A Survey on How Code Empowers LLM Agents

Anonymous authors
Paper under double-blind review

Abstract

Large Language Models (LLMs) have rapidly evolved into general-purpose agents capable of reasoning, planning, and acting across diverse tasks. While much progress has focused on scaling model size and aligning behavior through natural language, a growing body of research reveals that code—with its structured, executable, and compositional nature—plays a uniquely powerful role in shaping and augmenting LLM capabilities. The emerging synergy between code and LLMs is transforming how models reason, act, and collaborate—both individually and as agents. This survey systematically examines how code acts as both a medium and a mechanism to empower LLM agents. We synthesize a growing body of work where code is not only the output but also the internal mechanism that improves an agent's ability to decompose tasks, form plans, use tools, coordinate with others, and ground actions in real or digital environments.

1 Introduction

Large Language Models (LLMs) have emerged as versatile agents, capable of performing a wide range of cognitive and interactive tasks—reasoning through problems, planning sequences of actions, and engaging with dynamic environments. While advances in scale, instruction tuning, and alignment have significantly improved their linguistic competence, these models still face limitations when it comes to structured reasoning, long-term consistency, verifiability, and interaction with the external world (Liang et al., 2023a).

In parallel, a striking trend has gained momentum: LLMs that can generate, interpret, and use code demonstrate markedly stronger capabilities across tasks that require precision, abstraction, modularity, and interaction (Zhao et al., 2023). Rather than being treated solely as a modality for generation (e.g., in code completion or synthesis), code is increasingly central to how LLMs reason, act, and collaborate Wang et al. (2024). Code provides a structured and executable interface that allows language models to offload complex reasoning, formalize plans, manipulate memory, orchestrate tools, and coordinate with other agents.

This survey aims to explain the widespread adoption of code-specific training in the general LLM training paradigm and how code enhances LLMs to act as Agents, based on the taxonomy of relevant papers (see Figure 1).

Organization of this Survey We define code as formal language that is both machine-executable and human-interpretable (see our detailed definition of code and typical methods for LLM code training in Appendix 2). With insights from characteristics of code (see our case studies in Appendix A.1), our literature review reveals that integrating code into LLM training *i)* enhances their programming and reasoning capabilities (§3.1); *ii)* enables the models to directly generate executable, fine-grained steps during decision-making(§3.2); *iii)* boosts agents' capabilities tointeract with real and digital environments through structured actions, tools, and cross-agent coordination(§3.3); *iv)* empowers memory management strategies by code-based solutions, improving long-term memory retention, self-awareness, and adaptive learning(§3.4); and *iv)* situates the LLMs within a code execution environment, allowing them to receive automated feedback from integrated evaluation modules and self-improve (§3.5).

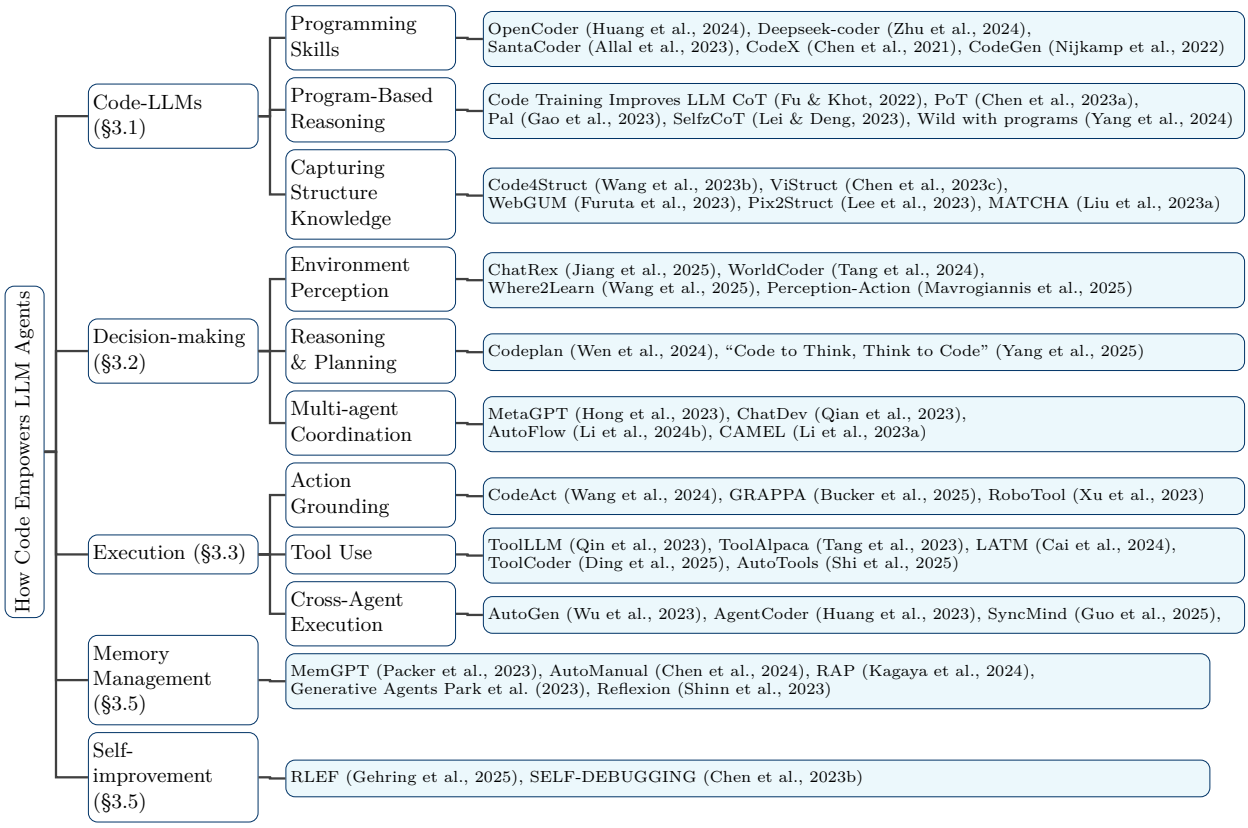

Figure 1: The organization of our paper, with a curated list of the most representative works.

