# Dedication

This work is dedicated to my parents, Grigory and Raisa Nedilko, for their continuous and unconditional love, kindness, dedication, and self-sacrifice throughout every step of my educational journeys, as well as for their strong belief in my knowledge and ability to reach my goals. Thank you for all the wonderful things that you have done for me—I will be forever grateful.

To my wife, Sandy Nedilko, whose endless beautiful love, unprecedented support, and boundless patience have been my anchor and my inspiration. Even when my studies pulled me away from our family, your supportive understanding gave me strength. You are the heart behind every success that I celebrate.

To my lovely daughters, Sophia and Daria, who are my greatest joy and inspiration to always reach new frontiers and to demonstrate that dedication and hard work can transform dreams into reality. You are my guiding stars, and your happiness and laughter light my path forward.

To my sister, Tatiana Nedilko who, though thousands of miles lie between us, has never wavered in her love and support. And in those moments when I couldn’t be there, she carried out the very acts of care I longed to give but could not. You have been a constant guiding light, proving that true closeness knows no distance.

Thank you all—for every sacrifice, every word of encouragement, and every moment of love. This journey belongs to you as much as it does to me.

# Acknowledgements

The author wishes to express his deepest gratitude to the George Washington University for always believing in his potential and providing continuous support. The university’s unwavering commitment to academic excellence, clear organizational structure, and professional guidance throughout the Dr. Eng. program have profoundly enriched this learning journey and empowered the author to achieve more than he ever imagined.

The author is also deeply grateful to all the professors, faculty, and staff whose passion and expertise have shaped every step of this doctoral program. Their insightful lectures, thoughtful feedback, and generous sharing of resources sharpened the author’s analytical skills and inspired creativity. Thank you for fostering an environment where curiosity thrives and for dedicating your time and energy to make this Dr. Eng. program a dynamic and transformative experience.

The author would also like to express his sincere thanks to his Praxis Director, Dr. Abreu-Castellanos, for his exceptional mentorship throughout the research phase. His patient guidance, probing questions, and unwavering belief in the author’s ideas challenged the author to think more deeply and refine his work. The author is especially grateful for the countless hours he spent reviewing drafts, discussing new directions, and encouraging.

The author is also grateful to his fellow D. Eng. students whose encouragement, and shared determination created a community of support. Also, thank you to all friends, colleagues, and well-wishers whose support—whether big or small—has been a continuous source of power.

# Abstract of Praxis

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

Automating code generation promises to substantially accelerate software development while reducing costs and minimizing human error. This study evaluates the viability of small language models (SLMs)—augmented with agentic workflows—as a privacy-preserving, cost-effective alternative to proprietary large language models (LLMs) for generating Python code based on natural-language descriptions.

24 open-source SLMs (2.8B - 22B parameters) were compared across four benchmarks—HumanEval, MBPP, LBPP, and BigCodeBench—using a uniform inference pipeline implemented with PyTorch and HuggingFace. The experiments were conducted with three stages of post-processing (raw output, fence extraction, full cleaning) and a tiered prompt-engineering framework (basic, instructional, full prompts). Two agentic workflows were introduced—a two-stage reflection agent and a multi-agent collaboration chain—to iteratively refine generated code. Key hyperparameters (temperature, top-p) were systematically tuned, and selected SLMs underwent fine-tuning via QLoRA on the MBPP training dataset.

Results demonstrate that full cleaning achieves the maximum mean pass@1 scores. Basic prompts outperform more elaborate prompts. Agentic workflows yield further gains: reflection adds +6 pp on average. Fine-tuning on MBPP delivers +3–5 pp improvements. In addition, latency and cost analyses reveal substantial practical advantages of SLMs.

# List of Acronyms

|  |  |
| --- | --- |
| AHA | Agents Help Agents |
| AI | Artificial Intelligence |
| APE | Automatic Prompt Engineering |
| API | Application Programming Interface |
| APPS | Automated Programming Puzzles & Solutions |
| BART | Bidirectional and Auto-Regressive Transformers |
| BERT | Bidirectional Encoder Representations from Transformers |
| BLEU | Bilingual Evaluation Understudy (Method) |
| BLOOM | BigScience Large Open-science Open-access Multilingual Language Model |
| CEO | Chief Executive Officer |
| CLM | Causal Language Modeling |
| CoNaLa | Code/Natural Language Challenge |
| CoT | Chain-of-Thought |
| CRAG | Corrective Retrieval-Augmented Generation |
| DL | Deep Learning |
| DPO | Direct Preference Optimization |
| FLAN | Fine-tuned LAnguage Net |
| FP16 | 16-bit floating-point (half-precision) |
| GB | Gigabyte |
| GPT | Generative Pre-Trained Transformers |
| GPU | Graphics Processing Unit(s) |
| HP | Hyperparameters |
| HumanEval | Human Evaluation Dataset |
| IoA | Internet of Agents |
| JSONL | JSON Lines |
| KTO | Kahneman-Tversky Optimization |
| LaMDA | Language Model for Dialogue Applications |
| LBPP | Less Basic Python Problems (Dataset) |
| LLaMA | Large Language Model Meta AI |
| LLM | Large Language Model |
| LMA | LLM-based multi-agent |
| LoRA | Low-Rank Adaptation |
| MathQA | Math Question Answering |
| MBPP | Mostly Basic Python Problems (Dataset) |
| MLM | Masked Language Modeling |
| MMLU | Massive Multitask Language Understanding |
| NL | Natural Language |
| NLP | Natural Language Processing |
| NSP | Next Sentence Prediction |
| PaLM | Pathways Language Model |
| PC | Personal Computer |
| PEFT | Parameter-Efficient Fine-Tuning |
| PII | Personally Identifiable Information |
| PL | Programming Language |
| QA | Quality Assurance |
| QLoRA | Quantized Low-Rank Adaptation |
| RAG | Retrieval-Augmented Generation |
| RALM | Retrieval-Augmented Language Model |
| RAM | Random-Access Memory |
| RLHF | Reinforcement Learning from Human Feedback |
| RNN | Recurrent Neural Network |
| RoBERTa | Robustly optimized BERT pretraining approach |
| SFT | Supervised Fine-Tuning |
| SLM | Small Language Model |
| SQuAD | Stanford Question Answering Dataset |
| T5 | Text-to-Text Transfer Transformer |
| ToT | Tree-of-Thought |
| US | United States |
| VPC | Virtual Private Cloud |

# 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 truly proved themselves to be extremely capable of automating code generation from natural language descriptions, thus enhancing the software engineering efficiency.

The McKinsey report (2023) indicates that generative artificial intelligence (AI) can reduce developer coding time by as much as 45%. Companies can achieve cost reductions through this capability. The efficiency gain becomes most apparent in big projects because even small improvements result in substantial cost savings.

The process of automating code generation results in better code quality because it maintains strict adherence to established coding standards and best practices. The research conducted by (Almeida Y. et al, 2024) and (Martinović B. and Rozić R., 2024) demonstrates that AI-enhanced tools produce consistent well-structured code which minimizes both human mistakes and system 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. However, developer productivity goes beyond speed - according to (Kalliamvakou, 2024) 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 the 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-assisted tools already have a significant impact on software development, and more than 25% of new code at Google is AI-generated.

AI is used for assistance in coding tasks not only by Google developers. According to the 2024 Developer Survey (Stack Overflow, 2024), more than 76% of respondents are using or intend to use AI tools in the development process this year, while as many as 62% are actively using them. A 2023 GitHub survey (Shani S. & GitHub Staff. 2023) showed that 92% of software engineers in the US are using AI tools for coding tasks for work and at home.

The use of proprietary LLMs creates major challenges for protecting sensitive data and respecting intellectual property rights. Developers often have to use proprietary or confidential information when training models or making inferences which creates risks for data breaches and unauthorized access to intellectual property. The approach endangers business competitive positions and creates legal exposure for companies.

To address these challenges, companies could implement small language models (SLMs) boosted by agents and deployed in resource-constrained, but secure environments. SLM-based agents provide organizations with an economical solution that protects privacy while replacing proprietary LLMs. Organizations can use these agents to automate basic code routine creation which shortens developer time spent on manual coding. The method delivers efficient solutions to understaffed projects too through automated code generation without requiring additional software engineers for routine tasks.

The implementation of SLM-based agents solves the dual problem of improving coding efficiency while maintaining sensitive data protection. Companies with small budgets who want to maintain a small workforce can use this technology to generate code efficiently and accomplish tasks that were impossible before. The approach helps organizations boost developer productivity while improving code quality and shortening time-to-market and maintains data confidentiality.

## 1.2 Research Motivation

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

The outstanding ability of LLMs to generate code from natural language descriptions comes with major challenges regarding sensitive data protection and intellectual property rights. The use of proprietary LLMs necessitates sending confidential information to third-party servers which creates risks of data breaches and unauthorized access to proprietary code. The practice endangers a company's competitive position while making it vulnerable to legal consequences.

On the other hand, these companies could use SLMs to provide a similar level of solution quality. This research aims 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** makes sure that any sensitive data, personally identifiable information (PII), or intellectual property stay 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 (Nguyen et al., 2024).
3. Due to their much smaller size on disk (because of significantly fewer training parameters) SLMs don’t require **massive clusters of Graphics Processing Units (GPUs)** to fine-tune (Nguyen et al., 2024).
4. Agents can ensure continuous adherence to **coding best practices and standards** which reduces human errors and bugs (Jin et al. 2024).
5. **Faster development cycles** allow companies to take products to market more quickly which provides a competitive edge in the industries that evolve rapidly.
6. In understaffed projects, SLM-based agents can compensate for **limited human resources** via efficient code generation which reduces the need to hire additional programmer for performing 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 main 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.

The purpose of this research is to create a feasible, secure, and efficient alternative to the use of proprietary LLMs for automated code generation by achieving the goals enumerated above.

## 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 the use of agentic workflows, such as reflection or multi‐agent collaboration, improve code‐generation quality, as measured by test‐case pass rates across multiple benchmarks?

**Research question 2:** Will adjusting SLM inference parameters, such as temperature and top-p, ensure greater code-generation quality, as measured by test-case pass rates across multiple benchmarks?

**Research question 3:** Will fine-tuning SLMs enhance code-generation quality, as measured by test-case pass rates across multiple benchmarks?

**Hypothesis 1**: Agentic workflows will result in higher test-case pass rates than single-pass SLM inference, as measured across multiple benchmarks.

**Hypothesis 2**: Adjusting SLM parameters, such as temperature and top-p, will improve code generation quality, as measured by test-case pass rates across multiple benchmarks.

**Hypothesis 3:** Fine-tuning SLMs on domain-specific data will increase test-case pass rates across multiple benchmarks compared to using SLMs without fine-tuning.

## 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. Such experiments will include the use of:
  + different SLMs;
  + several benchmark datasets;
  + several agentic workflows;
  + tuning different model hyperparameters;
  + fine-tuning SLMs, etc.
* **Using quantifiable metrics** to evaluate the correctness and quality of the generated code.
* **Documenting the experimentation methodologies and findings** in a comprehensive way in order to contribute to the academic knowledge base.

## 1.8 Research Limitations

The research on SLMs enhanced by agent-based architectures for automated code generation has several limitations that may affect the scope, applicability, and generalizability of the findings. It is important to recognize these limitations in order to interpret the results accurately and to identify 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. The research concentrates on natural language description-based code generation without exploring other software engineering automation aspects including code refactoring, bug detection and code optimization.

The research depends on publicly available datasets for both model training and evaluation purposes. The models' generalization to real-world applications might be limited because they lack access to proprietary domain-specific or large-scale datasets and the quality and diversity of available datasets could impact performance results. The training of SLMs on public data may result in the unintentional learning and propagation of biases which exist in the training data. The research does not address bias detection or mitigation strategies which could affect the ethical acceptance and fairness of generated code in sensitive applications.

The research does not provide detailed information about implementing strong data protection measures despite data privacy being a primary reason to use SLMs. Also, the fast-paced development of AI technologies indicates that new SLMs or alternative methods could appear before publication which might outperform the models studied here and affect the relevance and usefulness of the research 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. The complexity of programming challenges leads these assistants to enhance developer efficiency while reducing errors and optimizing the coding workflow. The support system delivers its assistance through automated code suggestions and error detection and resolution as well as code generation and documentation and context-based recommendations. In this field, language models have become essential, enabling developers to access informed hints, produce code segments, and generally improve their coding expertise (Soliman, 2024).

In a recent study (Coutinho et al., 2024), the authors explore how generative AI tools affect productivity in real-world software development settings. By distributing licenses for different generative AI solutions (e.g., ChatGPT, GitHub Copilot) to professionals working in different roles (developers, designers, data scientists, QA specialists, and coaches), the authors gather qualitative insights on how these tools fit into day-to-day activities. The participants expressed positive outcomes regarding their perceived productivity because the tools saved their time and generated efficient artifacts while providing immediate access to information. The respondents encountered three main issues which included maintaining reliability and improving output quality and protecting sensitive data from security risks. The research provides initial findings that will guide future investigations although it has certain boundaries. The study indicates that generative AI can boost workflow efficiency and knowledge acquisition but additional empirical research is required to measure productivity gains accurately (Coutinho et al., 2024).

