# Chapter 2—Literature Review

## 2.1 Introduction

Code assistance encompasses a broad range of tools, techniques, and methodologies aimed at supporting developers through the code development. As programming challenges become more intricate, these assistants significantly boost developer efficiency, minimize mistakes, and streamline the coding process. Such support may appear in multiple forms, including automated code suggestions, error identification and resolution, code generation, documentation, and context-specific recommendations. In this field, language models have become essential, enabling developers to access informed hints, produce code segments, and generally improve their coding expertise (Soliman, 2024).

In a recent study (Coutinho et al., 2024), the authors explore how generative AI tools affect productivity in real-world software development settings. By distributing licenses for different generative AI solutions (e.g., ChatGPT, GitHub Copilot) to professionals working in different roles (developers, designers, data scientists, QA specialists, and coaches), the authors gather qualitative insights on how these tools fit into day-to-day activities. Participants generally report positive impact on their perceived productivity, particularly through time savings, efficient artifact generation, and quick access to information. However, respondents also face challenges such as ensuring reliability, refining outputs, and addressing security concerns when handling sensitive data. While the study is limited, it sets the groundwork for more comprehensive future research. Its findings suggest that generative AI can enhance workflow efficiency and knowledge acquisition, but further empirical studies are needed to fully understand and quantify the productivity gains (Coutinho et al., 2024).

The study in (Martinović & Rozić, 2024) investigates how developers perceive the influence of AI-based tools on the quality of produced code. The authors present findings drawn from a survey targeting developers in various tech companies, exploring their experiences and satisfaction levels with AI-driven coding assistants. They focus particularly on metrics such as code readability, maintainability, efficiency, and accuracy, examining whether AI support can enhance these code quality dimensions. Their results highlight that developers, for the most part, recognize a positive impact on their productivity and overall coding experience, though improvements in certain quality aspects remain uneven. More than three-quarters of developers stated that their adoption of AI tools improved their day-to-day development work. While respondents reported noticeable gains in maintainability and efficiency, perceptions of improved readability and accuracy were more modest (Martinović & Rozić, 2024).

Additionally, the paper compares users who frequently rely on AI with those who have chosen not to adopt these tools. Non-users cite concerns about affordability, trust, and clarity of the potential benefits as key reasons for their hesitation. As AI capabilities become more refined—offering consistent code improvements, stronger accuracy, and better integration into established workflows—more developers may embrace these solutions.

Another interesting study was conducted in (Ciniselli et al., 2024) where the authors envision how AI-driven assistance will reshape software developers’ daily work by the year 2030. They compare current AI-based coding practices - exemplified by tools like GitHub Copilot and ChatGPT - with a future scenario in which developers rely on a hypothetical augmented tool, “HyperAssistant,” for end-to-end support. The proposed future assistant addresses a broad set of needs: it proactively manages developers’ mental well-being, detects and fixes complex faults, streamlines code optimization and reuse, facilitates dynamic team collaboration, and offers personalized learning and skill development resources. By examining this transition from human-led coding toward orchestrating AI-driven ecosystems, the authors highlight how developers’ roles may evolve, ultimately leading to more efficient, secure, and sustainable software engineering processes.

## 2.2 Automatic Code Generation Before Generative Pre-Trained Transformers (GPT)

Recently, LLMs have revolutionized automated code generation by effectively translating natural language intent into code. Early methods relied on Recurrent Neural Networks (RNNs) or syntax-driven approaches, which struggled with complex, long-range dependencies. The introduction of transformers, originally developed for natural language processing (NLP), significantly improved performance. By integrating encoder models like BERT or RoBERTa with Marian decoders, researchers achieved state-of-the-art results on benchmarks such as Code/Natural Language Challenge (CoNaLa) and DJANGO dataset (Oda et. al., 2015). These hybrid models enhance syntax, semantics, and developer productivity through intelligent autocompletion, context-aware suggestions, and inline documentation. Additionally, built-in linting, formatting, and error-checking further streamline development. Overall, LLMs are dramatically boosting the accuracy and efficiency of modern code generation (Soliman, 2024).

CodeBERT, a transformer-based model, exemplifies the potential of pre-trained models in integrating natural language (NL) and programming language (PL) tasks. By training on paired NL-PL data (e.g., code snippets and documentation) and standalone code, it employs masked language modeling and replaced token detection to capture semantic correspondences between NL descriptions and code functionality. CodeBERT excels in tasks like code search and documentation generation, producing accurate and informative outputs by leveraging its joint understanding of NL and PL. Its ability to generalize across multiple programming languages, including those unseen during training, highlights the promise of combining bimodal pre-training objectives with large-scale NL-PL resources. This paradigm sets a foundation for advancing code-related models with structural insights, advanced reasoning, and domain-specific customizations (Feng, Z. 2020).

In another study (Defferrard et al., 2024), the authors explore how to build and refine code generation models entirely from scratch, without relying on human-created code corpora. The authors develop a self-improvement approach that combines a neural language model with a search-based procedure, following an “expert iteration” paradigm. In this setup, search methods (such as Monte Carlo Tree Search or sampling-and-filtering approaches) discover programs that solve given programming problems, and these newly found solutions are used as training data to improve the language model. As the model becomes better at coding tasks, the search becomes more efficient at finding higher-quality solutions, enabling the model to tackle even more challenging problems. The study systematically examines how factors like search budget, problem complexity, and the relative allocation of computation to search versus training affect the learning process. Results show that even small, randomly initialized language models can gradually internalize programming competencies through this iterative search-and-learn framework, advancing their code generation abilities without human-written examples.

