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).

(Coutinho M. et al., 2024) conduct a preliminary case study exploring how generative AI tools influence productivity within real-world software development settings. By distributing licenses for various 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 initial qualitative insights on how these tools fit into day-to-day activities. Participants generally report positive impacts on their perceived productivity, particularly through time savings, efficient artifact generation, and quick access to information. However, respondents also highlight challenges such as ensuring reliability, refining outputs, and addressing security concerns when handling sensitive data. While the study is limited in scale and primarily focuses on individual perceptions, 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 these productivity gains in broader, more complex development contexts.

The study in (Martinović B. and Rozić R., 2024) investigates how developers perceive the influence of AI-based tools on the quality of code produced during software development. 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 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.

The data show that most developers using AI tools express relatively high overall satisfaction, with more than three-quarters stating that their adoption of such tools improved their day-to-day development work. However, when it comes to specific elements of code quality, improvements are not uniform. While respondents reported noticeable gains in maintainability and efficiency, perceptions of improved readability and accuracy were more modest. This indicates that while AI assistants can streamline certain coding tasks and reduce time-consuming chores, their ability to consistently produce top-tier, flawless code is still maturing.

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. The authors suggest that

as AI capabilities become more refined—offering consistent code improvements, stronger accuracy, and better integration into established workflows—more developers may embrace these solutions. The study concludes by calling for broader and more diverse research efforts, as well as further innovations that can translate developer satisfaction into measurable and uniform code quality improvements.

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. Unlike today's limited aids, which focus mainly on code suggestions and debugging, 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

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.

(Almeida Y., 2024) introduces a GPT-3.5-powered IntelliJ IDEA plugin 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 involving 12 undergraduate participants showed that AICodeReview significantly outperformed manual reviews, reducing review time (15.2 vs. 22.5 minutes), detecting more code smells (28 vs. 20), and improving refactoring outcomes (25 vs. 13). The study highlights the tool's potential to enhance code quality, productivity, and reliability while proposing future expansions to integrate additional LLMs and tools like SonarQube. This work emphasizes the effectiveness of LLMs in streamlining software development processes.

(Ottens L. et. al., 2024.) explores the use of pre-trained language models for Python code generation, 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 fine-tuned GPT-2 model demonstrates superior performance, achieving a 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 like functions, conditionals, and method calls. This work underscores the potential of large language models in automating software development tasks while improving efficiency and quality.

The paper by Le T. et al. (2020) offers a comprehensive review of deep learning (DL) applications in source code modeling and generation. 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.

DL Frameworks: 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.

An interesting approach is proposed by (Zhou S. et al., 2023) who introduce DocPrompting, a novel method designed to enhance automatic code generation by leveraging code documentation. Recognizing that programming libraries and APIs are continually evolving, and existing models cannot effectively utilize newly introduced functions and libraries that were not part of training data. Therefore, DocPrompting mimics the human approach of consulting documentation to understand and implement unfamiliar code components.

DocPrompting integrates existing documentation retrieval into the code generation process by fetching relevant documentation snippets based on a given natural language (NL) intent. Then it uses the retrieved documents and the original NL intent to generate the desired code. This approach is versatile, applicable to various programming languages, and compatible with different underlying neural architectures.

DocPrompting significantly improved the performance of several strong base models, including CodeT5, GPT-Neo, and Roberta. These results highlight DocPrompting's effectiveness in enabling models to generate accurate and functional code, even when dealing with previously unseen functions and libraries. It offers a promising advancement in the field of automatic code generation by enhancing the adaptability and accuracy of LLMs in dynamic programming environments.

A similar approach is used the authors of (Li et al., 2023) who introduce SKCODER, a novel approach for automatic code generation that simulates how human developers reuse code. Instead of merely copying from a retrieved similar code snippet, SKCODER first extracts a code sketch—the relevant portions of the retrieved code most aligned with the given natural language requirements—and then refines this sketch to produce the final

solution. This two-step process mimics the real-world coding practice of identifying useful patterns from existing code and then tailoring them to new tasks.

