IF LLM IS THE WIZARD, THEN CODE IS THE WAND: A SURVEY ON HOW CODE EMPOWERS LARGE LANGUAGE MODELS TO SERVE AS INTELLIGENT AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The prominent large language models (LLMs) of today differ from past language models not only in size, but also in the fact that they are trained on a combination of natural language and code. As a medium between humans and computers, code translates high-level goals into executable steps, featuring standard syntax, logical consistency, abstraction, and modularity. In this survey, we present an overview of the various benefits of integrating code into LLMs' training data. In addition, we trace how these profound capabilities of LLMs, brought by code, have led to their emergence as intelligent agents (IAs). Finally, we present several key challenges and future directions of empowering code-LLMs to serve as IAs.

1 INTRODUCTION

Code has become an integral component in the training data of large language models (LLMs), not only because the acquired programming skills facilitate commercial applications, such as Github Copilot[1], but also because it improves the models' previously lacking reasoning abilities (Liang et al., 2023b), enabling the models to handle a wider range of more complex tasks. Consequently, LLMs rapidly emerge as a primary decision-making hub for intelligent agents (IAs) dealing with tasks that involve more intricate steps (Zhao et al., 2023). As depicted in Figure 1, 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 IAs, based on the taxonomy of relevant papers (see Figure 2).

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 A). With insights from characteristics of code (see our case studies in Appendix B.1), our literature review reveals that integrating code into LLM training *i)* enhances their programming and reasoning capabilities (§2); *ii)* enables the models to directly generate executable, fine-grained steps during decision-making, thereby facilitating their scalability in incorporating various tool modules through function calls (§3); and *iii)* situates the LLMs within a code execution environment, allowing them to receive automated feedback from integrated evaluation modules and self-improve (§4). In addition, as LLMs are becoming key decision-makers for IAs in complex real-world tasks, our survey also explores how these advantages facilitate their functioning as IAs (§5). We discuss several open challenges and promising future directions in Appendix C.

2 CODE PRE-TRAINING BOOSTS LLMS' PERFORMANCE

Pre-training LLMs on code, exemplified by OpenAI's GPT Codex (Chen et al., 2021), expands their task scope beyond natural language. Code's requirement for logically coherent, ordered sequences of executable steps enhances LLMs' chain-of-thought (CoT) performance, improving complex reasoning skills (Lyu et al., 2023; Zhou et al., 2023a; Fu & Khot, 2022). By implicitly learning from code's structured format, code LLMs excel in commonsense structured reasoning tasks (Furuta et al., 2023; Liu et al., 2023a).

[1]https://github.com/features/copilot.

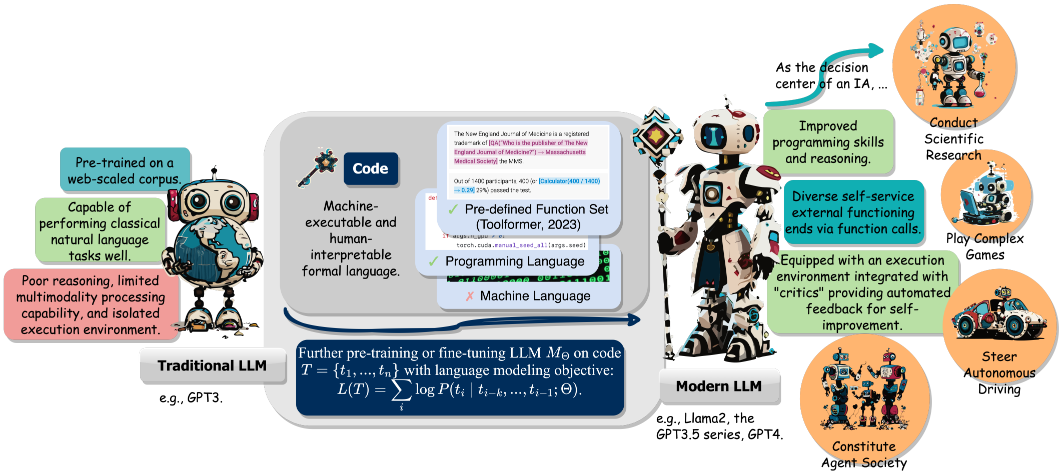


Figure 1: An illustration of how code empowers large language models (LLMs) and enhances their downstream applications as intelligent agents (IAs). While traditional LLMs excel in conventional natural language tasks like document classification and question answering, further pre-training or fine-tuning LLMs with human-interpretable and machine-executable code serves as an additional power-up — akin to equipping wizards with mana-boosting wands. This significantly boosts their performance as IAs through intricately woven operational steps.

In the following sections, we outline three key areas where code pre-training benefits LLMs: *i)* improving programming proficiency in §2.1, *ii)* enhancing complex reasoning capabilities in §2.2, and *iii)* facilitating the capture of structured commonsense knowledge in §2.3, as depicted in Figure 3.

2.1 Strengthen LLMs' Programming Skills

LLM as a strong coder. Earlier language models only generate domain-specific programs (Ellis et al., 2019) or restrict to one of the generic programming languages, such as Java or C# (Alon et al., 2020). Empowered by the increasing number of parameters and computing resources, recent LLM-based code generation models (such as AlphaCode (Li et al., 2022), CodeGen (Nijkamp et al., 2022), SantaCoder (Allal et al., 2023), PolyCoder (Xu et al., 2022)) could master more than 10 languages within the same model and show unprecedented success. A well-known work is CodeX (Chen et al., 2021), with 12 billion parameters that reads the entire GitHub database and is able to solve 72.31% of challenging Python programming problems created by humans. Recent studies (Zan et al., 2023; Xu et al., 2022; Du et al., 2023; Vaithilingam et al., 2022; Wong et al., 2023; Fan et al., 2023) have provided systematic surveys and evaluations of existing code-LLMs.

With its strong code generation ability, LLMs benefit various applications that rely on code, such as database administration (Zhou et al., 2023b), embedded control (Liang et al., 2023a), game design (Roberts et al.), spreadsheet data analysis (Liu et al., 2023c), and website generation (Calò & De Russis, 2023).

LLM as a state-of-the-art code evaluator. Interestingly, LLMs themselves could also serve as state-of-the-art evaluators (i.e., analyze and score) for human or machine-generated codes. Kang et al. (2023a) leverage LLM-based models for code fault localization, while Zhuo (2023) uses GPT-3.5 to evaluate the functional correctness and human preferences of code generation. In addition, Deng et al. (2023a) design a LLM-based penetration testing tool and find that LLMs demonstrate proficiency in using testing tools, interpreting outputs, and proposing subsequent actions. Two recent efforts (Li et al., 2023a; Mohajer et al., 2023) also utilize LLM for examining and analyzing source code without executing it. Furthermore, LLMs are used for automatic bug reproduction in Kang et al. (2023b) and vulnerable software evaluation in Noever (2023).

Multi-LLM collaboration solves complex coding problems. Collaborative coding among several role-specific LLM agents exhibits more accurate and robust performance towards complex tasks.

