# Chapter 1 - Introduction

## 1.1 Background

The modern technological world is driven by the software development industry, and millions of software engineers worldwide are among the highest-paid professionals. Companies invest heavily in this workforce to develop and maintain software, leading to substantial labor costs.

Recently, large language models (LLMs) have emerged as powerful tools capable of automating code generation from natural language descriptions, thus enhancing the software engineering efficiency.

According to (McKinsey Report, 2023), generative AI can significantly decrease the time developers spend on coding - by up to 45% - which can enable companies to reduce labor costs. This efficiency gain is particularly impactful for large-scale projects where even marginal improvements translate into significant cost reductions.

Automating code generation also leads to improved code quality through consistent adherence to coding standards and best practices. Studies like (Almeida Y. et al, 2024) and (Martinović B. and Rozić R., 2024) highlight how consistent and well-structured code generated by AI-enhanced tools contributes to minimizing human errors and bugs. In (Kalliamvakou, 2024), GitHub concluded that developers using GitHub Copilot finished their tasks 55% faster than the developers who preferred not to use GitHub Copilot. But developer productivity goes beyond speed - according to the same study between 60–75% of developers reported that they feel more fulfilled with their job, feel less frustrated, and can focus on more satisfying work when using GitHub Copilot.

In accordance with (Gartner Report, 2023), accelerated development cycles allow companies to bring products to market more quickly, providing a significant competitive edge by reducing manual coding time and streamlining development processes.

According to Google's CEO Sundar Pichai (Pichai, 2024) AI tools are already having a sizable impact on software development, and more than 25% of new code at Google is AI-generated. This helps Google engineers achieve more and work faster.

Google developers aren't the only programmers using AI to assist with coding tasks. According to Stack Overflow's 2024 Developer Survey (Stack Overflow, 2024), over 76% of all respondents are already using or plan to use AI tools in the development process this year, with 62% actively using them. A 2023 GitHub survey (Shani S. & GitHub Staff. 2023) showed that 92 percent of US software engineers are using AI tools for coding tasks in and outside of work.

However, the use of proprietary LLMs poses significant challenges regarding sensitive data protection and intellectual property rights. Developers often need to use proprietary or confidential information either to train these models or at the time of inference, risking data breaches and unauthorized access to intellectual property. This not only jeopardizes a company's competitive advantage but also exposes it to legal liabilities.

To address these concerns, companies could use small language models (SLMs) boosted by agents and deployed in resource-constrained and secure environments. SLM-based agents offer a cost-effective and privacy-preserving alternative to proprietary LLMs. They enable organizations to automate the creation of basic code routines without compromising sensitive data, which reduces the time developers spend on manual coding. This approach is especially beneficial for understaffed projects, providing efficient solutions without the need to hire additional software engineers for routine tasks.

Leveraging SLM-based agents for automated code generation addresses the dual challenge of making the process of writing code more efficient and protecting sensitive data. Furthermore, companies with limited financial resources that prefer not to hire many software engineers, can leverage this technology to efficiently write code while using a small workforce and achieve what would have been impossible just several years ago. This approach enables companies to optimize developer productivity, enhance code quality, and accelerate time-to-market, all while ensuring data confidentiality.

## 1.2 Research Motivation

The primary motivation for this research is ensuring efficiency, cost reduction, and data privacy in software development. As someone who worked for several large companies that had classified proprietary information or intellectual property, we can state that there is an evident trend that companies are reluctant to use LLMs for data privacy and security reasons.

While LLMs have demonstrated remarkable capabilities in automating code generation from natural language descriptions, they pose significant challenges related to sensitive data protection and intellectual property rights. Using proprietary LLMs often requires transmitting confidential information to third-party servers, raising concerns about data breaches and unauthorized access to proprietary code. This not only risks a company's competitive advantage but also exposes it to potential legal liabilities.