A GPT-3.5-powered IntelliJ IDEA plugin is introduced in (Almeida Y., 2024) designed to automate code reviews. AICodeReview identifies syntax errors, logic flaws, and vulnerabilities while offering actionable improvement suggestions with detailed explanations. The tool supports multiple programming languages, customizable suggestions, and integration with JetBrains products. A preliminary evaluation showed that AICodeReview significantly outperformed manual reviews, reducing review time (15.2 minutes vs. 22.5 minutes), detecting more code smells (28 vs. 20), and improving refactoring outcomes (25 vs. 13). This work emphasizes the effectiveness of LLMs in streamlining software development processes.

The use of pre-trained language models for Python code generation is explored in (Ottens et al., 2024), focusing on completing function bodies given function signatures and docstrings. Using the CodeSearchNet dataset, the authors compare baseline models, including a sequence-to-sequence RNN and character-level embeddings, against a fine-tuned GPT-2 model. The latter demonstrates superior performance, achieving a Bilingual Evaluation Understudy (BLEU) score of 0.22—a 46% improvement over the baseline. The study highlights GPT-2's ability to adapt to programming languages by leveraging transfer learning techniques originally designed for natural language processing. Generated code is both novel and syntactically correct, showcasing understanding of Python structures. This work emphasizes the potential of LLMs in automating software development tasks while improving efficiency and quality.

A comprehensive review of deep learning (DL) applications in source code modeling and generation is offered in (Le et al., 2020). The authors analyze the evolution of DL techniques, particularly in Natural Language Processing (NLP), and their adaptation to programming tasks such as source code generation, bug detection, and program synthesis. They present a framework for understanding common program learning tasks using encoder-decoder architectures, emphasizing their strengths in capturing both syntactic and semantic structures of code.

The review categorizes Big Code tasks under the encoder-decoder framework, exploring advancements in attention mechanisms, memory-augmented networks, and open-vocabulary models that address challenges like handling large code vocabularies and maintaining syntactic correctness.

DocPrompting is proposed in (Zhou et al., 2023), a method that enhances automatic code generation by incorporating code documentation into the process, addressing the limitations of models that struggle with unfamiliar or newly introduced libraries and functions. Mimicking the human practice of consulting documentation, DocPrompting retrieves relevant documentation snippets based on a natural language (NL) intent and combines them with the NL input to generate accurate code. By enabling models to adapt to evolving programming environments, DocPrompting represents a significant step forward in enhancing the adaptability and functionality of code generation systems.

SKCODER is introduced in (Li et al., 2023), a similar approach to automatic code generation that emulates human developers' practice of reusing code. Rather than simply copying similar code snippets, SKCODER extracts a high-level code sketch from retrieved snippets that align with natural language (NL) requirements and refines this sketch into a complete solution. The system consists of three components: a retriever to locate relevant code, a sketcher to create a structured skeleton, and an editor to adapt the sketch to the desired task. Experiments on multiple datasets, including a new large-scale Java dataset, show that SKCODER outperforms models like CodeT5 and REDCODER in accuracy and quality metrics. Its sketch-based method generalizes well across architectures, producing more precise, maintainable, and functional code compared to copy-based or purely generative models, advancing automated code generation toward human-like code reuse.

## 2.3 LLMs

### 2.3.1 LLMs: Overview

A comprehensive survey of Large Language Models (LLMs) is provided in (Minae et al., 2012), how LLMs have evolved to revolutionize NLP and AI at large, while acknowledging existing limitations and pointing towards the active research needed to address scalability, efficiency, reliability, and broader applicability. The study describes underlying technologies and popular model families, such as encoder-only (BERT, RoBERTa, etc.), decoder-only (GPT), and encoder-decoder transformers (T5, BART), GPT, LLaMA, and PaLM families of models, other representative LLMs like FLAN, LaMDA, BLOOM, Orca, StarCoder, Gemini, etc. These models and frameworks focus on various aspects such as efficient training, multilingual support, multimodal inputs, retrieval augmentation, and improved reasoning.

The survey then describes various techniques used to develop and augment LLMs, such as positional embeddings, mixture-of-experts, subword-based tokenization, and the methods, datasets and benchmarks for training and evaluation including BLEU, ROUGE, Pass@k, Stanford Question Answering Dataset (SQuAD), Massive Multitask Language Understanding (MMLU), HumanEval, etc. LLM pre-training objectives include masked language modeling (MLM), next sentence prediction (NSP), causal language modeling (CLM), and fine-tuning and alignment techniques include supervised fine-tuning (SFT), instruction tuning (e.g., InstructGPT), Reinforcement Learning from Human Feedback (RLHF), Direct Preference Optimization (DPO), Kahneman-Tversky Optimization (KTO).