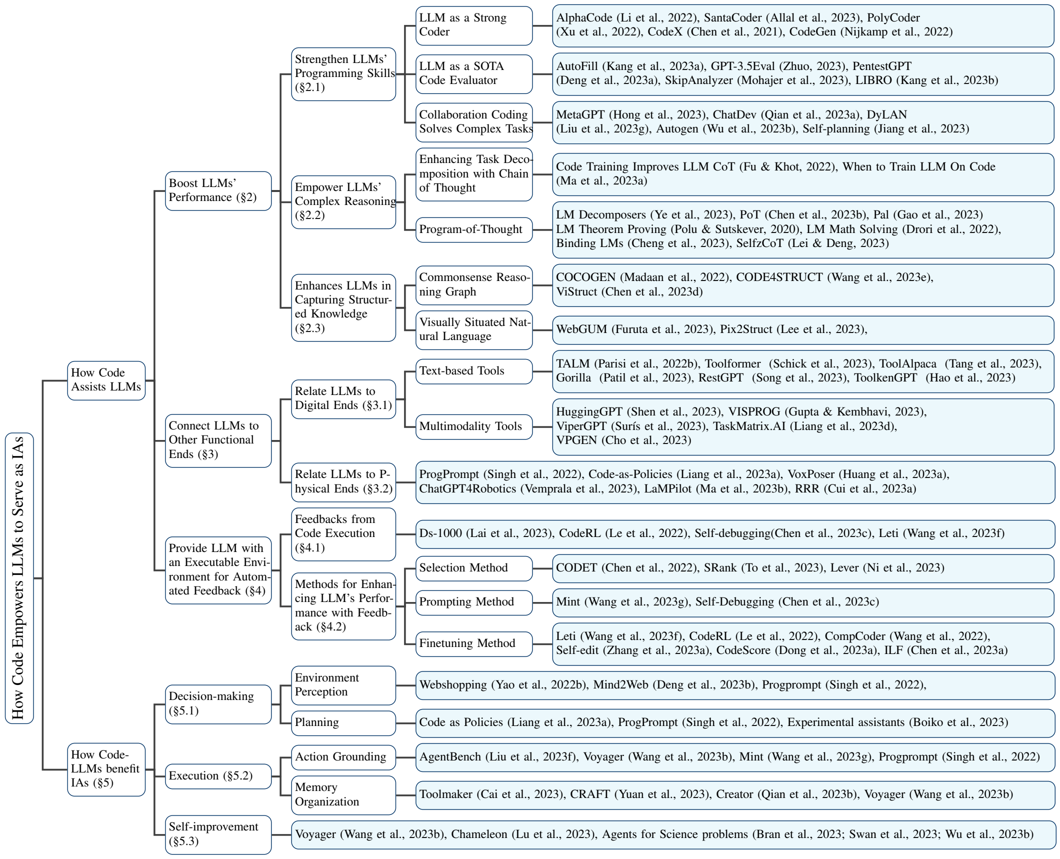
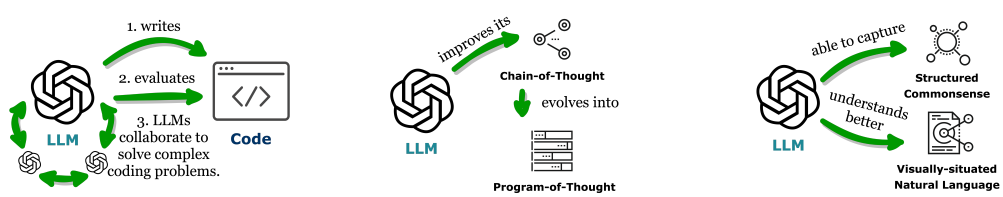


Figure 2: The organization of our paper, with a curated list of the most representative works. The complete work list is provided in Appendix D.



(a) Strengthen LLMs' programming and code evaluation skills (§2.1).

(b) Empower LLMs' complex reasoning, decoupling computation from language understanding (§2.2).

(c) Enable LLM to better capture structured knowledge and better understand complex multimedia data (§2.3).

Figure 3: How code pre-training boosts LLMs' performance.

Hong et al. (2023) incorporates human programming workflows as guides to coordinate different agents. Dong et al. (2023b) assigned three roles: analyst, coder, and tester to three distinct "GPT-3.5"s, which surpasses GPT-4 in code generation. Meanwhile, Qian et al. (2023a) designs a chat-powered software development process, assigning more than three roles to separate LLM agents. Other similar methods (Liu et al., 2023g; Talebirad & Nadiri, 2023; Wu et al., 2023b; Jiang et al., 2023) all employ multiple code-LLM agents or different phases of the same agent for code generation, software developments, or leveraging generated intermediate codes for other general purpose tasks.

2.2 Empower LLMs' Complex Reasoning

Code pre-training improves chain-of-thought performance. CoT prompting, where prompt inputs are designed with chains of reasoning, allows the LLM to condition its generation with further steps of reasoning (Wei et al., 2023). CoT has seen successful in the task decomposition of many problem settings, including planning (Huang et al., 2022b) and evidence-based question answering (Dua et al., 2022; Ye et al., 2023).

While LLM CoT ability was originally mainly attributed to dramatically increased model sizes (Wei et al., 2022b), recent evidence compiled by Fu & Khot (2022) suggests that much of the performance improvements from CoT stems from its pre-training on code. In support of this hypothesis, Ma et al. (2023a) show that pre-training on code in small-sized LLMs (2.6B) (Zeng et al., 2021) enhances performance when using CoT, and even more remarkably that smaller code-pretrained LLMs outperform their larger non-code counterparts across many different tasks. Furthermore, their study indicates that incorporating a greater volume of code during the initial phases of LLM training significantly enhances its efficacy in reasoning tasks. Notably, both Fu & Khot (2022) and Ma et al. (2023a) show that pre-training on code improves LLM performance in both standard and CoT prompting scenarios across downstream tasks.

Program-of-thought outperforms chain-of-thought. Furthermore, in comparison to vanilla CoT methods, LLMs that first translate and decompose a natural language task into code (Chen et al., 2023b; Gao et al., 2023), typically termed program-of-thought (PoT) prompting or program-aided language model, see sizable gains in tasks that require disambiguation in both language and explicit longitudinal structure. This approach is especially effective in complex areas such as theoretical mathematics (Polu & Sutskever, 2020), undergraduate mathematics (Drori et al., 2022), and question answering with data retrieval (Sun et al., 2023b; Cheng et al., 2023).

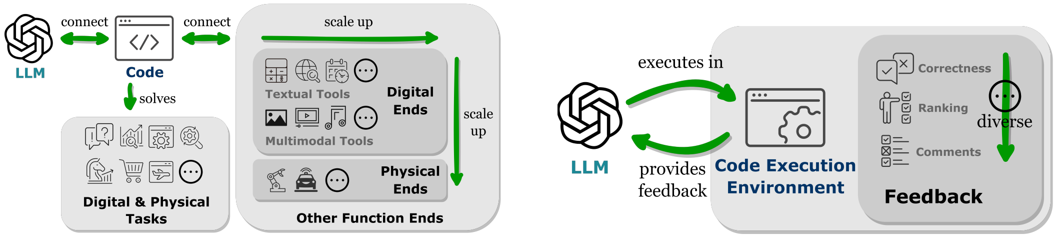
PoT enhances performance due to the precision and verifiability inherent in code as a machine-executable language. PoT implementations from Chen et al. (2023b), Gao et al. (2023), and Ye et al. (2023) show that by directly executing code and verifying outcomes post translation by LLMs, one can effectively mitigate the effects of incorrect reasoning in CoT (Ji et al., 2023). Such improvements are not limited to purely executable coding languages such as Python or SQL, nor are they limited to tasks that are specifically rigid in structure such as mathematics (Drori et al., 2022) and data retrieval (Rajkumar et al., 2022). Enhancements also extend to the realm where even translating into pseudo-code to decompose a task can improve zero-shot performance (Lei & Deng, 2023) in word problems containing numbers, and general reasoning tasks such as StrategyQA (Geva et al., 2021).

2.3 Enable LLMs to Capture Structured Knowledge

Code generation unveils superior structural commonsense reasoning. Given that code possesses the graph structure of symbolic representations, translating textual graphs, tables, or charts into code empowers a code-driven LLM to logically process such information according to code reasoning and generation principles. Previous work (Madaan et al., 2022; Wang et al., 2023e) shows that LLMs undergoing code pre-training may rival, or even exceed, their fine-tuned natural language counterparts in tasks involving structural commonsense reasoning, even with limited or no training data.

COCOGEN (Madaan et al., 2022) first reframed the commonsense reasoning graph completion task as a code generation task and demonstrated improved few-shot performance in reasoning graphs, table entity state tracking, and explanation graph generation. Building on this perspective, CODE4STRUCT (Wang et al., 2023e) applied code-LLMs to semantic structures, focusing on the event argument extraction task. By leveraging code's features such as comments and type annotation, it achieved competitive performance with minimal training instances. ViStruct (Chen et al., 2023d) extended this approach further to multimodal tasks, leveraging programming language for representing visually structural knowledge.

Markup code mastery evolves visually situated natural language understanding. An additional research stream involves using markup code (e.g., HTML and CSS) to delineate and derender structured graphical information in graphical user interfaces (GUIs) or visualizations in documents, aiding large vision-language models (LVLMs) in capturing visually situated natural language (VSNL). For LVLMs' markup code understanding, WebGUM (Furuta et al., 2023) exemplified autonomous web



(a) The code-centric tool-calling paradigm serves as a unified interface between LLMs and various functional ends, thus enabling many cross-modality and cross-domain tasks. (§3).

(b) LLMs can be embedded into a code execution environment, where they collect faithful, automatic, and customizable feedback for self-improvement. (§4).