On the other hand, these companies could use SLMs to provide a similar level of solution quality. By conducting this research, we aim to develop a solution that leverages the advantages of SLMs while minimizing their limitations compared to LLMs. To address these challenges, there is a strong motivation to explore the use of SLMs enhanced by agents for automated code generation within secure, resource-constrained environments. SLM-based agents offer several compelling benefits:

1. **Deploying SLMs in-house** ensures that sensitive data and intellectual property remain within the organization's secure environment.
2. **SLMs are generally more cost-effective** than proprietary LLMs which makes advanced code generation capabilities more accessible to organizations with limited resources.
3. SLMs don’t require **massive GPU clusters** to fine-tune.
4. SLM-based agents can consistently adhere to **coding standards and best practices**, reducing human errors and bugs.
5. **Faster development cycles** enable companies to bring products to market more quickly, providing a competitive edge in rapidly evolving industries.
6. In understaffed projects, SLM-based agents can compensate for **limited human resources** by efficiently generating code, reducing the need to hire additional developers for routine tasks.
7. Researching how to enhance SLMs with agent-based architectures **contributes to the broader field of AI and machine learning**, pushing the boundaries of what smaller models can achieve in specialized tasks like code generation.

## 1.3 Problem Statement

*Using proprietary large language models (LLMs) to automatically generate code is costly and not safe from the sensitive data protection and intellectual property standpoints forcing developers to spend twice as much time writing code manually.*

Proprietary LLMs are expensive in deployment and/or inference and expose sensitive data, pushing teams to code manually, slowing development and increasing costs. Data privacy and intellectual property risks with proprietary LLMs discourage their use, compelling developers to spend more time coding manually

## 1.4 Thesis Statement

*Agents based on open-source small language models (SLM) deployed in resource-constrained environments for automated code generation will ensure lower costs and sensitive data protection, reducing the manual coding time and speeding up development cycle.*

By paving the road for automated code generation, SLM-based agents reduce the overall time developers spend writing code while still preserving data privacy. This research introduces a novel approach by leveraging SLM-based agents to automate code generation from natural language descriptions, surpassing SLMs and attempting to approach the proprietary LLMs in code quality. Python software developers may use such a product to automatically generate code while ensuring sensitive data protection and reducing time for manual coding.

## 1.5 Research Objectives

The primary objective of this research is to develop and evaluate an agent-based system utilizing SLMs to automatically generate code from natural language descriptions. The study aims to bridge the performance gap between SLMs and proprietary LLMs in code generation tasks while ensuring data privacy and cost efficiency.

Since LLMs are costly and require investments into training data and large GPU clusters, companies can deploy more accessible SLMs. A tradeoff would be the lack of quality of LLMs, but using agents can make it competitive with LLMs to a certain degree Hence, the study aims to enhance and evaluate the code generation quality while ensuring data privacy and security, improving the cost efficiency and developer productivity, and accelerating time-to-market.

By achieving these objectives, the research aims to create a viable, secure, and efficient alternative to proprietary LLMs for automated code generation.

## 1.6 Research Questions and Hypotheses

Below is a list of research questions studied in the current Praxis, as well as hypotheses that need to be proved in the end of the Praxis cycle.

**Research question 1:** Will fine-tuning SLMs used by agents result in higher code generation quality as measured by the maintainability index?

**Research question 2:** Will changing SLM parameters, such as temperature and top-p, ensure greater code quality based on lower cyclomatic complexity??

**Research question 3:** Which agentic workflow, reflection or multi-agent collaboration, leads to a greater number of tests passed?

**Hypothesis 1**: Fine-tuning SLMs on domain-specific data will noticeably increase the maintainability index compared to using an LLM without fine-tuning.

**Hypothesis 2**: Adjusting SLM parameters, such as temperature and top-p, will noticeably improve the cyclomatic complexity of auto-generated code.

**Hypothesis 3:** Multi-agent collaboration will lead to a noticeably greater number of tests passed compared to the reflection agentic workflow.

## 1.7 Scope of Research

This research aims to assess the feasibility and competitiveness of using SLMs enhanced with agent-based architectures for automated code generation from natural language descriptions. It involves the following key activities:

* **Establishing the current benchmarks for code generation** by LLMs and SLMs using public leaderboards.
* **Selecting one or several SLMs** which can be used as is or which can be additionally fine-tuned on public code generation datasets in order to enhance their code generation capabilities.
* **Developing agent-based architectures** that integrate with SLMs to enhance their reasoning, planning, and problem-solving abilities facilitating the decomposition of complex coding tasks into manageable subtasks, enabling iterative refinement, and incorporating feedback mechanisms to improve code generation outputs.
* **Conducting systematic experiments** to assess the performance of the enhanced SLMs in automated code generation tasks on a variety of coding challenges based on natural language descriptions.
* **Utilizing quantifiable evaluation metrics** to evaluate the quality, correctness, and efficiency of the generated code.
* **Documenting the methodologies, experiments, and findings** comprehensively to contribute to the academic community.