The survey covers various prompt design and engineering techniques, such as chain-of-thought (CoT), tree-of-thought (ToT), Reflection, Expert Prompting, automatic prompt engineering (APE), and numerous tools for augmentation with external knowledge, including Retrieval-Augmented Generation (RAG) and LLM-based agents (integrating external tools, reasoning, and decision-making). Proposed efficiency and adaptation include, among others, parameter-efficient fine-tuning (PEFT), low-rank adaptation (LoRA), knowledge distillation, quantization, etc.

Future directions and open challenges listed in the paper include:

1. exploration of new model architectures beyond attention,
2. handling multi-modality (text, image, audio, video),
3. enhancing reliability and reducing hallucinations and, what is really important in the context of this Praxis,
4. development of smaller, more efficient models with similar capabilities as LLMs.

Other attempts at conducting surveys of LLMs in order to describe them based on various aspects are discussed in (Zhao et al., 2023) and (Naveeda et al., 2024). In continuation of the topic, (Han et al., 2024) provides a comprehensive survey of parameter-efficient fine-tuning methodologies for LLMs. Another technique that is very widely used with LLMs is retrieval-augmented generation (RAG). RAG involves using a trained retriever to fetch relevant structured information and feed it into the LLM’s prompt, thereby reducing hallucination and improving the quality and trustworthiness of the generated structured output (Béchard & Ayala, 2024). Corrective retrieval-augmented generation (CRAG) is a retrieval-augmented framework that adaptively evaluates and corrects the retrieval process, leveraging large-scale web searches and selective re-composition of documents to ensure robust and improved output quality from LLMs (Yan et al., 2024). (Huang & Huang, 2024), as well as (Hu & Lu, 2024.) conduct a survey on RAG systems: the former focuses on organizing the RAG process into four key phases - pre-retrieval, retrieval, post-retrieval, and generation - to offer a detailed, retrieval-oriented perspective on their operation and development and the latter discusses key components of the Retrieval-Augmented Language Model (RALM) (retrievers, language models, and augmentation methods), how these components interact, their evolution over time, as well as evaluation methods.

### 2.3.2. LLMs for Code Generation

A Survey on LLMs for Code Generation (Jiang et al., 2024.) offers a comprehensive review of the progress and capabilities of LLMs dedicated to code generation. It addresses the current gap in literature by systematically examining the complete lifecycle of LLMs for code-related tasks, from data preparation through advanced training and evaluation methodologies. The authors begin by outlining a taxonomy to structure recent developments, including how large-scale code datasets are curated and processed, how models are pre-trained and fine-tuned using diverse strategies, and what role instruction tuning, prompt engineering, and retrieval-augmented methods play. Special attention is given to novel frameworks that enable repository-level understanding, autonomous coding agents, and feedback-driven reinforcement learning to improve code correctness and adapt to real-world coding challenges.

The survey also discusses evaluation practices, including standard benchmarks like Human Evaluation Dataset (HumanEval) and Mostly Basic Python Problems (MBPP), newly proposed metrics, and human or LLM-based assessments, highlighting the complexity of assessing code quality, correctness, and broader software engineering attributes. This survey serves as a key reference for researchers and practitioners seeking a thorough understanding of the state-of-the-art in LLMs for code generation, pinpointing opportunities for future advancements and offering a structured roadmap for progressing toward more reliable, adaptable, and context-aware AI-driven coding assistants.

A less recent article on code generation (Wodecki B. 2023) surveys the emerging landscape of text-to-code generative AI models, outlining how these systems are poised to reshape software development by converting natural language instructions into executable code. Several prominent models and tools are highlighted, including StarCoder, Codex, Copilot, Code Interpreter, CodeT5, Polycoder, and Replit’s Ghostwriter. Each system has distinct origins, technical foundations, and capabilities. For instance, StarCoder, a collaborative effort between ServiceNow and Hugging Face, is trained on a broad array of code and demonstrates notable performance on standard benchmarks. Codex, from OpenAI, underpins GitHub Copilot, enabling developers to use plain English prompts to produce code snippets in multiple programming languages. Code Interpreter, also from OpenAI, extends ChatGPT’s functionality to execute code, aiding tasks like data analysis within the chatbot interface.

CodeT5 (Salesforce) and Polycoder (Carnegie Mellon University) represent research-driven efforts to enhance code understanding and generation, focusing on tasks like defect detection and code completion, while Polycoder also emphasizes openness and surpasses Codex in some niche cases. Commercial offerings such as Replit’s Ghostwriter and Tabine integrate into existing development workflows, providing autocomplete, code translation, and conversational interfaces that assist developers in real time.

AI-assisted code generation tools, such as GitHub Copilot, Amazon CodeWhisperer, and OpenAI’s ChatGPT, are increasingly used to produce code from natural language prompts. The study in (Yetiştiren et al., 2023) compares their performance using the HumanEval Dataset and evaluates the generated code on metrics like code validity, correctness, security, reliability, and maintainability. ChatGPT, GitHub Copilot, and Amazon CodeWhisperer produce correct code 65.2%, 46.3%, and 31.1% of the time, respectively, with newer versions of GitHub Copilot and Amazon CodeWhisperer showing 18% and 7% improvements. Average technical debt from code smells is 8.9 minutes for ChatGPT, 9.1 minutes for GitHub Copilot, and 5.6 minutes for Amazon CodeWhisperer. These findings highlight the tools’ strengths and limitations and can guide practitioners in choosing the best generator for their specific development needs.