Figure 4: How code connects LLMs to other function ends and how code execution environments provide LLMs with feedback.

navigation. It employed a pre-training approach using webpage screenshots and the corresponding HTML as input, and navigation action as output, showcasing the effectiveness of pre-training model with markup code augmentation in webpage understanding. For markup code generation, Pix2Struct (Lee et al., 2023) achieved SOTA in VSNL understanding by pre-training an image-to-text model on masked website screenshots, and further training with OCR, language modeling, and image captioning objectives.

3 Code Connects LLMs to Other Function Ends

Recent studies reveal that connecting LLMs to diverse functional ends enhances their task performance (Mialon et al., 2023; Parisi et al., 2022a; Peng et al., 2023; Gou et al., 2023). These functional ends enable LLMs to access external knowledge, engage with different modalities, and interact effectively with various environments. As shown in Table 1 in the appendix, a prevalent trend is observed where LLMs generate programming languages or use pre-defined functions to connect with other functional ends—a phenomenon we termed the *code-centric paradigm*, which provides a simple and clear interaction method for LLMs, boosting flexibility and scalability. Notably, as illustrated in Figure 4a, it allows LLMs to interact with functional ends across diverse modalities and domains, expanding their capacity to handle complex tasks.

In §3.1, we explore textual and multimodal (digital) tools connected to LLMs, while §3.2 focuses on physical-world functional ends, including robots and autonomous driving. This showcases the versatility of LLMs in addressing challenges across various modalities and domains.

3.1 Relate LLMs to Digital Ends

Text-Based Tools. The code-centric framework initially drive progress in text-based tools. Prior to the popularity of this framework, research on augmenting LMs with single tools like information retrivers (Guu et al., 2020; Lewis et al., 2020; Izacard et al., 2022) required a hardcoded-in-inference-mechanism (e.g. always calling a retriever before the generation starts), limiting flexibility and scalability. TALM (Parisi et al., 2022b) first incorporates multiple text-based tools by invoking API calls with a pre-defined delimiter, enabling unambiguous calls to any text-based tools at any position of generation. Following their work, Toolformer (Schick et al., 2023) marks API calls with <API> </API>along with their enclosed contents. Later, diverse tool-learning approaches were introduced to facilitate the integration of numerous text-based tools across various foundational models (Song et al., 2023; Hao et al., 2023; Tang et al., 2023). The code-centric framework facilitates the invocation of a diverse range of external text modules. These include calculators, calendars, machine translation systems, web navigation tools, as well as APIs from HuggingFace and TorchHub (Thoppilan et al., 2022; Yao et al., 2022c; Shuster et al., 2022; Jin et al., 2023; Yao et al., 2022a; Liu et al., 2023e; Jin et al., 2023; Patil et al., 2023).

Multimodal Tools. The high scalability of the code-centric LLM paradigm extends tool-learning to other modalities. Early work (Gupta & Kembhavi, 2023; Surís et al., 2023; Subramanian et al., 2023) utilizes this paradigm for visual question answering. For instance, VISPROG (Gupta & Kembhavi, 2023) compiles pretrained computer vision models and functions into APIs, creating programs for question-targeted image understanding via in-context learning with LLMs. Similar work, including ViperGPT (Surís et al., 2023) and CodeVQA (Subramanian et al., 2023), directly generates more flexible Python code with Codex, allowing for potentially complex control flows using LLMs' pre-trained knowledge. Beyond visual reasoning, code connects LLMs with multi-modal generative tools in image generation tasks (Cho et al., 2023; Feng et al., 2023; Wu et al., 2023a), leveraging code's unambiguous nature in generating images matching text prompts.

Across modalities, recent collaborative efforts (Shen et al., 2023; Yang et al., 2023; Liang et al., 2023d) consider diverse tools. For example, MM-REACT (Yang et al., 2023) integrates video recognition models, Chameleon (Lu et al., 2023) includes tools like visual text detectors or web search, and HuggingGPT (Shen et al., 2023) connects LLMs to various Hugging Face models. TaskMatrix.AI (Liang et al., 2023d) expands API diversity, including visual, figure, music, and game APIs. The flexibility of code enables LLMs to jointly use diverse multimodal tools, enhancing their versatility as general-purpose multimodal problem solvers.

3.2 Relate LLMs to Physical Ends

The code-centric paradigm has liberated the connection between LLMs and the physical world, enabling adaptable calls to tools and execution modules in real-world scenarios. This sparked a wave of research integrating LLMs with robotics and autonomous driving.

PaLM-SayCan (Ahn et al., 2022) stands out as a successful approach, employing LLMs to generate policy codes for robotic tasks. Following this, recent developments showcase LLMs serving as the brain for robotics planning and control. ProgPrompt (Singh et al., 2022) introduced program-like specifications for robot task planning, while others extended this approach to human-robot interaction and drone control (Huang et al., 2023a; Liang et al., 2023a; Vemprala et al., 2023).

LLMs trained with code exhibit potential in complex tasks like human-vehicle interactions and autonomous driving (Cui et al., 2023b; Huang et al., 2023b; Li et al., 2023d). Wayve's LINGO-1 (Wayve, 2023) serves as an industry example, using an open-loop vision, language, and action LLM to enhance the explainability of driving models. LLMs can understand intricate commands, generate actionable codes (Ma et al., 2023b), and execute them by calling low-level vehicle planning and control APIs (Cui et al., 2023a; Sha et al., 2023; Mao et al., 2023).

Despite challenges, including latency and accuracy issues, as well as the lack of adequate simulation environments, datasets, and benchmarks (Kannan et al., 2023; Chen & Huang, 2023; Cui et al., 2023b), LLMs exhibit promise in understanding high-level instructions and executing code-related APIs in complex domains like robotics and autonomous driving.

4 Code Provides LLM with an Executable Environment for Automated Feedback

LLMs showcase enhanced performance beyond their training parameters, leveraging feedback in dynamic real-world applications (Liu et al., 2023f; Wang et al., 2023d). However, careful feedback selection is crucial, as noisy prompts can hinder downstream task performance (Zheng & Saparov, 2023). In real-world scenarios, automating feedback collection while maintaining fidelity is essential. Integrating LLMs into a code execution environment, illustrated in Figure 4b, allows for automated, deterministic feedback intake from executed code results (Chen et al., 2023a; Fernandes et al., 2023; Scheurer et al., 2022). Code interpreters provide an automatic pathway for LLMs to query internal feedback, eliminating the need for costly human annotations in tasks like debugging or optimizing faulty code (Chen et al., 2023a; Fernandes et al., 2023; Scheurer et al., 2022). Additionally, code environments enable LLMs to incorporate diverse external feedback forms, such as binary correctness critiques (Wang et al., 2023f), natural language explanations (Chen et al., 2023c), and ranking with reward values (Inala et al., 2022), offering highly customizable methods for performance enhancement.

We introduce various types of feedback from code execution in §4.1 and discuss common methods for utilizing this feedback to enhance LLMs in §4.2.

4.1 Various Feedback from Code Execution

Code execution enables more comprehensive assessment of LLM-generated content, utilizing deterministic execution results instead of relying solely on often ambiguous sequence-based metrics like BLEU (Papineni et al., 2002; Ren et al., 2020) and Rouge (Lin, 2004). Evaluation methods include creating unit tests (Chen et al., 2021; Hendrycks et al., 2021; Austin et al., 2021; Li et al., 2022; Huang et al., 2022a; Lai et al., 2023) and employing exact result matching techniques (Dong & Lapata, 2016; Zhong et al., 2017; Huang et al., 2022a). Feedback takes two primary forms: simple correctness feedback, indicating whether a program is correct or not, generated through critic models or rule-based methods (Wang et al., 2023f; Chen et al., 2023c), and textual feedback, provided by language models offering explanations about the program (Chen et al., 2023c) or summarizing comments on the program and its execution (Wang et al., 2023f; Chen et al., 2023c; Zhang et al., 2023a). Execution results can be translated into reward functions suitable for reinforcement learning approaches (Le et al., 2022). Additional feedback can be extracted through static analysis using software engineering tools, such as obtaining information from execution traces (Wang et al., 2017; Gupta et al., 2017) or using surface-form constraints on function calls (Lai et al., 2023).

4.2 Methods for Enhancing LLM's Performance with Feedback

Feedback from code execution and external evaluation modules enhances LLMs through three major approaches.