## 1.8 Research Limitations

This research on SLMs enhanced by agent-based architectures for automated code generation has several limitations that may impact the scope, applicability, and generalizability of the findings. Recognizing these limitations is essential for interpreting the results accurately and identifying areas that require further investigation.

First of all, a fundamental limitation of this study is the fact that due to their smaller size and fewer training parameters, SLMs may not achieve the same level of sophistication, contextual understanding, and code generation quality as LLMs. Despite enhancing SLMs with agentic workflows, there may still be a noticeable gap in complex code generation tasks where LLMs excel. Also, the research focuses exclusively on SLMs and does not include the implementation of similar experiments using LLMs. Any comparisons drawn between SLM-based agents and proprietary LLMs rely on existing literature or reported benchmarks.

The study is conducted within the confines of limited hardware and computational resources, which are representative of resource-constrained environments typical for organizations without extensive infrastructure. This restricts the extent of model fine-tuning, the size of datasets processed, and the complexity of agentic architectures employed which could impact the final results. The one year allocated for this research may limit the depth and breadth of exploration possible - not all SLMs, programming languages, agentic workflows, or evaluation metrics can be exhaustively examined during this relative short period of time. That is why this study is confined to Python code generation and specific agentic workflows — namely, reflection and multi-agent collaboration — which may not capture the full spectrum of potential strategies. Also, the research specifically focuses on code generation from natural language descriptions and does not address other aspects of software engineering automation, such as code refactoring, bug detection, or code optimization.

The datasets used for fine-tuning and evaluating SLMs are limited to publicly available ones. The absence of proprietary, domain-specific, or larger-scale datasets may affect the models' ability to generalize to real-world applications, and the quality and diversity of these datasets may influence performance outcomes. In addition, SLMs trained on publicly available data may inadvertently learn and propagate biases present in the training data. The research does not specifically address bias detection or mitigation strategies, which could impact the fairness, ethical considerations, and acceptance of the generated code in sensitive applications.

Although a key motivation for using SLMs is to enhance data privacy by keeping computations in-house, the research does not delve deeply into the implementation of robust data protection measures. Additionally, the rapid evolution of AI technologies means that newer SLMs or alternative methods may emerge by the time of publication, potentially surpassing the models and approaches evaluated in this study and affecting the relevance and applicability of the findings.

## 

## 1.9 Organization of Praxis

This Praxis has the following structure: the current *Introduction* chapter will be followed by Chapter 2 *Literature Review* describing the research performed in the field to date to solve similar problems. It includes a careful, but critical comparison of available work described in the literature that is directly related to the problem at hand.

Chapter 3 *Research Methodology* conveys a complete understanding of the methodology used to conduct the research capturing assumptions, ease of use, input data, expected output results, constraints, required adaptations, and other important aspects.

Chapter 4 *Results* demonstrate the actual outputs of the steps described in the methodology highlighting the results accomplished after each step of the methodology. It may contain descriptive statistics, charts, tables, and other visual representations of the work conducted in the Praxis. This chapter also summarizes key findings and compares results of various methods examined, final performance, etc.

Chapter 5 *Discussion and Conclusions* outlines how the findings of the study are related to the research questions and hypotheses.

The *References* section lists all information sources used to justify or conduct the research or which were mentioned / cited in the Praxis.

