of dialogue, not supporting true single-round completion. Even with `reflect_on_tool_use=True`, the second inference is only encapsulated within the system, resulting in a "pseudo one-shot" experience for the user—in reality, two model inferences are still performed.