2 Preliminaries

2.1 Our Definition of Code

Code represents a significant abstraction layer beyond natural language, characterized by structured formal language systems that are simultaneously machine-executable and human-interpretable. While natural language serves as an abstraction of human thought, formal languages (our definition of code) constitute abstractions that can be systematically parsed and executed according to well-defined rules. These formal languages exhibit distinctive attributes including structural rigidity, symbolic representation, modularity, and non-ambiguity—features largely absent in natural language communication.

The inherent organization, deterministic interpretation, and compositional nature of code enable precise execution paths that natural language cannot guarantee. This fundamental difference in symbolic organization explains why, even when trained using identical paradigms, code-trained language models demonstrate capabilities distinct from those trained exclusively on natural language corpora.

For our purposes, human-readable programming languages (Python, JavaScript, etc.) represent archetypal examples of code. Conversely, we exclude low-level machine languages composed of binary instructions due to their limited human comprehensibility. Our definition also encompasses pre-constructed formal languages such as function sets implemented in systems like WebGPT (Nakano et al., 2021), as these can be deterministically parsed and executed through rule-based mechanisms.

Additionally, we extend our definition to include other formal expression systems—predefined function libraries, mathematical deduction formalisms, and similar rule-based notational systems—recognizing that language models trained on these structured symbolic systems exhibit advantages comparable to those trained on conventional programming languages. This expanded definition enhances this survey's comprehensiveness in addressing multiple contemporary research directions.

2.2 LLM Code Training Methods

Language models acquire code capabilities through training with language modeling objectives applied to structured code repositories. The sequential readability characteristic of code makes this approach conceptually similar to training models for natural language generation and comprehension.

Formally, for a language model architecture M_Θ with parameters Θ operating on a code corpus $T = t_1, ..., t_n$, optimization typically proceeds by minimizing the language modeling loss:

$$L(T) = \sum_i -\log P(t_i | t_{i-k}, ..., t_{i-1}; \Theta)$$

When utilizing conventional programming languages as training material (Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022), researchers typically extract datasets from open-source repositories like GitHub. This approach yields training corpora with scale comparable to natural language datasets, constituting what we term *code pre-training*. Implementation strategies include either continuing training of pre-trained language models using code repositories (as with Codex (Chen et al., 2021)), or developing models from initialization using combined natural language and code datasets (illustrated by CodeLLM (Ma et al., 2023)).

For specialized formal languages with more constrained expression spaces, such as domain-specific function libraries (Schick et al., 2023), mathematical notation systems (Wu et al., 2022), or database query languages (Sun et al., 2023), available training material is typically more limited. We characterize training on these specialized languages as *code fine-tuning*.

Beyond supervised learning approaches, reinforcement learning—particularly Proximal Policy Optimization (PPO)—has emerged as a critical methodology for training code-generating LLM-based agents. The PPO algorithm optimizes model parameters through iterative policy improvements while restricting update magnitudes to prevent performance deterioration.

In the context of LLM code generation, PPO training typically follows a structured workflow: First, the model generates candidate code sequences based on input prompts. These outputs are then evaluated through reward functions that assess code correctness, efficiency, or other desired attributes. The computed rewards are converted to advantages, which calibrate how much better or worse actions were relative to the model's current policy.

The heart of PPO is its objective function that incorporates clipped probability ratios:

$$L(\theta) = \mathbb{E}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A(s,a),; \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\epsilon, 1+\epsilon\right)A(s,a)\right)\right]$$

Where π_θ represents the current policy, $\pi_{\theta_{old}}$ the previous policy, $A(s,a)$ the advantage function, and ϵ a hyperparameter (typically 0.1-0.2) controlling clip range. This formulation prevents excessively large policy changes by clipping the probability ratio.

For LLM-based code agents, the PPO training loop integrates several crucial components: *i)* sequence generation using the actor model, *ii)* reward computation combining model-based and rule-based evaluations, *iii)* advantage estimation incorporating temporal relationships between code tokens, *iv)* critic model updates to improve value predictions, and *v)* actor model updates that optimize the policy while respecting KL-divergence constraints between consecutive policies. This approach enables models to learn from execution feedback and environmental interaction rather than solely from static examples—a critical advantage for developing robust code generation capabilities.

3 Code-empowered LLMs Agents

In the preceding sections, our discussion highlighted the various ways in which code integration enhances LLM Agents. Beyond these general improvements, we recognize that code-empowered LLMs play an especially

critical role when these foundation models are used to instantiate agents. Therefore, in the following sections, we will explore how code training provides capability edges that benefit code-LLM agents.

We will examine embodied agents in a broad sense, which require processing and generating multimodal inputs. Due to language's abstraction properties, many existing approaches use language for supervision, processing, and integration of multimodal signals. We observe that the enhancements provided by code training in LLMs typically manifest at each stage of the IA application. For instance: *i)* improving the agent's decision-making capabilities regarding environmental perception, reasoning, and planning (§3.2), *ii)* enabling agents to more flexibly ground natural language actions to machine-interpretable formal languages, thereby facilitating more generalized tool use practices (§3.3), and *iii)* optimizing performance through automatic feedback derived from code execution environments (§3.5). The following sections will provide detailed explanations of each of these aspects.