# Chapter 2—Literature Review (18-20 pp.)

## 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).

**2.2 Automatic Code Generation**

Recently, large language models (LLMs) have emerged as a key technology in bridging the gap between human intent expressed in natural language and the automated code generation. Early approaches often relied on recurrent neural networks (RNNs) or specialized, syntax-driven methods that did not fully reflect long-range dependencies and contextual subtleties present in complex coding tasks. With the introduction of transformer-based architectures, researchers have noticed significant performance boost by leveraging pre-trained language models for code more effectively that were originally developed for natural language processing (NLP) tasks. For example, recent studies have integrated well-known encoder transformer models such as BERT, RoBERTa, ELECTRA, and LUKE with Marian decoder transformers, achieving state-of-the-art results on such standard benchmarks like CoNaLa and DJANGO. These hybrid models improve the syntactic and semantic quality of the generated code and reduce the manual effort by offering capabilities such as intelligent autocompletion, context-aware suggestions, and inline documentation support. Moreover, researchers have emphasized the importance of refining the generated code through linting, formatting, and error-checking utilities, ensuring that the outputs adhere to established coding standards and facilitate seamless integration into real-world software development workflows. Together, these improvements highlight the huge potential of LLMs in improving both the accuracy and efficiency of modern code generation solutions (Soliman, 2024).

Pre-trained models have demonstrated strong generalization in both natural and programming languages, leading to the development of models specifically designed to handle code-related tasks. One notable example is CodeBERT, a transformer-based model that learns joint representations of natural language (NL) and programming language (PL) inputs. Rather than relying solely on text, CodeBERT is trained on paired NL-PL data—such as code snippets coupled with documentation—as well as unimodal resources like standalone code. This training scheme incorporates masked language modeling and a replaced token detection objective, allowing CodeBERT to capture rich semantic correspondences between NL descriptions and code functionality.

Evaluations on benchmarks demonstrate that CodeBERT can effectively perform code-related understanding and generation tasks. For instance, it achieves strong results in code search, where a natural language query must be matched with a relevant code snippet. Moreover, CodeBERT excels in tasks like documentation generation, producing summaries of code behavior that are more fluent and informative than those from models trained only on text or code in isolation. Additionally, probing experiments indicate that CodeBERT internalizes both NL and PL semantics, enabling zero-shot reasoning about programming constructs and natural language descriptions.

In essence, CodeBERT’s approach—integrating bimodal pre-training objectives and large-scale NL-PL resources—demonstrates that jointly modeling programming languages and their natural language descriptions can improve downstream performance. Its adaptability across multiple programming languages and its ability to generalize to languages unseen during training further highlight the potential of this paradigm. As the field progresses, models like CodeBERT pave the way for more sophisticated NL-PL integration strategies, potentially incorporating structural information, advanced reasoning techniques, and domain-specific customization to further enhance code understanding and generation tasks (Feng, Z. 2020).

(Defferrard M. et al., 2024) explores 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. Over time, this leads to a virtuous cycle: 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.

## 2.3 LLMs

**2.3.1 LLMs: Overview**

Lorem ipsum dolor sit amet, consectetur adipiscing elit (Fox, 2012).

**2.3.2. LLMs for Code Generation**

Lorem ipsum dolor sit amet, consectetur adipiscing elit (Fox, 2012).

**2.4 SLMs**

**2.4.1 SLMs: Overview**

While Large Language Models (LLMs) have demonstrated impressive capabilities, their massive size leads to drawbacks in efficiency, cost, and customizability. Small Language Models (SLMs) address these issues by providing a more efficient, cost-effective, and customizable alternative without significantly compromising performance.

Research indicates that models with as few as 1–10 million parameters can demonstrate basic language competencies.. Despite their smaller size, SLMs aim to perform similar tasks such as text generation, summarization, translation, and question-answering. In SLMs, it is important to balance model size with performance and flexibility. While challenges such as responsible deployment and maintenance exist, the potential benefits position SLMs to drive the next phase of AI innovation and productivity across various industries.

Motivations for developing SLMs include:

* efficiency which is based on faster inference, lower resource requirements (less memory and storage space), and smaller training datasets: SLMs can be effectively trained on less data, reducing the data acquisition burden.
* Cost reflected in reduced training costs and affordable deployment.
* Customizability as SLMs can be tailored to specific domains or tasks more easily than LLMs. Organizations are increasingly developing proprietary SLMs tailored to their specific domains, such as finance, healthcare, and education. Also, SLMs allow for quicker iteration via faster experimentation and refinement. They can be modified to suit niche tasks through such techniques as pretraining or fine-tuning on domain-specific datasets, optimizing prompts for specific applications, faster and easier adjustment of the model's structure to better suit certain tasks.

