Artificial Intelligence-Based System for Boosting Automated Code Generation from Natural Language Descriptions

By Andrew Nedilko

M.S. in Computer Science, August 2018, University of Illinois at Urbana Champaign

A Praxis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Engineering

May 1, 2025

Praxis directed by

Dr. Kevin Abreu-Castellanos, D.Eng.
Professorial Lecturer in Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Andrew Nedilko has passed the Final Examination for the degree of Doctor of Engineering as of May 1, 2025. This is the final and approved form of the Praxis.

Artificial Intelligence-Based System for Boosting Automated Code Generation from Natural Language Descriptions

Andrew Nedilko

Praxis Research Committee:

Dr. Kevin Abreu-Castellanos, Professorial Lecturer in Engineering and Applied Science, Praxis Director

Last. M. First, Title of 2nd committee member, Praxis Co-Director

Last. M. First, Title of 3rd committee member, Committee Member

Dedication

The author wishes to Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur sem id massa aliquam, et scelerisque lacus finibus. Donec lacus orci, scelerisque at tincidunt at, suscipit vitae nulla. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Curabitur ut feugiat erat, in efficitur mi. Sed at placerat arcu. Nulla et dolor vel velit tincidunt ornare. Curabitur imperdiet tortor ligula, quis ultrices erat dapibus ac.

Mauris aliquet lorem sed nisi ullamcorper tincidunt. Morbi ornare pellentesque turpis, at tristique sapien elementum in. In tristique ipsum et neque efficitur sagittis. Nullam interdum pulvinar convallis. Nulla ac erat mollis, faucibus urna sed, finibus tellus. Aenean volutpat finibus egestas. Nam scelerisque mollis libero sed blandit. Integer vel massa augue. Nam non cursus ipsum. Aliquam varius sodales elit, in faucibus sem pharetra sit amet. Nam eget varius libero. Ut in fermentum lacus. Cras cursus non mi nec elementum. Suspendisse dapibus lectus nec congue lobortis. Praesent pellentesque erat nibh, eget varius nunc suscipit ut.

Acknowledgements

The author wishes to Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur sem id massa aliquam, et scelerisque

Abstract of Praxis

Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur sem id massa aliquam, et scelerisque lacus finibus. Donec lacus orci, scelerisque at tincidunt at, suscipit vitae nulla. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Curabitur ut feugiat erat, in efficitur mi. Sed at placerat arcu. Nulla et dolor vel velit tincidunt ornare. Curabitur imperdiet tortor ligula, quis ultrices erat dapibus ac.

Mauris aliquet lorem sed nisi ullamcorper tincidunt. Morbi ornare pellentesque turpis, at tristique sapien elementum in. In tristique ipsum et neque efficitur sagittis. Nullam interdum pulvinar convallis. Nulla ac erat mollis, faucibus urna sed, finibus tellus. Aenean volutpat finibus egestas. Nam scelerisque mollis libero sed blandit. Integer vel massa augue. Nam non cursus ipsum. Aliquam varius sodales elit, in faucibus sem pharetra sit amet. Nam eget varius libero. Ut in fermentum lacus. Cras cursus non mi nec elementum. Suspendisse dapibus lectus nec congue lobortis. Praesent pellentesque erat nibh, eget varius nunc suscipit ut.

Table of Contents

List of Figures

List of Tables

List of Symbols

x	State of the system
y	Output of the system
z	Noise

List of Acronyms

AI	Artificial Intelligence
LLM	Large Language Model
SLM	Small Language Model
HP	Hyperparameters
BERT	Bidirectional Encoder Representations from Transformers
GPT	Generative Pre-Trained Transformers
LoRA	Low-Rank Adaptation
RNN	Recurrent Neural Network
DL	Deep Learning
NLP	Natural Language Processing
RAG	Retrieval-Augmented Generation
CRAG	Corrective Retrieval-Augmented Generation
MBPP	Mostly Basic Python Problems (Dataset)
LBPP	Less Basic Python Problems (Dataset)
BLEU	Bilingual Evaluation Understudy (Method)

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

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

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

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

Another interesting study was conducted in (Ciniselli et al., 2024) where the authors envision how AI-driven assistance will reshape software developers' daily work by the year 2030. They compare current AI-based coding practices - exemplified by tools like GitHub Copilot and ChatGPT - with a future scenario in which developers rely on a hypothetical augmented tool, "HyperAssistant," for end-to-end support. The proposed

future assistant addresses a broad set of needs: it proactively manages developers' mental well-being, detects and fixes complex faults, streamlines code optimization and reuse, facilitates dynamic team collaboration, and offers personalized learning and skill development resources. By examining this transition from human-led coding toward orchestrating AI-driven ecosystems, the authors highlight how developers' roles may evolve, ultimately leading to more efficient, secure, and sustainable software engineering processes.

2.2 Automatic Code Generation (Before GPT)

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

CodeBERT, a transformer-based model, exemplifies the potential of pre-trained models in integrating natural language (NL) and programming language (PL) tasks. By training on paired NL-PL data (e.g., code snippets and documentation) and standalone code, it employs masked language modeling and replaced token detection to capture

semantic correspondences between NL descriptions and code functionality. CodeBERT excels in tasks like code search and documentation generation, producing accurate and informative outputs by leveraging its joint understanding of NL and PL. Its ability to generalize across multiple programming languages, including those unseen during training, highlights the promise of combining bimodal pre-training objectives with large-scale NL-PL resources. This paradigm sets a foundation for advancing code-related models with structural insights, advanced reasoning, and domain-specific customizations (Feng, Z. 2020).

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

(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 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). This work emphasizes the effectiveness of LLMs in streamlining software development processes.

(Ottens et al., 2024) explore 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 latter 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. This work emphasizes the potential of LLMs in automating software development tasks while improving efficiency and quality.

(Le 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.

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.

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

(Li et al., 2023) introduce SKCODER, a similar approach to automatic code generation that emulates human developers' practice of reusing code. Rather than simply copying similar code snippets, SKCODER extracts a high-level code sketch from retrieved snippets that align with natural language (NL) requirements and refines this sketch into a complete solution. The system consists of three components: a retriever to locate relevant code, a sketcher to create a structured skeleton, and an editor to adapt the sketch to the desired task. Experiments on multiple datasets, including a new large-scale Java dataset, show that

SKCODER outperforms models like CodeT5 and REDCODER in accuracy and quality metrics. Its sketch-based method generalizes well across architectures, producing more precise, maintainable, and functional code compared to copy-based or purely generative models, advancing automated code generation toward human-like code reuse.