3.1 Code-LLMs

Large Language Models pre-trained or fine-tuned on code—commonly referred to as code-LLMs—exhibit enhanced capabilities that go beyond natural language understanding. Exemplified by models like OpenAI's Codex (Chen et al., 2021), this class of LLMs leverages the structural, formal, and executable nature of code to develop more robust reasoning and compositional generalization. Unlike natural language, code imposes strict syntactic and semantic constraints: it requires producing logically coherent, sequentially valid instructions that can be executed and verified at each step. By learning from these properties, code-augmented LLMs demonstrate stronger capabilities in symbolic manipulation, mathematical reasoning (Wu et al., 2022), general-purpose programming (Chen et al., 2021), and structured data querying (Sun et al., 2023; Cheng et al., 2023).

Crucially, these models also show marked improvements on non-code tasks that benefit from structured reasoning. The discipline of code—its demand for precision, modularity, and step-wise logic—enhances chain-of-thought (CoT) prompting across a variety of domains (Lyu et al., 2023; Zhou et al., 2023; Fu & Khot, 2022). This influence extends to multimodal and semi-structured inputs such as charts, markup, and HTML documents, where code-augmented models outperform their natural language counterparts (Furuta et al., 2023; Liu et al., 2023a).

In the remainder of this section, we organize the capabilities of code-LLMs into three subcategories: *(i) Programming Skills*, which assess the core coding competencies and generalization of these models; *(ii) Program-Based Reasoning*, which explores how code can scaffold and improve complex reasoning; and *(iii) Capturing Structured Knowledge*, which highlights how code enhances model understanding of structured and multimodal information.

Programming Skills A foundational benefit of code-LLMs lies in their ability to learn and perform core programming tasks with high competence. These models are not only trained to understand and generate code but also to generalize across languages, frameworks, and problem domains—enabling them to serve as reliable coding assistants and integral components of intelligent agents.. Recent models such as OpenCoder (Huang et al., 2024), Deepseek-Coder-V2 (Zhu et al., 2024), and SantaCoder (Allal et al., 2023) are trained specifically to improve multilingual code generation and comprehension across languages and frameworks. DeepSeek-Coder-V2 Zhu et al. (2024) is an open-source Mixture-of-Experts (MoE) code language model that achieves performance comparable to GPT-4 Turbo in code-specific tasks. It supports an extensive range of 338 programming languages and extends the context length from 16K to 128K, enhancing its capability to handle complex coding tasks. CodeGen (Nijkamp et al., 2022) and Codex (Chen et al., 2021) demonstrate that models trained on diverse programming corpora can generalize to unseen coding problems and languages. These models often excel in standard programming benchmarks, including HumanEval and MBPP, showcasing their capacity for code synthesis, completion, and repair tasks. They form the technical backbone of many code-empowered agents.

Program-Based Reasoning Beyond raw coding ability, code pretraining significantly enhances LLMs' reasoning capabilities, especially for tasks requiring structured, step-wise problem decomposition. Fu et al. (2022) and Ma et al. (2023a) empirically demonstrate that code training improves chain-of-thought (CoT)

reasoning in math and logic tasks. Building upon this, PAL (Program-Aided Language Models) (Gao et al., 2023) introduces a method where LLMs generate Python programs as intermediate reasoning steps, offloading execution to a Python interpreter. Similarly, Program of Thoughts (PoT) prompting (Chen et al., 2023a) disentangles reasoning from computation by having LLMs generate programs that are executed externally. SelfzCoT (Lei & Deng, 2023) further refines this idea by using program-based rationales for self-consistency. (Yang et al., 2024) introduces the task of reasoning in the wild, where LLMs must solve problems of unknown types by identifying subproblems and generating appropriate programs These approaches treat code not just as output but as a reasoning substrate, enabling verifiability and modularity in inference.

Capturing Structured Knowledge Code's strong bias for structure also helps LLMs acquire and generalize over structured data modalities. Code4Struct (Wang et al., 2023b) shows that code-formatted supervision improves parsing and semantic extraction from semi-structured text. ViStruct (Chen et al., 2023c) extends this to visual documents by encoding layout and content into structured prompts. MATCHA (Liu et al., 2023a) and WebGUM (Furuta et al., 2023) leverage code-like representations to handle HTML and autonomous web navigation. Pix2Struct (Lee et al., 2023) is pre-trained as an image-to- text model on masked website screenshots, and further training with OCR, language modeling, and image captioning objectives. These models demonstrate that code inductive biases can bridge language with tabular, visual, and markup representations, enhancing LLMs' comprehension in semi-structured environments.

3.2 Decision Making

Environment Perception It's important to emphasize that perception in intelligent agents occurs continuously and should be organized in a structured manner where different modalities are processed simultaneously without temporal disparities, with important elements receiving heightened attention. This principle of "structured perception" often contradicts the training paradigm of LLMs that lack code training, as unprocessed language datasets are unstructured, and the temporal sequentiality of language is emphasized by the need to interpret text, making perception time-dependent. Conversely, for formally structured languages, when LLMs are trained on such texts, they partially shed the temporal dependency assumptions imposed by natural language text datasets. The most intuitive example is class definitions in code, where attributes are established, stored, and accessed in a tree structure without a concept of "sequence." Additional examples include webpage layouts.

In agent development, researchers have explicitly or implicitly utilized the capacity of code-LLMs to perceive the world in a structured manner, enhancing the precision and effectiveness of input processing. ChatRex (Jiang et al., 2025) segments visual inputs and transcribes them into text corresponding to layout-content pairs, strengthening the agent's structured information perception and making its reasoning more grounded. WorldCoder (Tang et al., 2024) stores world models in code format, enabling flexible addition, deletion, and querying of environmental perceptions. Where2Learn (Wang et al., 2025) abstracts embedded environmental perceptual knowledge into steps (code) for environment exploration, creating learning datasets guided by structured knowledge, helping perception models learn more effectively. Perception-Action (Mavrogiannis et al., 2025) encapsulates perceptions that cannot be described linguistically into APIs, using API return values to inform agents about more specialized environmental perception.