Selection-Based Method. Proven effective in tasks like code generation, selection-based methods leverage code execution outcomes to choose the best-performing code snippet. Majority voting (Chen et al., 2018; Li et al., 2022; Shi et al., 2022; Chen et al., 2022) and re-ranking schemes (Zhang et al., 2023b; Yin & Neubig, 2019; Zeng et al., 2023; Ni et al., 2023; Inala et al., 2022; To et al., 2023) have demonstrated efficacy in tasks and interactive environments, although they introduce inefficiencies requiring multiple rounds of generation and additional re-ranking models.

Prompting-Based Method. Modern LLMs integrate feedback into prompts, improving self-debugging with in-context learning (Wang et al., 2023g; Chen et al., 2023c). This approach, favored by LLM-based agents, relies on natural language explanations or error messages for better comprehension. However, its effectiveness depends on the LLM's in-context learning capabilities.

Finetuning-Based Method. Finetuning fundamentally improves LLMs by updating their parameterized knowledge. Wang et al. (2023f) exemplifies a direct optimization approach where the LLM is fine-tuned with a feedback-conditioned objective. Other methods include finetuning with synthetic unit tests data (Haluptzok et al., 2022; Dong et al., 2023a), training external modules like editors (Zhang et al., 2023a) or "refine models" (Chen et al., 2023a) to aid fine-tuning. Reinforcement learning is applied in CodeRL (Le et al., 2022) and Wang et al. (2022), with the former using fixed reward values based on unit tests and the latter employing compiler feedback. Despite effective refinement, finetuning involves resource-intensive data collection, and assessing predefined reward values, as in CodeRL, presents challenges (Le et al., 2022).

5 Application: Code-empowered LLMs Facilitate Intelligent Agents

In the preceding sections, our discussion highlighted the various ways in which code integration enhances LLMs. Going beyond, we discern that the benefits of code-empowered LLMs are especially pronounced in one key area: the development of IAs (Wang et al., 2023d; Xi et al., 2023; Zhao et al., 2023). Figure 5 helps to illustrate a standard operational pipeline of an IA, specifically serving as an embodied general daily assistant. We observe that the improvements brought about by code training in LLMs are firmly rooted in their practical operational steps when serving as IAs. These steps include (i) enhancing the IA's decision-making in terms of environment perception and planning (§5.1), (ii) streamlining execution by grounding actions in modular and explicit action primitives and efficiently organizing memory (§5.2), and (iii) optimizing performance through feedback automatically derived from the code execution environment (§5.3).

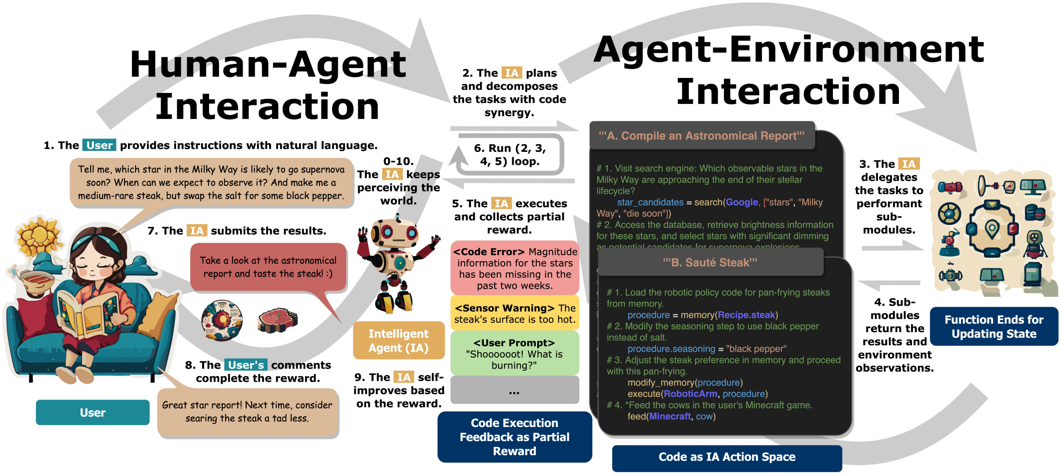


Figure 5: This figure illustrates the complete working pipeline of a LLM-based intelligent agent, mapping code-LLM abilities to specific phases: code-based planning in step (2), modular action parsing and tool creation in step (3), and automated feedback collection for enhanced agent self-improvement in step (5). Collectively, steps 0-10 in the entire loop benefit from code-LLMs' improved structured information understanding and perception.

5.1 Decision Making

Environment Perception As depicted in Figure 5 at step (0-10), the IA continuously perceives the world, engaging in interactions with humans and the environment, responding to relevant stimuli (e.g., human instructions for meal preparation), and planning and executing actions based on the observed environmental conditions (e.g., the kitchen layout). Utilizing LLMs as decision centers for IAs requires translating environmental observations into text, such as tasks based in the virtual household or Minecraft (Shridhar et al., 2020; Côté et al., 2018; Wang et al., 2023b; Zhu et al., 2023). Through pre-training on code, LLMs acquire the ability to better comprehend and generate structured representations .

One such intuitive example is web-page-based environments which are highly structured around HTML code. In agent tasks like web shopping (Yao et al., 2022b), web browsing (Deng et al., 2023b), and web-based QA (Nakano et al., 2021; Liu et al., 2023d), it is preferred to translate the web-based environment into HTML code rather than natural language, directly encompassing its structural information and thereby improving the LLM agent's overall perception. Moreover, in robotics research by Singh et al. (2022) and Liang et al. (2023a), the IAs are prompted with program-like specifications for objects in the environment, enabling the LLM to generate situated task plans based on the virtual objects they perceived.

Planning As illustrated in Figure 5 at step (2), IAs must break down intricate tasks into finer, manageable steps. As discussed in §2.2, when code-LLMs are employed for planning agent tasks, they exhibit enhanced reasoning capabilities. In addition, they generate the sub-tasks as executable programs when necessary, yielding more robust intermediate results, which the IA conditions on and refines its planning with greater precision. Furthermore, the IA seamlessly integrates performant tool APIs into planning, addressing the limitations such as poor mathematical reasoning and outdated information updates faced by vanilla LLMs during planning.

Typical examples that utilize code for planning are in two main categories. Progprompt (Singh et al., 2022) and Code as Policies (Liang et al., 2023a) represent the work utilizing code for better robot control. Both work highlight the benefits brought by code-based planning as they not only enable direct expressions of feedback loops, functions, and primitive APIs, but also facilitate direct access to third-party libraries. Another stream of work is concerned with the scenario when the agents' programming and mathematical reasoning abilities are crucial, like solving maths-related problems (Gao et al., 2023; Wang et al., 2023g) or doing experiments in the scientific domain (Boiko et al., 2023; Liffiton et al., 2023).

5.2 EXECUTION

Action Grounding As depicted in Figure 5 at step (3), when the IA interfaces with external function ends according to the planning, it must invoke action primitives from a pre-defined set of actions (i.e., functions). Connecting the IA with other function ends requires grounding actions into formalized function-like primitives. For instance, in a benchmark evaluating LLMs as agents in real-world scenarios (Liu et al., 2023f), seven out of eight scenarios involve code as the action space. Previous work generating agent plans with pure natural language necessitate an additional step-to-primitive module to ground those planning steps into code (Wang et al., 2023c; Yin et al., 2023). In contrast, IAs that plan with code-LLMs generate atomic action programs (Yao et al., 2022d; Wang et al., 2023g; Liang et al., 2023a; Singh et al., 2022), and can have their generation quickly parsed for execution.

Memory Organization As depicted in Figure 5 at step (3) and the component labeled "Function Ends for Updating State", the IA typically necessitates an external memory organization module to manage exposed information (Wang et al., 2023d), including original planning, task progress, execution history, available tool set, acquired skills, and users' early feedback. In this context, Code-LLM aids the IA's memory organization by employing highly abstract and modular code to record, organize, and access memory, especially for expanding the available tool set and manage acquired skills.

Typically, agent-written code snippets can serve as parts of the toolset, integrated into the memory organization of agents. This stream of research is known as tool creation approaches. TALM (Cai et al., 2023) proposes to use stronger agents (e.g. GPT-4 based agents) to write code as part of memory for weaker agents (e.g. GPT-3.5 based agents). In Creator (Qian et al., 2023b), agents themselves are highlighted as not only users of the tools but also their creators. Going further, Craft (Yuan et al., 2023) focuses on ensuring the created tools are indeed executable, making the framework more robust. Another work sharing this idea is Voyager (Wang et al., 2023b), in which the agent store learned skills in code format and execute them in the future when faced with similar tasks.