2.3 LLMs

2.3.1 LLMs: Overview

(Minae et al., 2012) provide 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), 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 et al., 2023) and (Naveeda et al., 2024) discuss other attempts at conducting surveys of LLMs in order to describe them based on various aspects. In continuation of the topic, (Han et al., 2024) provides a comprehensive survey of parameter-efficient fine-tuning methodologies for LLMs. Another technique that is very widely used with LLMs is retrieval-augmented generation (RAG). RAG involves using a trained retriever to fetch relevant structured information and feed it into the LLM's prompt, thereby reducing hallucination and improving the quality and trustworthiness of the generated structured output (Béchard & Ayala, 2024). Corrective retrieval-augmented generation (CRAG) is a

retrieval-augmented framework that adaptively evaluates and corrects the retrieval process, leveraging large-scale web searches and selective re-composition of documents to ensure robust and improved output quality from LLMs (Yan et al., 2024). (Huang & Huang, 2024), as well as (Hu & Lu, 2024.) conduct a survey on RAG systems: the former focuses on organizing the RAG process into four key phases - pre-retrieval, retrieval, post-retrieval, and generation - to offer a detailed, retrieval-oriented perspective on their operation and development and the latter discusses 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 LLMs for Code Generation (Jiang et al., 2024.) offers a comprehensive review of the progress and capabilities of LLMs dedicated to code generation. It addresses the current gap in literature by systematically examining the complete lifecycle of LLMs for code-related tasks, from data preparation through advanced training and evaluation methodologies. The authors begin by outlining a taxonomy to structure recent developments, including how **large-scale code datasets** are curated and processed, how models are pre-trained and fine-tuned using diverse strategies, and what role instruction tuning, prompt engineering, and retrieval-augmented methods play. Special attention is given to novel frameworks that enable repository-level understanding, autonomous coding agents, and feedback-driven reinforcement learning to improve code correctness and adapt to real-world coding challenges.

The survey also discusses evaluation practices, including standard benchmarks like 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. This survey serves as a key reference for researchers and practitioners seeking a thorough understanding of the state-of-the-art in LLMs for code generation, pinpointing opportunities for future advancements and offering a structured roadmap for progressing toward more reliable, adaptable, and context-aware AI-driven coding assistants.

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

CodeT5 (Salesforce) and Polycoder (Carnegie Mellon University) represent research-driven efforts to enhance code understanding and generation, focusing on tasks like defect detection and code completion, while Polycoder also emphasizes openness and surpasses Codex in some niche cases. Commercial offerings such as Replit's Ghostwriter and Tabine

integrate into existing development workflows, providing autocomplete, code translation, and conversational interfaces that assist developers in real time.

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

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

tools, potentially offering educators an alternative assessment format less susceptible to student over-reliance on AI assistance.

2.4 SLMs

2.4.1 SLMs: Overview

(Nguyen et al., 2024) 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.

Other papers on SLMs: (Wang et al., 2024), (Lee 2024), (Ghosh 2023), (Mok 2023), (Szczygło 2024) (Morris et al., 2024), (Kili Technology Guide, 2024), (Abbas 2024) repeat the same very important advantages of SLMs as contrasted with LLMs. According to all of these studies, SLMs have emerged as a compelling alternative to LLMs, addressing the key challenges posed by LLMs' massive size and resource requirements. While LLMs like GPT-4 and LLaMA have demonstrated impressive capabilities in general-purpose tasks,

their computational demands raise issues related to cost, scalability, privacy, and real-time inference on edge devices. Furthermore, their broad focus often leads to insufficient domain specialization and suboptimal performance in fields like healthcare or law.

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

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

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

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

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

2.4.2. SLMs for Code Generation

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

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

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

such a division of labor can substantially boost performance on complex reasoning tasks (Juneja, G., 2024).

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

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

The study identifies the following models as great candidates for this task: Apple's OpenELM Family (set of small language models for mobile and local deployment),

DeepSeek V2.5, Qwen2.5-Coder-32B-Instruct (by Alibaba), Nxcode-CQ-7B-orpo (fine-tuned Qwen model optimized for simpler coding tasks), OpenCodeInterpreter-DS-33B, Artigenz-Coder-DS-6.7B. Benchmarks and evaluation tools discussed include HumanEval, MBPP, BigCodeBench, LiveCodeBench, EvoEval.

2.5 Agents

2.5.1 Agents: Overview

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

(Xi et al., 2023) survey recent advances in LLM-based AI agents, arguing that LLMs—with their strong language understanding, reasoning, and planning capabilities—can serve

as a robust “brain” for intelligent agents. The authors propose a three-part framework: an LLM-based cognitive core (“brain”), a perception module for ingesting multimodal inputs, and an action module for complex outputs such as tool usage or environmental manipulation. (Masterman et al., 2024) also present a thorough overview of recent AI agent architectures that build on LLMs to achieve complex tasks involving intricate reasoning, planning, and external tool usage.

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

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

(Huang et al., 2024) explores 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. Various solution strategies include step-by-step reasoning, planning, memory retrieval, reflection, and external tool usage.

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

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

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

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

When it comes to evaluation of LLM-based dialog agents, (Wason et al., 2024) provide a critical examination of LLM-based dialogue agents, arguing that their real-world success depends not only on technical advancements—such as improved training pipelines and

state management techniques—but also on responsible design, thorough validation, and ongoing refinement of their prompting strategies and underlying models. This work thus positions LLM-based dialogue agents as promising yet still maturing tools in domains like customer support, virtual assistance, and automatic code generation, each with its own set of unique challenges and performance criteria.

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

2.5.2 Agents for Code Generation

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

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

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

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

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

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

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

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

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

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

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

2.6 Evaluation of Generated Code

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

(Chen et al., 2021) introduce 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 with a limited number of function-level problems and a few test cases each, APPS comprises thousands of more complex and varied coding problems sourced from real coding competitions, each backed by extensive and diverse test inputs.

(Austin et al., 2021) investigate the capabilities of 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 containing tens of thousands of math-related coding questions. They examine both few-shot prompting—providing only a handful of examples—and fine-tuning on a small subset of tasks. Their findings show that performance on code generation improves substantially as model size increases, and that fine-tuning further boosts accuracy. Also, their analysis reveals that models struggle with deeper program "understanding," as evidenced by poor results on tasks requiring them to predict code outputs given specific inputs.

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

(Dong 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 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.

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

Some of the **common problems with current evaluation datasets** are described in (Gao L., 2023). They include such facts as the data can be contaminated which means that both the HumanEval and MBPP datasets have been discussed very frequently online, and the latest language models most probably "saw" the test data from these datasets during training which may invalidate the evaluation results. Another problem is that the datasets contain coding questions that are too trivial and don't reflect real engineering challenges. Also, the unit tests that come with these datasets may be too weak and may need improvement.

The authors of (Matton et al., 2024) raise a similar concern - data leakage in code generation which occurs when popular evaluation benchmarks (like HumanEval and MBPP) appear in a model's training data—whether intentionally or unintentionally—thereby compromising the validity of test scores as measures of this model's generalization. This contamination can arise through direct inclusion of test examples in the training corpus, via synthetic data creation pipelines that inadvertently reproduce evaluation samples, or through overfitting models to a narrow set of public benchmarks during

checkpoint selection. As a result, reported improvements on these benchmarks may reflect memorization or over-optimization rather than genuinely improved coding capability. To address these challenges, the authors introduce a set of 161 Less Basic Python Problems (LBPP), a new Python code generation benchmark designed to avoid overlap with existing training data and provide a more trustworthy measure of code generation performance.