Reasoning & Planning Code-LLM based agents demonstrate significant advantages in planning and reasoning capabilities due to the inherent characteristics of code itself. The symbolic, modular nature of programming languages facilitates well-decoupled, logically structured reasoning steps between different functionalities. These advantages manifest in several ways: *i)* When code serves as the reasoning medium, thought chains become more concise and precise with minimal redundancy; *ii)* The syntax allows for functional composition and nesting, making it easier to express complex logic through established grammatical structures; *iii)* Execution results from code steps can be derived from external modules such as calculators or deterministically verified by compilers, making intermediate steps more accurate and reliable.

For example, Codeplan (Wen et al., 2024) utilizes code as the planning format, overcoming the limitations of natural language planning such as lack of analysis, verification, and clear logical structure, thereby enhancing overall performance on reasoning tasks. The survey "Code to Think, Think to Code" (Yang et al., 2025)

provides detailed descriptions of how code properties enhance reasoning capabilities in LLMs. Chain of Code (Li et al., 2024a) replaces natural language thoughts with code-formatted reasoning, reducing hallucination in intermediate steps and improving performance on tasks requiring sophisticated reasoning. "Steering Text and Code" (Chen et al., 2025) addresses some negative examples of converting text to code for reasoning and proposes solutions to these challenges. "World Models for Model-based Task Planning" (Guan et al., 2023) represents the planning components of all covered tasks using a formal task language, leveraging human-LLM collaboration to continuously refine code-form plans.

Multi-Agent Coordination Code plays a pivotal role in orchestrating multi-agent collaboration by serving as a shared interface for communication, task planning, and synchronized execution. Systems like MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2023) introduce code-centric workflows where agents specialize in roles (e.g., architect, coder, tester) and collaborate through structured code generation pipelines, mimicking real-world software engineering teams. These frameworks use code not just as output but as a medium of coordination, enabling modular task decomposition and collective progress tracking. On the other hand, frameworks such as CAMEL (Li et al., 2023a) and AutoFlow (Li et al., 2024b) focus on autonomous collaboration protocols, where agents negotiate and align through natural language and code-grounded dialogue to co-design task plans and workflows. CAMEL (Li et al., 2023a) in particular emphasizes role-play and dialogue-based alignment to simulate social dynamics among agents, while AutoFlow (Li et al., 2024b) automates the creation of executable workflows by converting high-level objectives into interdependent code-based subtasks. Across these systems, code serves both as a shared language and a structuring mechanism, enabling agents to coordinate not through fixed rules, but via adaptable, executable artifacts.

3.3 Execution

Action Grounding Action grounding in agents requires translating planning or steps from unstructured, machine-unreadable natural language into a formalized language format that machines can interpret and execute. Code-LLM agents hold a distinct advantage in this domain as their action space already exists within well-defined formal language structures, such as Python code or API calls.

Several works demonstrate this advantage effectively. CodeAct (Wang et al., 2024) enables LLM agents to utilize executable Python code for performing actions, revising decisions, and combining tools flexibly through multi-turn interactions. This approach grounds abstract reasoning in concrete, executable operations. GRAPPA (Bucker et al., 2025) introduces a team of specialized conversational agents, prominently featuring code-based instruction writing that enhances model performance. These agents assist robots in grounding their actions to the environmental context and adjusting behavior based on visual and motor feedback, eliminating the need for additional human demonstrations or custom configurations. RoboTool (Xu et al., 2023) employs LLMs to transform natural language instructions into executable standardized code-form robot actions. This system enables creative tool utilization by integrating task understanding, planning, parameter computation, and code generation into a cohesive framework that bridges the gap between human intent and machine execution.

Tool Use In the context of code-LLM agents, tool utilization refers to the ability of LLMs to appropriately invoke external tools or APIs through specifically formatted commands. This capability involves recognizing when tool assistance is beneficial, selecting the most suitable tool for a given scenario, and constructing syntactically valid invocation statements that adhere to the tool's interface requirements. From this perspective, many research efforts in tool learning essentially focus on enabling agents to communicate with specialized execution modules through formal, machine-interpretable languages. These approaches allow the language model to delegate specific tasks to dedicated components, creating a more effective division of labor where the LLM handles high-level reasoning while specialized tools manage domain-specific operations. This tool integration framework represents a fundamental capability for code-generating agents that need to interact with external systems and resources to accomplish complex tasks.

For example, ToolLLM (Qin et al., 2023) addresses the tool-use challenge for open-source LLMs by creating a diverse instruction-tuning dataset (ToolBench) and fine-tuning models like LLaMA, allowing them to effectively interact with external APIs and generalize to new tools, achieving performance on par with advanced

models like ChatGPT. ToolAlpaca (Tang et al., 2023) empowers compact LLMs to acquire generalized tool-use skills by automatically generating a diverse tool-use corpus through multi-agent simulations, enabling these models to apply previously unseen tools without specific training. GPT4Tools (Yang et al., 2023) enhances the tool-use capabilities of open-source LLMs by training them on self-instructed data generated from advanced models, which structures tool interactions using code to solve visual tasks efficiently through fine-tuning and zero-shot generalization. LATM (Cai et al., 2024) equips LLMs with the ability to generate reusable tools as executable code for solving tasks, splitting the tool creation and usage process between models to enable cost-effective and efficient problem-solving through modular tool reuse and functional caching. Gorilla (Patil et al., 2023) refines LLaMA-based models for more accurate API call generation by mitigating hallucinations and adapting to evolving documentation, providing a robust and flexible tool-use system when integrated with a document retriever. ToolCoder (Ding et al., 2025) allows LLMs to approach tool usage as a code generation task by converting natural language queries into Python function scaffolds, offering precise, modular, and reusable tool interactions grounded in executable code. AutoTools (Shi et al., 2025) enables LLMs to autonomously convert tool documentation into executable code, validate its correctness, and integrate tools into scalable, flexible problem-solving workflows, enhancing the efficiency of tool use.