Advantages of SLMs include: a) superior accuracy in specialized tasks, confidentiality because in-house models prevent exposure of sensitive data, ensuring compliance, rapid iteration and alignment with organizational needs as well as cost efficiency by reducing reliance on external models and services.

Challenges associated with SLMs include data sufficiency, model governance, and maintenance costs as models need regular updates to address data drift and maintain reliability (Ghosh, 2023).

Small Language Models (SLMs) play a crucial role in enhancing business efficiency. While LLMs have garnered significant attention for their expansive capabilities, SLMs offer a domain-specific alternative that is more efficient and cost-effective. Businesses are recognizing that while LLMs are powerful, they may not always be the most practical solution for domain-specific tasks due to their size, cost, and resource requirements.

On the other hand, SLMs are gaining traction because they are more efficient, less costly to implement, and better suited for specialized tasks which can easily enhance business operations. SLMs are designed with fewer parameters and trained on domain-specific data. As a result, they require less computational power and can be deployed efficiently on a range of devices, including mobile and edge systems. This domain-focused efficiency not only reduces operational costs but also strengthens data security since smaller models can often be run locally, minimizing reliance on cloud infrastructures.

In essence, the rise of SLMs underscores a balanced approach to AI adoption: businesses gain the precision and agility needed for specific goals, while avoiding the resource-intensive drawbacks of larger-scale models. For developers and engineers, this trend points toward an ecosystem where smaller, domain-optimized language models complement the landscape already shaped by large, general-purpose counterparts (Szczygło, 2024)

(Quach, S. 2024) highlights the growing importance of Small Language Models (SLMs) as leaner, domain-targeted alternatives to their larger, more general-purpose counterparts. Unlike massive Large Language Models (LLMs) that rely on extensive and often unwieldy datasets, SLMs focus on specific domains and tasks, reducing computational overhead and costs while maintaining strong performance within their target areas. This efficiency is achieved through techniques such as knowledge distillation, pruning, and quantization, resulting in models that are typically just a few gigabytes in size. Although SLMs may lack the broad adaptability of LLMs, their tighter specialization leads to faster processing, lower latency, and a reduced risk of generating irrelevant or misleading outputs. These attributes make SLMs attractive for enterprise applications, particularly those involving proprietary or sensitive data, where customization, cost-effectiveness, and data security are paramount. As organizations increasingly recognize these advantages, SLMs are becoming a prominent choice for deploying AI solutions that are both powerful and practical.

(Fatima, F. 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. In contrast to large language models (LLMs) with billions of parameters and significant computational demands, 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.

Each of the highlighted models adopts an open or accessible development philosophy, encouraging customization, on-device processing, and domain-specific fine-tuning. This focus on resource-efficiency and adaptability enables them to be deployed in environments with limited hardware, power constraints, or strict privacy requirements. Compared to larger models, SLMs are often easier to update, maintain, and trust, aligning with responsible and transparent AI practices. As a result, these emerging SLMs are not only democratizing AI access for startups, researchers, and smaller enterprises, but also influencing the broader trajectory of model design, paving the way for more sustainable, secure, and versatile AI solutions.

(Kili Technology Guide, 2024) provides a practical overview of small language models (SLMs) and their role in business applications, contrasting them with their larger counterparts (LLMs). SLMs are essentially scaled-down versions of large models that, despite having fewer parameters, can effectively handle focused tasks while demanding less computational power and infrastructure. They are more cost-efficient, agile, and simpler to integrate, making them suitable for organizations that prioritize resource management, data privacy, or niche language-processing scenarios.

The guide outlines when and why SLMs are appropriate, such as for specialized language tasks, quick-response applications, and settings with constrained budgets or strict data confidentiality requirements. It also shows that these smaller models can be tailored more easily through fine-tuning or used in tandem with retrieval-augmented generation methods to incorporate external information sources. Several examples of existing SLMs illustrate their capabilities in areas like customer support automation, content creation, and basic code assistance.

Crucially, the guide emphasizes the need for careful dataset preparation, continuous evaluation, and systematic monitoring of performance. By maintaining close oversight, businesses can ensure their chosen SLM stays aligned with evolving goals and consistently delivers value. In summary, SLMs offer a balanced approach to deploying language-based AI solutions—one that trades some of the expansive capabilities of large models for accessibility, adaptability, and overall efficiency.