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

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

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

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

The recent advancements in LLMs have transformed automated code generation through their ability to convert natural language inputs into actual code. The initial methods used Recurrent Neural Networks (RNNs) and syntax-driven approaches which struggled with complex long-range dependencies. The introduction of transformers that were originally introduced to solve tasks related to natural language processing (NLP), significantly improved performance. Researchers achieved state-of-the-art results on benchmarks such as Code/Natural Language Challenge (CoNaLa) and DJANGO dataset (Oda et. al., 2015) through the combination of encoder models BERT or RoBERTa with Marian decoders. These hybrid models improve syntax and semantics and developer productivity through intelligent autocompletion and context-aware suggestions and inline documentation. The development process becomes more efficient through built-in linting and formatting and error-checking features. LLMs have created a significant improvement in the accuracy and efficiency of contemporary code generation (Soliman, 2024.

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

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

A GPT-3.5-powered IntelliJ IDEA plugin is introduced in (Almeida Y., 2024) designed to automate code reviews. AICodeReview identifies syntax errors, logic flaws, and vulnerabilities while offering actionable improvement suggestions with detailed explanations. The tool enables support for various programming languages and allows users to customize suggestions and integrate with JetBrains products. The evaluation results demonstrated that AICodeReview performed better than manual reviews by cutting review time to 15.2 minutes from 22.5 minutes and detecting 28 code smells instead of 20 while improving refactoring outcomes to 25 from 13. The research demonstrates how LLMs can effectively optimize software development workflows.

The use of pre-trained language models for Python code generation is explored in (Ottens et al., 2024), focusing on completing function bodies given function signatures and docstrings. The authors evaluate three models using CodeSearchNet data: a sequence-to-sequence RNN and character-level embeddings and a fine-tuned GPT-2 model. The fine-tuned GPT-2 model outperformed the baseline models by achieving a Bilingual Evaluation Understudy (BLEU) score of 0.22 which represented a 46% improvement over the baseline. The research demonstrates how GPT-2 adapts to programming languages through transfer learning methods that were initially developed for natural language processing. The generated code demonstrates both originality and correct syntax structure which indicates the model's understanding of Python programming elements. The research demonstrates how LLMs can automate software development tasks while enhancing both efficiency and quality standards.

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

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

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

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

## 2.3 LLMs

### 2.3.1 LLMs: Overview

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

The survey then describes various techniques used to develop and augment LLMs, such as positional embeddings, mixture-of-experts, subword-based tokenization, and the methods, datasets and benchmarks for training and evaluation including BLEU, ROUGE, Pass@k, Stanford Question Answering Dataset (SQuAD), Massive Multitask Language Understanding (MMLU), HumanEval, etc. The objectives targeted in the process of pre-training LLMs cover masked language modeling (MLM) along with next sentence prediction (NSP), as well as causal language modeling (CLM), while 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), and another technique called Kahneman-Tversky Optimization (KTO).

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

Future directions and open challenges listed in the paper include:

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

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

### 2.3.2. LLMs for Code Generation

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

The survey evaluates different assessment methods which include Human Evaluation Dataset (HumanEval) and Mostly Basic Python Problems (MBPP) as well as newly proposed metrics and human or LLM-based assessments. It emphasizes the challenges of evaluating code quality alongside correctness and software engineering attributes. This survey functions as a fundamental resource that enables both researchers and practitioners to understand current LLM code generation techniques while identifying development paths for improved AI coding assistants with reliability and adaptability and context-aware features.

The study by Wodecki (Wodecki 2023) examines the emerging text-to-code generative AI models which show potential to transform software development through natural language instructions conversion into executable code. The report evaluates major tools including StarCoder and Codex and Copilot and Code Interpreter and CodeT5 and Polycoder and Replit’s Ghostwriter. The different systems originate from separate sources and use different technical frameworks with unique functional characteristics. StarCoder emerges as a collaborative project between ServiceNow and Hugging Face through its training on diverse code datasets leading to exceptional results on standard evaluation metrics. The OpenAI-developed Codex serves as the foundation for GitHub Copilot to let developers generate code snippets through natural language inputs across various programming languages. The Code Interpreter tool from OpenAI expands ChatGPT capabilities by letting users execute code for data analysis tasks directly from 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 tools used to generate code, such as Amazon CodeWhisperer, OpenAI’s ChatGPT, or GitHub Copilot are increasingly utilized to produce code from natural language prompts. assesses code generation tools including Amazon CodeWhisperer and OpenAI’s ChatGPT and GitHub Copilot through HumanEval Dataset analysis to evaluate their code output quality across metrics including correctness and validity and maintainability and reliability. The study shows ChatGPT and Amazon CodeWhisperer and GitHub Copilot generate correct code at rates of 65.2% and 31.1% and 46.3% respectively while updated GitHub Copilot and Amazon CodeWhisperer achieve 18% and 7% improvement respectively. The calculated technical debt from code smells reaches 5.6 minutes for Amazon CodeWhisperer but reaches 8.9 minutes for ChatGPT and 9.1 minutes for GitHub Copilot. These results demonstrate the tools' capabilities along with their constraints which help developers select appropriate generators for their programming requirements.

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

## 2.4 SLMs

### 2.4.1 SLMs: Overview

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

The survey document presents common datasets together with evaluation metrics that focus on SLM scenarios and stresses the need to measure factors such as latency and memory footprint and privacy and energy efficiency. The paper investigates real-world deployments of SLMs on edge devices and in interactive systems while analyzing problems with model biases and hallucinations alongside privacy concerns.

Research studies on SLMs can be found in (Wang et al., 2024), (Lee 2024), (Ghosh 2023), (Mok 2023), (Szczygło 2024) (Morris et al., 2024), (Kili Technology Guide, 2024), (Abbas 2024) which repeat the essential benefits of SLMs when compared to LLMs. All research indicates SLMs present themselves as viable alternatives to LLMs since they resolve the major problems stemming from LLMs' extensive size and resource needs. The impressive general-purpose capabilities of GPT-4 and LLaMA come with significant challenges regarding operational expenses as well as scalability issues and privacy concerns and real-time operation on edge devices. Their broad approach results in weak domain-specific capabilities that lead to inferior results for medical or legal applications.

The advantages of SLMs become evident because these models possess a much lower parameter count. They are less expensive to operate and easier to integrate and can efficiently run on different devices which includes both local and edge systems thus maintaining data privacy. The compact model architecture simplifies domain-specific fine-tuning procedures which enhances accuracy and application responsiveness while decreasing dependence on extensive cloud infrastructure. Current studies indicate that despite their simpler design SLMs demonstrate equivalent or superior performance to larger models in particular tasks through the application of knowledge distillation and pruning along with quantization and parameter-efficient fine-tuning (LoRA) and retrieval-augmentation strategies. Through these methods SLMs can preserve high performance while needing reduced parameters and training data along with diminished computing power which makes them suitable for specific domain applications.

The compact structure of SLMs enables quicker processing together with reduced latency and improved cost efficiency although they provide slightly restricted capabilities. The ability of SLMs to adapt quickly to business evolution allows organizations to speed up model development and maintain alignment with changing requirements. SLMs reduce operational expenses and environmental impact because they operate with smaller computational resources and simple architectures and enable better data privacy and security through local model execution.

As a result, SLMs are increasingly viewed as a preferred solution for both business organizations and research groups because they offer practical access to their capabilities. Organizations can select SLMs instead of large general-purpose models because these models require substantial resources and expenses. Organizations can select SLMs which they customize for specific needs and update regularly to meet their particular requirements. The right combination of size and performance and flexibility will propel SLMs into the next AI innovation era by making NLP technology available to all.

The research paper by Fatima (2024) studies how small language models (SLMs) have gained popularity in the 2024 AI environment through five significant examples: Meta’s Llama 3, Microsoft’s Phi 3, Mistral AI’s Mixtral 8x7B, Google’s Gemma, and Apple’s OpenELM family. SLMs deliver powerful linguistic functions by using reduced architectures together with training methods that include transfer learning and knowledge distillation and sparse mixtures of experts. The developed models serve as a cost-efficient option for wide device integration and application compatibility which enables customization as well as on-device processing and domain-specific fine-tuning. An SLM example that replaces an LLM is presented in (Murallie 2024).

Future SLM research indicates the development of blended systems that unite small and large models with enhanced architectures for computation optimization and on-device learning methods for adaptive processing. Research indicates SLMs will serve as a fundamental solution to advance NLP technology while preserving environmental and computational boundaries (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. Despite the impressive ability of LLMs to generate complex code snippets, they are computationally expensive and thus difficult to deploy in resource-limited environments. The small models have been proposed as a promising alternative to the traditional LLMs in code generation by using data curation, prompt optimization, and domain-specific fine-tuning techniques.

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

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

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

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

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

## 2.5 Agents

### 2.5.1 Agents: Overview

According to (Park et al., 2023), researchers have begun exploring generative agents, computational entities built on top of LLMs, to create realistic simulations of human-like behavior in interactive environments. The agents operate independently from traditional non-player character rules because they create memories through experience and reflect on past events while modifying their plans dynamically. Generative agents achieve believable thought patterns and social coordination through their ability to manage long-term memory and perform higher-level reasoning and recursive planning. The first demonstrations of these agents in virtual communities based on The Sims show they can perform sophisticated social actions including information sharing and relationship building and event organization without human intervention. The research indicates a new paradigm for code generation and AI-based interactions which enables authentic simulations in user interfaces, game worlds, educational platforms, and social computing systems.

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

The frameworks enable both single-agent operations for basic tasks and open-ended exploration as well as multi-agent systems that produce cooperative and competitive interactions. The authors emphasize essential components which enable strong performance across these designs including defined roles, phases of plan development, improvement and flexible team member changes, and effective communication methods. The authors also acknowledge that properly selecting between single- or multi-agent paradigms depends on problem characteristics.

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

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

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

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

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

A study in (Chen, You et al., 2024) proposes the Internet of Agents (IoA), a novel framework designed to facilitate LLM-driven multi-agent collaboration in a manner reminiscent of the Internet’s global connectivity. The IoA system differs from previous multi-agent systems because it enables the integration of various third-party agents with different skills and tools that operate across multiple devices and environments. The framework enables flexible team formation through autonomous agent collaboration for task evolution. 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 comprising the use of autonomous language agents described in (Zhou W. et al., 2023). It introduces AGENTS, an open-source framework designed to make creating, customizing, and deploying LLM-based autonomous language agents more accessible. AGENTS facilitates key features such as long-term memory management, versatile tool and web usage, multi-agent communication, and the ability for human users to interact with these agents. It also offers a novel concept of a symbolic plan (SOP) that provides a structured, state-based approach to controlling an agent’s actions, thereby enabling greater predictability and stability in behavior.

### 2.5.2 Agents for Code Generation

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

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

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

The perception module accepts different input types which it converts into representations that LLMs can understand. The memory module controls various knowledge types which include enduring semantic information (such as documentation or Application Programming Interface (API) references) and episodic and procedural memories that store recent events and learned actions. The action module then enables reasoning and planning—often improved by chain-of-thought prompting—and tool usage for activities like code retrieval or debugging. They also highlight that agents can operate individually or collaboratively, with multi-agent systems dividing tasks and sharing knowledge for greater efficiency.

The study in (He et al., 2024) outlines a forward-looking perspective on employing multi-agent systems powered by LLMs to tackle complex software engineering tasks. The growing complexity of software projects that include requirements definition and code implementation and quality assurance and maintenance exceeds the capabilities of single LLM-driven agents to handle domain knowledge variety and depth. LLM-based multi-agent (LMA) systems demonstrate potential to solve complex problems through teams of agents who specialize in different capabilities.

The authors explain how LMA systems deliver three essential benefits for software engineering applications: a) Multiple agents working together in LMA systems enhance both reliability and robustness and reduce hallucinations, b) These systems provide independent task decomposition which enables autonomous process management for design, coding and testing without continuous human supervision, c) LMA systems exhibit natural scalability properties. As project scope evolves, the system can incorporate more agents or adapt roles, enhancing its ability to handle large-scale, diverse software initiatives efficiently.

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

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

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

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

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

## 2.6 Evaluation of Generated Code

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

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

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

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

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

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

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

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

## 2.7 Conclusion

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

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

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

By integrating these promising new SLM architectures with agent-based systems—where models interact with external tools, retrieve relevant data, and break down problems into manageable steps—the authors plan to achieve even more substantial efficiency and accuracy gains. Agents acting as orchestrators can direct an SLM’s inference strategies, selecting when and how to apply iterative refinement or verification procedures. They can determine which domain-specific resources to query and how to adaptively allocate additional compute where it matters most. The synergy enables small models to perform intelligent thinking during testing while operating in dynamic environments that require context-rich information to achieve complex tasks with improved success rates. The combination of SLMs with agents enhances their core advantages of cost-effectiveness and flexibility and specialization through strategic parameter count compensation to create more powerful efficient and responsive AI systems.