2.7 Conclusion

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

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

The core idea is to use dynamic inference strategies, such as iterative self-refinement or verifier-based search methods, to guide a model toward correct answers. While large models rely on their vast parameters for accuracy, small models can offset this disadvantage by systematically applying more reasoning steps and better filtering mechanisms at test-time. Crucially, these methods show that tiny 1B and 3B-parameter

models can outperform models as large as 70B parameters if given enough “time to think” - that is, enough test-time search and verification cycles. This opens the door to resource-efficient LLM deployments where you don’t need massive compute for training; instead, you invest your compute at inference time, unlocking high performance from much smaller models.

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

Chapter 3 — Methodology (16 pages)

3.1 Introduction

This chapter outlines the methodology employed in evaluating the performance of small language models (SLMs) on code generation tasks. We describe the platforms and tools used for conducting the experiments, the selection of appropriate datasets for LLM fine-tuning and evaluation, and the specific hyperparameters tuned for optimizing inference accuracy. We also discuss prompt engineering strategies that shaped the model outputs and methods for measuring SLM performance.

Additionally, the chapter presents an overview of the models evaluated, how they were integrated within the execution framework, and how agent-based methods were integrated for iterative code refinement. Through this methodology, we aim to establish a reproducible, efficient, and comprehensive framework for assessing SLM capabilities in practical, code-centric tasks.

3.2 Experimentation Platforms

We ran our experiments using several different platforms.

3.2.1. Google Colab

We ran a series of Jupyter notebooks to experiment with small language models (SLMs) in the form of HuggingFace transformers. For this, we utilized Google Colab Pro+, a reliable, high-performance cloud-based Jupyter notebook subscription tier that builds on the free version of Google Colab by providing more robust resources for computationally intensive tasks: longer runtimes, increased memory, dedicated compute, accelerated GPU access, including NVIDIA K80, P100, and T4 instances. A100 GPU instances provide the most optimal runtimes.

This service offers faster GPU assignment and higher priority for resource allocation, making it less likely for sessions to be interrupted during peak usage, high-memory runtimes - often up to 52GB of RAM - making it suitable for large-scale data analysis or deep learning workloads. Colab Pro+ sessions can remain active for up to 24 hours, even if the browser is closed, allowing for extended training jobs or prolonged analysis without manual intervention.

3.2.2. Personal PC Using API for Hosted Models

On the other hand, we ran another series of Jupyter notebooks locally using API calls to SLMs hosted in the cloud.

Mistral AI is a next-generation AI platform built on advanced transformer architectures, including the 7B and 8x7B Mistral models under a permissive license with varying context windows from 8K to 32K tokens. The platform is known for its flexibility, highlighted by customizable deployments and straightforward integration via an API. Using this powerful, scalable cutting-edge language technology, we combined the efficient transformer models and a developer-friendly approach. We built a code generation solution that worked faster than transformer models in Google Colab since SLMs were already hosted and we were calling them via an API.

Another platform, Replicate, facilitates easy access to cutting-edge AI models and is a cloud-based AI model hosting and inference platform designed for running, fine-tuning, and deploying machine learning models at scale through a simple and efficient API. The platform dynamically adjusts compute resources based on demand. This makes it highly scalable while being cost-efficient, where users only pay for the compute usage as required. Replicate hosts thousands of community-contributed models for image, text,

speech, and music generation which can be invoked on-demand, including the ones that we used for code generation: Nous-hermes-2-solar-10.7b, Phixtral-2x2_8, Qwen1.5-7b, Llama 3 8B, Gemma 7B, Gemma 2B, and others.

3.2.3. Describe HPC if I end up using it (Placeholder)

3.3 Small Language Models Used in Experiments

In this experiment, a broad selection of small language models (SLMs) was evaluated on multiple code-generation benchmarks (HumanEval, MBPP, LBPP, and BigCodeBench). Model sizes ranged from ~2.8B to ~22B parameters, including Nxcode-CQ-7B-orpo (~7B), Codestral Mamba (~7.3B), various Mistral models (3B, 7B, 8B, 12B, 22B), and others like Deepseek-Coder-6.7B and CodeQwen1.5-7B. Each model's version and commit number were tracked for versioning completeness, crucial for reproducibility in real-world experiments.

Inference parameters, such as temperature and top_p settings, were optimized for the models based on their performance, with a default setting of 1.0 for both parameters, and subsequent experiments with lower values (e.g., 0.3 for temperature) to control output diversity and reduce hallucinations.

The output from all models underwent minimal post-processing: code fences and triple backticks were removed, and missing import statements (e.g., from typing import List) were reinserted if the model omitted them. See Table 1 below for a list of SLMs used in experiments.

Table 1. Small Language Models Used in Experiments

Model	Hosted By	Model Size	Temp / top_p	Estimated cost, $. (per experiment)
Small Language Models (SLMs)				
Nxcode-CQ-7B-orpo	Google Colab	7.25B	1.0 / 1.0	$50/month

Model	Hosted By	Model Size	Temp / top_p	Estimated cost, $. (per experiment)
Codestral Mamba	mistral.ai	7.3B	0.7 / 1.0	0.02
Ministral 8B Instruct	mistral.ai	8B	0.3 / 1.0	0.01
Deepseek-Coder-6.7B-Instruct	Google Colab		1.0 / 1.0	$50/m
Ministral 3B Instruct	mistral.ai	3B	0.3 / 1.0	0.01
Mistral-Nemo-Instruct-2407	mistral.ai	12B	0.3 / 1.0	0.01
Llama 3.1 8B Instruct	Google Colab	8B		
CodeQwen1.5-7B-Chat	Google Colab		1.0 / 1.0	$50/m
OpenCodeInterpreter-DS-6.7B	Google Colab		1.0 / 1.0	$50/m
Mistral 7B, open-mistral-7b	mistral.ai	7B	0.7 / 1.0	0.01
Nous-hermes-2-solar-10.7b	replicate.com	10.7B	0.95 / 1	0.61
Phixtral-2x2_8 (4.5B)	replicate.com	4.5B	0.95 / 1	2.77
Artigenz-Coder-DS-6.7B	Google Colab		1.0 / 1.0	$50/m.
Code Gemma 7b IT	Google Colab	7B	1.0 / 1.0	
Slightly Bigger SLMs				
Mistral-Small-2409	mistral.ai	22B	0.7 / 1.0	0.03
Codestral latest	mistral.ai	22.2B	0.7 / 1.0	0.15
Mixtral-8x7B-v0.1	mistral.ai	12B active (47B total)	0.7 / 1.0	0.05
SLMs That Were Not Found Useful				
Qwen1.5-7b	replicate.com	7B	0.95 / 1	3.55
Llama 3 8B	Replicate	8B	0.95 / 1	0.29
Gemma 7B	replicate.com	7B	0.95 / 1	0.05
Gemma 2B	replicate.com	2B	0.95 / 1	0.05
Flan-T5	replicate.com		0.95 / 1	
Phi-2	replicate.com		0.95 / 1	
Mamba 2.8B	replicate.com	2.8B	0.95 / 1	0.02 (20 calls)

3.4 Test Datasets

The scope of this praxis is limited to the Python programming language. We used four different datasets to make a comprehensive evaluation of the code generation

capabilities of SLMs. First, we used two popular Python benchmark datasets: HumanEval and MBPP.