**2.4.2. SLMs for Code Generation**

Recent achievements in the field of large language models (LLMs) have significantly improved code generation, particularly via “Chain-of-Thought” method that breaks problems into smaller reasoning steps. However, the practical deployment of massive LLMs is hampered by high costs and data security concerns, prompting interest in transferring LLM reasoning abilities to smaller, more manageable models. 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 shift emphasizes equipping smaller models with the underlying reasoning patterns of LLMs to improve their code generation quality and efficiency without the burdens of large-scale deployment (Sun, Z. et al. 2024).

In recent work, researchers have begun exploring ways to break down complex reasoning tasks for code and math problem-solving into more manageable parts. Traditional strategies often rely on a single large language model (LLM) to both decompose a problem into subproblems and then solve those subproblems. While this approach can deliver strong results, it remains computationally expensive and restricts fine-tuning options, since many of the largest models are not openly available for retraining. More importantly, it ties both “understanding” and “solution” stages to a single massive model, which may not be optimal.

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.

The decomposition model is trained using a combination of supervised fine-tuning and reinforcement learning (RL) methods. Initially, it learns to produce relevant subproblems by observing high-quality reasoning paths. It then refines its approach by interacting with the solver, receiving feedback on how well its generated subproblems guide the larger model toward a correct final answer. Through RL-based optimization, the decomposition model adapts to the solver’s behavior—improving its ability to identify particularly effective subproblems, focusing on steps that correct earlier solver mistakes, and ultimately enhancing the solver’s overall performance.

Evaluations have demonstrated that such a division of labor can substantially boost performance on complex reasoning tasks. Smaller models, once aligned to decompose problems effectively, can enable large solvers to approach or even surpass the capabilities of newer, more powerful LLMs. In some cases, these composite systems rival or outperform standard prompting methods and even begin to close the gap with top-tier models like GPT-4. This approach opens the door to more efficient reasoning pipelines, reduces the reliance on ever-larger single models, and illustrates the potential of modular architectures for code generation and other intricate tasks (Juneja, G., 2024).

(Anonymous authors. 2024) introduces a training-free framework, called Agents Help Agents (AHA), for transferring knowledge from large language models (LLMs) to smaller, locally run language models (SLMs) in the domain of data science code generation. 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 on multiple complex, tabular data analysis benchmarks show that AHA’s approach significantly improves SLM performance. Moreover, this distilled knowledge can be applied to other SLMs not originally involved in the training process, suggesting that the method is both model-agnostic and scalable.

**2.5 Agents**

**2.5.1 Agents: Overview**

Researchers have begun exploring generative agents, computational entities built on top of large language models, 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. (Park, J.S. 2023)

## 2.5.2 Agents for CodeGen

Recent advances in large language models (LLMs) have begun to reshape the way complex software is developed, moving beyond specialized, single-purpose models toward more comprehensive, integrated workflows. Existing approaches to leveraging deep learning in software development have often focused on optimizing isolated stages—such as design, coding, or testing—within the traditional waterfall model. Although these techniques can improve individual phases, this compartmentalized approach tends to create technical gaps and inconsistencies across the development lifecycle. To address this limitation, recent work proposes adopting a unified communication paradigm that treats natural language as a bridge among agents performing distinct roles. In particular, the ChatDev framework integrates large language models (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. By merging phases through natural and programming-language exchanges, ChatDev fosters a more coherent, flexible, and efficient software development process than the fragmented methods that preceded it (Qian C. et al., 2024).

(Zhang K et al. 2024) describes a training-free framework, called Agents Help Agents (AHA), for transferring knowledge from large language models (LLMs) to smaller, locally run language models (SLMs) in the domain of data science code generation. 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 on multiple complex, tabular data analysis benchmarks show that AHA’s approach significantly improves SLM performance. Moreover, this distilled knowledge can be applied to other SLMs not originally involved in the training process, suggesting that the method is both model-agnostic and scalable.

(Zhang K. 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, CODEAGENT allows a Large Language Model (LLM) to interact with an integrated toolkit. This toolkit includes searching through online resources, navigating repository documentation, analyzing code symbols, checking code formatting, and running tests. To make effective use of these tools, the framework explores four different agent-based strategies that help the LLM break down complex coding tasks into manageable steps and adapt its approach dynamically.