This study in (Reeves et al., 2023) examines the capability of an LLM-based code generation model (OpenAI Codex) to solve Parsons problems, a type of exercise where learners must reorder given code fragments into a correct solution. While previous work has shown these models can outperform many students in traditional code-writing tasks, the results here indicate a significantly lower success rate on Parsons problems—Codex correctly rearranges the code about half the time, and even small prompt changes influence outcomes. The model struggles particularly with problems that include extra, unused lines of code, but it rarely alters or adds lines on its own. These findings suggest that, unlike free-form coding tasks, Parsons problems may resist easy solutions from code generation tools, potentially offering educators an alternative assessment format less susceptible to student over-reliance on AI assistance.

## 2.4 SLMs

### 2.4.1 SLMs: Overview

A structured overview of small language models (SLMs) and how they can be developed and optimized to operate efficiently under various constraints is provided in (Nguyen et al., 2024). The authors outline SLM model architectures designed for compactness, discuss training techniques that maintain performance while reducing computational demand, and review model compression methods such as pruning, quantization, and knowledge distillation. They introduce a taxonomy that classifies methods based on stages (e.g., pre-processing, training, post-processing) and on the optimization goals they address (e.g., memory use, inference speed, or resource limitations).

Additionally, the survey enumerates common datasets and evaluation metrics tailored to SLM scenarios, emphasizing the importance of measuring factors like latency, memory footprint, privacy, and energy efficiency. They also explore practical use cases, such as deploying SLMs on edge devices or in real-time interactive settings, and they discuss open research challenges related to model biases, hallucinations, and privacy protection.

Other papers on SLMs: (Wang et al., 2024), (Lee 2024), (Ghosh 2023), (Mok 2023), (Szczygło 2024) (Morris et al., 2024), (Kili Technology Guide, 2024), (Abbas 2024) repeat the same very important advantages of SLMs as contrasted with LLMs. According to all of these studies, SLMs have emerged as a compelling alternative to LLMs, addressing the key challenges posed by LLMs’ massive size and resource requirements. While LLMs like GPT-4 and LLaMA have demonstrated impressive capabilities in general-purpose tasks, their computational demands raise issues related to cost, scalability, privacy, and real-time inference on edge devices. Furthermore, their broad focus often leads to insufficient domain specialization and suboptimal performance in fields like healthcare or law.

In contrast, SLMs—models with significantly fewer parameters—deliver a range of advantages. They are cheaper to run, simpler to integrate, and can be efficiently deployed on a variety of devices, including local and edge systems, thereby ensuring data confidentiality. Their smaller size also makes them easier to fine-tune for specific domains, often improving accuracy and responsiveness in specialized applications while reducing reliance on large-scale cloud infrastructures. Recent research has shown that, despite their reduced complexity, SLMs can match or even surpass the performance of larger models in certain tasks, particularly when combined with techniques such as knowledge distillation, pruning, quantization, parameter-efficient fine-tuning (e.g., LoRA), or retrieval-augmentation strategies. These approaches enable SLMs to maintain strong performance using fewer parameters, less training data, and less computing power, making them ideal for domain-specific use cases.

Although SLMs may have slightly narrower capabilities than their larger counterparts, their leaner nature results in faster processing, lower latency, and improved cost-effectiveness. They are also more agile in addressing evolving business needs, enabling organizations to rapidly iterate and align models with changing requirements. SLMs’ smaller computational footprint and simplified architectures reduce operational expenses and environmental impact, while the ability to run models locally enhances data privacy and security.

As a result, SLMs are increasingly viewed as a practical and accessible path forward for businesses and research teams. Instead of committing to the resource-heavy and often costly route of massive general-purpose models, organizations can adopt SLMs that are carefully tailored, continuously evaluated, and regularly updated to meet their unique demands. By striking the right balance between size, performance, and flexibility, SLMs stand poised to drive the next wave of AI innovation, democratizing language-based intelligence and making high-quality NLP technology more broadly attainable.

A study in (Fatima 2024) examines the increasing prominence of small language models (SLMs) in the 2024 AI landscape, focusing on five notable examples: Meta’s Llama 3, Microsoft’s Phi 3, Mistral AI’s Mixtral 8x7B, Google’s Gemma, and Apple’s OpenELM family. These SLMs offer advanced linguistic capabilities through more lightweight architectures and refined training techniques such as transfer learning, knowledge distillation, and sparse mixtures of experts. The result is an efficient, cost-effective class of models that can be integrated into a wider range of devices and applications, encouraging customization, on-device processing, and domain-specific fine-tuning. An example of using an SLM to replace an LLM is discussed in ([Murallie](https://thuwarakesh.medium.com/?source=post_page---byline--7ce1e5619f3d--------------------------------) T. 2024).

Looking ahead, SLM research points toward hybrid approaches that combine small and large models, improved architectures to streamline computation, and on-device training methods that support dynamic adaptation. This reorientation toward efficiency, sustainability, and accessibility positions SLMs as a crucial avenue for advancing NLP while respecting environmental and computational constraints (Abbas 2024).