5.3 SELF-IMPROVEMENT

As illustrated in Figure 5 at step (5), when the IA's decision center, i.e., the LLM, operates within a code execution environment, the environment can integrate various evaluation modules to offer automated feedback (e.g., correctness, ranking, detailed comments). This significantly enhances the IA's early error correction and facilitates self-improvement. Voyager (Wang et al., 2023b) is a good example for agents that use feedback from the simulated environment. The agent learns from failure task cases and further horn its skills in Minecraft. Chameleon (Lu et al., 2023) receives feedback from a program verifier to decide whether it should regenerate an appropriate program. Mint (Wang et al., 2023g) can receive feedback from proxies, and the agent can thus self-improve in a multi-turn interactive setting. Importantly, this ability to self-improve from execution feedback is fundamental for agents' success at solving scientific problems (Bran et al., 2023; Swan et al., 2023; Wu et al., 2023b).

6 CONCLUSION

In this survey, we compile literature that elucidates how code empowers LLMs, as well as where code assists LLMs to serve as intelligent agents (IAs). To begin with, code possesses natural language's sequential readability while also embodying the abstraction and graph structure of symbolic representations, rendering it a conduit for knowledge perception and reasoning as an integral part of the LLMs' training corpus based on the mere language modeling objective. Through a comprehensive literature review, we observe that after code training, LLMs *i)* improve their programming skills and reasoning, *ii)* could generate highly formalized functions, enabling flexible connections to diverse functional ends across modalities and domains, and *iii)* engage in interaction with evaluation modules integrated in the code execution environment for automated self-improvement. Moreover, we find that the LLMs' capability enhancement brought by code training benefits their downstream application as IAs, manifesting in the specific operational steps of the IAs' workflow regarding decision-making, execution, and self-improvement. Beyond reviewing prior research, we put forth several challenges in this field to serve as guiding factors for potential future directions.

REFERENCES

Rajas Agashe, Srini Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. In Conference on Empirical Methods in Natural Language Processing, 2019. URL https://api.semanticscholar.org/CorpusID:202776931.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022.

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.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code, 2020.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot, 2017.

Daniil A. Boiko, Robert MacKnight, and Gabe Gomes. Emergent autonomous scientific research capabilities of large language models. ArXiv, abs/2304.05332, 2023. URL https://api.semanticscholar.org/CorpusID:258059651.

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools, 2023.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. ArXiv, abs/2305.17126, 2023. URL https://api.semanticscholar.org/CorpusID:258947222.

Tommaso Calò and Luigi De Russis. Leveraging large language models for end-user website generation. In International Symposium on End User Development, pp. 52–61. Springer, 2023.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback, 2023a.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022.

Juo-Tung Chen and Chien-Ming Huang. Forgetful large language models: Lessons learned from using llms in robot programming, 2023.

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.

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

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In International Conference on Learning Representations, 2018.

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

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

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.

Jaemin Cho, Abhay Zala, and Mohit Bansal. Visual programming for text-to-image generation and evaluation, 2023.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben A. Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew J. Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. ArXiv, abs/1806.11532, 2018. URL https://api.semanticscholar.org/CorpusID:49552345.

Can Cui, Yunsheng Ma, Xu Cao, Wenqian Ye, and Ziran Wang. Receive, reason, and react: Drive as you say with large language models in autonomous vehicles, 2023a.

Can Cui, Yunsheng Ma, Xu Cao, Wenqian Ye, Yang Zhou, Kaizhao Liang, Jintai Chen, Juanwu Lu, Zichong Yang, Kuei-Da Liao, Tianren Gao, Erlong Li, Kun Tang, Zhipeng Cao, Tong Zhou, Ao Liu, Xinrui Yan, Shuqi Mei, Jianguo Cao, Ziran Wang, and Chao Zheng. A survey on multimodal large language models for autonomous driving, 2023b.

Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. Pentestgpt: An llm-empowered automatic penetration testing tool, 2023a.

Xiang Deng, Yu Gu, Bo Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. ArXiv, abs/2306.06070, 2023b. URL https://api.semanticscholar.org/CorpusID:259129428.

Li Dong and Mirella Lapata. Language to logical form with neural attention, 2016.

Yihong Dong, Ji Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. ArXiv, abs/2301.09043, 2023a. URL https://api.semanticscholar.org/CorpusID:256105296.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2023b.

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.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.

Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions, 2022.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. Advances in Neural Information Processing Systems, 32, 2019.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems, 2023.

Weixi Feng, Wanrong Zhu, Tsu-jui Fu, Varun Jampani, Arjun Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. Layoutgpt: Compositional visual planning and generation with large language models. arXiv preprint arXiv:2305.15393, 2023.

Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José GC de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, et al. Bridging the gap: A survey on integrating (human) feedback for natural language generation. arXiv preprint arXiv:2305.00955, 2023.

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-t

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. Complexity-based prompting for multi-step reasoning, 2023.

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.

Mor Geva, Daniel Khashabi, Elad Segal, Tushar Khot, Dan Roth, and Jonathan Berant. Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies. Transactions of the Association for Computational Linguistics, 9:346–361, 2021. URL https://api.semanticscholar.org/CorpusID:230799347.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. May 2023.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In Proceedings of the aaai conference on artificial intelligence, volume 31, 2017.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 14953–14962, 2023.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In Hal Daumé III and Aarti Singh (eds.), Proceedings of the 37th International Conference on Machine Learning, volume 119 of Proceedings of Machine Learning Research, pp. 3929–3938. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/guu20a.html.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. arXiv preprint arXiv:2207.14502, 2022.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. May 2023.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. arXiv preprint arXiv:2105.09938, 2021.

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.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. Execution-based evaluation for data science code generation models. arXiv preprint arXiv:2211.09374, 2022a.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In International Conference on Machine Learning, pp. 9118–9147. PMLR, 2022b.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. arXiv preprint arXiv:2307.05973, 2023a.

Yu Huang, Yue Chen, and Zhu Li. Applications of large scale foundation models for autonomous driving. arXiv preprint arXiv:2311.12144, 2023b.

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.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. Fault-aware neural code rankers. Advances in Neural Information Processing Systems, 35:13419–13432, 2022.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. August 2022.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. ACM Computing Surveys, 55(12):1–38, mar 2023. doi: 10.1145/3571730. URL https://doi.org/10.1145%2F3571730.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. arXiv preprint arXiv:2303.06689, 2023.

Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. GeneGPT: Augmenting large language models with domain tools for improved access to biomedical information. April 2023.

Sungmin Kang, Gabin An, and Shin Yoo. A preliminary evaluation of llm-based fault localization. arXiv preprint arXiv:2308.05487, 2023a.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2312–2323. IEEE, 2023b.

Shyam Sundar Kannan, Vishnunandan LN Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. arXiv preprint arXiv:2309.10062, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In International Conference on Machine Learning, pp. 18319–18345. PMLR, 2023.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. Advances in Neural Information Processing Systems, 35:21314–21328, 2022.

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.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), Advances in Neural Information Processing Systems, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. The hitchhiker's guide to program analysis: A journey with large language models. arXiv preprint arXiv:2308.00245, 2023a.

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

Sha Li, Chi Han, Pengfei Yu, Carl Edwards, Manling Li, Xingyao Wang, Yi Fung, Charles Yu, Joel Tetreault, Eduard Hovy, and Heng Ji. Defining a new NLP playground. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 11932–11951, Singapore, December 2023c. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.799. URL https://aclanthology.org/2023.findings-emnlp.799.

Xin Li, Yeqi Bai, Pinlong Cai, Licheng Wen, Daocheng Fu, Bo Zhang, Xuemeng Yang, Xinyu Cai, Tao Ma, Jianfei Guo, et al. Towards knowledge-driven autonomous driving. arXiv preprint arXiv:2312.04316, 2023d.

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.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In 2023 IEEE International Conference on Robotics and Automation (ICRA), pp. 9493–9500. IEEE, 2023a.

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, 2023b. 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, 2023c.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. arXiv preprint arXiv:2303.16434, 2023d.

Mark H. Liffiton, Brad E. Sheese, Jaromir Savelka, and Paul Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes. ArXiv, abs/2308.06921, 2023. URL `https://api.semanticscholar.org/CorpusID:260886891`.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In Text summarization branches out, pp. 74–81, 2004.