Cross-Agent Execution While multi-agent coordination often emphasizes planning and role assignment, cross-agent execution focuses on how agents collaboratively generate, test, and refine executable code within shared environments. This paradigm treats code not merely as an output but as a dynamic medium for interaction and synchronization among agents. Frameworks like AgentCoder (Huang et al., 2023) exemplify this approach by assigning specialized roles to agents—such as programmer, test designer, and test executor—that iteratively collaborate to produce and validate code. Each agent contributes its expertise, and through continuous feedback loops, they enhance code quality and robustness. AutoGen (Wu et al., 2023) extends this concept by enabling agents to engage in multi-turn dialogues, combining natural language and code to accomplish complex tasks. This conversational framework allows for flexible agent behaviors and interaction patterns, facilitating the development of diverse applications ranging from mathematics to software engineering. Addressing the challenges of maintaining synchronization in collaborative environments, SyncMind (Guo et al., 2025) introduces a framework to measure and recover from out-of-sync scenarios among agents. By analyzing real-world software engineering tasks, it highlights the importance of shared understanding and adaptability in multi-agent systems. Collectively, these systems illustrate the evolution of multi-agent frameworks where shared code spaces serve as the foundation for collaborative execution. By leveraging code as both a communication medium and an execution platform, agents can achieve higher levels of coordination, adaptability, and efficiency in complex tasks.

3.4 Memory Management

Effective memory management is crucial for large language model (LLM) agents to handle extended contexts, adapt to dynamic environments, and perform complex tasks. Recent advancements have introduced code-centric frameworks that structure and evolve memory systems to enhance agent capabilities. For LLM agents, learning from past experiences can also be viewed as managing the episodic memory (Shinn et al., 2023). Retrieval-Augmented Planning (RAP) Kagaya et al. (2024) retrieves past experiences corresponding to the current situation. MemGPT Packer et al. (2023) draws inspiration from traditional operating systems to address the limitations of LLMs' context windows. By implementing a virtual context management system, it simulates hierarchical memory structures, allowing agents to manage different memory tiers. It allows LLM to select content to retain in working memory and to search for information in long-term memory. Generative Agents Park et al. (2023) retrieve memories based on recency, importance, and relevance to the current situation. Generative Agents also generate tree-structured reflections, but they focus on a continuous scenario rather than task-oriented rules. AutoManual Chen et al. (2024) enables LLM agents to autonomously build their understanding through interaction and adapt to new environments. It categorizes environmental knowledge into diverse rules and optimizes them in an online fashion.

3.5 Self-improvement

In code environment-integrated agent learning frameworks, agents can observe the environment, issue actions, and receive update signals through a unified program execution process. This integration is evident in numerous frameworks designed for training general-purpose agents using reinforcement learning. A prime exemplification of this approach is Reinforcement Learning with Execution Feedback (RLEF) (Gehring et al., 2025), which enables large language models to progressively enhance their capabilities by utilizing execution results as feedback. This methodology substantially improves the models' problem-solving abilities while minimizing the need for human intervention. When evaluated on competitive programming challenges, RLEF demonstrates exceptional performance, establishing new benchmarks while significantly reducing sample efficiency requirements and facilitating effective multi-stage self-improvement processes. Complementary to this, the Self-Debugging framework proposed by **?** teaches LLMs to automatically identify and fix their own code mistakes by iteratively analyzing execution errors and refining their outputs. This approach underscores the growing potential of execution feedback—not just as a training signal but as an active loop for runtime correction and continual self-improvement within agentic workflows.

4 Challenges and Future Directions

Despite the promising synergy between code and LLMs, several challenges remain. Current models still struggle with reliably generating syntactically correct and semantically coherent code in complex, multi-turn, or multimodal contexts. Robust program-based reasoning requires stronger grounding and debugging abilities, especially in open-ended, real-world scenarios. Memory management through code structures remains underexplored, particularly for long-horizon planning and persistent agent behavior. Moreover, aligning code-executing agents with user intent and safety constraints remains a key concern. Future work should focus on improving the modularity and generalization of code-based reasoning, developing more transparent and adaptive execution-feedback loops, and scaling multi-agent collaboration via code as a shared coordination protocol. Building open, interpretable, and continually learning agents will be essential for realizing the full potential of code-powered intelligence.

5 Conclusion

In this survey, we have systematically explored how code functions not merely as an output modality but as a foundational catalyst that empowers large language models (LLMs) to reason, act, and improve more effectively. By synthesizing findings across a wide array of recent literature, we highlight how the integration of code into pretraining and prompting pipelines transforms LLMs into more capable, agentic systems. Code provides a dual advantage: it inherits the compositional semantics and readability of natural language while introducing the formal structure, executability, and verifiability of symbolic systems. These properties make code an ideal training signal for developing more precise, structured, and interpretable model behavior. Through this lens, we examined five core dimensions in which code enhances LLM agents: i) Code-LLMs: Code-trained models display superior programming skills, compositional generalization, and cross-domain capabilities, forming the technical backbone of many agentic systems; ii) Decision-Making: Code-aware models demonstrate more reliable planning, reasoning, and problem decomposition—often via program-based prompting techniques that yield modular and verifiable thought processes; iii) Execution: Code enables LLMs to interface with tool APIs, control environments, and translate natural language into executable formal commands, facilitating grounded and robust task completion; iv) Memory Management: Through structured function definitions and calls, code improves how models organize, retrieve, and reuse information—providing scaffolds for modular memory strategies and persistent workflows; and v) Self-Improvement: The integration of execution environments allows LLMs to observe, debug, and iteratively refine their outputs using programmatic feedback, paving the way for continual learning and autonomous skill acquisition. By viewing code as a core enabler of reasoning, structure, and agency, we aim to provide a unified perspective on how code is transforming the capabilities and design of LLM agents—and to inspire future research at the intersection of language, logic, and computation.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars!, 2023.