The HumanEval dataset is a benchmark created by OpenAI specifically for assessing the code generation capabilities of large language models (LLMs) (Chen et al., 2021). It comprises 164 carefully hand-crafted Python programming problems, each having a function signature, a descriptive docstring, and a set of unit tests (averaging 7.7 tests per problem) to verify functional correctness. Each generated solution is run against the provided unit tests to determine its correctness. This design emphasizes not only syntactic accuracy but also the semantic and logical correctness of the generated code, making it a robust measure of an LLM's ability to understand natural language descriptions and translate them into executable code. This integrated evaluation framework provides a reproducible and scalable methodology to benchmark LLMs on practical programming challenges.

The Mostly Basic Programming Problems (MBPP) dataset is a benchmark for evaluating program synthesis capabilities in large language models (LLMs). It comprises 974 self-contained programming tasks designed to be solvable by entry-level programmers. Each task includes a concise natural language description, a canonical function signature, and a reference solution that passes three assert-based test cases. The dataset's problems span simple numeric manipulations, list processing, and string operations, with an average of 6.8 lines of code per solution and a median of 5 lines.

The dataset is partitioned into distinct subsets: a small held-out set (10 problems) for use as examples in few-shot prompts, 500 tasks for testing, and 374 examples for fine-tuning LLMs and 180 examples for validation during fine-tuning. In our research for this

Praxis, we used the 500 testing data points to evaluate the code generated by SLMs. The dataset design ensures that LLMs are evaluated not merely on their ability to generate syntactically correct code, but also on their capability to capture the precise semantics implied by the natural language description.

In response to concerns raised by the authors of (Matton et al., 2024), namely data contamination and data leakage, which we touched upon in Section 2.6 of this Praxis, we also used the Less Basic Python Problems (LBPP) which is considered to be a more objective and trustworthy measure of code generation performance.

Data leakage through advertent or inadvertent inclusion of the popular evaluation benchmarks like HumanEval and MBPP into other models' training data compromises the validity of test scores for those model. This can happen through direct inclusion of test examples in the training corpus, via synthetic data creation reproducing evaluation samples, or through overfitting models on the public benchmarks during checkpoint selection. As a result, the reported improvements on these benchmarks may reflect memorization or over-optimization rather than genuinely improved coding capability.

To address these concerns, the paper introduces LBPP (Less Basic Python Problems), an uncontaminated benchmark comprising 161 carefully curated coding tasks and associated Python solutions. It's a new Python code generation benchmark designed to avoid overlap with existing training data and provide a more trustworthy measure of code generation performance. LBPP is designed to be more challenging and initial results show that state-of-the-art models, which perform strongly on HumanEval and MBPP, suffer significant performance drops on LBPP—highlighting the urgent need for cleaner, more robust evaluation benchmarks in code generation research.

Another less known dataset that we used in an attempt to prevent the data leakage issue was the BigCodeBench dataset. The dataset comprises 1,140 finely curated tasks divided into two distinct variants - code completion based on detailed structured docstrings and instruction-driven code generation from natural language prompts. Each task includes comprehensive prompts (both complete and instructive), canonical solutions, code-only prompts, and unit tests (averaging 5.6 per task) with an overall branch coverage of approximately 99%. Each task is precisely defined and evaluation metrics reflect both syntactic correctness and functional accuracy.

BigCodeBench emphasizes the diversity and complexity of real-world programming challenges by incorporating function calls from 139 libraries across 7 distinct domains. This level of detail mirrors authentic software development scenarios and facilitates an end-to-end evaluation framework for LLMs, pushing them to demonstrate robust compositional reasoning and precise tool-use capabilities.

BigCodeBench aims to advance the responsible development of code-centric AI. It focuses on practical challenges that demand both code synthesis and correct usage of external libraries, going well beyond standard function-level benchmarks like HumanEval. As such, BigCodeBench is a key resource for measuring and improving LLM's capabilities in real-world programming contexts.

Other coding datasets that we explored, but decided not to use include:

- **LiveCodeBench** has a huge size (~10GB) and an excruciating number of test cases: some don't even fit into a Jupyter Notebook cell causing output errors. Private test cases are scrambled and look unusable.

- **Taco** requires 5GB on disk; there are no test cases per se, but only inputs and expected outputs. Since they come from different sources, it may take time to parse them properly and write assert statements or test cases for them.

3.4 Prompt Engineering

We utilized zero-shot prompts to generate code completions - an LLM was asked to generate Python code based on a coding task description, a function header or a docstring describing what the function does. Our initial prompt was simple: "Complete the following code." Section 4 of (Austin et al., 2023) and (Google Research, 2023) describe a three-shot prompt strategy for the MBPP dataset where Task IDs 2, 3, and 4 were used as examples provided in the prompt. This was outside of our scope, but it can be an interesting continuation of our work.

After running experiments using the above simple prompt, we noticed that SLMs include a lot of extraneous text into their output with the intention to clarify the generated code. Therefore, we had to modify the prompt to ask SLMs to output only the runnable code and nothing else by specifically advising SLMs not to output any extraneous content other than runnable code. With this approach, besides the ability to instruct how to generate code, the prompts also measured the ability of SLMs to follow instructions. The following is an example of a more detailed prompt that we used for the HumanEval and the BigCodeBench datasets:

"'1. Complete the following starter code consisting of a function signature and docstring.

2. Your output must include only the function body with proper Python code and indentation.

3. Do not include phrases like “Completion:” or any other headings, do not show example usage, do not show test cases, do not provide explanatory text.

4. Your output must be with no additional text, no extra symbols, no code fences (```), and no other content before or after the function body except for import statements.

5. Stop generating immediately after the return statement or final line of the function body.”

Since both MBPP and LBPP datasets contain specific test cases to be satisfied, we used this extended prompt:

“1. You are an expert Python programmer, and here is your task: { task description }.

2. Your output must include only the correct import statements, the function with proper Python code, and nothing after it. Stop generating immediately after the return statement or final line of the function.

3. Your output should be runnable Python code. Do not include phrases like "Solution:" or any other headings, do not show example usage, do not show test cases, do not provide explanatory text.

4. Your output must be with no additional text, no extra symbols, and no code fences (```).

5. Your code must satisfy these tests: {test cases}”

Surprisingly, not all SLMs were able to follow these instructions (see Section 4 Results). Some of them followed the instructions quite well, but some did this only in a percentage of cases. There were models where this percentage was very low. SLMs tend to output additional explanations and clarifications like: “Here is the requested code completion:” etc. which breaks the automatic code execution during the verification stage. Adding more specific instructions like: “Output only the runnable code and nothing else:” would still lead to non-runnable content like triple backticks in the output. To mitigate this,

we provided some very light general assistance to SLMs by removing the initial or trailing code fences and triple backticks or by adding missing import statements like “from typing import List”, removing lines starting with assert statements and “print(this_function())“ test cases because the models hallucinated them.

Having witnessed a lot of deviations from the instructions provided in the prompts, we assumed a general approach to evaluating all SLMs: create one extensive and comprehensive prompt for all models; if any model fails to fully follow it and still outputs human phrases or non-runnable content, it should be considered the drawback of the model due to its poor ability for instruction following.