Jiawei Lin, Jiaqi Guo, Shizhao Sun, Weijiang Xu, Ting Liu, Jian-Guang Lou, and Dongmei Zhang. A parse-then-place approach for generating graphic layouts from textual descriptions, 2023.

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.

Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. "what it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, pp. 1–31, 2023c.

Xiao Liu, Hanyu Lai, Hao Yu, Yifan Xu, Aohan Zeng, Zhengxiao Du, P. Zhang, Yuxiao Dong, and Jie Tang. Webglm: Towards an efficient web-enhanced question answering system with human preferences. Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2023d. URL `https://api.semanticscholar.org/CorpusID:259144903`.

Xiao Liu, Hanyu Lai, Hao Yu, Yifan Xu, Aohan Zeng, Zhengxiao Du, Peng Zhang, Yuxiao Dong, and Jie Tang. WebGLM: Towards an efficient web-enhanced question answering system with human preferences. June 2023e.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Yuxian Gu, Hangliang Ding, Kai Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Shengqi Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. ArXiv, abs/2308.03688, 2023f. URL `https://api.semanticscholar.org/CorpusID:260682249`.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. arXiv preprint arXiv:2310.02170, 2023g.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. arXiv preprint arXiv:2304.09842, 2023.

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?, 2023a.

Yunsheng Ma, Can Cui, Xu Cao, Wenqian Ye, Peiran Liu, Juanwu Lu, Amr Abdelraouf, Rohit Gupta, Kyungtae Han, Aniket Bera, et al. Lampilot: An open benchmark dataset for autonomous driving with language model programs. arXiv preprint arXiv:2312.04372, 2023b.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners, 2022.

Jiageng Mao, Junjie Ye, Yuxi Qian, Marco Pavone, and Yue Wang. A language agent for autonomous driving. arXiv preprint arXiv:2311.10813, 2023.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented language models: a survey. ArXiv, abs/2302.07842, 2023. URL https://api.semanticscholar.org/CorpusID:256868474.

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.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: An embodied agent for code analysis with large language models. arXiv preprint arXiv:2310.18532, 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.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In International Conference on Machine Learning, pp. 26106–26128. PMLR, 2023.

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.

David Noever. Can large language models find and fix vulnerable software? arXiv preprint arXiv:2308.10345, 2023.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp. 311–318, 2002.

Aaron Parisi, Yao Zhao, and Noah Fiedel. TALM: Tool augmented language models. May 2022a.

Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. ArXiv, abs/2205.12255, 2022b. URL https://api.semanticscholar.org/CorpusID:249017698.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive APIs. May 2023.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. Check your facts and try again: Improving large language models with external knowledge and automated feedback. February 2023.

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

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. arXiv preprint arXiv:2307.07924, 2023a.

Cheng Qian, Chi Han, Yi Ren Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. 2023b. URL https://api.semanticscholar.org/CorpusID:258841653.

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

Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-Context retrieval-augmented language models. January 2023.

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.

Jasmine Roberts, Andrzej Banburski-Fahey, Microsoft Jaron, and Lanier Microsoft. Surreal vr pong: Llm approach to game design. URL https://api.semanticscholar.org/CorpusID:265045268.

Alex Robinson. Sketch2code: Generating a website from a paper mockup, 2019.

Jérémy Scheurer, Jon Ander Campos, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. Training language models with natural language feedback. arXiv preprint arXiv:2204.14146, 8, 2022.

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.

Hao Sha, Yao Mu, Yuxuan Jiang, Li Chen, Chenfeng Xu, Ping Luo, Shengbo Eben Li, Masayoshi Tomizuka, Wei Zhan, and Mingyu Ding. Languagempc: Large language models as decision makers for autonomous driving. arXiv preprint arXiv:2310.03026, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. arXiv preprint arXiv:2303.17580, 2023.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. Natural language to code translation with execution. arXiv preprint arXiv:2204.11454, 2022.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew J. Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. ArXiv, abs/2010.03768, 2020. URL https://api.semanticscholar.org/CorpusID:222208810.

Kurt Shuster, Jing Xu, Mojtaba Komeili, Da Ju, Eric Michael Smith, Stephen Roller, Megan Ung, Moya Chen, Kushal Arora, Joshua Lane, Morteza Behrooz, William Ngan, Spencer Poff, Naman Goyal, Arthur Szlam, Y-Lan Boureau, Melanie Kambadur, and Jason Weston. BlenderBot 3: a deployed conversational agent that continually learns to responsibly engage. August 2022.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models, 2022.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. RestGPT: Connecting large language models with real-world RESTful APIs. June 2023.

Davit Soselia, Khalid Saifullah, and Tianyi Zhou. Learning ui-to-code reverse generator using visual critic without rendering, 2023.

Abishek Sridhar, Robert Lo, Frank F. Xu, Hao Zhu, and Shuyan Zhou. Hierarchical prompting assists large language model on web navigation, 2023.

Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. Modular visual question answering via code generation. arXiv preprint arXiv:2306.05392, 2023.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3d-gpt: Procedural 3d modeling with large language models. arXiv preprint arXiv:2310.12945, 2023a.

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, 2023b.

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

Melanie Swan, Takashi Kido, Eric Roland, and Renato P. dos Santos. Math agents: Computational infrastructure, mathematical embedding, and genomics. ArXiv, abs/2307.02502, 2023. URL https://api.semanticscholar.org/CorpusID:259360593.

Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents. arXiv preprint arXiv:2306.03314, 2023.

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. June 2023.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, Yaguang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. LaMDA: Language models for dialog applications. January 2022.

Hung Quoc To, Minh Nguyen, and Nghi D. Q. Bui. Neural rankers for code generation via inter-cluster modeling. 2023. URL https://api.semanticscholar.org/CorpusID:265043836.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In Chi conference on human factors in computing systems extended abstracts, pp. 1–7, 2022.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. Microsoft Auton. Syst. Robot. Res, 2:20, 2023.

Boshi Wang, Sewon Min, Xiang Deng, Jiaming Shen, You Wu, Luke Zettlemoyer, and Huan Sun. Towards understanding chain-of-thought prompting: An empirical study of what matters, 2023a.

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, 2023b.

Huaxiaoyue Wang, Gonzalo Gonzalez-Pumariega, Yash Sharma, and Sanjiban Choudhury. Demo2code: From summarizing demonstrations to synthesizing code via extended chain-of-thought. arXiv preprint arXiv:2305.16744, 2023c.

Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. arXiv preprint arXiv:1711.07163, 2017.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. arXiv preprint arXiv:2308.11432, 2023d.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. arXiv preprint arXiv:2203.05132, 2022.

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

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. Leti: Learning to generate from textual interactions. arXiv preprint arXiv:2305.10314, 2023f.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. arXiv preprint arXiv:2309.10691, 2023g.

Wayve. LINGO-1: Exploring Natural Language for Autonomous Driving, September 2023. URL https://wayve.ai/thinking/lingo-natural-language-autonomous-driving/.

Colin Wei, Sang Michael Xie, and Tengyu Ma. Why do pretrained language models help in downstream tasks? an analysis of head and prompt tuning, 2022a.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. Natural language generation and understanding of big code for AI-assisted programming: A review. Entropy, 25(6):888, jun 2023. doi: 10.3390/e25060888. URL https://doi.org/10.3390%2Fe25060888.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. arXiv preprint arXiv:2303.04671, 2023a.

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, 2023b.

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

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, 2023.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, pp. 1–10, 2022.

Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. Mm-react: Prompting chatgpt for multimodal reasoning and action. arXiv preprint arXiv:2303.11381, 2023.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), Advances in Neural Information Processing Systems, volume 35, pp. 20744–20757. Curran Associates, Inc., 2022a. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/82ad13ec01f9fe44c01cb91814fd7b8c-Paper-Conference.pdf.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. ArXiv, abs/2207.01206, 2022b. URL https://api.semanticscholar.org/CorpusID:250264533.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. October 2022c.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. ArXiv, abs/2210.03629, 2022d. URL https://api.semanticscholar.org/CorpusID:252762395.

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.

Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Lumos: Towards language agents that are unified, modular, and open source. 2023.

Pengcheng Yin and Graham Neubig. Reranking for neural semantic parsing. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019.

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.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Ren Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. ArXiv, abs/2309.17428, 2023. URL https://api.semanticscholar.org/CorpusID:263310662.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet nl2code: A survey. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 7443–7464, 2023.