To evaluate CODEAGENT’s effectiveness, the authors introduce CODEAGENTBENCH, a new benchmark that captures realistic repository-level coding challenges from real open-source projects. 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.

**2.6 Agents + LLMs / SLMs for Code Generation**

**2.7 Summary and Conclusion**

SLMs are en route to become 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, we expect to observe even faster progress in this space (Quach, S. 2024.

**References**

McKinsey Report. 2023. [Unleashing developer productivity with generative AI](https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/unleashing-developer-productivity-with-generative-ai).

Yonatha Almeida, Danyllo Albuquerque, Emanuel Dantas Filhob, Felipe Munizb, Katyusco de Farias Santosb, Mirko Perkusichc, Hyggo Almeidac, Angelo Perkusichc. AICodeReview: Advancing code quality with AI-enhanced reviews. 2024.

Boris Martinović and Robert Rozić. Impact of AI Tools on Software Development

Code Quality. 2024.

Eirini Kalliamvakou, GitHub. 2024. Quantifying GitHub Copilot’s impact on developer productivity and happiness.

Shani S. & GitHub Staff. 2023. Survey reveals AI’s impact on the developer experience.

Gartner Report. 2023. Top Strategic Technology Trends. <http://emtemp.gcom.cloud/ngw/globalassets/en/publications/documents/2023-gartner-top-strategic-technology-trends-ebook.pdf>

Pichai, S. 2024. Q3 earnings call: CEO’s remarks. <https://blog.google/inside-google/message-ceo/alphabet-earnings-q3-2024/#full-stack-approach>

Morris, S. 2023. AI, cloud boost Alphabet profits by 34 percent. <https://arstechnica.com/gadgets/2024/10/ai-cloud-boost-alphabet-profits-by-34-percent/>

Stack Overflow. 2024. AI. <https://survey.stackoverflow.co/2024/ai>

Ahmed Soliman, Samir Shaheen, Mayada Hadhoud. 2024. Leveraging pre-trained language models for code generation.

Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, Zhi Jin. 2024. Enhancing Code Generation Performance of Smaller Models by Distilling the Reasoning Ability of LLMs.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng,  
Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages

Juneja, G., Subhabrata Dutt, Soumen Chakrabarti, Sunny Manchhanda, Tanmoy Chakraborty. 2024. Small Language Models Fine-tuned to Coordinate Larger Language Models Improve Complex Reasoning.

Qian C., Xin Cong, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, Maosong Sun. 2024. Communicative Agents for Software Development.

Bijit Ghosh. 2023. The Rise of Small Language Models— Efficient & Customizable. <https://medium.com/@bijit211987/the-rise-of-small-language-models-efficient-customizable-cb48ddee2aad>

Szczygło, P. 2024. Small Language Models Examples Boosting Business Efficiency. <https://www.netguru.com/blog/small-language-models-examples>.

Park, J.S. 2023. Generative Agents: Interactive Simulacra of Human Behavior. 2023. Meredith Ringel Morris, Joseph C. O’Brien, Percy Liang, Carrie J. Cai, Michael S. Bernstein

Quach, S. 2024. Mini Models, Major Impact: How Small Language Models Outshine LLMs.

<https://www.knowi.com/blog/mini-models-major-impact-how-small-language-models-outshine-llms/>

Fatima, F. 2024. The 5 leading small language models of 2024: Phi 3, Llama 3, and more.

Anonymous authors. 2024. Agents Help Agents: Exploring Training-Free Knowledge Distillation for Small Language Models in Data Science Code Generation.ICLR 2025 Conference Submission. <https://openreview.net/forum?id=hREMYJ5ZmD>.

Zhang, K., Jia Li∗, Ge Li†, Xianjie Shi, Zhi Jin. 2024. CODEAGENT: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges.

Defferrard M., Corrado Rainone, David W. Zhang, Blazej Manczak, Natasha Butt, Taco Cohen. 2024. Towards Self-Improving Language Models for Code Generation.

Kili Technology. 2024. A Guide to Using Small Language Models for Business Applications.