# Chapter 3 — Methodology

## 3.1 Introduction

This chapter describes the methodology used in evaluating the performance of small language models (SLMs) on code generation tasks, as well as the implementation of agentic workflows to enhance the performance of these SLM models. It describes 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. It also discusses 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. This methodology establishes a reproducible, efficient, and comprehensive framework for assessing SLM capabilities in code generation tasks.

## 3.2 Experimentation Platforms

Several platforms were used to run the experiments.

### 3.2.1 Google Colab

A series of Jupyter notebooks was used to experiment with small language models (SLMs) in the form of HuggingFace transformers. For this, the author 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 Random-Access Memory (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 Computer (PC) Using API for Hosted Models

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

Mistral AI (Mistral AI, 2024) 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, the efficient transformer models were combined with a developer-friendly approach. A code generation solution was built that worked faster than transformer models in Google Colab since SLMs were already hosted and called via an API.

Another platform, Replicate (Replicate, 2024), 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 were 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.

## 3.3 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.

A diagram of a process

AI-generated content may be incorrect.

##### Figure 3-1. Simple Flowchart Describing the Experiments in This Research

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 3-1 below for a list of SLMs used in experiments.

#### Table 3-1. Small Language Models Used in Experiments

|  | **Model** | **Hosted By** | **Model Size** | **Temp / top\_p** | **Estimated cost, ($ per experiment)** |
| --- | --- | --- | --- | --- | --- |
|  | **Small Language Models (SLMs)** | | | | |
| 1 | Nxcode-CQ-7B-orpo | Google Colab | 7.25B | 1.0 / 1.0 | $50/month |
| 2 | Codestral Mamba | mistral.ai | 7.3B | 0.7 / 1.0 | 0.02 |
| 3 | Ministral 8B Instruct | mistral.ai | 8B | 0.3 / 1.0 | 0.01 |
| 4 | Deepseek-Coder-6.7B-Instruct | Google Colab |  | 1.0 / 1.0 | $50/m |
| 5 | Ministral 3B Instruct | mistral.ai | 3B | 0.3 / 1.0 | 0.01 |
| 6 | Mistral-Nemo-Instruct-2407 | mistral.ai | 12B | 0.3 / 1.0 | 0.01 |
| 7 | Llama 3.1 8B Instruct | Google Colab | 8B |  |  |
| 8 | CodeQwen1.5-7B-Chat | Google Colab |  | 1.0 / 1.0 | $50/m |
| 9 | OpenCodeInterpreter-DS-6.7B | Google Colab |  | 1.0 / 1.0 | $50/m |
| 10 | Mistral 7B, open-mistral-7b | mistral.ai | 7B | 0.7 / 1.0 | 0.01 |
| 11 | Nous-hermes-2-solar-10.7b | replicate.com | 10.7B | 0.95 / 1 | 0.61 |
| 12 | Phixtral-2x2\_8 (4.5B) | replicate.com | 4.5B | 0.95 / 1 | 2.77 |
| 13 | Artigenz-Coder-DS-6.7B | Google Colab |  | 1.0 / 1.0 | $50/m. |
| 14 | Code Gemma 7b IT | Google Colab | 7B | 1.0 / 1.0 |  |
|  | **Slightly Bigger SLMs** | | | | |
| 15 | Mistral-Small-2409 | mistral.ai | 22B | 0.7 / 1.0 | 0.03 |
| 16 | Codestral latest | mistral.ai | 22.2B | 0.7 / 1.0 | 0.15 |
| 17 | Mixtral-8x7B-v0.1 | mistral.ai | 12B active (47B total) | 0.7 / 1.0 | 0.05 |
|  | **SLMs That Were Not Found Useful** | | | | |
| 18 | Qwen1.5-7b | replicate.com | 7B | 0.95 / 1 | 3.55 |
| 19 | Llama 3 8B | Replicate | 8B | 0.95 / 1 | 0.29 |
| 20 | Gemma 7B | replicate.com | 7B | 0.95 / 1 | 0.05 |
| 21 | Gemma 2B | replicate.com | 2B | 0.95 / 1 | 0.05 |
| 22 | Flan-T5 | replicate.com |  | 0.95 / 1 |  |
| 23 | Phi-2 | replicate.com |  | 0.95 / 1 |  |
| 24 | 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. In order to have a fairer and broader evaluation of SLMs, four different datasets were utilized to make a comprehensive evaluation of SLMs’ code generation capabilities. First, two popular Python benchmark datasets were used: HumanEval and MBPP.

The HumanEval dataset is a benchmark created by OpenAI specifically for assessing the capabilities of large language models (LLMs) to generate code (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. The generated solutions undergo unit test evaluation to verify their accuracy. The design approach evaluates both syntactic accuracy and semantic and logical correctness of generated code to provide a strong assessment of LLMs' natural language understanding and code generation capabilities. This integrated evaluation framework provides a reproducible and scalable methodology to benchmark LLMs on practical programming challenges.

The Mostly Basic Programming Problems (MBPP) dataset (Google Research, 2023) 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. 500 testing data points were used in the research for this Praxis 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 were touched upon in Section 2.6 of this Praxis, the Less Basic Python Problems (LBPP) dataset was used, 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 (Matton et al., 2024), 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 was used in an attempt to prevent the data leakage issue was the BigCodeBench dataset (Zhuo et al., 2024). 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 benchmark datasets that were explored, and it was 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.

## 3.5 Measuring Performance

The pass@1 metric (Chen 2021) was used to evaluate performance across all four datasets. It is 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. The implementation by OpenAI presented in (Chen M., 2021) was utilized. The OpenAI code didn’t work out of the box, so it had to be modified 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, the Replicate client had to be integrated 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 no easy way to adapt it to work with any model or any agent-based application could be found. 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 Prompt Engineering

Code completions were generated using zero-shot prompts - 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. The 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 the scope, but it can be an interesting continuation of this work.

The prompts for the HumanEval and the BigCodeBench datasets asked SLMs to complete the starter code provided in the datasets. Since both MBPP and LBPP datasets contained specific test cases to be satisfied, the prompts for these datasets asked to solve the corresponding task and satisfy the specific test cases, both provided in the datasets (see Section 3.3 for dataset details).

### 3.6.1 Initial Observations

After running initial experiments using the above simple prompt, it was noticed that SLMs included a lot of extraneous text into their output with the intention to clarify the generated code, provide additional test cases, and even to react to the prompt (“Sure! I can do that!”). These passages made the generated code non-executable and would lower the pass rate unnecessarily. Therefore, the prompt had to be modified 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.

Surprisingly, not all SLMs were able to follow these instructions (see Section 4 Results) – they still tended to output additional explanations and clarifications like: “Here is the requested code completion:” etc. which broke 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, some very light general post-processing of the generated code was introduced 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, a general approach to evaluating all SLMs was assumed: 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 the prompting strategy - SLMs are inclined very much to put the generated code inside the code fences: ```python … ```. At first, the author 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, it was decided to use this weakness of SLMs to the advantage – encourage 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.

### 3.6.2. Final Strategy

A screenshot of a computer

AI-generated content may be incorrect.

##### Figure 3-2. Final Prompt Templates Utilized in this Research with Reflection Agents

The final prompt-engineering strategy presented in Fig. 3-1 was based on the following principles. The reliable generation of correct, executable code by SLMs requires both carefully crafted prompts and a robust evaluation pipeline. A systematic prompt engineering framework was tailored to the four code-generation benchmarks—HumanEval, BigCodeBench, MBPP, and LBPP—and utilized an automated methodology for quantifying the impact of each prompt design decision. The approach is organized into two complementary components: a tiered prompt-template hierarchy and dataset-specific wrappers. An end-to-end evaluation flow was then used to measure pass-rates and characterize common error modes for each prompt type.

A screenshot of a computer program

AI-generated content may be incorrect.

##### Figure 3-3. Examples of Real Basic and Instructional Prompts

A screenshot of a computer program

AI-generated content may be incorrect.

##### Figure 3-4. Example of a Real Full Prompt

To isolate the influence of instruction strength on output quality, three levels of prompt complexity were constructed for each task:

* **Basic:** a minimal directive such as a request to complete a starter code. This bare prompt establishes a performance baseline under unconstrained generation.
* **Instructional:** incorporates a developer persona and formatting constraints, such as to act as an experienced Python software developer, enclose the output in code fences. etc. By specifying role and formatting stopping criteria, this level aims to reduce spurious text.
* **Full:** adds a step-by-step procedure plus strict “no non-code content” rules, such as to Read and understand the starter code, integrate the completion with correct syntax and indentation, exclude any explanations or test cases, stop immediately after the final line of the function, output only valid Python code within fenced blocks. These guardrails enforce maximum consistency and minimize post-processing.

Comparing model behavior across these tiers reveals how much instruction is necessary to achieve acceptable pass-rates and clean outputs.

### 3.6.3 Dataset-Specific Wrappers

Different benchmarks present code-generation tasks in distinct formats, therefore the same three-tier hierarchy was applied to obtain dataset-appropriate wrappers:

* **HumanEval & BigCodeBench:** Function headers and docstrings are embedded directly in the prompt. The model completes the starter code to satisfy hidden tests.
* **MBPP & LBPP:** Natural-language task descriptions are accompanied by explicit test cases. The prompt instructs the model to generate code that passes these listed tests without including them in the output.

By unifying the instruction strategy across function headers and natural language descriptions, the prompt format was decoupled from dataset content and enable apples-to-apples comparisons of the final code generation results using the proposed metrics.

### 3.6.4 Reflection Agent Approach

Even with strong prompts, initial outputs can exhibit logic errors, off-by-one bugs, or inefficiencies. A two-stage pipeline was introduced:

1. **Generation Stage:** Produce a candidate solution using one of the prompt templates.
2. **Reflection Stage:** Add the candidate solution along with the original requirements into a meta-prompt that instructs the model to:
   1. Conduct an analysis of the proposed solution for errors or inefficiencies.
   2. Produce a single, optimized version—or return the original if it is already optimal.
   3. Obey the same “code-only” and early-stop constraints.

This self-critique mechanism leverages the model’s own reasoning capabilities to recover from common failure modes and improve overall pass-rates.

### 3.6.3. Multi-Agent Collaboration Approach

A modular, role‐based prompt strategy enables systematic, incremental refinement of AI‐generated code, yielding higher overall quality and maintainability. Each agent is assigned a narrowly scoped role. First, a Generator agent receives a specification (e.g., “Implement function X according to specification Y”) and produces Python code. That output is then passed to a sequence of Refiner agents, each charged with a distinct aspect of code quality: several agents refactor for the ability to execute code—parsing from code fences, removing non-code portions, removing unnecessary test cases which are a product of hallucination, etc.; another agent is looking for errors and inefficiencies and optimizes the solution, annotating changes and justifying each optimization; yet another could audit the code for security vulnerabilities (outside of the scope in the Praxis), and documents both risks and fixes. Throughout this process, each agent receives only the code and a concise change request, enforcing strict role definitions that prevent scope overlap and eliminate prompt ambiguity.

This chaining workflow ensures that changes remain traceable and that each code quality dimension is addressed systematically rather than haphazardly. It is designed to enforce a clear separation of concerns and reduce prompt ambiguity by splitting one monolithic prompt into several steps with smaller and more understandable prompts. Empirical evaluation can compare multi‐agent performance against monolithic prompts by measuring code metrics (e.g., pass rate as pass@1, cyclomatic complexity, etc.).

### 3.6.4. Automated Evaluation Framework

To measure the effectiveness of each prompt variant and the agentic workflows, an end-to-end pipeline was implemented:

1. **Prompt Execution:** Automatically invoke the LLM with every prompt template across all problems in each benchmark.
2. **Output Extraction:** Strip code fences and normalize whitespace.
3. **Test Harness Integration:** Execute the generated code against the benchmark’s reference tests in a sandboxed environment.
4. **Metric Calculation:** Compute standard pass@k metrics (e.g., pass@1, pass@10. Note: only pass@1 is in scope for this Praxis) and exact-match rates for each configuration.
5. **Error Analysis:** Aggregate failure cases by category (syntax error, test assertion, logic bug) and quantify the recovery rate provided by the reflection agent.

By systematically varying prompt strength and dataset wrapper, one can attribute improvements in code-generation performance to each design element. Together, these components enable rigorous measurement of how instructional specificity contributes to the reliability and correctness of SLM-generated code.

## 3.7 Post-Processing of Generated Code

In the workflow used for this Praxis, the generated code may be subject to one of three increasingly aggressive cleaning stages before evaluation:

### 3.7.1. Raw Output (No Post-Processing)

In the simplest setup, the model’s entire response is passed through unchanged. Any markdown fences, explanatory text, inline test cases, or ad hoc print statements remain in place. This approach evaluates the reliability of the model’s original output.

### 3.7.2. Fence Extraction Only

A lightweight filter parses away everything except the first Python‐style code block. Internally, this routine scans for an opening marker like “```python” (or its variants) and captures only the content up to the next closing backticks. By removing preamble, commentary, or trailing prose, it yields a cleaner snippet while still leaving assertions, prints, and other non-essential lines intact.