Arthur Bucker, Pablo Ortega-Kral, Jonathan Francis, and Jean Oh. Grappa: Generalizing and adapting robot policies via online agentic guidance, 2025.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers, 2024. URL https://arxiv.org/abs/2305.17126.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. Advances in Neural Information Processing Systems, 37:589–631, 2024.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023a.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023b.

Yangyi Chen, Xingyao Wang, Manling Li, Derek Hoiem, and Heng Ji. Vistruct: Visual structural knowledge extraction via curriculum guided code-vision representation, 2023c.

Yongchao Chen, Harsh Jhamtani, Srinagesh Sharma, Chuchu Fan, and Chi Wang. Steering large language models between code execution and textual reasoning, 2025. URL https://arxiv.org/abs/2410.03524.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages, 2023.

Hanxing Ding, Shuchang Tao, Liang Pang, Zihao Wei, Jinyang Gao, Bolin Ding, Huawei Shen, and Xueqi Chen. Toolcoder: A systematic code-empowered tool learning framework for large language models, 2025. URL https://arxiv.org/abs/2502.11404.

Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, Roman Wang, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. Proceedings of the National Academy of Sciences, 119(32), aug 2022. doi: 10.1073/pnas.2123433119. URL https://doi.org/10.1073%2Fpnas.2123433119.

Hao Fu, Yao; Peng and Tushar Khot. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. Yao Fu's Notion, Dec 2022. URL https://yaofu.notion.site/How-does-GPT-Obtain-its-Ability-Tracing-Emergent-Abilities-of-Language-Models-to-their-Sources-b9a57

Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models, 2023.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In International Conference on Machine Learning, pp. 10764–10799. PMLR, 2023.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2025. URL https://arxiv.org/abs/2410.02089.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning, 2023. URL https://arxiv.org/abs/2305.14909.

Xuehang Guo, Xingyao Wang, Yangyi Chen, Sha Li, Chi Han, Manling Li, and Heng Ji. Syncmind: Measuring agent out-of-sync recovery in collaborative software engineering. arXiv preprint arXiv:2502.06994, 2025.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, et al. Metagpt: Meta programming for multi-agent collaborative framework. arXiv preprint arXiv:2308.00352, 2023.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv preprint arXiv:2312.13010, 2023.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. Opencoder: The open cookbook for top-tier code large language models. arXiv preprint arXiv:2411.04905, 2024.

Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp. 6700–6709, 2019.

Qing Jiang, Gen Luo, Yuqin Yang, Yuda Xiong, Yihao Chen, Zhaoyang Zeng, Tianhe Ren, and Lei Zhang. Chatrex: Taming multimodal llm for joint perception and understanding, 2025. URL https://arxiv.org/abs/2411.18363.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents. arXiv preprint arXiv:2402.03610, 2024.

Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding, 2023.

Ioktong Lei and Zhidong Deng. Selfzcot: a self-prompt zero-shot cot from semantic-level to code-level for a better utilization of llms, 2023.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator, 2024a. URL https://arxiv.org/abs/2312.04474.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. Advances in Neural Information Processing Systems, 36:51991–52008, 2023a.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. arXiv preprint arXiv:2301.12597, 2023b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. Science, 378(6624):1092–1097, 2022.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. Autoflow: Automated workflow generation for large language model agents. arXiv preprint arXiv:2407.12821, 2024b.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher R'e, Diana Acosta-Navas, Drew A. Hudson, E. Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel J. Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan S. Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas F. Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. Annals of the New York Academy of Sciences, 1525:140 – 146, 2023a. URL https://api.semanticscholar.org/CorpusID:253553585.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models, 2023b.

Fangyu Liu, Francesco Piccinno, Syrine Krichene, Chenxi Pang, Kenton Lee, Mandar Joshi, Yasemin Altun, Nigel Collier, and Julian Martin Eisenschlos. Matcha: Enhancing visual language pretraining with math reasoning and chart derendering, 2023a.

Jiateng Liu, Sha Li, Zhenhailong Wang, Manling Li, and Heng Ji. A language-first approach for procedure planning. In Findings of the Association for Computational Linguistics: ACL 2023, pp. 1941–1954, 2023b.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. arXiv preprint arXiv:2301.13379, 2023.

Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At which training stage does code data help llms reasoning?, 2023.

Angelos Mavrogiannis, Dehao Yuan, and Yiannis Aloimonos. Discovering object attributes by prompting large language models with perception-action apis, 2025. URL https://arxiv.org/abs/2409.15505.

Swaroop Mishra, Matthew Finlayson, Pan Lu, Leonard Tang, Sean Welleck, Chitta Baral, Tanmay Rajpurohit, Oyvind Tafjord, Ashish Sabharwal, Peter Clark, and Ashwin Kalyan. Lila: A unified benchmark for mathematical reasoning, 2023.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. arXiv preprint arXiv:2112.09332, 2021.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474, 2022.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: Towards llms as operating systems. 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In Proceedings of the 36th annual acm symposium on user interface software and technology, pp. 1–22, 2023.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023. URL https://arxiv.org/abs/2305.15334.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. arXiv preprint arXiv:2307.07924, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023. URL https://arxiv.org/abs/2307.16789.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models, 2022.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297, 2020.