One interesting observation led to a modification in our prompting strategy - SLMs are inclined very much to put the generated code inside the code fences: ```python … ```. At first, we tried to strictly prohibit this by including into the prompt the corresponding instructions to exclude the code fences and asking the model to output the clean code, but it didn’t help – the models would still include the code fences no matter what. And this was true for several different models, not just one or two. Therefore, we decided to use this weakness of SLMs to our advantage – we encouraged SLMs to output the proper code inside the code fences, and the rest of the output outside of them. This made parsing the code from the SLM output more straightforward.

3.5 Measuring Performance

To evaluate performance across all four datasets, we relied on the pass@1 metric, the probability that at least one of the top k generated code samples correctly passes all tests; in other words, it’s the percentage of passed test cases when the model had only one chance to generate the code based on a natural language description of the task. We utilized

the implementation by OpenAI presented in (Chen M., 2021). The OpenAI code didn't work out of the box, so we had to modify it as described in (Nedilko A., 2024). In addition, the same code was modified for use with all other datasets, and not just HumanEval, which required writing additional evaluation functions for each dataset (Nedilko A., 2024). Also, because the Replicate API was not directly compatible with the default script, we integrated the Replicate client through LangChain, which facilitated seamless interaction with multiple SLMs.

Although there is a more general framework available for the evaluation of code generated based on the HumanEval and MBPP datasets called *bigcode-evaluation-harness* (Loubna B. A., 2022), it is designed to work with HuggingFace models, and we didn't find an easy way to adapt it to work with any model or any agent-based application. On the contrary, the modified OpenAI evaluation code mentioned above can be used with any model or agent with some slight modifications for additional datasets.

3.6 Tunable Model Hyperparameters for Optimized Inference

Below, we outline key hyperparameters used to optimize inference across the small language models (SLMs) discussed, such as transformer models Nxcode-CQ-7B-orpo, Mistral 7B/8B, Deepseek-Coder-6.7B, CodeQwen1.5-7B, and others. The content of this section is based on the following sources: OpenAI API Reference 2025, Siddique 2023, Sadani 2023, Cohere Team 2022. While **temperature** and **top_p** are typically the most influential hyperparameters for controlling code style and correctness, there are also other parameters that may be very important for maximizing functional correctness.

1. **Temperature**

Governs the randomness of next-token sampling. A higher temperature (e.g., 1.0–1.2) increases output variability, whereas a lower value (e.g., 0.2–0.5) makes the model more deterministic and can mitigate hallucinations and extraneous text, particularly in coding tasks.

2. **Top-p (Nucleus Sampling)**

Controls the cumulative probability threshold (usually between 0.8 and 1.0) for token sampling, effectively limiting the token distribution to the highest-probability subset. Setting top_p below 1.0 prunes the "long tail" of unlikely tokens. This can yield more coherent outputs and reduce random code fragments, but if set too low (e.g., 0.5), the coverage of probable tokens may diminish which can impact functional correctness.

3. **Top-k**

Restricts sampling to the top k most probable tokens at each decoding step. It is often used in conjunction with top_p or temperature.

4. **Repetition Penalty**

A multiplicative factor applied to token logits to discourage repeated tokens. It is useful in code generation where repeated lines or duplicated function signatures may occur.

5. **Max New Tokens**

Specifies the maximum number of tokens generated in a single inference call. In code generation tasks, limiting output length can prevent overly verbose completions. However, it should be sufficiently large to accommodate full function definitions and docstrings.

6. **Presence or Frequency Penalty**

Penalizes tokens that already appeared in the text, encouraging the model to introduce new tokens. It helps mitigate repeated code segments, but if set too high, it may cause the model to omit necessary repeated keywords in code.

7. **Context Truncation / Truncation Strategy**

The number of tokens from the prompt or conversation context that the model can access to ensure the context window is not inadvertently truncated.

8. Inference Optimization

We will use the following approach when tuning model parameters: start with default values for temperature (τ=1.0) and top_p = 1.0 (or top_p = 0.9) to gauge the baseline performance, then iteratively lower temperature (e.g., to 0.5) if the model produces irrelevant text or incomplete code.

Next, if the model still exhibits excessive variability, we will adjust top_p from 1.0 down to 0.7 and apply a mild repetition penalty (in the range of 1.1–1.2) – care must be taken not to harm legitimate repeated tokens.

3.7 Fine-Tuning SLMs for Code Generation

Small language models (SLMs) can have significant gains in code-generation quality when they undergo fine-tuning on relevant, high-quality code datasets (Napalkova 2024). By exposing the model to domain-specific syntax, idioms, and problem-solving patterns, fine-tuning aligns the model's parameters more closely with the target domain. This alignment helps reduce hallucinations and ensures that the output code follows common language constructs, libraries, and design patterns frequently encountered in practical tasks.

Another key advantage of fine-tuning is the improvement in prompt sensitivity and instruction following. Datasets specifically designed for code instructions enable the model to learn how to respond accurately to step-by-step prompts. As a result, SLMs are better equipped to parse problem statements, reason about function requirements, and produce well-structured, executable solutions without superfluous commentary (Napalkova 2024).

Additionally, fine-tuning on curated code corpora ensures higher correctness by filtering out incomplete or erroneous snippets. When combined with advanced techniques such as Low-Rank Adaptation (LoRA) or prefix tuning, models can achieve these improvements even in resource-constrained environments (e.g., single-GPU setups), reducing both memory footprint and training time (Khalusova 2025).

Finally, fine-tuning offers a practical pathway to incorporate specialized libraries, frameworks, or coding styles relevant to a particular domain. For instance, a model focused on web development might benefit from fine-tuning on datasets emphasizing HTTP requests, server-side logic, or specific JavaScript libraries. This targeted adaptation ultimately expands the model's range of competency, enabling it to generate code that is not just syntactically valid, but also domain-aware and closely aligned with real-world development practices (Wang & Rishi 2023).

There are several datasets that can be used for SLM fine-tuning on the code generation task specifically, and there is a training set available as part of the MBPP dataset too. Below, we discuss several approaches—each leveraging the provided datasets—to optimize model performance while balancing computational cost and training complexity.

3.7.1. Supervised Fine-Tuning on Curated Code Datasets

This method should be used with high-quality code samples. For instance, datasets like **Tested-143k-Python-Alpaca** or **Magicoder-Evol-Instruct-110K** (HuggingFace 2024) already contain Python code that has been tested or decontaminated. Once fine-tuned, the model will learn correct syntax, library usage, and best practices.

3.7.2. Instruction Tuning for Enhanced Code Completions

This is especially relevant for tasks requiring step-by-step problem solving or explanation. Datasets like **CodeFeedback-Filtered-Instruction** and **Just-write-the-code-Python-GenAI-143k** (HuggingFace 2024) contain prompts that mimic real user instructions for code generation. Fine-tuning on these resources helps SLMs learn the prompt–response alignment and iterative refinement.

3.8 Agents

In addition to using SLMs directly for single-pass code generation, we explored agentic workflows that enable iterative improvement and collaboration. Two main approaches were tested: (1) a Reflection framework, where a single agent revisits and refines its output, and (2) a Multi-Agent Collaboration framework, where multiple specialized agents coordinate to improve code correctness by verifying imports, stripping extraneous text, removing leftover test statements, and introducing other optimizations.