### 2.4.2. SLMs for Code Generation

According to (Chen & Varoquaux, 2024), in the rapidly evolving landscape of artificial intelligence, the relationship between LLMs and SLMs is becoming increasingly nuanced, particularly in the domain of code generation. While LLMs have demonstrated remarkable capabilities in producing complex code snippets, they come with significant computational overhead, making them challenging to deploy in resource-constrained environments. Small models have emerged as a compelling alternative, offering a more efficient approach to code generation by leveraging techniques like data curation, prompt optimization, and domain-specific fine-tuning.

According to (Sun et al. 2024), rather than relying on brute-force scaling, recent work distills the LLM’s internal “solution plans” - obtained through techniques like backward reasoning - into smaller models. By training these models to generate both the reasoning steps and the final code, researchers have demonstrated substantial performance gains on challenging benchmarks, even surpassing standard fine-tuning methods. This equips smaller models with the underlying reasoning patterns of LLMs to improve their code generation quality and efficiency without the burdens of large-scale deployment.

A promising direction involves treating problem decomposition and solution derivation as distinct capabilities, handled by separate models. For instance, DaSLaM is a framework that splits the reasoning process into two specialized modules: a smaller, fine-tuned model dedicated to decomposing a complex problem into simpler subproblems, and a larger solver model that answers these subproblems and ultimately the original question. This modular setup is solver-agnostic, meaning the decomposition model is not tailored to any one solver and can work with a variety of large models or tools. Evaluations have demonstrated that such a division of labor can substantially boost performance on complex reasoning tasks (Juneja, G., 2024).

A training-free framework, called Agents Help Agents (AHA), for transferring knowledge from LLMs to smaller, locally run SLMs in the domain of data science code generation is introduced in (Anonymous authors, 2024). Rather than using traditional fine-tuning, AHA relies on in-context learning and a staged orchestration process. First, an LLM serves as a “Teacher Agent,” guiding an SLM “Student Agent” through a problem-solving interface. By exploring code generation tasks and refining problem-solving strategies, AHA’s orchestration system collects successful examples into a memory database. During inference, this memory is mined to produce both general-purpose and query-specific instructions that help the SLM generate accurate code without extensive retraining. Evaluations show that AHA’s approach significantly improves SLM performance.

The authors of (Williams 2024) explore the growing popularity of locally hosted language models and SLMs for coding tasks, highlighting their privacy advantages, cost savings, and customization potential compared to cloud-based solutions. These models are optimized for speed and lower hardware requirements. Their ongoing improvements and the involvement of major players like Apple and Meta hint at a future with more accessible, efficient local coding models. The study covers ways to evaluate these models, lists several top contenders, and explains how each caters to different needs. While these models may not yet match the raw power of big tech offerings, they provide developers with control, privacy, and flexibility.

The study identifies the following models as great candidates for this task: Apple’s OpenELM Family (set of small language models for mobile and local deployment), DeepSeek V2.5, Qwen2.5-Coder-32B-Instruct (by Alibaba), Nxcode-CQ-7B-orpo (fine-tuned Qwen model optimized for simpler coding tasks), OpenCodeInterpreter-DS-33B, Artigenz-Coder-DS-6.7B. Benchmarks and evaluation tools discussed include HumanEval, MBPP, BigCodeBench, LiveCodeBench, EvoEval.

## 2.5 Agents

### 2.5.1 Agents: Overview

According to (Park et al., 2023), researchers have begun exploring generative agents, computational entities built on top of LLMs, to create realistic simulations of human-like behavior in interactive environments. Unlike traditional non-player characters that rely on manually scripted rules, these agents autonomously form memories of their experiences, reflect on past events, and dynamically adjust their plans over time. By incorporating mechanisms for long-term memory management, higher-level reasoning, and recursive planning, generative agents can demonstrate remarkably believable patterns of thought, social interaction, and coordination. Early demonstrations, such as populating virtual communities inspired by The Sims, show that these agents can engage in complex social behaviors—spreading information, forming relationships, and even organizing group events—without explicit human direction. This line of research suggests a paradigm shift for code generation and AI-based interactions, opening possibilities for more authentic simulations in user interfaces, game worlds, educational platforms, and social computing systems.

Recent advances in LLM-based AI agents are surveyed in (Xi et al., 2023). The paper argus that LLMs—with their strong language understanding, reasoning, and planning capabilities—can serve as a robust “brain” for intelligent agents. The authors propose a three-part framework: an LLM-based cognitive core (“brain”), a perception module for ingesting multimodal inputs, and an action module for complex outputs such as tool usage or environmental manipulation. (Masterman et al., 2024) also present a thorough overview of recent AI agent architectures that build on LLMs to achieve complex tasks involving intricate reasoning, planning, and external tool usage.

Both frameworks support single-agent use cases (from simple tasks to open-ended exploration) as well as multi-agent systems, where cooperation and competition emerge. Across these designs, the authors highlight key elements that facilitate robust performance: clear role assignments, structured phases of plan creation and refinement, dynamic adjustments to team composition, and efficient communication strategies. The authors also acknowledge that properly selecting between single- or multi-agent paradigms depends on problem characteristics.

The authors of (Li et al., 2024) examine how simply increasing the number of independently sampled outputs from a large language model (i.e., instantiating more “agents” from the same underlying model) and then applying a majority-vote selection can significantly boost task performance. Their comprehensive experiments span arithmetic and general reasoning challenges as well as code generation tasks, showing that this “sampling-and-voting” ensemble approach enables smaller models—when queried multiple times—to match or even surpass the performance of larger ones.