### 3.7.3. Full Cleaning Pipeline

The most comprehensive strategy orchestrates a sequential series of edits to make sure the resulting output is executable Python code:

1. **Fence parsing:** Extract only the fenced code.
2. **Main‐block removal:** Strip out any if \_\_name\_\_ == "\_\_main\_\_": section to focus solely on definitions.
3. **Test-line pruning:** Eliminate lines beginning with assert or with print(function\_name(, removing unnecessary hallucinated test harnesses.
4. **Fence and placeholder cleanup:** Discard stray fence markers and tags such as ``` or [code].
5. **Signature and import reconciliation:** If the snippet lacks its original function signature, reattach it (properly indented); otherwise, merge in imports from the prompt.
6. **Typing import injection:** Ensure a “from typing import \*” statement appears at the top if not already present – SLMs often drop it and this leads to code execution failures.
   1. This full-pipeline approach delivers a self-contained, clean function body free of superfluous artifacts—ideal for automated execution and benchmarking.

A comparison of evaluation results across the three cleaning tiers would enable one to precisely gauge how much post-processing improves code usability: running raw outputs reveals the baseline failure modes caused by stray markdown, tests, or print calls; extracting only fenced blocks shows whether simply isolating the core snippet is sufficient to eliminate most formatting noise; and applying the full code cleaning pipeline demonstrates the additional gains from stripping main-blocks, assertions, fence markers, and reconciling signatures and imports. By measuring pass rates, syntax errors, and extraneous output at each stage, one can determine which cleanup steps are truly necessary and how aggressively one must sanitize LLM-generated code before benchmarking or deployment. See Chapter 4 for a discussion of the final results.

## 3.8 Tunable Model Hyperparameters for Optimized Inference

Below is an overview of key hyperparameters used to optimize inference across the small language models (SLMs) listed above. 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. This Praxis utilizes only temperature and top\_p, and the other hyperparameters are included for any future stages of this research.

1. **Temperature:** regulatesthe randomness for sampling the next token. A higher temperature (e.g., 1.0–1.2) makes the output more variable, while a lower value (e.g., 0.2–0.5) leads to more deterministic results and can reduce hallucinations and extraneous text, particularly in coding tasks.
2. **Top-p (Nucleus Sampling):** representsthe overall probability threshold (usually between 0.8 and 1.0) for sampling the next token, 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 the functional correctness.
3. **Top-k:r**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:** Meant to penalize those tokens that were already encountered in the text which encourages the model to use 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.

## 3.9 Fine-Tuning SLMs for Code Generation

It is expected that small language models (SLMs) can improve the 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).

The process of fine-tuning on curated code corpora leads to better correctness because it removes incomplete or erroneous code snippets. Models can achieve these improvements in resource-constrained environments (e.g., single-GPU setups) through the combination of Low-Rank Adaptation (LoRA) or prefix tuning techniques which reduce both memory usage and training duration (Khalusova 2025).

The practical application of fine-tuning enables developers to integrate specialized libraries and frameworks and coding styles that match specific domains. A model dedicated to web development would gain advantages from training on datasets containing HTTP requests and server-side logic and specific JavaScript libraries. The model's competency range expands through targeted adaptation which produces code that is both syntactically valid and domain-aware and closely matches 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. First, there is a training set available as part of the MBPP dataset which was used in the experiments for this Praxis. The other datasets can be used to continue this research in the future:

* High quality 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 can learn correct syntax, library usage, and best practices.
* Datasets like **CodeFeedback-Filtered-Instruction** and **Just-write-the-code-Python-GenAI-143k** (HuggingFace 2024) can be used to fine-tune a model for tasks requiring step-by-step problem solving or explanation. They already 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.

## 3.10 Agents

In addition to using SLMs directly for single-pass code generation, this work explores 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 agentic 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). The purpose of adopting these agentic workflows is to reduce error rates, enhance code validity, and boost the model performance in code generation tasks.

### 3.10.1 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). The following architecture was used:

1. **Initial Code Generation Prompt:** The agent is given a problem statement and generates a first-pass solution in code form.
2. **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.

A screen shot of a computer program

AI-generated content may be incorrect.

##### Figure 3-5. Wrapper for PyTorch Implementation of the Reflection Agentic Workflow.

An interesting observation was that when the first implementation of the reflection prompts was used, SLMs used to respond that the initially generated code was already optimal without quoting the actual code. Since this kind of textual output is not runnable code, it broke the code execution and decreased the pass@1 rate. To mitigate, the reflection prompt had to be modified by adding a request to include the initial proposed solution, and the experimentation logic was altered to check the final output if it was executable, and to save the initial proposed solution if the improved solution was not executable.

### 3.10.2 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). The idea to use this agentic workflow was a natural continuation of the reflection agentic workflow implementation, when the most typical errors made by reflection agents were reviewed and an opportunity to achieve improved results through the use of multiple specialized agents was noticed. 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.

The multi-agent collaboration workflow used in this Praxis orchestrates a sequence of agents, each responsible for a specific cleanup or verification step:

* **Code Fence Removal Agent**: Extracts the raw code from Markdown-style fences (```), stripping out the surrounding markers.
* **Extraneous Text Remover Agent**: Deletes any human-style commentary or system messages (e.g. “Here’s an explanation…”) that aren’t valid Python.
* **Import Checker Agent**: Verifies that all required imports (e.g. from typing import List) are present and injects them if they’re missing.
* **Potential Code Execution Agent** (if plausible): Execute the snippet in a sandbox and collect and rectify any syntax/indentation failures.
* **Test-Case Removal Agent**: Deletes inline test scaffolding—assert statements, print(func\_name) calls, or any other hidden test code—that would otherwise interrupt execution. Makes sure the remaining code is executable Python code.

A screenshot of a computer program

AI-generated content may be incorrect.

##### Figure 3-6. Functions in PyTorch Implementation to Call SLMs at Inference Time (used for Regular SLMs and Agents).

# Chapter 4 - Results

## 4.1 Introduction

This chapter presents the comprehensive results of evaluating small language models (SLMs) on code generation tasks across four benchmark datasets: HumanEval, MBPP, LBPP, and BigCodeBench. The initial list of 24 SLMs was reduced to 15 and included the models that showed better results in terms of the pass score and latency. The evaluation encompasses 15 different SLMs ranging from 2.8B to 22B parameters, tested under various experimental conditions including different post-processing techniques, prompt engineering strategies, hyperparameter configurations, and agentic workflows.

The results are organized into several key sections: the analysis begins by establishing optimal post-processing techniques through comparative evaluation of raw output, fence extraction, and full cleaning pipelines across all models, using pass@1 scores to identify the most effective approach for maximizing code executability. Subsequently, the impact of prompt engineering strategies is assessed by comparing basic, instructional, and full prompt complexity tiers to determine the most effective prompting approach. These steps establish baseline performance metrics across all evaluated models and datasets, providing a controlled foundation for subsequent comparative analysis of advanced techniques including hyperparameter tuning for inference optimization, SLM fine-tuning, and agentic workflow implementations including reflection and multi-agent collaboration approaches.

Additionally, this chapter examines the latency and cost-effectiveness of different hosting platforms and provides insights into model selection criteria for practical deployment scenarios.

## 4.2 Experimentation Workflow

The initial experiments evaluated all 24 models across three hosting platforms (see Table 3-1). However, because of very high latency on Replicate.com, that platform was dropped and the focus was narrowed down to the 15 top-performing models (Table 4-1) due to time constraints.

Even so, the first pass@1 results were disappointing—partly because the example inference scripts on Hugging Face provided by the developers of the models varied wildly between models and were often inefficient. After extensive testing, the author developed a single, unified inference routine using the PyTorch transformers library (Fig. 3-5) that works efficiently across all Hugging Face–hosted SLMs. This streamlined code was then used for all subsequent single-pass evaluations.

For both the reflection and multi-agent workflows, the following four lowest-performing models were excluded. Since these agentic approaches incur multiple inference calls per task, removing the slowest models helped keep overall execution time more manageable:

* Mistral 7B (lowest results)
* Solar-10.7B (a larger 10.7B model, but very low results).
* Phixtral-4x2\_8 (duplicates the results of Phixtral-2x2\_8).
* Mistral codestral\_mamba – suddenly discontinued by Mistral AI for unknown reasons (coincided with the announcement that OpenAI was buying WindSurf, after which Anthropic closed access to its best code generation LLMs).

Each model required three separate notebook runs—one per prompt template—across four datasets and 15 SLMs, for a total of 15 models × 3 prompts × 4 datasets = 180 notebook runs.

Each run issued between 160 (HumanEval, LBPP) and 500 (MBPP, BigCodeBench) SLM inference calls—about 2,320 calls per model-prompt-dataset combination. Over 180 runs, that adds up to 417,600 SLM inference requests in a single-pass cycle. When switched to the agentic workflows (only the top 11 models), the total dropped to roughly 300,000 calls per cycle.

Five types of experimental cycles were run—single-pass, reflection, multi-agent, temperature sweeps, and top\_p sweeps—which together required roughly 1.6 million inference calls. Because many of these cycles were repeated (e.g. single-pass runs with 24 models, then 15 models, then revised inference code; similarly, several repetitive runs for both agentic workflows), the total number of SLM inference calls made during this Praxis is on the order of 12 million.

The above GPU/API-based inference calls generated for each model a JSONL file listing all problems alongside the model’s code completions. Next, three post‐processing pipelines were run on each completion—raw, partial, and full cleaning—and each code completion variant was executed in a sandbox, producing a secondary JSONL with pass/fail indicators and any error messages. This rich dataset allowed measuring not only each model’s pass@1 score, but also the impact of cleaning strategies, prompt designs, error‐type distributions, and average inference latency. All of these analyses appear in this Chapter 4.

## 4.3 Effectiveness of Post-Processing Pipelines

### 4.3.1 Cleaning Stage Effectiveness

The three-stage post-processing pipeline (Raw Output, Fence Extraction Only, Full Cleaning Pipeline) demonstrated varying levels of improvement across different models. Table 4-1 compares pass@1 scores across the three cleaning tiers:

#### Table 4-1. Pass@1 Performance by Post-Processing Stage

|  | **Model** | **Raw Output** | **Fence Extraction** | **Full Cleaning** | **Best Method** |
| --- | --- | --- | --- | --- | --- |
| **1** | **Solar-10.7B** | 0.0268 | 0.5066 | 0.5417 | full |
| **2** | **mistral\_7b** | 0.0000 | 0.5608 | 0.5599 | partial |
| **3** | **phixtral-2x2\_8** | 0.0952 | 0.5122 | 0.6216 | full |
| **4** | **phixtral-4x2\_8** | 0.0952 | 0.5117 | 0.6220 | full |
| **5** | **Codegemma-7b-it** | 0.0000 | 0.6735 | 0.7020 | full |
| **6** | **mistral\_nemo** | 0.0488 | 0.6741 | 0.7153 | full |
| **7** | **Llama-3.1-8B** | 0.0027 | 0.7550 | 0.7770 | full |
| **8** | **mistral\_3b** | 0.0000 | 0.5790 | 0.7920 | full |
| **9** | **OpenCodeInterpreter-DS-6.7B** | 0.1182 | 0.8045 | 0.8256 | full |
| **10** | **deepseek-coder-6.7b** | 0.0997 | 0.7782 | 0.8300 | full |
| **11** | **mistral\_8B** | 0.0000 | 0.8472 | 0.8300 | full |
| **12** | **Artigenz-Coder-DS-6.7B** | 0.2807 | 0.8357 | 0.8450 | full |
| **13** | **codestral\_mamba** | 0.0638 | 0.7545 | 0.8455 | full |
| **14** | **Nxcode-CQ-7B-orpo** | 0.0018 | 0.8402 | 0.8556 | full |
| **15** | **CodeQwen1.5-7B-Chat** | 0.0018 | 0.8568 | 0.8652 | full |
| **16** | **mean** | 0.0556 | 0.6993 | 0.7486 |  |

A graph of a number of cleaning methods

AI-generated content may be incorrect.

##### Figure 4-1. Comparative Performance of Cleaning Methods

Figure 4-1 visualizes the same information. Mean results:

1. **Raw Output:** no cleaning; the results are definitely not useful - there are ~94% of non-executable code (syntax errors or human-like output).
2. **Fence Extraction:** stripping to first code block – much better as there are only ~30% of syntax errors. The overall improvement over raw output is almost 70%.
3. **Full Cleaning:** Comprehensive artifact removal – this method improves the pass@1 scores on average by ~5%.

The full cleaning pipeline improved executable code yield and accounted for nearly three-quarters of the syntax-error reduction, underscoring the paramount importance of post-processing in the automated evaluation of generated code.