The authors design SKCODER with three main components: a retriever to find a similar code snippet based on the given description, a sketcher to extract a structured, high-level "skeleton" from the snippet, and an editor that integrates the developer's requirements to modify this sketch into the desired code. Experiments on two established datasets and a newly collected large-scale Java dataset demonstrate that SKCODER significantly outperforms state-of-the-art code generation models, such as CodeT5 and REDCODER, in multiple metrics, including exact match and test-based correctness. The study further shows that the proposed sketch-based approach generalizes well to different neural architectures and results in code that is more accurate, of higher quality, and more maintainable than code produced by conventional copy-based or purely generative models. By effectively capturing and reusing relevant code patterns while filtering out irrelevant elements, SKCODER represents a substantial step forward in bridging the gap between automated code generation and human-like code reuse behavior.

2.3 LLMs

2.3.1 LLMs: Overview

(Minaee S. et al., 2012) provides a comprehensive survey of Large Language Models (LLMs), 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-2), 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, SQuAD, 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: a) exploration of new model architectures beyond attention, b) handling multi-modality (text, image, audio, video), c) enhancing reliability and reducing hallucinations, and, what is really important in the context of this Praxis, d) **development of smaller, more efficient models with similar capabilities as LLMs**.

(Zhao W. X. et al., 2023) and (Naveeda H. et al. 2024) contain other attempts at conducting surveys of LLMs in order to describe them based on various aspects. In continuation of the topic, (Han Z. 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 P. and Ayala O. M., 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 S. et al., 2024). (Huang Y. and Huang J. X., 2024), as well as (Hu Y. and Lu Y. 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 RALM's key components (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 Large Language Models for Code Generation (Jiang J., 2024.) offers a comprehensive review of the progress and capabilities of large language models (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.

In addition to describing state-of-the-art techniques and architectural choices—both encoder-decoder and decoder-only models—the survey provides a historical overview, illustrating how LLMs for code generation have evolved, often dramatically improving benchmark performance over time. It also discusses evaluation practices, including standard benchmarks like HumanEval and MBPP, newly proposed metrics, and human or LLM-based assessments, highlighting the complexity of assessing code quality, correctness, and broader software engineering attributes.

Importantly, the review identifies key research directions and open problems, such as ensuring that models can handle repository-scale contexts, developing more holistic evaluation metrics, improving data quality and diversity, and scaling to less-resourced

programming languages. The authors emphasize bridging the gap between academic approaches, which often focus on function-level tasks and standard benchmarks, and the practical demands of real-world software development, which require more complex reasoning, continuous learning, and robust, verifiable code solutions.
Overall, 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.
Overall, this ecosystem of text-to-code models shows rapid advancement and increasing sophistication. As these technologies evolve, they promise to streamline coding tasks, expedite development cycles, and fundamentally transform the way programmers interact with code.

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 B. 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 B. 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

(Nguyen C. V.) provide a structured overview of small language models (SLMs) and how they can be developed and optimized to operate efficiently under various constraints. 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. Overall, the paper consolidates current knowledge and offers guidance for

researchers aiming to build smaller yet effective language models that are easier to train, faster to run, and suitable for resource-constrained environments.

Another survey of SLMs is provided in (Wang F. et al., 2024). According to this study, LLMs have displayed striking capabilities across many tasks, but their sheer size and computational demands necessitate running them on powerful cloud infrastructure which can create issues with privacy, hinder real-time inference on edge devices, and drive up the costs of fine-tuning. Moreover, LLMs frequently underperform in specialized fields (e.g., healthcare, law) due to insufficient domain knowledge.

These challenges have led to growing interest in SLMs that use fewer parameters, delivering several advantages: they are cheaper to run, easier to customize, and can be efficiently deployed on resource-limited devices. They enable privacy preservation through local inference, minimal latency for applications needing rapid responses, and domain knowledge integration via cost-effective fine-tuning. Recent work has shown that SLMs can approach or even exceed the performance of much larger models. Additionally, they can be combined with retrieval-augmentation methods or instruction-tuned using carefully curated data to match or surpass larger counterparts, all while running locally with lower memory footprints.