The idea of enabling language-based autonomous agents to dynamically select and employ different problem-solving mechanisms, rather than being limited to a fixed or pre-defined sequence of steps, is explored in (Huang et al., 2024). Various solution strategies include step-by-step reasoning, planning, memory retrieval, reflection, and external tool usage.

Collaboration among agents as one of the agentic architectures is discussed in the four papers listed next. (Wu et al, 2023) introduce AutoGen, an open-source framework for building advanced LLM-based applications by having multiple agents converse with one another. AutoGen provides a standard way for agents to exchange messages, coordinate their actions, and use external tools or human inputs. Developers can easily customize agents and program interactions using both natural language instructions and code. By breaking problems into subtasks and delegating them across different agents—such as coding assistants, reasoning specialists, or safety checkers—AutoGen streamlines the development of more capable and efficient LLM-driven systems.

MetaGPT, a multi-agent cooperation framework designed to organize LLM-driven agents into a structured “virtual team” that follows human-like Standardized Operating Procedures (SOPs), is presented in (Hong et al., 2023). Instead of relying on unstructured conversation, MetaGPT encodes workflows into a series of role-specific prompts, clearly assigning domain experts (e.g., product managers, architects, engineers) to tackle different aspects of a software engineering project - requirements documents, system designs, and code drafts. The authors show that MetaGPT outperforms prior multi-agent chat-based systems in code generation tasks, producing more coherent and reliable solutions. This work emphasizes the potential of combining human-inspired process standards and modular role assignments with LLM-based agents, resulting in more accurate and efficient collaborative code generation processes.

The authors of (Chen, Su et al., 2024) introduce AGENTVERSE, a multi-agent coordination framework that leverages LLMs to orchestrate a team of specialized “expert” agents for complex task-solving scenarios. Rather than relying on a single agent to handle all aspects of a problem, AGENTVERSE mimics the dynamics of human groups by breaking tasks into subtasks and assigning them to different agents, each with domain-specific expertise. Evaluations show that this multi-agent approach outperforms single-agent baselines. The work highlights that carefully structured multi-agent collaboration can achieve higher efficiency and better solutions in complex, real-world tasks than solitary LLM-based agents.

A study in (Chen, You et al., 2024) proposes the Internet of Agents (IoA), a novel framework designed to facilitate LLM-driven multi-agent collaboration in a manner reminiscent of the Internet’s global connectivity. Unlike previous multi-agent systems that operate within isolated ecosystems or on a single device, IoA supports the integration of a wide array of third-party agents—each with diverse skills and tools—distributed across multiple devices and environments. The framework provides flexible mechanisms for dynamic team formation, where agents autonomously locate and recruit additional collaborators as tasks evolve. By enabling heterogeneous agents to discover each other, form nested sub-teams when needed, and efficiently manage shared dialogue states, IoA pushes beyond traditional limitations of multi-agent frameworks, thus paving the way for more scalable, robust, and versatile collaborative intelligent systems.

When it comes to evaluation of LLM-based dialog agents, (Wason et al., 2024) provide a critical examination of LLM-based dialogue agents, arguing that their real-world success depends not only on technical advancements—such as improved training pipelines and state management techniques—but also on responsible design, thorough validation, and ongoing refinement of their prompting strategies and underlying models. This work thus positions LLM-based dialogue agents as promising yet still maturing tools in domains like customer support, virtual assistance, and automatic code generation, each with its own set of unique challenges and performance criteria.

In this context, it is also worth mentioning a paper dedicated to an open-source framework for autonomous language agents described in (Zhou W. et al., 2023). It introduces AGENTS, an open-source framework designed to make creating, customizing, and deploying LLM-based autonomous language agents more accessible. AGENTS facilitates key features such as long-term memory management, versatile tool and web usage, multi-agent communication, and the ability for human users to interact with these agents. It also offers a novel concept of a symbolic plan (SOP) that provides a structured, state-based approach to controlling an agent’s actions, thereby enabling greater predictability and stability in behavior.

### 2.5.2 Agents for Code Generation

The authors of (Jin et al. 2024) conduct a broad investigation into the use of LLMs and LLM-based agents in the field of software engineering, highlighting the distinctions between these two categories and examining their evolving roles across a range of tasks. Their survey categorizes existing work into six key areas: requirements engineering, code generation and development, autonomous decision-making, design and evaluation, test generation, and security and maintenance. Within each of these domains, the authors analyze how standard LLMs and more complex LLM-based agents—capable of autonomous planning, tool usage, and self-improvement—differ in their approaches, requirements, and results.

The paper notes that while LLMs have achieved promising results in tasks like code completion and vulnerability detection, their lack of autonomy often limits them to more static, predefined tasks. LLM-based agents, on the other hand, integrate additional components and potentially multiple cooperating agents, allowing them to interact with external tools, perform multi-turn reasoning, and adapt dynamically to feedback. This shift enables more complex scenarios, such as automated software design, continuous improvement in test coverage, and robust security evaluations.