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

Zhengliang Shi, Shen Gao, Lingyong Yan, Yue Feng, Xiuyi Chen, Zhumin Chen, Dawei Yin, Suzan Verberne, and Zhaochun Ren. Tool learning in the wild: Empowering language models as automatic tool agents, 2025. URL https://arxiv.org/abs/2405.16533.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36:8634–8652, 2023.

Ruoxi Sun, Sercan O. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. Sql-palm: Improved large language model adaptation for text-to-sql, 2023.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. arXiv preprint arXiv:2303.08128, 2023.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment, 2024. URL https://arxiv.org/abs/2402.12275.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases, 2023. URL https://arxiv.org/abs/2306.05301.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291, 2023a.

Juan Wang, Di Guo, and Huaping Liu. Where to learn: Embodied perception learning planned by vision-language models. IEEE Transactions on Cognitive and Developmental Systems, 2025.

Xingyao Wang, Sha Li, and Heng Ji. Code4struct: Code generation for few-shot event structure prediction, 2023b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL https://arxiv.org/abs/2402.01030.

Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. Unlocking reasoning potential in large langauge models by scaling code-form planning, 2024. URL https://arxiv.org/abs/2409.12452.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. arXiv preprint arXiv:2308.08155, 2023.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models, 2022.

Mengdi Xu, Peide Huang, Wenhao Yu, Shiqi Liu, Xilun Zhang, Yaru Niu, Tingnan Zhang, Fei Xia, Jie Tan, and Ding Zhao. Creative robot tool use with large language models, 2023. URL https://arxiv.org/abs/2310.13065.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian McAuley. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms, 2025. URL https://arxiv.org/abs/2502.19411.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. Gpt4tools: Teaching large language model to use tools via self-instruction, 2023. URL https://arxiv.org/abs/2305.18752.

Yuan Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. Can llms reason in the wild with programs? arXiv preprint arXiv:2406.13764, 2024.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning, 2023.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2019.

Pengyu Zhao, Zijian Jin, and Ning Cheng. An in-depth survey of large language model-based artificial intelligence agents, 2023.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. arXiv preprint arXiv:2308.07921, 2023.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv preprint arXiv:2406.11931, 2024.

A Discussions

A.1 Intrinsic Qualities of Code that Contribute to LLM Empowerment

Reflecting on our definition of code in the introduction section (§1) as formal languages that are both human-interpretable and machine-executable, we highlight that while some features are shared by all code, programming language, as the most well-known and most established type of code, enjoy some unique advantages. In Figure 2, we provide a case study comparing code and natural language.

```
class IntelligentAgent
    """
    An intelligent agent utilizing a decision center (default: LLM) and a toolbox of available tools.

    Parameters:
    - decision_center: The decision-making center for the agent, defaults to an LLM.
    - toolbox: A list of available tools for the agent, default includes GOOGLE, Minecraft, and RoboticArm.

    Methods:
    - cs_rookie_ritual(): Executes a rookie ritual for a computer science rookie, obtaining plans from the decision
center and performing actions with the specified tools.
    """
    def __init__(self, decision_centor=LLM, toolbox=[GOOGLE, Minecraft, RoboticArm]):
        """
        Initialize an IntelligentAgent instance.

        Args:
        - decision_center: The decision-making center for the agent, defaults to an LLM.
        - toolbox: A list of available tools for the agent, default includes GOOGLE, Minecraft, and RoboticArm.
        """
        self.decision_centor = decision_centor
        self.toolbox = toobox

    def cs_rookie_ritual(self):
        """
        Execute a rookie ritual for a computer science rookie using the decision center and specified tools.
        """
        plans = self.decision_centor("Hello, World!", toolbox=self.toolbox)
        for tool, action in plans:
            action(tool)
```

1. Object-Oriented Programming
Adv: Structured

2. Functional Programming
Adv: Modular & Explicit

3. Procedural Programming
Adv: Step-by-Step

Figure 2: We generate pseudo-code for the "IntelligentAgent" class and employ ChatGPT to compile its docstring. By contrasting the self-explanatory code with its natural language docstring, we observe that code exhibits greater structure, expressiveness, and logical coherence, underscoring certain advantages of code over natural language.

First, we talk about the core feature shared by all code within the range of our definition. The inherent nature of code is that they are explicit and have clear definitions for every single line, while natural language is generally in free form and can be very ambiguous. Consequently, code is significantly better at expressing detailed commands, signifying a specific step, and transmitting control signals. This generally led to the improvement in §**??**, the improvement for more controlled planning (cf. planning part in §3.2), and also helped with action execution (§3.3).

Programming languages, a critical component of the code family, are specifically designed for machine communication. Their advantages extend beyond mere explicitness and clarity. One overwhelming feature of programming languages (though some formal languages also define logical commands and loops) is that they contain structural definitions. Some well-known features are logical operands (If & Else), loops (For & While), nesting (within Functions), and even class definition and class inheritance (Object Oriented Programming). This feature makes them super suitable for expressing nesting and complicated structures (cf. §**??** and the perception part in §3.2). Another feature is that programming languages are often paired with a very powerful execution environment. This executable feature benefits much as it naturally delegates some harder tasks to lower level, like arithmetic computing or interacting with a simulated environment when connecting to a Database, Minecraft, and so on, also facilitating reasoning discussed in §**??**. What's more, the execution often includes feedback mechanisms, which can be valuable for further refining the generator (§**??** and §3.5).