This is a comprehensive survey that clarifies the definition of SLMs, details how best to obtain or train them, and reviews methods to enhance their performance, reliability, and adaptability. The survey also offers a taxonomy of existing models and methodologies, including compression techniques (pruning, quantization, low-rank factorization), knowledge distillation, and parameter-efficient fine-tuning approaches like LoRA.

Similar conclusions are made in (Lee L. 2024), specifically the fact that SLMs can outperform LLMs, they are typically deployed for a single specific task, and they are far less expensive, more efficient, higher performing and, often, more accurate than LLMs.

An example of using an SLM to replace an LLM is discussed in (Murallie T. 2024).

In (Ghosh, 2023) the authors state that while LLMs have demonstrated impressive capabilities, their massive size leads to drawbacks in efficiency, cost, and customizability. 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).

As if in unison, the author of (Mok K., 2023) discusses the growing trend of small language models (SLMs) as a viable alternative to large language models (LLMs). While LLMs like OpenAI's GPT-4 or Meta's LLaMA have demonstrated impressive capabilities in various NLP tasks—ranging from content generation and translation to advanced reasoning and conversation—they also come with significant downsides. Chief among these drawbacks are their enormous computational requirements and the immense energy, time, and cost needed for training and deployment. This complexity and expense can place these models out of reach for smaller organizations.

In contrast, SLMs, which are essentially scaled-down versions of LLMs, are gaining ground precisely because they mitigate these issues. Smaller models need fewer parameters, less training data, and can be fine-tuned more quickly, sometimes in minutes or a few hours rather than days. Their reduced size and simplified architectures not only

make them more resource-efficient and cost-effective, but also easier to implement on-site or on smaller devices. For organizations that prioritize privacy and security, SLMs present a simpler codebase with fewer potential vulnerabilities. Since SLMs can be tailored to more targeted, domain-specific tasks rather than broad, general-purpose applications, they can deliver better performance in niche scenarios without requiring the massive datasets needed for LLM training.

Although there may be trade-offs in terms of slightly diminished language processing capabilities, several recent SLMs are proving themselves to be on par with, or even surpass, larger counterparts in certain benchmarks. Examples include Microsoft's Phi-2, a 2.7-billion-parameter model that excels in mathematical reasoning, logic, and language understanding, and Orca 2, which was fine-tuned from LLaMA 2 on synthetic data to achieve enhanced reasoning abilities. The article also notes the existence of well-known smaller variants, like DistilBERT and the Mini/Tiny/Medium BERT models, as well as open source projects GPT-Neo and GPT-J, which further illustrate the promise of this leaner approach.

In the end, the rise of SLMs suggests an industry shift away from costly, resource-intensive LLMs toward smaller, more specialized models. This makes high-quality NLP technology more accessible and affordable, empowering more businesses and organizations to customize and incorporate advanced language AI into their workflows. As the field advances, it appears that "going small" may indeed be the most practical and cost-effective path forward.

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.

(Abbas H., 2024) discusses the emerging paradigm of small language models (SLMs) as a response to the resource-heavy approach of scaling large-scale language models. Traditionally, advancing NLP performance has centered on ever-larger models like GPT-3 and PaLM, which require significant computational resources and raise environmental concerns. In contrast, SLMs prioritize parameter efficiency and computational lean-ness, employing strategies such as knowledge distillation, low-rank factorization, and pruning to maintain competitive accuracy while shrinking model size. They also leverage parameter-efficient fine-tuning methods (e.g., LoRA, adapters) to customize for specific tasks without full retraining. As a result, SLMs often consume less energy, run at lower latency, and can be deployed on more modest hardware, making advanced NLP capabilities more widely available.

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.

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.

(Williams A. T. 2024) explores the growing popularity of locally hosted language models and small language models (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

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)

(Huang Z. et al., 2024) explore 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. Their work introduces a unified framework that represents various solution strategies—such as step-by-step reasoning, planning, memory retrieval, reflection, and external tool usage—through a single action space. They then propose a training strategy that allows the agent to

improve its ability to choose appropriate mechanisms on its own through self-exploration, rather than relying on curated expert data.