A structured overview of how LLM-based agents are integrated into various software engineering tasks is provided in (Huang et al. 2024) outlining their key design elements. They observe that recent approaches increasingly rely on the concept of autonomous agents—systems that perceive their environment, store and recall information, and take actions guided by large language models—to handle tasks like code generation, vulnerability detection, and requirement analysis. The authors propose a conceptual framework for LLM-based agents within software engineering, breaking it down into three primary modules: perception, memory, and action.

The perception module handles diverse input formats and transforms them into representations suitable for LLMs. The memory module manages different types of knowledge, ranging from persistent semantic information (like documentation or Application Programming Interface (API) references) to episodic and procedural memories that capture recent events or learned actions. The action module then enables reasoning and planning—often improved by chain-of-thought prompting—and tool usage for activities like code retrieval or debugging. They also highlight that agents can operate individually or collaboratively, with multi-agent systems dividing tasks and sharing knowledge for greater efficiency.

The study in (He et al., 2024) outlines a forward-looking perspective on employing multi-agent systems powered by LLMs to tackle complex software engineering tasks. As software projects become increasingly intricate—spanning requirements definition, code implementation, quality assurance, and maintenance—single LLM-driven agents often struggle to cope with the variety and depth of domain knowledge needed. By contrast, LLM-based multi-agent (LMA) systems promise to leverage teams of specialized agents, each with distinct capabilities, to collaboratively solve multifaceted problems.

The authors highlight three key advantages of LMA systems in software engineering contexts: a) multi-agent collaboration can improve robustness and reliability and battle hallucinations, b) such systems offer greater autonomy – task decomposition tackled independently by specialized agents, streamlining processes like design, coding, and testing without constant human oversight, c) LMA systems are naturally scalable. As project scope evolves, the system can incorporate more agents or adapt roles, enhancing its ability to handle large-scale, diverse software initiatives efficiently.

AGENTLESS, a streamlined method for tackling repository-level software development tasks using LLMs without relying on complex autonomous agents, is presented in (Xia et al., 2024). AGENTLESS follows a simple two-step workflow of localization and repair. By first narrowing down the search space (localizing the edit region within a large codebase) and then generating a patch, this approach avoids the overhead of tool orchestration or dynamic decision-making by the LLM. Surprisingly, on the SWE-bench Lite benchmark, this simpler agentless solution not only achieves superior or competitive success rates compared to advanced agent-based systems, but also does so at substantially lower cost.

The authors of (Qian et al., 2024) introduce ChatDev. Recent advances in LLMs have begun to reshape the way complex software is developed, moving beyond specialized, single-purpose models toward more comprehensive, integrated workflows. The ChatDev framework integrates LLMs into a chat-based environment, enabling agents to engage in multi-turn, language-driven collaboration for end-to-end software production. Rather than developing specialized models tailored to each phase, ChatDev relies on LLM-powered agents guided by a “chat chain” of subtasks and a process called “communicative dehallucination.” This ensures that the agents coordinate effectively, refine their outputs through dialogue, and proactively seek clarity when instructions are ambiguous. Thus, ChatDev fosters a more coherent, flexible, and efficient software development process than the fragmented methods that preceded it.

The study in (Zhang et al., 2024) introduces CODEAGENT, a framework designed to tackle code generation tasks at the level of entire software repositories, a setting that goes beyond the simpler function- or statement-level generation commonly examined in prior research. Recognizing that real-world code often depends on multiple interconnected components. Results show that CODEAGENT substantially improves performance over standard LLM baselines and even outperforms some commercial coding assistants. Furthermore, tests on both the new benchmark and a widely used function-level dataset demonstrate that CODEAGENT’s capabilities are both robust and transferable. Overall, this work highlights the importance of an agent-based approach paired with domain-specific tools for enabling LLMs to handle more complex, context-rich code generation scenarios common in real-world software development.

AGILECODER, a multi-agent software development system that uses Agile practices to better model real-world programming workflows, is presented in (Nguyen et al., 2024). Existing approaches, such as ChatDev and MetaGPT, rely on a waterfall-like process and often assume LLMs can handle entire codebases and decision-making without iteration. In contrast, AGILECODER assigns roles like Product Manager, Scrum Master, Developer, Senior Developer, and Tester to different agents, who then plan, build, and refine software in iterative sprints. Each sprint includes planning, development, testing, and review phases, allowing for continuous improvement and adjustments to changing requirements.

AGILECODER surpasses existing benchmarks on standard datasets like HumanEval and MBPP, as well as on a new, more complex dataset (ProjectDev).

## 2.6 Evaluation of Generated Code

Evaluation of the generated code is a very important aspect of the automatic code generation process as it makes it possible to understand the quality of the code generation process.

The **HumanEval** dataset which offers a relatively small set of hand-crafted programming tasks with hidden tests, useful for quick and controlled assessments, is introduced in (Chen et al., 2021). The Automated Programming Puzzles & Solutions(**APPS**) benchmark introduced in (Hendrycks D. et al., 2021) positions it as a more expansive and challenging alternative. Unlike HumanEval with a limited number of function-level problems and a few test cases each, APPS comprises thousands of more complex and varied coding problems sourced from real coding competitions, each backed by extensive and diverse test inputs.