Lu Zeng, Sree Hari Krishnan Parthasarathi, and Dilek Hakkani-Tur. N-best hypotheses reranking for text-to-sql systems. In 2022 IEEE Spoken Language Technology Workshop (SLT), pp. 663–670. IEEE, 2023.

Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation, 2021.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. arXiv preprint arXiv:2305.04087, 2023a.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. Coder reviewer reranking for code generation. In International Conference on Machine Learning, pp. 41832–41846. PMLR, 2023b.

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

Hongyi Zheng and Abulhair Saparov. Noisy exemplars make large language models more robust: A domain-agnostic behavioral analysis, 2023.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. arXiv preprint arXiv:1709.00103, 2017.

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, 2023a.

Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. Llm as dba. arXiv preprint arXiv:2308.05481, 2023b.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyuan Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Y. Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. ArXiv, abs/2305.17144, 2023. URL `https://api.semanticscholar.org/CorpusID:258959262`.

Terry Yue Zhuo. Large language models are state-of-the-art evaluators of code generation. arXiv preprint arXiv:2304.14317, 2023.

A Preliminaries

A.1 Our Definition of Code

We consider code as any formal language that is both machine-executable and human-interpretable. For instance, human-readable programming languages fall within the scope of our discussion, whereas low-level languages, such as machine language based on binary instructions, are excluded due to their lack of human interpretability. Additionally, pre-defined formal languages, such as function sets employed in WebGPT (Nakano et al., 2021), are included as they can be parsed and executed in a rule-based manner.

LLMs trained with expressions formulated within a defined set of symbols and rules (e.g., pre-defined function sets, mathematical deduction formula, etc.), i.e., formal languages, exhibit advantages akin to those trained with programming languages. Therefore, we expand our definition of code to incorporate these homogeneous training corpora, enhancing the comprehensiveness of this survey to align with current research needs.

A.2 LLM Code Training Methods

LLMs undergo code training by following the standard language modeling objective, applied to code corpora. Given that code possesses natural language-like sequential readability, this parallels the approach to instruct LLMs in understanding and generating free-form natural language. Specifically, for an LLM M_Θ with parameters Θ and a code corpus $T = \{t_1, ..., t_n\}$, the language modeling loss for optimization is:

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

When employing programming language (e.g., Python, C, etc.) as the corpus (Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022), training data is typically sourced from publicly accessible code repositories, such as GitHub. This process yields a corpus with a volume comparable to that of natural language pre-training, and thus we call training with such an abundance of code as *code pre-training*. The training strategy entails either training code on a pre-trained natural language LLM, as exemplified by Codex (Chen et al., 2021), or training a LLM from scratch with a blend of natural language and code corpora, as demonstrated by CodeLLM (Ma et al., 2023a).

Conversely, when utilizing other pre-defined formal language for training, the objective shifts to acquainting the model with the application of specific functions (Schick et al., 2023), mathematical proof formulas (Wu et al., 2022), SQL (Sun et al., 2023b), and similar constructs. As the dataset for this is smaller compared to the pre-trained natural language corpus, we refer to such training process as *code fine-tuning*. Researchers apply the language modeling loss to optimize LLMs during this process, similarly.

```
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      1. Object-Oriented Programming
        self.toolbox = toobox                           Adv: Structured

    def cs_rookie_ritual(self):      2. Functional Programming
                                         Adv: Modular & Explicit
        """
        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:                                            3. Procedural Programming
            action(tool)                                                          Adv: Step-by-Step
```

Figure 6: 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.

B Discussions

B.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 6, we provide a case study comparing code and 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 §3, the improvement for more controlled planning (cf. planning part in §5.1), and also helped with action execution (§5.2).

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. §2.3 and the perception part in §5.1). 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 §2.2. What's more, the execution often includes feedback mechanisms, which can be valuable for further refining the generator (§4 and §5.3).

B.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.

B.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 §5, 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., 2023b) 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., 2023b; Lei & Deng, 2023). Leveraging this understanding, we aim to explore novel research avenues in LLM reasoning inspired by the utilization of "code".

C Challenges

In our survey, we identify several intriguing and promising avenues for future research.

C.1 The Causality between Code Pre-training and LLMs' Reasoning Enhancement

Although we have categorized the most pertinent work in §2.2, a noticeable gap persists in providing explicit experimental evidence that directly indicates the enhancement of LLMs' reasoning abilities through the acquisition of specific code properties. While we intuitively acknowledge that certain code properties likely contribute to LLMs' reasoning capabilities, the precise extent of their influence on enhancing reasoning skills remains ambiguous. In the future research endeavors, it is important to investigate whether reinforcing these code properties within training data could indeed augment the reasoning capabilities of trained LLMs. If it is indeed the case, that pre-training on specific properties of code directly improves LLMs' reasoning abilities, understanding this phenomenon will be key to further improving the complex reasoning capabilities of current models.

C.2 Acquisition of Reasoning Beyond Code

Despite the enhancement in reasoning achieved by pre-training on code, foundational models still lack the human-like reasoning abilities expected from a truly generalized artificial intelligence. Importantly, beyond code, a wealth of other textual data sources holds the potential to bolster LLM reasoning abilities, where the intrinsic characteristics of code, such as its lack of ambiguity, executability, and logical sequential structure, offer guiding principles for the collection or creation of these datasets. However, if we stick to the paradigm of training language models on large corpora with the language modeling objective, it's hard to envision a sequentially readable language that is more abstract, highly structured, and closely aligned with symbolic language than formal languages, exemplified by code, which are prevalent in a substantial digital context. We envision that exploring alternative data modalities, diverse training objectives, and novel architectures would present additional opportunities to further enhance the reasoning capabilities of these models.

C.3 Challenges of Applying Code-centric Paradigm

The primary challenge in LLMs using code to connect to different function ends is learning the correct invocation of numerous functions, including selecting the right function end and passing the correct parameters at an appropriate time. Even for simple tasks like simplified web navigation with a limited set of action primitives like mouse movements, clicks, and page scrolls, few shot examples together with a strong underlying LLM are often required for the LLM to precisely grasp the usage of these primitives (Sridhar et al., 2023). For more complex tasks in data-intensive fields like chemistry (Bran et al., 2023), biology, and astronomy, which involve domain-specific Python libraries with diverse functions and intricate calls, enhancing LLMs' capability of learning the correct invocation of these functions is a prospective direction, empowering LLMs to act as IAs performing expert-level tasks in fine-grained domains.

C.4 Learning from multi-turn interactions and feedback

LLMs often require multiple interactions with the user and the environment, continuously correcting themselves to improve intricate task completion (Li et al., 2023c). While code execution offers reliable and customizable feedback, a perfect method to fully leverage this feedback has yet to be established. As discussed in § 4.2, we observed that selection-based methods, though useful, do not guarantee improved performance and can be inefficient. Prompting-based methods heavily depend on the in-context learning abilities of the LLM, which might limit their applicability. Fine-tuning methods show consistent improvement, but data collection and fine-tuning are resource-intensive and thus prohibitive. We hypothesize that reinforcement learning could be a more effective approach for utilizing feedback and improving LLMs. This method can potentially address the limitations of current techniques by providing a dynamic way to adapt to feedback through well-designed reward functions. However, significant research is still needed to understand how reward functions should be designed and how reinforcement learning can be optimally integrated with LLMs for complex task completion.

D The Comprehensive Paper List

To complement the core paper list presented in Figure 2, we have included a comprehensive list of papers in Figure 8. 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.

Table 1: Representative work connecting LLMs to different function ends for performing non-trivial tasks. Initial efforts embed tool calls rigidly within the LLMs' inference mechanism (indicated by "$*$"), resulting in diminished flexibility and constrained tool accessibility. More recently, the *code-centric paradigm* establishes connections between LLMs and function ends through programming languages or pre-defined functions (indicated by "$\dagger$"). This approach enhances the scalability of LLMs' function end invocation across diverse tools and execution modules.