These agentic methods were hosted in the same manner as the SLMs described earlier—some were run via Google Colab notebooks, while others utilized cloud-hosted APIs (e.g. Mistral AI, Replicate). The baseline performance of SLMs without agentic intervention serves as a reference point for assessing any gains achieved through these iterative or collaborative techniques.

Beyond calling a language model, agentic workflows can incorporate memory, multiple tool integrations, and multi-step planning. This allows the system to automatically refine code or remove extraneous content (like leftover human-style clarifications or debug statements). By adopting these agentic workflows, we aim to reduce error rates, enhance code validity, and boost the model performance in code generation tasks.

3.8.1 Reflection Agentic Workflow

Reflection agents (sometimes spelled as reflexion agents) implement a straightforward yet effective approach to iterative code refinement (Galileo 2024). Conceptually, a single agent produces an initial code solution, then immediately revisits that solution to identify and correct errors or improve structure - akin to an author proofreading and editing their own text. The reflection approach can result in substantial improvements over a baseline performance (e.g. for coding, reasoning, etc.). For instance, researchers achieved a 91% pass@1 rate on HumanEval using reflection, which is considerably better than the previous best result by GPT-4 of 80% (Shinn 2023). We used the following architecture:

a) Initial Code Generation Prompt

- The agent is given a problem statement – a description of the coding task with preliminary inputs (e.g. a function header, docstring, etc.).
- It generates a first-pass solution in code form.

b) Reflection Prompt

- The agent receives its own code output, along with an instruction to optimize it: fill missing imports, remove extraneous text, fix logic errors, etc.
- The agent then outputs a revised solution, presumably more accurate.

In our implementation, we tested a plain reflection setup where each code-generation attempt uses two prompts. Then we attempted a LangChain-based architecture.

3.8.2 Multi-Agent Collaboration Agentic Workflow

The multi-agent collaboration approach generalizes the idea of iterative refinement by introducing multiple specialized agents that coordinate to improve the generated code (Talebirad & Nadiri 2023, Galileo 2024). We came up with this idea as a natural continuation of our implementation of the reflection agentic workflow, when we reviewed the most typical errors made by reflection agents and saw an opportunity to achieve improved results through the use of multiple specialized agents. Examples of somewhat similar systems highlighting the potential of multi-agent systems for iterative testing, optimization, and debugging in code generation tasks are described in (Huang et al., 2024) and (Islam et al., 2024) as the AgentCoder and MapCoder frameworks, respectively.

In our version of multi-agent collaboration, we orchestrate a sequence of agents, each responsible for a specific cleanup or verification step:

- **Import checker agent** - ensures necessary imports (e.g., from typing import List) are included if missing.
- **Extraneous text remover agent** - strips out any human-like explanation lines or system messages the SLM might have inserted.
- **Code fence / test removal agent** - removes triple backticks, leftover test statements, or any assert lines that could interfere with the final code snippet.
- **Miscellaneous correction agents** - additional specialized modules that handle corner cases.

For each of the two agentic workflows, we developed a “plain” version using a sequence of direct API calls to an LLM and a LangChain-based version of the agent to simplify chaining prompts and storing partial outputs (LangChain 2025). LangChain is a development framework that simplifies building language model applications by chaining prompts, managing context and memory, and integrating external tools. While LangGraph is a framework that organizes multi-agent workflows using graph-based structures to manage complex, branched interactions and robust error handling.

In these frameworks, the developers can define each agent as a step in a LangChain pipeline or a node in a LangGraph system (AI-Agent-Dev 2025). For instance, the output from the import checker feeds into the extraneous text remover, which in turn feeds the code fence remover, culminating in a final refined snippet. This architecture is flexible enough to incorporate other modules - like a syntax validator or an additional reflection step - should more sophisticated corrections be needed.

LangChain and LangGraph streamline the development of LLM-based agent systems by providing:

- **Prompt templates**: for reusing prompts.
- **Memory, state, and context management**: to keep track of the model’s first output so it can be fed back as context, so that each agent step can automatically access or update relevant conversation history and code output.
- **Tool integration**: agents can call external APIs, test code, or parse partial solutions without manual orchestration.
- **Graph-based workflows**: complex multi-agent flows can be organized into graphs, enabling branching logic and robust error handling.

- **Logging & debugging**: automated tracking of intermediate steps, making it easier to observe how the reflection agent modifies its code.

Hence, employing LangChain for reflection and multi-agent collaboration simplifies the creation, debugging, and scaling of code-generation workflows that rely on iterative improvements (AI-Agent-Dev 2025).

Chapter 4—Results (28 pages)

4.1 Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur sem id massa aliquam, et scelerisque lacus finibus. Donec lacus orci, scelerisque at tincidunt at, suscipit vitae nulla. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Curabitur ut feugiat erat, in efficitur mi. Sed at placerat arcu. Nulla et dolor vel velit tincidunt ornare. Curabitur imperdiet tortor ligula, quis ultrices erat dapibus ac.

4.2 Another Section

One single round for three prompts with minimal code cleaning (parse from code fences only)

Another single round for three prompts with full code cleaning – how how code cleaning is important, that the models almost do it well, but just a few error cases lower their score because these few cases happen in many test cases.

Hypothesis 3: Same for agents – one simple and one full-code-cleaning round for each of the two agents. Demonstrate the effect of agents, especially for more advanced models is equal or exceeds the code cleaning effort.

Hypothesis 2: Select a subset of models for temperature and top_p experiments (three values each).

Hypothesis 1: Select a around 5 models, fine-tune on the MBPP train set and retest on the test set – should be better results.

(Fox, 2012).

4.3 Leaderboards

Known official baselines hosted publicly. If Chapter 3 is short, include them here. Otherwise, include them in Chapter 4 with the actual results.

4.4 Model Results

Some SLMs struggled with instruction adherence, producing extraneous text or incomplete code, which lowered their pass@1 scores. Nxcode-CQ-7B-orpo emerged as a top performer (82–87% on HumanEval), while Ministral 8B and Deepseek-Coder-6.7B also showed strong results. Conversely, certain SLMs provided near-zero performance or frequent API failures, rendering them unsuitable for subsequent agent-based experiments.

Latency differed greatly: Nxcode, Mistral, and Code Gemma often returned completions in under a minute, whereas Qwen1.5-7B, Phixtral, and others could take 200–300 seconds per call. These slow latencies significantly hindered model scaling for datasets with large numbers of code generation test cases, such as MBPP, which contains around 500 samples.

Curious fact: I had a short question for each model before starting a long code generation session: “What is the capital of California?”

Every model answered correctly except for “deepseek-coder-6.7b-instruct” whose answer was: “I'm sorry, but as an AI Programming Assistant, I'm specialized in answering questions related to computer science. I'm not equipped to provide information about geography or other non-computer science topics.” Seems like it’s a very specialized model.