### 4.3.2 Common Output Issues

Here are the common output issues associated with the generated code along with their frequencies:

1. **Fence presence:** fences occur in approx. 80% of the output. As item 4.2.1.3 states above, simply using the first fenced code block improves the pass rate by ~37%
2. **Main‐block removal:** if \_\_name\_\_ == "\_\_main\_\_" section occurs in ~15% of all the cases of generated code.
3. **Test-line pruning:** lines starting with assert or with print(function\_name) statements - unnecessary hallucinated test harnesses – occur in approx. 20% of the generated code. It was established empirically, that these lines don’t actually break the code execution at the code verification stage, so it should be fine to leave these lines in the generated code.
4. **Fence and placeholder cleanup:** stray fence markers and tags such as ``` or [code] occur in approx. 5% of cases. They would actually break the code execution and need to be removed.
5. **Signature and import reconciliation:** there were ~3% of generated code snippets that were lacking their original function signatures. SLMs would either “forget” to attach them to the final output or, in the case of code completion (HumanEval and BigCodeBench datasets), SLMs were completing the starter code from the point where it ended, completely omitting the starter code including all the import statements in the generated code.
6. **Typing import injection:** omitting the “from typing import \*” statement in ~7% of cases was one of the most common mistakes by SLMs which lead to code execution failures because this statement was needed either in the function header or less frequently in the tests.

## 4.4 Prompt Engineering Results

### 4.4.1 Impact of Prompt Complexity Tiers

The three-tier prompt hierarchy (Basic, Instructional, Full) revealed significant differences in model performance and output quality across all evaluated SLMs. Table 4-2 reports pass@1 scores under three prompt tiers (Basic, Instructional, Full) on the four benchmarks.

#### Table 4-2. Average Performance by Prompt Complexity Tier

| **prompt** | **HumanEval** | **BigCode** | **MBPP** | **LBPP** | **Mean** |
| --- | --- | --- | --- | --- | --- |
| **basic\_prompt** | 0.804831 | 0.759498 | 0.778919 | 0.726389 | 0.767409 |
| **prompt** | 0.807246 | 0.778853 | 0.755135 | 0.698611 | 0.759961 |
| **full\_prompt** | 0.745411 | 0.671326 | 0.752432 | 0.704167 | 0.718334 |

#### Table 4-3. Average Pass@1 Scores by SLMs by Prompt

|  | **Model** | **Basic Prompt** | **Instructional Prompt** | **Full Prompt** | **Best Method** |
| --- | --- | --- | --- | --- | --- |
| **1** | **olar-10.7B** | 0.561787 | 0.54499 | 0.518401 | basic\_prompt |
| **2** | **mistral\_7b** | 0.589476 | 0.572106 | 0.518262 | basic\_prompt |
| **3** | **phixtral-2x2\_8** | 0.604155 | 0.657863 | 0.602756 | prompt |
| **4** | **phixtral-4x2\_8** | 0.605499 | 0.657863 | 0.602756 | prompt |
| **5** | **Codegemma-7b-it** | 0.697922 | 0.714365 | 0.693777 | prompt |
| **6** | **mistral\_nemo** | 0.79985 | 0.770507 | 0.575523 | basic\_prompt |
| **7** | **mistral\_3b** | 0.815625 | 0.779999 | 0.780409 | basic\_prompt |
| **8** | **Llama-3.1-8B** | 0.820449 | 0.712323 | 0.798337 | basic\_prompt |
| **9** | **deepseek-coder-6.7b** | 0.834196 | 0.841352 | 0.814506 | prompt |
| **10** | **mistral\_8B** | 0.84499 | 0.82515 | 0.819989 | basic\_prompt |
| **11** | **OpenCodeInterpreter-DS-6.7B** | 0.848225 | 0.845216 | 0.783304 | basic\_prompt |
| **12** | **codestral\_mamba** | 0.852342 | 0.833569 | 0.850472 | basic\_prompt |
| **13** | **Nxcode-CQ-7B-orpo** | 0.873879 | 0.887842 | 0.805148 | prompt |
| **14** | **Artigenz-Coder-DS-6.7B** | 0.877397 | 0.858942 | 0.798542 | basic\_prompt |
| **15** | **CodeQwen1.5-7B-Chat** | 0.885347 | 0.897335 | 0.812829 | prompt |
| **16** | **mean** | 0.767409 | 0.759961 | 0.718334 | NaN |

* **Basic prompts** demonstrate the best pass score with a mean value of 0.77 across all models.
* **Instructional prompts** are the next best option with a slightly lower mean value of 0.76 across all models.
* **Full prompts** (strict “code-only” rules and stepwise instructions) shows a performance that is by ~6% lower than the basic prompt results.

These results demonstrate that simpler prompt can do a relatively good job in code generation. One would probably need a lot of experiments to come up with an efficient full prompt that would contain many rules and that would beat a simple prompt. Also, it should be noted that in reality, the prompt performance varies from model to model: the basic prompt works best for some models, while the instructional prompt works better for other models. In addition, the section dedicated to agentic workflow results demonstrates that the full prompt is most efficient to select models too.

A graph with green orange and blue lines and text

AI-generated content may be incorrect.

##### Figure 4-2. Comparative Performance of Prompting Strategies

### 4.4.2 Instruction Following Analysis

An important finding emerged regarding the varying ability of different SLMs to follow formatting and content instructions: in 50% of the cases, smaller SLMs like phixtral-2x2\_8 were more “talkative” and struggled with instruction adherence when the basic prompt was used. On the other hand, such top performers as Artigenz-Coder-DS-6.7B, CodeQwen1.5-7B-Chat, or Nxcode-CQ-7B-orpo, when coupled with the full prompt and even with the instructional prompt, generated much cleaner code that contained little to no human-like explanations in ~90% of the cases.

## 4.5 Baseline Performance Results

### 4.5.1 Overall Performance Across Benchmark Datasets

Figure 4-1 shows an example of a results table that was obtained for each benchmark dataset per each one-model run, each reflection and multi-agent run, each temperature setting run, etc. Overall, there is a total of 15 such tables to analyze.

A graph of numbers and a number

AI-generated content may be incorrect.

##### Figure 4-3. Example of Results for Each Run.

Table 4-4 summarizes the pass@1 results for each SLM on the HumanEval, MBPP, LBPP and BigCodeBench datasets computed as an average for all prompts. The results demonstrate significant variation in performance both across models and datasets, with several key patterns emerging from the evaluation. As expected, larger models tended to achieve higher accuracy on the public, but all models suffered substantial drops on the uncontaminated LBPP, confirming concerns about data leakage in more popular datasets.

#### Table 4-4. Ranking by Mean Pass@1 Across All Prompts (Single-Pass)

| **Model / Dataset** | **BigCode** | **HumanEval** | **LBPP** | **MBPP** | **Mean** | **Rank** |
| --- | --- | --- | --- | --- | --- | --- |
| **CodeQwen1.5-7B-Chat** | 0.707885 | 0.968599 | 0.798611 | 0.985586 | 0.86517 | 1 |
| **Nxcode-CQ-7B-orpo** | 0.691756 | 0.963768 | 0.777778 | 0.989189 | 0.855623 | 2 |
| **codestral\_mamba** | 0.817204 | 0.888889 | 0.902778 | 0.772973 | 0.845461 | 3 |
| **Artigenz-Coder-DS-6.7B** | 0.715054 | 0.874396 | 0.944444 | 0.845946 | 0.84496 | 4 |
| **mistral\_8B** | 0.885305 | 0.922705 | 0.75 | 0.762162 | 0.830043 | 5 |
| **deepseek-coder-6.7b** | 0.767025 | 0.917874 | 0.784722 | 0.85045 | 0.830018 | 6 |
| **OpenCodeInterpreter-DS-6.7B** | 0.673835 | 0.888889 | 0.881944 | 0.857658 | 0.825582 | 7 |
| **mistral\_3b** | 0.88172 | 0.886473 | 0.694444 | 0.705405 | 0.792011 | 8 |
| **Llama-3.1-8B** | 0.706093 | 0.782609 | 0.819444 | 0.8 | 0.777037 | 9 |
| **mistral\_nemo** | 0.598566 | 0.748792 | 0.777778 | 0.736036 | 0.715293 | 10 |
| **Codegemma-7b-it** | 0.713262 | 0.688406 | 0.6875 | 0.718919 | 0.702022 | 11 |
| **phixtral-4x2\_8** | 0.650538 | 0.640097 | 0.541667 | 0.655856 | 0.622039 | 12 |
| **phixtral-2x2\_8** | 0.648746 | 0.640097 | 0.541667 | 0.655856 | 0.621591 | 13 |
| **mistral\_7b** | 0.87276 | 0.456522 | 0.388889 | 0.521622 | 0.559948 | 14 |
| **Solar-10.7B** | 0.718638 | 0.519324 | 0.354167 | 0.574775 | 0.541726 | 15 |

The table displays the normalized average Pass@1 scores of 15 SLMs evaluated across the four datasets. The models are ranked based on their average performance (mean score across datasets).

At the top of the ranking is CodeQwen1.5-7B-Chat, with the highest overall mean score of 0.865, driven by consistently strong performance across all four datasets, particularly on MBPP and HumanEval. It is closely followed by Nxcode-CQ-7B-orpo and codestral\_mamba, both of which also achieve high scores above 0.84, showing that they are well-rounded and capable across diverse coding tasks.

Mid-ranking models such as mistral\_8B and deepseek-coder-6.7b perform well on certain datasets (e.g., HumanEval and MBPP) but have slightly lower scores on LBPP or BigCodeBench, which brings down their average. OpenCodeInterpreter-DS-6.7B rounds out the top 7 with a respectable mean of 0.825, showing consistent but not top-tier performance.

In the lower tier, models like mistral\_3b, Llama-3.1-8B, and mistral\_nemo show more variation in performance across datasets, leading to lower average scores in the range of 0.71–0.79. At the bottom of the ranking, Solar-10.7B and the phixtal variants score significantly lower, with Solar-10.7B having the lowest mean score of 0.542 — largely due to poor results on HumanEval and LBPP.

The analysis suggests that certain models, particularly those tuned for code (e.g., CodeQwen, Nxcode, Codestral), offer significantly better performance across benchmarks, while others lag behind likely due to weaker specialization or less effective fine-tuning for coding tasks.

While Table 4-4 presents the mean scores across all prompts for each dataset, Table 4-5 below does the same for the maximum pass@1 scores.

#### Table 4-5. Ranking by Max Pass@1 Across All Prompts (Single-Pass)

| **Model / Dataset** | **BigCode** | **HumanEval** | **LBPP** | **MBPP** | **Max** | **Rank** |
| --- | --- | --- | --- | --- | --- | --- |
| **CodeQwen1.5-7B-Chat** | 0.784946 | 1 | 0.8125 | 0.991892 | 1 | 1 |
| **Nxcode-CQ-7B-orpo** | 0.774194 | 0.985507 | 0.791667 | 1 | 1 | 2 |
| **Artigenz-Coder-DS-6.7B** | 0.763441 | 0.905797 | 0.979167 | 0.875676 | 0.979167 | 3 |
| **deepseek-coder-6.7b** | 0.801075 | 0.963768 | 0.791667 | 0.87027 | 0.963768 | 4 |
| **Llama-3.1-8B** | 0.801075 | 0.84058 | 0.958333 | 0.821622 | 0.958333 | 5 |
| **OpenCodeInterpreter-DS-6.7B** | 0.77957 | 0.905797 | 0.958333 | 0.862162 | 0.958333 | 6 |
| **codestral\_mamba** | 0.860215 | 0.905797 | 0.958333 | 0.789189 | 0.958333 | 7 |
| **mistral\_8B** | 0.892473 | 0.956522 | 0.791667 | 0.802703 | 0.956522 | 8 |
| **mistral\_3b** | 0.892473 | 0.913043 | 0.708333 | 0.748649 | 0.913043 | 9 |
| **mistral\_7b** | 0.88172 | 0.5 | 0.479167 | 0.57027 | 0.88172 | 10 |
| **mistral\_nemo** | 0.860215 | 0.818841 | 0.791667 | 0.756757 | 0.860215 | 11 |
| **Codegemma-7b-it** | 0.752688 | 0.724638 | 0.75 | 0.721622 | 0.752688 | 12 |
| **phixtral-2x2\_8** | 0.736559 | 0.65942 | 0.5625 | 0.672973 | 0.736559 | 13 |
| **phixtral-4x2\_8** | 0.736559 | 0.65942 | 0.5625 | 0.672973 | 0.736559 | 14 |
| **Solar-10.7B** | 0.736559 | 0.536232 | 0.395833 | 0.605405 | 0.736559 | 15 |

As you can see, the best performing and the worst performing models are the same, while rest of SLMs are slightly perturbed in their positions.