Major Type of Function Ends	Representative Work	Connecting Paradigm	Learning Method	Objectives or Problems to Solve
Single Tool	Retriever in REALM Guu et al. (2020)	Hardcoded in Inference Mechanism*	Example Fine-tuning	Augment LLMs with Tools
	Verifier in GSM8K Cobbe et al. (2021)	Hardcoded in Inference Mechanism*	Example Fine-tuning	
Limited Text-based Tools	Blenderbot3 Shuster et al. (2022)	Hardcoded in Inference Mechanism*	Example Fine-tuning	Open-domain Conversation
	LamDA Thoppilan et al. (2022)	Generate Pre-defined Functions†	Example Fine-tuning	
Text-based Tools	TALM Parisi et al. (2022b)	Generate Pre-defined Functions†	Iterative Self-play	Efficient and Generalizable Tool Using
	ToolFormer Schick et al. (2023)	Generate Pre-defined Functions†	Self-supervised Training	
Multi-modal Modules	MM-React Yang et al. (2023)	Generate Pre-defined Functions†	Zero-shot Prompting	Multi-modal Reasoning Tasks
	CodeVQA Subramanian et al. (2023)	Generate Python Functions†	Zero-shot & Few shot	
	VISPROG Gupta & Kembhavi (2023)	Generate Python Functions†	Zero-shot Prompting	
	ViperGPT Surís et al. (2023)	Generate Python Functions†	Zero-shot Prompting	
Real-World APIs	Code as Policies Liang et al. (2023a)	Generate Python Functions†	Few-shot Prompting	Better Robot Control
	Progprompt Singh et al. (2022)	Generate Python Functions†	Zero-shot Prompting	
	SayCan Ahn et al. (2022)	Generate Pre-defined Functions†	Zero-shot Prompting	
	RRR Cui et al. (2023a)	Generate Pre-defined Functions†	Zero-shot Prompting	Autonomous Driving Ecosystems
	Agent-Driver Mao et al. (2023)	Generate Pre-defined Functions†	Few-shot Prompting	
	LaMPilot Ma et al. (2023b)	Generate Python Functions†	Zero-shot & Few-shot	

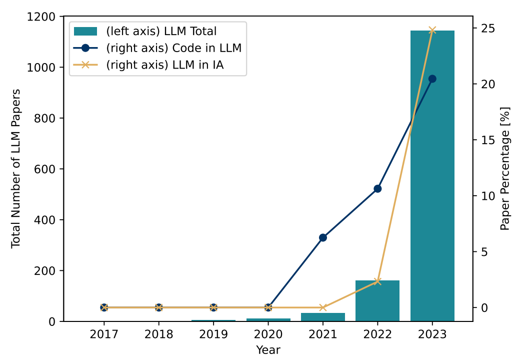


Figure 7: Paper statistics from Arxiv. We identified a significant and growing trend in recent research focused on code-based large language models (LLMs) and LLM-based intelligent agents (IAs). Code usage contributes much to the success of these cutting-edge models and systems.

E 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 7, intending to provide insights into the trends and relationships between LLMs, code-related topics, and IAs in the past few years.

F Benchmarks for Evaluating Complex Reasoning with Code:

While there exist many benchmarks used to evaluate the abilities of LLMs (Liang et al., 2023c) 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, 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

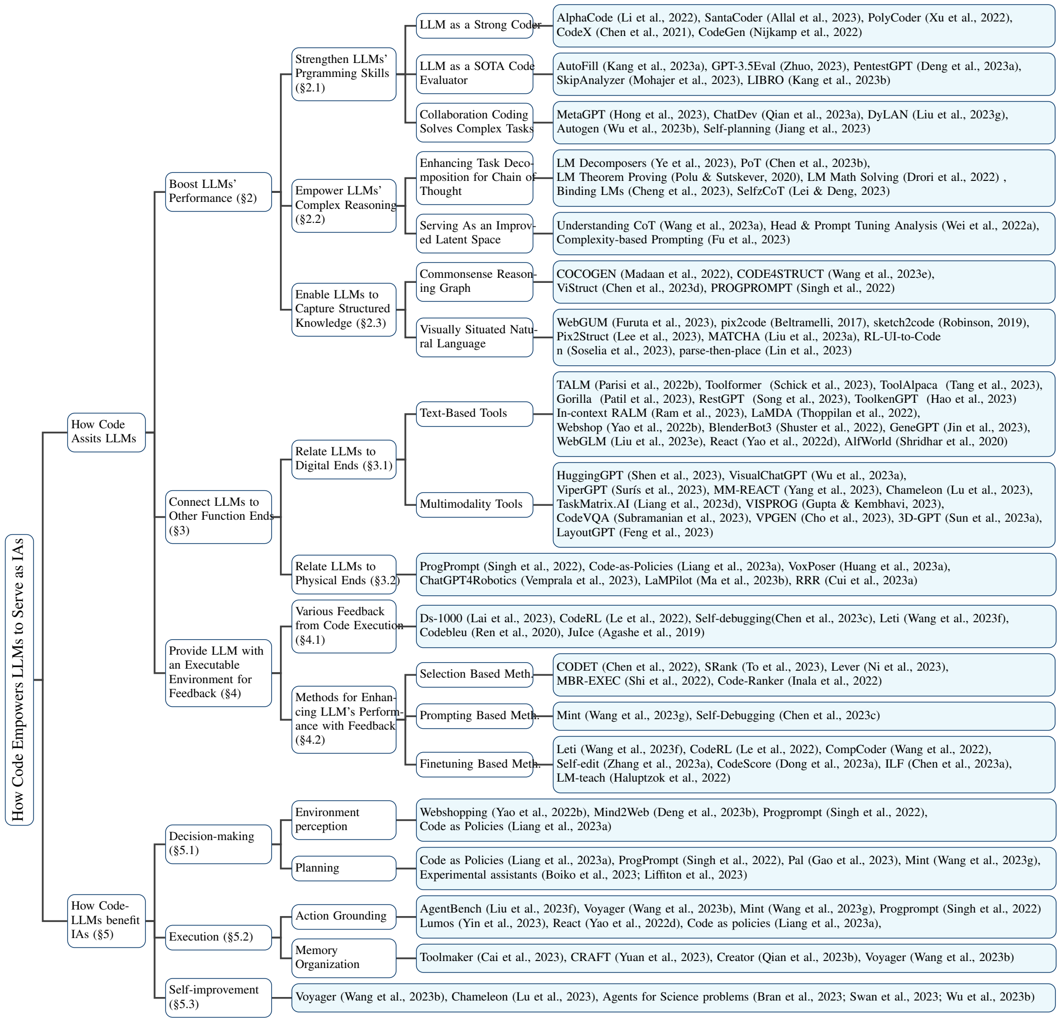


Figure 8: We hereby provide a complete list of the papers included in our survey.

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).

G 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 2. 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.

Major Functionalities Facilitated	Key Features of Code
Strengthen LLMs' Programming Skills Correspond to §2.1 and Figure 3 (a)	**Machine Executable:** Pretraining with Code, the LLM is able to write code, evaluate code, and utilize collaborative coding to solve complex tasks.
Empower LLMs' Complex Reasoning Correspond to §2.2 and Figure 3 (b)	**Structured and Expressive:** The step-by-step nature of code benefits CoT. **Machine Executable:** LLMs can utilize code to help with certain capabilities like mathematical reasoning.
Enhance LLMs' Structured Knowledge Correspond to §2.3 and Figure 3 (c)	**Structured and Expressive:** Code can be used to express complex structures, some programming language features like logical expressions and class inheritance will be especially useful.
Connect LLMs to Other Functional Ends Correspond to §3 and Figure 4a	**Explicit and Unambiguous**: Code is more explicit and clear than natural language, thus can better express clear instructions of connecting to any other functional ends.
Provide LLMs w/ Environmental Feedback Correspond to §4 and Figure 4b	**Machine Executable**: Code execution result can be treated as feedback to finetune the LLMs further and make their performance more desirable
Help with IAs' Decision-Making Correspond to §5.1 and Figure 5	**Structured and Expressive**: Code pretraining Enhances agents' ability to precept structural knowledge, step by step feature improves CoT planning, logical expressions and nesting help with better control flow. **Machine Executable**: Help with solving mathematical tasks.
Help with IAs' Action Execution Correspond to §5.2 and Figure 5	**Explicit and Unambiguous:** Unambiguous function calling help with action grounding. **Machine Executable:** Agents can write reusable code snippets as its memory
Help with IAs' Self-improving Correspond to §5.3 and Figure 5	**Machine Executable:** Agent code execution results reflect potential environment change and can be utilized for self-improving

Table 2: We conclude the three major key features of code and correspond them to the major functionalities of LLMs and IAs they facilitated. The three key features are namely **Structured and Expressive**, **Machine Executable** and **Explicit and Unambiguous**. More details of how these features assist LLMs and IAs can be found in the preamble of each section.