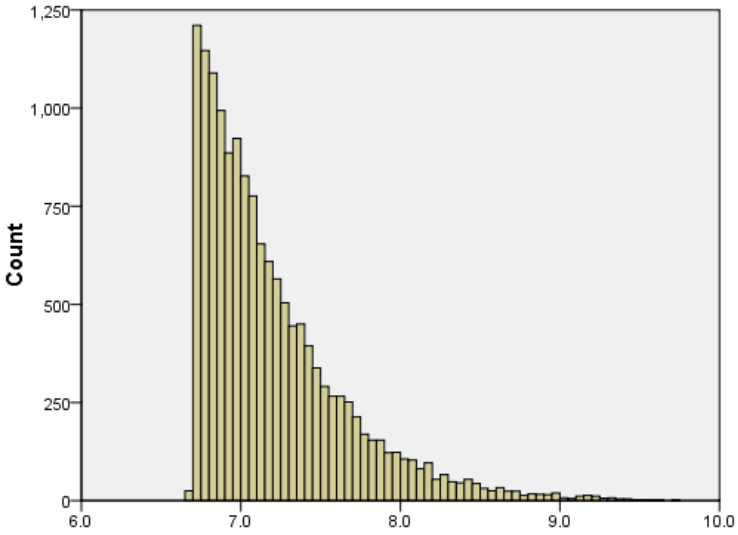

Figure 4-1. Histogram of XYZ.

Table 4-1. Pearson Correlations Between W and T

X	Correlation Coefficient	r^2	*df*	*p*
Cost	0.55	.45	200	< .001
Schedule	0.44	.45	234	0.04

Note. Add some note here.

Chapter 5—Discussion and Conclusions (3 pages)

5.1 Discussion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur sem id massa aliquam, et scelerisque lacus finibus. Donec lacus orci, scelerisque at tincidunt at, suscipit vitae nulla. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Curabitur ut feugiat erat, in efficitur mi. Sed at placerat arcu. Nulla et dolor vel velit tincidunt ornare. Curabitur imperdiet tortor ligula, quis ultrices erat dapibus ac.

Mauris aliquet lorem sed nisi ullamcorper tincidunt. Morbi ornare pellentesque turpis, at tristique sapien elementum in. In tristique ipsum et neque efficitur sagittis. Nullam interdum pulvinar convallis. Nulla ac erat mollis, faucibus urna sed, finibus tellus. Aenean volutpat finibus egestas. Nam scelerisque mollis libero sed blandit. Integer vel massa augue. Nam non cursus ipsum. Aliquam varius sodales elit, in faucibus sem pharetra sit amet. Nam eget varius libero. Ut in fermentum lacus. Cras cursus non mi nec elementum. Suspendisse dapibus lectus nec congue lobortis. Praesent pellentesque erat nibh, eget varius nunc suscipit ut.

5.2 Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec efficitur sem id massa aliquam, et scelerisque lacus finibus. Donec lacus orci, scelerisque at tincidunt at, suscipit vitae nulla. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Curabitur ut feugiat erat, in efficitur mi. Sed at placerat arcu.

Nulla et dolor vel velit tincidunt ornare. Curabitur imperdiet tortor ligula, quis ultrices erat dapibus ac.

Mauris aliquet lorem sed nisi ullamcorper tincidunt. Morbi ornare pellentesque turpis, at tristique sapien elementum in. In tristique ipsum et neque efficitur sagittis. Nullam interdum pulvinar convallis. Nulla ac erat mollis, faucibus urna sed, finibus tellus. Aenean volutpat finibus egestas. Nam scelerisque mollis libero sed blandit. Integer vel massa augue. Nam non cursus ipsum. Aliquam varius sodales elit, in faucibus sem pharetra sit amet. Nam eget varius libero. Ut in fermentum lacus. Cras cursus non mi nec elementum. Suspendisse dapibus lectus nec congue lobortis. Praesent pellentesque erat nibh, eget varius nunc suscipit ut.

5.3 Contributions to Body of Knowledge

1. Mauris aliquet lorem sed nisi ullamcorper tincidunt. Morbi ornare pellentesque turpis, at tristique sapien elementum in.
2. In tristique ipsum et neque efficitur sagittis. Nullam interdum pulvinar convallis. Nulla ac erat mollis, faucibus urna sed, finibus tellus. Aenean volutpat finibus egestas. Nam scelerisque mollis libero sed blandit.
3. Integer vel massa augue. Nam non cursus ipsum. Aliquam varius sodales elit, in faucibus sem pharetra sit amet.

5.4 Recommendations for Future Research

Mauris aliquet lorem sed nisi ullamcorper tincidunt. Morbi ornare pellentesque turpis, at tristique sapien elementum in. In tristique ipsum et neque efficitur sagittis. Nullam

interdum pulvinar convallis. Nulla ac erat mollis, faucibus urna sed, finibus tellus. Aenean volutpat finibus egestas. Nam scelerisque mollis libero sed blandit. Integer vel massa augue. Nam non cursus ipsum. Aliquam varius sodales elit, in faucibus sem pharetra sit amet. Nam eget varius libero. Ut in fermentum lacus. Cras cursus non mi nec elementum. Suspendisse dapibus lectus nec congue lobortis. Praesent pellentesque erat nibh, eget varius nunc suscipit ut.

Number of Pages

- Front Matter: Cover page to Ch 1 16
- Main Body of Praxis:
    - Ch 1 (Introduction) 7
    - Ch 2 (Literature) 18
    - Ch 3 (Methodology) 16
    - Ch 4 (Results) 28
    - Ch 5 (Conclusions) 3
- Main body total 72 (expected to be 70-90)
- Back Matter
- References 7
- Appendices 21 (prefer <10 pages)
- Total 116 (prefer < 110)

References

START OF EXAMPLES

Bradshaw, J., & Chang, S. (2013). Past performance as an indicator of future performanc e: Selecting an industry partner to maximize the probability of program success. *Defense Acquisition Research Journal*, *20*(1), 4980. Retrieved from http://dau.dodlive.mil/files/2014/11/ARJ_65-Bradshaw.pdf

Chaplain, C. T. (2011). Space acquisitions: DoD delivering new generations of satellites, but space system acquisition challenges remain. *Hampton Roads International Se curity Quarterly*, 52.

Fox, J. R. (2012). *Defense acquisition reform, 1960-2009: An elusive goal* (Vol. 51). Retrieved from https://www.fas.org/sgp/crs/natsec/R43566.pdf

Guigues, V., Sagastizábal, C., & Zubelli, J. P. (2014). Robust management and pricing of liquefied natural gas contracts with cancelation options. *Journal of Optimization Theory and Applications*, 161(1), 179-198. doi:10.1007/s10957-013-0309-5

END OF EXAMPLE

McKinsey Report. 2023. Unleashing developer productivity with generative AI.

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

Morris M. R., 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.

Austin J., 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.

Google Research. 2023. Mostly Basic Python Problems Dataset. https://github.com/google-research/google-research/tree/master/mbpp

Chen M., 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. Code repository: https://github.com/openai/human-eval. MIT license.

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

Jin H., Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future.

Huang Y., Wanjun Zhong, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, Zibin Zheng, Yanlin Wang. 2024. Agents in Software Engineering: Survey, Landscape, and Vision.