### 4.5.2 Dataset-Specific Performance

* **HumanEval:**
  + CodeQwen1.5-7B-Chat leads with 0.9686, followed closely by Nxcode-CQ-7B-orpo and mistral\_8B, both above 0.92.
  + Even mid-tier models like codestral\_mamba and OpenCodeInterpreter-DS-6.7B maintain strong scores around 0.89.
  + The worst performance on HumanEval comes from mistral\_7B (0.4565) and Solar-10.7B (0.5193), suggesting limited general code-writing ability or weaker reasoning capacity.
* **MBPP** shows a wide spread, but many models perform well on the 500 basic programming problems:
  + Nxcode-CQ-7B-orpo again excels (0.9892), along with CodeQwen1.5-7B-Chat (0.9856).
  + OpenCodeInterpreter-DS-6.7B also performs strongly here with 0.8577.
  + mistral\_7B and Solar-10.7B are again weakest, scoring just 0.5216 and 0.5748, respectively.
* **LBPP:** This was a critical stage of measuring SLM performance on the uncontaminated benchmark dataset, highlighting the significant performance drops observed when data leakage concerns are addressed, as anticipated in the methodology.
  + The highest score comes from Artigenz-Coder-DS-6.7B at 0.9444, making it the best LBPP performer, and not the usual suspects above - Nxcode-CQ-7B-orpo or CodeQwen1.5-7B-Chat – which raises legitimate concerns that maybe the latter two models were trained on the HumanEval and MBPP datasets. Hence, their perfect performance on them, but relatively moderate performance on LBPP and BigCodeBench!
  + codestral\_mamba also stands out at 0.9028, indicating strong long-form code generation.
  + Several models fall below 0.7, including mistral\_3b, Codegemma-7b-it, and mistral\_nemo, indicating challenges in longer context or structured outputs.
* **BigCodeBench** shows the most compressed distribution of scores:
  + mistral\_8B achieves the top score at 0.8853, followed closely by mistral\_3b and codestral\_mamba.
  + The lowest scoring model here is mistral\_nemo at 0.5986, which is significantly behind the top performers.
  + Interestingly, CodeQwen1.5-7B-Chat does not lead this dataset but maintains a respectable 0.7079 which again may confirm the fact the datasets where this model excels are contaminated.

### 4.5.3 Model Size vs. Performance Correlation

The results demonstrate a complex relationship between model size and code generation performance. First of all, small models can be surprisingly competitive:

* mistral\_3B (3B) ranks 8th overall with a mean score of 0.792, outperforming larger models like mistral\_nemo (mean: 0.715), Solar-10.7B (mean: 0.541), and even some mid-sized models.
* phixtal-4x2\_8 (small) ranks 12th, with a lower score (0.622) but still ahead of mistral\_7B and Solar, both of which are larger.

Conclusion: model size is not the sole determinant of performance. Secondly, largest models perform relatively poorly:

* Solar-10.7B, a larger model than 7B models, ranks 15th (last), with the lowest mean score (0.541).
* mistral\_nemo, another large model, ranks 10th with a moderate score (0.715), underperforming compared to many 7B models.

This suggests that larger size does not guarantee better performance. And finally, mid-sized (~7B) models dominate. The top 7 models — including CodeQwen1.5-7B-Chat, Nxcode-CQ-7B-orpo, codestral\_mamba, and others — are all around 7 billion parameters. These models consistently achieve mean scores above 0.82, with the top model scoring 0.865.

## 4.6 Reflection Agent Results

Table 4-6 shows the gains from the two-stage reflection workflow (baseline → self-critique). The reflection agent approach demonstrated certain improvements over baseline single-pass generation. On average, reflection agents increased pass@1 by +6 pp over the baseline (mean pass@1 score across all models is 0.805707, while for single-pass models it is 0.748568).

Table 4-6 and 4-7 present the results of ranking SLMs by their mean scores across all prompts and datasets and by their maximum scores, respectively.

#### Table 4-6. Ranking by Mean Pass@1 Across All Prompts (Reflection)

| **Model / Dataset** | **BigCode** | **HumanEval** | **LBPP** | **MBPP** | **Max** | **Rank** |
| --- | --- | --- | --- | --- | --- | --- |
| **Nxcode-CQ-7B-orpo** | 0.94605 | 0.973788 | 0.78519 | 0.973298 | 0.919581 | 1 |
| **CodeQwen1.5-7B-Chat** | 0.949904 | 0.982341 | 0.757187 | 0.964842 | 0.913568 | 2 |
| **Artigenz-Coder-DS-6.7B** | 0.867052 | 0.871537 | 0.899609 | 0.8251 | 0.865824 | 3 |
| **OpenCodeInterpreter-DS-6.7B** | 0.870906 | 0.860536 | 0.888169 | 0.831776 | 0.862847 | 4 |
| **mistral\_8B** | 0.947977 | 0.9014 | 0.823865 | 0.747664 | 0.855226 | 5 |
| **deepseek-coder-6.7b** | 0.849711 | 0.877636 | 0.767762 | 0.787717 | 0.820706 | 6 |
| **mistral\_3b** | 0.944123 | 0.889381 | 0.737605 | 0.683578 | 0.813672 | 7 |
| **Llama-3.1-8B** | 0.808285 | 0.694775 | 0.839564 | 0.686248 | 0.757218 | 8 |
| **mistral\_nemo** | 0.705202 | 0.742753 | 0.787161 | 0.70939 | 0.736127 | 9 |
| **Codegemma-7b-it** | 0.919075 | 0.666122 | 0.58481 | 0.671562 | 0.710392 | 10 |
| **phixtral-2x2\_8** | 0.701349 | 0.5577 | 0.551056 | 0.620383 | 0.607613 | 11 |
| **MEAN** | 0.864512 | 0.819812 | 0.765634 | 0.772869 | 0.805707 |  |

#### Table 4-7. Ranking by Max Pass@1 Across All Prompts (Reflection)

| **Model / Dataset** | **BigCode** | **HumanEval** | **LBPP** | **MBPP** | **Max** | **Rank** |
| --- | --- | --- | --- | --- | --- | --- |
| **Artigenz-Coder-DS-6.7B** | 0.994220 | 0.879766 | 0.900792 | 0.871829 | 0.994220 | 1 |
| **Nxcode-CQ-7B-orpo** | 0.947977 | 0.987934 | 0.871026 | 0.979973 | 0.987934 | 2 |
| **CodeQwen1.5-7B-Chat** | 0.953757 | 0.980723 | 0.897502 | 0.982644 | 0.982644 | 3 |
| **Llama-3.1-8B** | 0.968208 | 0.843999 | 0.783297 | 0.827770 | 0.968208 | 4 |
| **OpenCodeInterpreter-DS-6.7B** | 0.968208 | 0.872555 | 0.958759 | 0.830441 | 0.968208 | 5 |
| **mistral\_8B** | 0.959538 | 0.956609 | 0.763715 | 0.807744 | 0.959538 | 6 |
| **Codegemma-7b-it** | 0.953757 | 0.656219 | 0.646220 | 0.664887 | 0.953757 | 7 |
| **mistral\_3b** | 0.953757 | 0.894189 | 0.704967 | 0.752203 | 0.953757 | 8 |
| **mistral\_nemo** | 0.945087 | 0.807654 | 0.802879 | 0.728972 | 0.945087 | 9 |
| **deepseek-coder-6.7b** | 0.942197 | 0.923033 | 0.854185 | 0.798398 | 0.942197 | 10 |
| **phixtral-2x2\_8** | 0.907514 | 0.677853 | 0.587473 | 0.662216 | 0.907514 | 11 |
| **MEAN** | 0.954020 | 0.861867 | 0.797347 | 0.809734 | 0.960278 |  |

Here again, the best and the worst performing SLMs are similar to the non-agentic runs, but their pass@1 score are improved.

Figures 4-3, 4-4, 4-5, and 4-6 below demonstrate a comparison of baseline single-pass results and reflection agentic workflow on a per-dataset basis.

A graph with a line and a red line

AI-generated content may be incorrect.

##### Figure 4-4. Comparison of Baseline and Reflection Results: HumanEval

A graph with a line and a red line

AI-generated content may be incorrect.

##### Figure 4-5. Comparison of Baseline and Reflection Results: MBPP

A graph with red and blue lines

AI-generated content may be incorrect.

##### Figure 4-6. Comparison of Baseline and Reflection Results: LBPP

A graph with a red line and blue line

AI-generated content may be incorrect.

##### Figure 4-7. Comparison of Baseline and Reflection Results: BigCodeBench

The above 4 figures show that using a reflection-based agentic approach consistently improves Pass@1 performance, though the degree of improvement varies by dataset.

On HumanEval and MBPP, improvements are modest but consistent, with average gains of 0.43% and 0.42%, respectively. These results suggest that reflection helps models refine their answers slightly more effectively in relatively straightforward code generation tasks.

The effect is more pronounced in LBPP, where the reflection workflow results in a 2.53% average improvement. This indicates that reflection offers greater value in longer, more complex problem-solving scenarios where planning and iterative self-review benefit generation quality.

The greatest gains are seen in BigCodeBench, with an average improvement of 3.39%, showing that reflection substantially boosts performance in tasks that may involve more diverse code contexts or require greater generalization.

It is a critical correlation that the effect from the reflection is considerably greater in the less contaminated datasets. This can simply mean that if SLMs were trained on HumanEval and MBPP, they get the right answers right away and they don’t need improvements in the form of agents. While agents really help with newer datasets as it should be.

These results prove Hypothesis 1 of this Praxis (Section 1.6) confirming that agentic workflows result in higher test-case pass rates than single-pass SLMs, as measured across multiple benchmarks.

## 4.7 Multi-Agent Collaboration Results

Building on reflection, the multi-agent pipeline further improved pass@1 by +0.2pp. This is a very marginal improvement which most probably means that one needs to spend more time improving the design of the multi-agent system and doing additional prompt engineering. Providing the same tables below for general information, although they are not much different from the reflection agent results.

#### Table 4-8. Ranking by Mean Pass@1 Across All Prompts (Multi-Agent)

| **Model / Dataset** | **BigCode** | **HumanEval** | **LBPP** | **MBPP** | **Max** | **Rank** |
| --- | --- | --- | --- | --- | --- | --- |
| **Nxcode-CQ-7B-orpo** | 0.943383 | 0.982443 | 0.792998 | 0.976428 | 0.923813 | 1 |
| **CodeQwen1.5-7B-Chat** | 0.928602 | 0.981720 | 0.732175 | 0.960809 | 0.900826 | 2 |
| **Artigenz-Coder-DS-6.7B** | 0.864354 | 0.883363 | 0.887219 | 0.828980 | 0.865979 | 3 |
| **OpenCodeInterpreter-DS-6.7B** | 0.875232 | 0.870060 | 0.879713 | 0.834628 | 0.864908 | 4 |
| **mistral\_8B** | 0.921520 | 0.902369 | 0.804913 | 0.763270 | 0.848018 | 5 |
| **deepseek-coder-6.7b** | 0.852550 | 0.883937 | 0.763021 | 0.803358 | 0.825716 | 6 |
| **mistral\_3b** | 0.950525 | 0.888856 | 0.731689 | 0.698016 | 0.817272 | 7 |
| **Llama-3.1-8B** | 0.811555 | 0.713681 | 0.831107 | 0.689585 | 0.761482 | 8 |
| **mistral\_nemo** | 0.679504 | 0.754806 | 0.792930 | 0.716596 | 0.735959 | 9 |
| **Codegemma-7b-it** | 0.911509 | 0.671543 | 0.588351 | 0.674972 | 0.711593 | 10 |
| **phixtral-2x2\_8** | 0.710147 | 0.565415 | 0.576865 | 0.623372 | 0.618950 | 11 |
| **MEAN** | 0.858989 | 0.827108 | 0.761907 | 0.779092 | 0.806774 |  |

#### Table 4-9. Ranking by Max Pass@1 Across All Prompts (Multi-Agent)

| **Model / Dataset** | **BigCode** | **HumanEval** | **LBPP** | **MBPP** | **Max** | **Rank** |
| --- | --- | --- | --- | --- | --- | --- |
| **CodeQwen1.5-7B-Chat** | 0.972880 | 0.993170 | 0.895523 | 0.992521 | 0.993170 | 1 |
| **Nxcode-CQ-7B-orpo** | 0.941284 | 0.986096 | 0.879681 | 0.985912 | 0.986096 | 2 |
| **Artigenz-Coder-DS-6.7B** | 0.984235 | 0.897558 | 0.869590 | 0.875690 | 0.984235 | 3 |
| **OpenCodeInterpreter-DS-6.7B** | 0.982817 | 0.875691 | 0.945285 | 0.852983 | 0.982817 | 4 |
| **mistral\_8B** | 0.977162 | 0.950401 | 0.744663 | 0.829404 | 0.977162 | 5 |
| **mistral\_3b** | 0.955745 | 0.905931 | 0.729169 | 0.757814 | 0.955745 | 6 |
| **Llama-3.1-8B** | 0.938487 | 0.847695 | 0.793785 | 0.848708 | 0.938487 | 7 |
| **Codegemma-7b-it** | 0.936929 | 0.662949 | 0.640521 | 0.675971 | 0.936929 | 8 |
| **deepseek-coder-6.7b** | 0.915310 | 0.921267 | 0.822841 | 0.808089 | 0.921267 | 9 |
| **mistral\_nemo** | 0.904780 | 0.813799 | 0.824099 | 0.723918 | 0.904780 | 10 |
| **phixtral-2x2\_8** | 0.870374 | 0.675552 | 0.567215 | 0.679320 | 0.870374 | 11 |
| **MEAN** | 0.943637 | 0.866374 | 0.792034 | 0.820939 | 0.950096 |  |

These results also prove Hypothesis 1 of this Praxis (Section 1.6) confirming that agentic workflows result in higher test-case pass rates than single-pass SLMs, as measured across multiple benchmarks.