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 Evaluation of Generated Code

Evaluation of the generated code is a very important aspect of the automatic code generation process as it allows to evaluate the quality of the auto-generation process.

(Chen M. et al., 2021) introduces the **HumanEval** dataset which offers a relatively small set of hand-crafted programming tasks with hidden tests—useful for quick and controlled assessments. The **APPS** benchmark introduced in (Hendrycks D. et al., 2021) positions it as a more expansive and challenging alternative. Unlike HumanEval, which

presents a limited number of function-level problems with only 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. In essence, while HumanEval has established a baseline for evaluating code synthesis on simpler tasks, the APPS dataset pushes these evaluations toward richer, more authentic problem-solving scenarios that more thoroughly probe a model's true coding competence.

(Austin J. et al., 2021) investigate the capabilities of large language models (LLMs) to synthesize code in general-purpose programming languages, focusing on Python. Their work introduces two benchmarks: **MBPP**, a dataset of nearly one thousand entry-level programming tasks, and MathQA-Python, which contains tens of thousands of math-related coding questions. Each task is accompanied by a textual prompt and test cases that verify correctness. 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, allowing the largest models to produce correct solutions for most of the tested programming problems. 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.

(Miah T., Zhu H. 2024) proposes 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 also measure user experience factors like number of attempts and completion time to better reflect real-world usability. Applying this methodology, they find ChatGPT generally performs well for R programming tasks, though it struggles with more complex challenges. The results show that a user-centric perspective—focusing on how easily and effectively a developer can use the model to reach a workable solution—yields valuable insights that go beyond standard performance metrics.

(Dong Y. et al., 2024) introduces CodeScore, 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 large language models (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.

CodeScore achieves state-of-the-art performance by ensuring up to a 58.87% improvement in correlation with functional correctness compared to existing metrics. It operates three orders of magnitude faster than execution-based metrics by avoiding direct code execution. CodeScore is also versatile since it is applicable across diverse scenarios and outperforms baselines in datasets like APPS-Eval, MBPP-Eval, and HE-Eval.

(Du X. et al., 2024) conduct the first evaluation of large language models (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 across 11 state-of-the-art LLMs 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. Moreover, most LLMs perform better generating classes in a step-by-step manner rather than all at once, and they particularly struggle with generating methods that invoke other methods. The authors highlight the need for more comprehensive benchmarks and improved LLM strategies to handle long, complex, and contextually dependent code generation scenarios.

2.7 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).

The potential of SLMs was further uncovered in a recent discovery made by HuggingFace researchers. (Beeching E. et al., 2024.) discusses a 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.

References

McKinsey Report. 2023. 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.

Nguyen C. V., Xuan Shen, Ryan Aponte, Yu Xia, Samyadeep Basu, Zhengmian Hu, Jian Chen, Mihir Parmar, Sasidhar Kunapuli, Joe Barrow, Junda Wu, Ashish Singh, Yu Wang, Jiuxiang Gu, Franck Dernoncourt, Nesreen K. Ahmed, Nedim Lipka, Ruiyi Zhang, Xiang Chen, Tong Yu, Sungchul Kim, Hanieh Deilamsalehy, Namyong Park, Mike Rimer, Zhehao Zhang, Huanrui Yang, Ryan A. Rossi, Thien Huu Nguyen1. 2024. A Survey of Small Language Models.

Jiang J., Fan Wang, Sungju Kim, Naver Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation.

Ben Wodecki, 2023. AI Code Generation Models: The Big List. https://aibusiness.com/nlp/ai-code-generation-models-the-big-list

Almeida Y., Albuquerque D., Emanuel Dantas Filho, Felipe Muniz, Katyusco de Farias Santos, Mirko Perkusich, Hyggo Almeida, Angelo Perkusich. 2024. AICodeReview: Advancing code quality with AI-enhanced reviews.

Ottens L., Perez L., Viswanathan S. 2024. Automatic Code Generation using Pre-Trained Language Models.