He J., Christoph Treude, David Lo. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Vision and the Road Ahead.

Xia C. S., Deng Y., Dunn S., Zhang L. 2024. AGENTLESS: Demystifying LLM-based Software Engineering Agents

Nguyen M. H., Thang Chau Phan, Phong X. Nguyen, Nghi D. Q. Bui. 2024. Agile Coder. Dynamic Collaborative Agents for Software Development based on Agile Methodology.

Xi Z., Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang and Tao Gui. 2023. The Rise and Potential of Large Language Model Based Agents: A Survey.

Tula Masterman, Sandi Besen, Mason Sawtell, Alex Chao. 2024. The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey.

Li J., Qin Zhang Yangbin Yu, Qiang Fu, Deheng Ye. 2024. More Agents Is All You Need.

Wason R., Parul Arora, Devansh Arora, Jasleen Kaur, Sunil Pratap Singh, M. N. Hoda. 2024. Appraising Success of LLM-based Dialogue Agents.

Wu Q., Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.

Hong S., Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau5, Zijuan Lin, Liyang Zhou, Chenyu

Ran, Lingfeng Xiao, Chenglin Wu, Jurgen Schmidhuber. 2023. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework.

Chen W., Su Y., Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, Jie Zhou. 2024. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors.

Chen W., You Z., Ran Li, Yitong Guan, Chen Qian, Chenyang Zhao, Cheng Yang, Ruobing Xie, Zhiyuan Liu, Maosong Sun. 2024. Internet of Agents: Weaving a Web of Heterogeneous Agents for Collaborative Intelligence.

Zhou W., Yuchen Eleanor Jiang, Long Li1, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, Shiding Zhu, Jiyu Chen, Wentao Zhang, Xiangru Tang, Ningyu Zhang, Huajun Chen, Peng Cui, Mrinmaya Sachan. 2023. Agents: An Open-source Framework for Autonomous Language Agents.

Chen L., Varoquaux G. 2024. What is the Role of Small Models in the LLM Era: A Survey.

Gao L. 2023. Cracking the Coding Evaluation.

Loubna B. A., Muennighoff N., Kumar Umapathi, L., Lipkin B., and von Werra, L. 2022. A framework for the evaluation of code generation models (bigcode-evaluation-harness) as part of the BigCode Project - an open scientific collaboration run by Hugging Face and ServiceNow Research. https://github.com/bigcode-project/bigcode-evaluation-harness. Apache license.

Nedilko A. 2024. Code Repository for this Praxis. https://github.com/agnedil/Praxis. Modified OpenAI code for HumanEval and MBPP evaluation: https://github.com/agnedil/Praxis/tree/main/code/modified-human-eval-by-openai.

Matton A., Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, Matthias Gallé. 2024. On Leakage of Code Generation Evaluation Datasets.

OpenAI. 2025. API Reference. https://platform.openai.com/docs/api-reference/chat/create.

Siddique Y. 2023. Understanding the controllable parameters to run/inference your Large Language Model. https://medium.com/@developer.yasir.pk/understanding-the-controllable-parameters-to-run-inference-your-large-language-model-30643bb46434

Sadani A. 2023. Mastering LLM Inference: A Closer Look at Request Parameters. https://medium.com/@anuj.sadani3/mastering-llm-inference-a-closer-look-at-request-parameters-68aba00914a3

Cohere Team. 2022. LLM Parameters Demystified: Getting the Best Outputs from Language AI. https://cohere.com/blog/llm-parameters-best-outputs-language-ai

Khalusova M. 2025. Hugging Face Open-Source AI Cookbook. Fine-Tuning a Code LLM on a Single GPU. https://huggingface.co/learn/cookbook/en/fine_tuning_code_llm_on_single_gpu

Napalkova L. 2024. Fine-Tuning Small Language Models: Practical Recommendations. https://medium.com/@liana.napalkova/fine-tuning-small-language-models-practical-recommendations-68f32b0535ca

Timothy Wang, Devvret Rishi. Predibase. 2023. Fine-Tune a Code Generation LLM with Llama 2 for Less Than the Cost of a GPU. https://predibase.com/blog/fine-tune-a-code-generation-llm-with-llama-2-for-less-than-the-cost-of-a

HuggingFace, 2024 (1). Coding dataset. Tested-143k-Python-Alpaca. https://huggingface.co/datasets/Vezora/Tested-143k-Python-Alpaca

HuggingFace, 2024 (2). Coding dataset. CodeFeedback-Filtered-Instruction. https://huggingface.co/datasets/m-a-p/CodeFeedback-Filtered-Instruction

HuggingFace, 2024 (3). Coding dataset. Magicoder-Evol-Instruct-110K. https://huggingface.co/datasets/ise-uiuc/Magicoder-Evol-Instruct-110K

Shinn N., Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning.

LangChain. 2025. LangChain Documentation. https://python.langchain.com/docs/introduction/

Galileo. 2024. Mastering AI Agents. https://www.galileo.ai/ebook-mastering-agents

Talebirad Y. & Nadiri A. 2023. Multi-Agent Collaboration: Harnessing the Power of Intelligent LLM Agents.

Huang D., Bu Q., Zhang J. M., Luck M., Cui H.. 2024. AgentCoder: Multiagent-Code Generation with Iterative Testing and Optimization.

Islam A., Ali M. E., Parvez R. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving.

AI-Agent-Dev. 2025. How LangChain and LangGraph Make the Life of AI Agent Developers Easier. https://habr.com/ru/articles/881372/

Appendix A

Table A-1. Parametric Correlations of X and Y

Pearson Correlation	1	.627**
Sig. (2-tailed)		.000
Sum of Squares and Cross-products	960.119	607.382
Covariance	.559	.354
N	1719	1719
Pearson Correlation	.627**	1

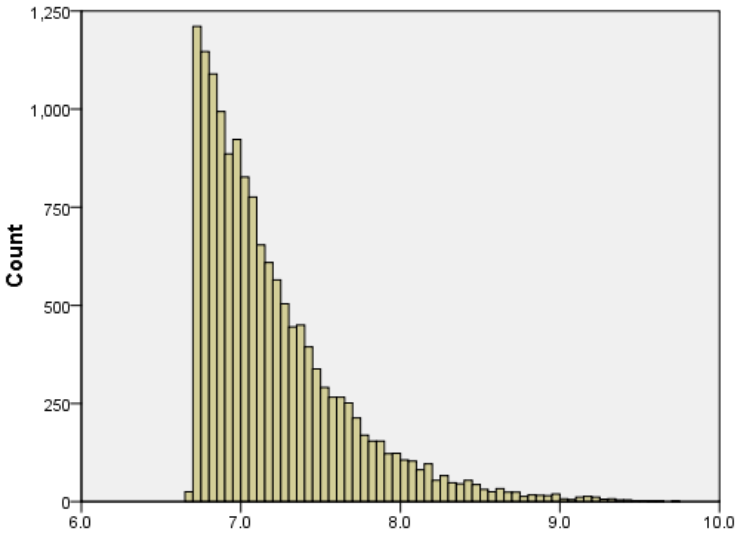

Figure A-1. Histogram of XYZ.