## 4.8 Hyperparameter Optimization Results

### 4.8.1 Temperature Settings Impact

The evaluation of different temperature settings (1.0, 0.5, 0.25, 0.1) across models run on the LBPP dataset revealed a certain impact on the code generation quality.

Figure 4-1 shows model performance on the LBPP dataset across different temperature settings. The temperature shows a steady effect on model performance according to the results. The average improvement from a baseline temperature of 1.0 (red line) reaches 0.34% when using a temperature of 0.5 but decreases by 0.07% when using a temperature of 0.25. The largest average improvement occurs when the temperature is set to 0.1 because it results in a 0.96% gain above the baseline.

A graph showing the difference between temperature and improvement

AI-generated content may be incorrect.

##### Figure 4-8. Temperature Effect

Model-specific trends vary. For lower-performing models such as Solar-10.7B and phixtal-2x2\_8, reducing the temperature consistently improves performance, suggesting that lower sampling entropy helps these models generate more accurate predictions. Similarly, models like deepseek-coder-6.7b and OpenCodeInterpreter-DS-6.7B show progressive gains as temperature decreases.

In contrast, certain models such as Llama-3.1-8B achieve their highest performance at the baseline temperature of 1.0, with performance declining at lower temperatures. This indicates that some models may already be well-optimized for open-ended generation under default conditions and might be negatively affected by reduced sampling diversity.

Artigenz-Coder-DS-6.7B demonstrates the highest individual performance overall at t = 0.1, showing that even high-performing models can benefit from temperature tuning.

These findings support the conclusion that, although optimal temperature may vary by model architecture and training regime, temperature scaling—particularly reducing to 0.1—can yield measurable improvements in the code generation task.

### 4.8.2 Top-p Parameter Effects

Several values for top-p (p∈{1.0, 0.9, 0.8}) were attempted and an analysis of results didn’t confirm any improvements in the pass@1 scores for any of the SLM models when top-p decreases.

### 4.8.3 Confirming Hypothesis 2

Hypothesis 2 states that adjusting SLM parameters, such as temperature and top-p, will improve code generation quality, as measured by test-case pass rates across multiple benchmarks. The above results confirm Hypothesis 2 for temperature.

## 4.9 SLM Fine-Tuning Results

Using a single GPU, the best-performing MBPP model Nxcode-CQ-7B-orpo, as well as a Mistral-family model that showed promising results Ministral 8B, were fine-tuned on the portion of the MBPP dataset that was not used for SLM evaluation. As shown in Table 4-2, the former achieved a +3pp improvement on MBPP, while the latter had a +5pp uplift on MBPP, demonstrating the effectiveness of domain-specific adaptation even in low-compute settings.

#### Table 4-10. Improved Performance of Fine-Tuned SLMs on the MBPP Dataset.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Baseline Pass@1 Score** | **Fine-Tuned Pass@1 Score** | **Gain** |
| **Nxcode-CQ-7B-orpo** | 0.74 | 0.7699 | +2.99% |
| **Ministral 8B** | 0.594 | 0.645 | +5.1% |

The task was achieved by fine-tuning a pre-trained SLM to follow human instructions more effectively, using modern techniques that optimize both performance and efficiency. These essential Python libraries were used: transformers, torch, datasets, peft, and trl. The model is loaded in an optimized format that significantly reduces memory usage, making it possible to fine-tune even on smaller machines. The process uses a parameter-efficient (PEFT) fine-tuning method utilizing a lightweight approach called Quantized Low-Rank Adaptation (QLoRA), which allows large models to be fine-tuned with minimal hardware resources by working with compressed (4-bit) versions of the model weights and modifying only a small subset of the parameters, instead of retraining everything from scratch. This approach is far more efficient and faster, while still allowing meaningful improvements in how the model behaves.

Each data point from the training dataset is converted into a clear format that the model can understand - beginning with a user instruction followed by the expected response. The training parameters such as the learning rate, batch size, and the number of steps were selected in such a way that it would balance the speed and performance. Once fine-tuned, the new model can be uploaded back to the Hugging Face Huband then easily deployed for inference.

This experiment uses existing open-source tools and datasets and demonstrates how advanced AI can be more adaptable and useful for specific business or user needs.

Hypothesis 3 states that fine-tuning SLMs on domain-specific data will increase test-case pass rates across multiple benchmarks compared to using SLMs without fine-tuning. The results described in this section confirm it.

## 4.10 Leaderboards

Figure 4-5 shows the top 25 LLMs on the EvalPlus Leaderboard (EvalPlus, 2025), ranked by their performance on the HumanEval and MBPP basic tests which were also used to score SLMs in this Praxis. The September 2024 “Q1 Preview” and “Q1 Mini” models lead both benchmarks, each scoring 96.3% pass@1 on HumanEval (tied for first) and 95.5%/93.1% on MBPP respectively. GPT-4o (Aug 2024) follows on HumanEval at 92.7%, while on MBPP the third spot is held by Qwen 2.5-Coder-32B-Instruct at 90.5%. Beyond the top three, specialized code-tuned models (e.g. Gemini 1.5 Pro, DeepSeek-Coder-V2-Instruct, Claude Sonnet 3.5) cluster in the high-80s on both suites, demonstrating that recent open and closed LLMs now routinely exceed 85–90% pass@1 on these core benchmarks.

A screenshot of a computer

AI-generated content may be incorrect.

##### Figure 4-9. EvalPlus Leaderboard for the Basic HumanEval and MBPP Tests

In the experiments under this Praxis, **CodeQwen1.5-7B-Chat** achieved a pass@1 of **84.15**% on HumanEval in a single‐pass setting and improved to **84.60**% when augmented with a reflection pass. Even more, **Nxcode-CQ-7B-orpo** reached **85.38**% under the multi-agent workflow, outperforming substantially larger models—such as Claude 3 Opus (Mar 2024), Gemini 1.5 Flash, DeepSeek-Coder-33B, Codestral-22B, OpenCodeInterpreter-DS-33B, Llama 3-70B, and speechless-codellama-34B—on the EvalPlus leaderboard.

On MBPP, the performance was more modest: **Nxcode-CQ-7B-orpo** achieved **74.0**% in a single-pass setting and **74.9**% with the multi-agent workflow, while **CodeQwen1.5-7B-Chat** reached **75.8**% under the multi-agent setup.

A screenshot of a computer

AI-generated content may be incorrect.

##### Figure 4-10. BigCodeBench Leaderboard

Figure 4-6 shows the results in the BigCodeBench leaderboard (BigCodeBench, 2025). It is a tight competition. GPT-4o (May 13, 2024) and DeepSeek-V3 share the top with a pass@1 score of 56.1%. A cluster of experimental and instruction-tuned models—Gemini-Exp-1202 (54.7%), Gemini-Exp-1114 (54.3%), DeepSeek-V2-Chat (54.1%) and DeepSeek-Coder-V2-Instruct (54.0%)—fills out the mid-54% range, while GPT-4-Turbo, Claude-3.5-Sonnet variants and other entrants are in low-50% band. The best performing SLM under this Praxis is Mistral 8B which scored 36.37% under the under the multi-agent workflow. The models in the leaderboard are significantly larger than Mistral 8B. Interestingly, one of the smallest models Mistral 3B scored 35.5% and looks very competitive.

The LBPP dataset, introduced by Matton A. et al. (2024), was designed as an uncontaminated, held-out benchmark for code generation. Although no official leaderboard has been published, the original paper reports that most models scored between 11% and 50% pass@1, with Claude-3.5-Sonnet as the sole outlier at 64%. In the Praxis experiments, **Artigenz-Coder-DS-6.7B** achieved **29.1**% under the multi-agent workflow, placing it in the middle of the reported performance range.

## 4.11 Analysis of Common Errors

Figure 4-7 shows the distribution of different exception types raised during the execution of the generated code.

A screenshot of a graph

AI-generated content may be incorrect.

##### Figure 4-11. Error Type and Counts Across Datasets.

The raw error messages were much longer; in the figure they are truncated so they could be grouped into buckets. The most frequent exception by far for three out the four benchmark datasets is AssertionError, which simply indicates that the generated code failed one or more test cases. Unfortunately, no further detail is available because the asserts did not include custom messages (they were written as assert <condition>). Even when a message like: “AssertionError: This prints if this assert fails 5 (good for debugging!)” is provided, it still does not reveal what specific condition was being evaluated in test case 5.

In addition, TypeError appears near the top in every dataset and becomes the number-one error on BigCodeBench, underscoring that argument mismatches and improper type usage are a universal weakness in autoregressive code models. SyntaxError and IndentationError show up most when the tasks require more elaborate code constructs (LBPP, BigCodeBench).

Below is an analysis of errors based on each of the four datasets:

1. HumanEval: The most frequent error is AssertionError (628 occurrences) which means that the generated code did not return the expected values. The next-most frequent errors are NameError (33), IndexError (31) and TypeError (29), indicating occasional typos or mis-indexed lists.
2. MBPP: AssertionError leads again (2,665), but TypeError (191) moves to the second place leaving NameError (159) behind. MBPP’s simpler function-generation tasks seem to produce more spurious type calls when arguments aren’t handled correctly.
3. LBPP: AssertionError still sits at the top (1,113), but TypeError (295) and SyntaxError (135) are both substantially more frequent than in HumanEval or MBPP, suggesting that longer or more complex problems in LBPP introduce more parsing mistakes along with failed test assertions. IndentationError (102) is also quite frequent.
4. BigCodeBench: This benchmark flips the script—TypeError is now the dominant failure (3,997), with SyntaxError (1,160) and IndentationError (127) in distant second and third. Because BigCodeBench covers a wide variety of real-world code snippets, many generations simply fail to run or misuse Python types before even reaching logical tests.

## 4.12 Latency and Cost Effectiveness

### 4.12.1 Latency

Table 4-3 shows the four benchmark datasets ranked by the average latency of one inference call, and Table 4-4 describes the same for each SLM. This information was parsed from the logs and aggregated by the corresponding group.

Across the four code-generation benchmarks, one can see a clear rise in latency as tasks become longer, more complex, and—critically—uncontaminated. MBPP runs fastest at 7.59 s per average sample and HumanEval at 7.74 s, both of which include some publicly exposed examples. In contrast, BigCodeBench (a non-contaminated, real-world code suite) averages 14.48 s, and LBPP (a strictly held-out, non-contaminated benchmark) is slowest at 18.40 s—reflecting their heavier test harnesses and the extra sandboxing overhead required to enforce clean evaluation.

#### Table 4-11. Average Latency by Dataset

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Average Latency** | **Rank** |
| MBPP | 7.587069 | 1 |
| HumanEval | 7.742248 | 2 |
| BigCode | 14.475852 | 3 |
| LBPP | 18.40398 | 4 |

#### Table 4-12. Average Latency by SLM

| **Model** | **Average Latency** | **Rank** |
| --- | --- | --- |
| nemo | 2.63 | 1 |
| mistral\_7b | 3.23 | 2 |
| mistral\_8B | 3.53 | 3 |
| codestral\_mamba | 3.83 | 4 |
| mistral\_3b | 4.32 | 5 |
| CodeQwen1.5-7B-Chat | 8.2 | 6 |
| deepseek-coder-6.7b-instruct | 8.84 | 7 |
| Nxcode-CQ-7B-orpo | 9.27 | 8 |
| codegemma-7b-it | 9.76 | 9 |
| OpenCodeInterpreter-DS-6.7B | 11.1 | 10 |
| Artigenz-Coder-DS-6.7B | 11.6 | 11 |
| Meta-Llama-3.1-8B-Instruct | 13.5 | 12 |
| phixtral-2x2\_8 | 27.6 | 13 |
| Nous-Hermes-2-Solar-10.7B | 30 | 14 |
| phixtral-4x2\_8 | 33.4 | 15 |

Inference latency also varies dramatically by model. The Mistral models (Nemo, 7B, 8B and Codestral Mamba) called via the Mistral AI API achieved the quickest turn-around (2.6–3.8 s) thanks to highly optimized remote endpoints. In contrast, all other 7B-scale models—CodeQwen1.5-7B-Chat (8.2 s), DeepSeek-Coder-6.7B (8.8 s), Nxcode-CQ-7B-orpo (9.3 s), Codegemma-7B-it (9.8 s) and Meta-Llama-3.1-8B-Instruct (13.5 s)—were run in Google Colab, where the inference on the A100 GPUs roughly double their per-sample times. Heavier or quantized variants like Phixtral-2x2\_8 (27.6 s) and Nous-Hermes-2-Solar-10.7B (30 s) suffered additional API and compute overheads.