The study in (Austin et al., 2021) investigates the capabilities of LLMs to synthesize code in general-purpose programming languages, focusing on Python. Their work introduces two benchmarks: Mostly Basic Python Problems (**MBPP)**, a dataset of nearly one thousand entry-level programming tasks, and Math Question Answering (MathQA-Python) containing tens of thousands of math-related coding questions. They examine both few-shot prompting—providing only a handful of examples—and fine-tuning on a small subset of tasks. Their findings show that performance on code generation improves substantially as model size increases, and that fine-tuning further boosts accuracy. Also, their analysis reveals that models struggle with deeper program “understanding,” as evidenced by poor results on tasks requiring them to predict code outputs given specific inputs.

The authors of (Miah & Zhu 2024) propose a user-focused method for evaluating large language models’ effectiveness as code generation tools, using ChatGPT’s R code generation capabilities as a case study. Unlike conventional benchmarks that primarily gauge accuracy or human-level skill, their approach integrates usage-related metadata, emulates realistic user interactions through multi-attempt processes, and assesses outputs on multiple quality aspects (e.g., completeness, readability, logic structure) rather than correctness alone. They find that ChatGPT generally performs well for R programming tasks, though it struggles with more complex challenges.

CodeScore is introduced in (Dong et al., 2024), a novel evaluation metric for code generation based on functional correctness, addressing limitations in traditional match-based metrics like BLEU and CodeBLEU, which emphasize surface-level similarities and fail to account for functional equivalence. CodeScore leverages LLMs fine-tuned to assess code execution through measures like PassRatio and Executability. The study highlights that CodeScore aligns closely with human judgment and effectively evaluates code in practical settings.

The authors of (Du X. et al., 2024) conduct the first evaluation of LLMs in generating Python classes composed of multiple, interdependent methods—a task more representative of real-world software development than typical function-level benchmarks like HumanEval. They introduce **ClassEval**, a manually constructed benchmark of 100 class-level code generation tasks, each with extensive tests and dependencies among methods. Their empirical study shows a substantial drop in performance compared to method-level code generation, and reveals that the best-in-class GPT models still dominate, though the relative ranking of other models changes when moving from method-level to class-level tasks.

Gao (2023) highlights several **key flaws in popular code‐generation benchmarks**: first, data contamination—HumanEval and MBPP problems are so widely circulated online that modern LMs have almost certainly encountered them during training, calling pass-rate results into question; second, oversimplified tasks—many questions are too trivial to reflect the complexity of real‐world engineering challenges; and third, weak test suites—the provided unit tests often miss edge cases and subtle logic errors, so they require significant strengthening.

The authors of (Matton et al., 2024) raise a similar concern - data leakage in code generation which occurs when popular evaluation benchmarks (like HumanEval and MBPP) appear in a model’s training data—whether intentionally or unintentionally—thereby compromising the validity of test scores as measures of this model’s generalization. This contamination can arise through direct inclusion of test examples in the training corpus, via synthetic data creation pipelines that inadvertently reproduce evaluation samples, or through overfitting models to a narrow set of public benchmarks during checkpoint selection. As a result, reported improvements on these benchmarks may reflect memorization or over-optimization rather than genuinely improved coding capability. To address these challenges, the authors introduce a set of 161 Less Basic Python Problems (LBPP), a new Python code generation benchmark designed to avoid overlap with existing training data and provide a more trustworthy measure of code generation performance.

## 2.7 Conclusion

SLMs are en route to becoming an important player in the realm of AI. They perform well on specialized tasks and show high efficiency and accessibility which makes both developers and companies consider them attractive alternatives to LLMs. As more businesses refine and fine-tune SLMs, even faster progress in this space is expected (Morris et al., 2024).

The potential of SLMs was further uncovered in a recent discovery made by HuggingFace researchers. (Beeching et al., 2024.) discusses an approach to improving language model performance that focuses on scaling test-time compute - essentially allowing a model to “think longer” or search more extensively during inference. By carefully allocating additional compute at test-time, even smaller models can achieve results that rival or exceed those of their much larger counterparts on challenging tasks like MATH benchmarks.

The core idea is to use dynamic inference strategies, such as iterative self-refinement or verifier-based search methods, to guide a model toward correct answers. While large models rely on their vast parameters for accuracy, small models can offset this disadvantage by systematically applying more reasoning steps and better filtering mechanisms at test-time. Crucially, these methods show that tiny 1B and 3B-parameter models can outperform models as large as 70B parameters if given enough “time to think” - that is, enough test-time search and verification cycles. This opens the door to resource-efficient LLM deployments where you don’t need massive compute for training; instead, you invest your compute at inference time, unlocking high performance from much smaller models.

By integrating these promising new SLM architectures with agent-based systems—where models interact with external tools, retrieve relevant data, and break down problems into manageable steps—the authors plan to achieve even more substantial efficiency and accuracy gains. Agents acting as orchestrators can direct an SLM’s inference strategies, selecting when and how to apply iterative refinement or verification procedures. They can determine which domain-specific resources to query and how to adaptively allocate additional compute where it matters most. This synergy allows small models not only to think more intelligently at test-time but also to operate in more dynamic, context-rich environments, performing complex tasks with higher confidence and success rates. In essence, coupling SLMs with agents amplifies their inherent strengths—cost-effectiveness, flexibility, and specialization—while strategically compensating for their smaller parameter counts, thus paving the way for more powerful, efficient, and responsive AI systems.