A.2 Breadth by Code Delegation or Depth by Multimodality Joint Learning

LLMs can swiftly and cost-effectively address tasks involving more data modalities by utilizing code to invoke tools. Simultaneously, joint fine-tuning on multimodal data enhances the model's precision and robustness in perceiving each modality, resulting in superior task performance. For instance, on the VQA dataset GQA (Hudson & Manning, 2019), ViperGPT (Surís et al., 2023), a typical code-centric paradigm, marginally surpasses the multimodal model BLIP-2 (Li et al., 2023b) in the zero-shot scenario after learning visual model API usages. However, its accuracy remains significantly lower than other supervised multimodal models. It is also still uncertain whether this approach will surpass the state-of-the-art models on multi-modal procedural planning (Liu et al., 2023b). One reason is that the code-centric paradigm's effectiveness hinges on the central decision model and individual task execution components. This makes code-delegation approaches susceptible to error accumulation across steps and highly influenced by the worst-performing sub-modules or tools. Nevertheless, code delegation remains essential, as certain tools' advantages, such as the precision of calculators and the flexibility of search engines, cannot be learned by training multimodality models alone. The high extensibility of the code-centric paradigm to various tools and modalities also makes it a perfect fit for domains where training data is hard to collect at scale. We anticipate that the central decision model, utilizing code to invoke tools, will evolve from text-only LLMs to multimodality models capable of comprehensively understanding and processing multimodal data.

A.3 The Potential of Using Code-centric Framework for Intelligent Agent Construction

We observed a rising trend in leveraging code in the construction of LLM-based intelligent agents. As shown in §3, we showed three major scenarios where agents can effectively benefit from code usage. We also identified that this trend mainly originated from the increasing need to evaluate agents in a real-world scenario, where executive environments and interactions are everywhere. A natural question arises: Does code have the potential to substitute natural language and become the dominant media in the construction of agents?

A lot of work has begun to adopt this approach, like Voyager (Wang et al., 2023a) in a simulated Minecraft environment. They used code for high-level planning, low-level control sequence, and execution to interact with the environment. Acquired skills are also organized in the format of code snippets. With the code-centric paradigm, the framework is highly automatic and efficient. However, it's also true that many framework today are still using natural language for planning, probably because they provide more human-interpretable reasoning steps. Human feedback in natural language is also widely used to harvest strong reward models that reflect real human preferences. We hypothesize that the integration of code will continue gaining popularity on our path to AGI, especially for facilitating interactions between agents and the real world. Nevertheless, natural language could hardly be replaced regarding the interaction between agents and humans.(Drori et al., 2022; Chen et al., 2023a; Lei & Deng, 2023). Leveraging this understanding, we aim to explore novel research avenues in LLM reasoning inspired by the utilization of "code".

B Paper Statistics from Arxiv

We write a Python script that serves as a web scraper to extract paper details from the ArXiv preprint server, specifically focusing on the field of computer science. The web scraper gathers information about papers related to specific topics, including code, LLM, and IA. The script navigates through the ArXiv website, fetching essential details such as paper title, abstract, authors, and subject categories. We analyze and visualize data related to these papers in Figure 3, intending to provide insights into the trends and relationships between LLMs, code-related topics, and IAs in the past few years.

C Benchmarks for Evaluating Complex Reasoning with Code:

While there exist many benchmarks used to evaluate the abilities of LLMs (Liang et al., 2023b) across many disciplines, the benchmarks that most directly evaluate LLMs pre-trained on code in complex reasoning tasks are programming benchmarks such as CodeBLEU (Ren et al., 2020), where metrics that better match a human's evaluation of what is good logical, interpretable, and syntactically concise code was created,

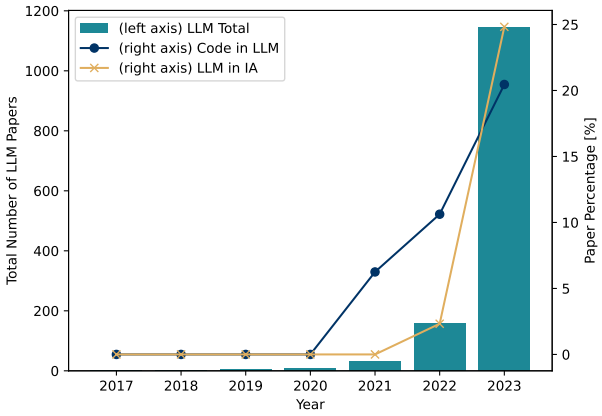

Figure 3: Paper statistics from Arxiv. We identified a significant and growing trend in recent research focused on code-based large language models (LLMs) and LLM-based intelligent agents (IAs). Code usage contributes much to the success of these cutting-edge models and systems.

and CodeXGLUE (Lu et al., 2021) where multiple programming tasks such as code repair and code defect detection were accumulated into one dataset. Other suitable benchmarks include math datasets such as many of MIT's undergraduate math courses such as calculus and linear algebra, (Drori et al., 2022), LILA, a compilation of 23 tasks that test for mathematical abilities, language format, language diversity, and external knowledge abilities of LLMs (Mishra et al., 2023), and theorem proving from the metamath theorem code language (Polu & Sutskever, 2020). Others include question-answering tasks that require complex abilities to perform data retrieval in SQL databases (Ye et al., 2023), such as those seen by the Spider dataset (Yu et al., 2019; Rajkumar et al., 2022).

D The Comprehensive Paper List

To complement the core paper list presented in Figure 1, we have included a comprehensive list of papers in Figure **??**. It is important to note that this list excludes papers used for performance comparisons between code and natural language. Instead, it focuses on papers that have utilized code to augment the capabilities of Large Language Models and intelligent agents.

E Mappings of Sections to Core Code features

In each section, we identify key code features that contribute to the success of enhancing Large Language Models and Intelligent Agents. The correlation between each section and its core features is detailed in Table **??**. We have classified code features into three main categories: Machine Executable, Structured and Expressive, and Explicit and Unambiguous. Various aspects of these core features play a pivotal role in the effective use of code. Detailed explanations of these aspects are provided in the right column of the table. Additionally, further information can be found in the preamble of each respective section.