Dong Y., Ding J., Jiang X., Li G., Li Zh., Jin Zh. 2024. CodeScore: Evaluating Code Generation by Learning Code Execution

Le T., Chen H., Babar A. B. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications and Challenges

Zhou S., Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, Graham Neubig. 2023. DocPrompting: Generating Code by Retrieving The Docs.

Du X., Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation.

Yetiştiren B., Işık Özsoy, Miray Ayerdem, Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT.

Reeves B., Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems with Small Prompt Variations.

Ciniselli M., Niccolò Puccinelli, Ketai Qiu, Luca Di Grazia. 2024. From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030.

Martinović B., Rozić R. 2024. Impact of AI Tools on Software Development Code Quality.

Li J., Zhi Jin, Yongmin Li, Yiyang Hao, Ge Li, Xing Hu 2023. SKCODER: A Sketch-based Approach for Automatic Code Generation.

Mok K. 2023. The Rise of Small Language Models.

Coutinho M., Lorena Marques, Anderson Santos, Marcio Dahia, Cesar França, Ronnie de Souza Santos. 2024. The Role of Generative AI in Software Development Productivity: A Pilot Case Study.

Minaee S., Mikolov T., Nikzad N., Chenaghlu M., Socher R., Amatriain X., Gao J. 2024. Large Language Models: A Survey.

Zhao W. X., Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie and Ji-Rong Wen. 2023. A Survey of Large Language Models.

Naveeda H., Asad Ullah Khana, Shi Qiub, Muhammad Saqibc, Saeed Anware, Muhammad Usmane, Naveed Akhtarg, Nick Barnesh, Ajmal Miani. 2024. A Comprehensive Overview of Large Language Models.

Han Z., Chao Gao, Jinyang Liu, Jeff (Jun) Zhang, Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey.

Béchard P., Ayala O. M. 2024. Reducing hallucination in structured outputs via Retrieval-Augmented Generation.

Yan S., Jia-Chen Gu, Yun Zhu, Zhen-Hua Ling. 2024. Corrective Retrieval Augmented Generation.

Huang Y., Huang J. X. 2024. A Survey on Retrieval-Augmented Text Generation for Large Language Models.

Wu K., Wu E., Zou J. 2024. How faithful are RAG models? Quantifying the tug-of-war between RAG and LLMs' internal prior.

Hu Y., Lu Y. 2024. RAG and RAU: A Survey on Retrieval-Augmented Language Model in Natural Language Processing.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, Michael Terry, Quoc Le, David Dohan, Ellen Jiang, Carrie Cai, Charles Sutton. 2021. Program Synthesis with Large Language Models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira, Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf 3 Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code.

Dan Hendrycks, Ethan Guo, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Horace He, Collin Burns, Dawn Song, Samir Puranik, Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS.

Miah T., Zhu H. 2024. Evaluation of ChatGPT Usability as A Code Generation Tool

Huang Z., Zhao J., Liu K. 2024. Towards Adaptive Mechanism Activation in Language Agent

Abbas H., 2024. How Small Language Models Are Redefining AI Efficiency. https://dev.to/hakeem/how-small-language-models-are-redefining-ai-efficiency-5dgo

Beeching E., Tunstall L., Rush S. 2024. Scaling Test Time Compute with Open Models. https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute .

Coding With SLMs and Local LLMs: Tips and Recommendations

Williams A. T. 2024. Small language models and local LLMs are increasingly popular with devs. We list the best models and provide tips for evaluation. https://thenewstack.io/coding-with-slms-and-local-llms-tips-and-recommendations/

Wang F., Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, Qi He, Yao Ma, Ming Huang, Suhang Wang. 2024. A Comprehensive Survey of Small Language Models in the Era of Large Language Models: Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness.

Lee L. 2024. Tiny Titans: How Small Language Models Outperform LLMs for Less. https://www.salesforce.com/blog/small-language-models/

Murallie T. 2024. I Fine-Tuned the Tiny Llama 3.2 1B to Replace GPT-4o. https://towardsdatascience.com/i-fine-tuned-the-tiny-llama-3-2-1b-to-replace-gpt-4o-7ce1e5619f3d

8