Inference via Replicate.com proved the slowest: models like Qwen1.5-7B and Phixtral often required 200–300 seconds per call. This bottleneck dramatically extended the overall runtime, especially on larger benchmarks such as MBPP (≈500 samples), where the total evaluation time became prohibitive.

#### Table 4-13. Average Latency by Prompt

|  |  |  |
| --- | --- | --- |
| **Prompt** | **Average Latency** | **Rank** |
| complete\_task\_prompt\_full | 6.787204 | 1 |
| complete\_code\_prompt\_full | 10.64445 | 2 |
| complete\_task\_prompt | 10.76385 | 3 |
| complete\_task\_prompt\_basic | 13.15127 | 4 |
| complete\_code\_prompt\_basic | 13.4104 | 5 |
| complete\_code\_prompt | 13.79237 | 6 |

Table 4-5 describes average latency by prompt. Inference latency clearly correlates with prompt specificity and verbosity. The full-instruction templates run fastest. Dropping the “full” guardrails slows things down, and the slowest are the basic prompts. These results demonstrate that it is important where the time is spent: almost all of the wall-clock delay in these calls is in the token generation step (not “reading” the prompt). Therefore, a longer, more detailed “full” prompt actually shrinks the model’s output by telling the model to emit only the exact code and nothing else—so it ends up generating far fewer tokens overall. On the contrary, a vague basic brief prompt causes the model to ramble, repeat itself, or wrap code in extra explanation.

### 4.12.2 Platform Cost Comparison

Table 4-6 highlights costs by experiment by hosting platform. The Mistral AI API is by far the most cost-effective way to run code-generation experiments (just $0.01–0.15 per full benchmark run, depending on model). Widely used Google Colab Pro+ comes next at an estimated $0.65–2.00 per experiment followed by Replicate ($0.02–3.55), reflecting both lightweight models and heavier, quantized variants.

This breakdown underscores that for large-scale or frequent benchmarking, API-based access to optimized hosted endpoints can slash your bill compared to general-purpose notebook subscriptions or ad-hoc cloud calls.

#### Table 4-14. Cost Analysis by Hosting Platform

|  |  |  |  |
| --- | --- | --- | --- |
| **Platform** | **Average Cost Per Experiment** | **Comments** | **Rank** |
| Mistral AI API | $0.01-0.15 | Varies by model | 1 |
| Google Colab Pro+ | $0.65-2.00 | $50/month | 2 |
| Replicate | $0.02-3.55 | Varies by model | 3 |

Although Google Colab Pro+ is a flat $50/month subscription, it comes with only 500 compute units. Running one 160-sample experiment (e.g. on HumanEval or LBPP) consumed about 6.5 compute units—equivalent to roughly $0.65 per run ($2.00 for a 500-sample run).

# Chapter 5—Discussion and Conclusions

## 5.1 Discussion of Results

### 5.1.1 Key Findings Summary

The comprehensive evaluation revealed several critical insights:

1. **Post-processing is critical:** the three-stage pipeline raised average pass@1 from 5.6% (raw) to 70.0% (fence-extracted) and 74.9% (full cleaning), with full cleaning yielding the best executability across all models.
2. **Simpler prompts often win:** basic prompts achieved a mean pass@1 of 0.767 versus 0.760 for instructional and 0.718 for “full” prompts, underscoring that concise, clear directives can outperform elaborate guardrails.
3. **7 B models dominate:** CodeQwen1.5-7B-Chat (mean 0.865) and Nxcode-CQ-7B-orpo (0.856) top the leaderboards, while very small (< 4 B) and very large (> 10 B) models underperform, revealing a “sweet spot” around 7 B parameters.
4. **Data-leakage penalty:** all models suffer steep drops on uncontaminated LBPP and BigCodeBench (e.g. top models fall from ~98% on MBPP/HumanEval to ~80% on LBPP), validating the need for held-out benchmarks.
5. **Reflection boosts quality:** a two-stage self-critique workflow adds **+6 pp on average** (up to +3.4 pp on BigCodeBench), while a multi-agent pipeline delivers a further marginal **+0.2 pp** gain, suggesting diminishing returns without more refined agent design.
6. **Temperature tuning helps selectively:** lowering temperature to **0.1** yields up to **+0.96%** on LBPP, though optimal settings vary by model; nucleus sampling (top\_p) showed no consistent benefit.
7. **Fine-tuning proves effective:** QLoRA-based adaptation on MBPP gave **+3 pp** (Nxcode) and **+5 pp** (Mistral 8B), demonstrating that even lightweight, domain-specific tuning materially improves pass@1.
8. **Error modes expose weaknesses:** AssertionError dominates unit-test failures on structured benchmarks (HumanEval 628, MBPP 2665, LBPP 1113), while TypeError leads on BigCodeBench (3997), highlighting universal challenges with test-case correctness and type handling.
9. **Latency scales with complexity:** per-sample inference takes ~7.6 s on MBPP/HumanEval, ~14.5 s on BigCodeBench, and ~18.4 s on LBPP, reflecting longer prompts and heavier test harnesses.
10. **Execution environment matters:** Mistral-API models are fastest (2.6–3.8 s), Colab-hosted 7 B models run in ~8–13 s, and Replicate calls can spike to 200–300 s—crucial for throughput planning.
11. **Cost trade-offs:** Mistral AI API is most economical at **$0.01–0.15** per full run, Google Colab Pro+ (500 compute units/month) costs **$0.65–2.00** per experiment, and Replicate varies **$0.02–3.55**, guiding platform choice for scale.

## 5.2 Conclusions

This thorough assessment of small language models for code generation tasks reveals important details about SLM capabilities and deployment best practices. The results show that model size typically relates to performance but prompt engineering and hyperparameter optimization together with post-processing steps enhance the practical value of these models.

The assessment of agentic workflows indicates potential for improving code generation quality through iterative refinement but its effectiveness depends heavily on the specific model and problem type.

The research findings establish essential criteria for selecting and implementing small language models for applications related to code generation by weighing performance needs against implementation costs and deployment feasibility.

### 5.2.1 Model selection recommendations:

If absolute correctness is the only goal, these 7 B models are the ones to pick:

* **CodeQwen1.5-7B-Chat** (mean 0.865 pass@1) and **Nxcode-CQ-7B-orpo** (mean 0.856) clearly lead the 15-model experiments, topping every public benchmark (HumanEval, MBPP) and holding the #1–2 spots even on held-out tests.
* **Codestral Mamba** (#3, mean 0.845) complements the “best-of-breed” triad.

For the lowest latency, these models must be picked:

* + The **Mistral-family** models hosted on the **Mistral AI API** dominate on speed: **Mistral Nemo** (2.6 s), **Mistral 7B** (3.2 s), **Mistral 8B** (3.5 s) and **Codestral Mamba** (3.8 s) all respond in under 4 s per sample.

For the **best performance vs. cost trade-off**, this model must be picked**:**

* + Codestral Mamba (mean 0.845, 3.8 s, $0.02–0.15/run) performs well and fast enough.
  + Mistral 8B sits squarely in the middle—≈0.83 pass@1, 3.5 s latency, $0.01–0.15/run.

### 5.2.2 Efficiency Gains from SLMs

**5.2.2.1 API Costs**

The results demonstrate that using optimized SLMs instead of high-cost LLMs such as GPT-4 or Claude 4 leads to substantial cost savings for code generation. By extracting SLM API-call costs (Table 2-12) and average latency figures (Table 2-14) and then computing the same metrics for LLMs, one can directly compare their cost-and-performance trade-offs.

#### Table 5-1. Characteristics of Average Experiment

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Dataset** | **Code Snippets Generated Per Experiment** | **Total Prompt Tokens Per Experiment** | **Total Generated Tokens Per Experiment** | **Average Prompt Tokens Per One API Call** | **Average Generated Tokens Per One API Call** | **Average Latency Per One LLM API Call (seconds)** |
| **HumanEval** | 164 | 29246 | 29054 | 178.32 | 177.15 | 3.19 |
| **BigCodeBench Bench** | 500 | 158665 | 159818 | 317.33 | 319.64 | 5.75 |
| **LBPP** | 162 | 100586 | 85263 | 620.9 | 526.32 | 9.47 |
| **MBPP** | 500 | 85790 | 71265 | 171.6 | 142.53 | 2.56 |
| **AVERAGE** | 331.5 | **106743** | **96602** | 322 | 291.41 | 5.24 |

Table 5-1 shows the total and average numbers of input (prompt) and output (generated) tokens per one experiment under this Praxis. Using OpenAI o3 pricing [OpenAI 2, 2025] - $2 per 1 M input tokens and $8 per 1 M output tokens - 331 code completions cost: (106,743\*2 + 96,602\*8)/1M ≈ $0.99. By comparison, the Mistral AI API charges between $0.01 and $0.15 per experiment (Table 2-12), with a midpoint of $0.08. Therefore, at $0.08 vs. $0.99 per run, the Mistral API is roughly 92% cheaper than the OpenAI o3 model.

To demonstrate this using a purely hypothetical example:

* Organization: 100 developers
* Completions per developer per day: 40 (≈5 per hour over an 8-hour day)
* Total completions per day: 100 × 40 = 4,000
* Working days in a year: 260
* Annual completions: 4,000 × 260 = 1,040,000

**OpenAI o3 model cost:**

* $0.99 per 331 completions
* **Annual cost** = (1,040,000 ÷ 331) × $0.99 ≈ **$3,110**

**Mistral AI API cost:**

* $0.08 per 331 completions (average)
* **Annual cost** = (1,040,000 ÷ 331) × $0.08 ≈ **$250**

**Annual savings:** $3,110 – $250 = **$2,860** (≈ 92% cost reduction)

For larger teams, savings scale linearly with the number of developers and completions.

If a public API (for example, Mistral AI) can’t be used because you must keep sensitive data (Personally Identifiable Information (PII), etc.) entirely in-house, you can instead deploy an SLM on a GPU instance inside your organization’s virtual private cloud (VPC). Annual costs will vary by GPU type—more powerful cards deliver lower latency but carry higher hourly rates. This on-premises deployment is recommended solely to protect sensitive information.

Deploying an SLM locally - using frameworks such as Ollama (Ollama, 2025) or llama.cpp (Llama.cpp, 2025) - offers both maximal cost savings and complete data privacy. Because inference runs on your existing hardware, there are effectively zero incremental costs for code generation (aside from routine laptop wear-and-tear). However, only a limited selection of models is supported by these frameworks, and one would need to benchmark their pass-rate performance separately to compare them with the SLMs evaluated in this Praxis.

**5.2.2.2 Infrastructure Cost Reductions**

An 8B-parameter SLM model can be fine-tuned on a single GPU, as demonstrated in this Praxis. By contrast, a 70B-parameter LLM holds about 140 GB of weights in FP16 (16-bit floating-point - bytes per parameter) and therefore requires four 40 GB A100 GPUs to fine-tune. According to Amazon (2025), an on-demand g4dn.12xlarge instance (4 GPUs) costs $3.90/hour, while a single-GPU g4dn.2xlarge instance (32 GB RAM) costs around $0.75/hour.

**Hardware cost savings**: deploying or fine-tuning an SLM on one GPU instead of the multi-GPU setup needed for a 70B LLM cuts hardware expenses by roughly **84%**.

**5.2.2.3 Latency**

According to (OpenAI 3, 2025) the average latency of the GPT4 model is 18 milliseconds per generated token. With an average of 291.41 tokens per run (Table 5-1), each GPT-4 inference takes about 5.25 seconds. By comparison, Mistral 8B completes the same workload in 3.5 seconds (Table 4-12), making it roughly 33% faster than GPT-4. This shorter turnaround time directly reduces developer wait times and can boost productivity.

**5.2.2.4 Other Improvements**

As it was demonstrated in this Praxis, the reflection agentic workflow improves pass rates by an average of 6% while fine-tuning an SLM can improve pass rates by additional 5%. All these performance gains lead to a better quality of generated code and, therefore, to a shorter time the developers need to debug the generated code.

### 5.2.3 Substantiating the Thesis Statement

Going back to the Thesis Statement from section 1.4, the above conclusions and results have clearly demonstrated that SLMs ensure lower costs as compared to the use of LLMs. SLMs also provide sensitive data protection through deployment in private cloud or in a local resource-constrained environment.

### 5.2.4 Hypotheses Validation

As stated in Sections 4.6, 4.7, 4.8, and 4.9 of this research, Hypotheses 1 and 3 were fully confirmed, and Hypothesis 2 was confirmed for model temperature.

These results described in Section 4.6 of this research proved the correctness of Hypothesis 1 because agentic workflows resulted in higher model performance vs. single-pass SLMs, as measured across multiple benchmarks.

Hypothesis 2 states that adjusting SLM parameters, such as temperature and top-p, will improve code generation quality. Section 4.8 proved this hypothesis for temperature as reducing it to 0.1 yielded measurable improvements in the code generation task, however there was not enough evidence to consider it proved for top-p because no improvements were observed when the top-p parameter was decreased.

Finally, Section 4.9 proved the correctness of Hypothesis 3 stating that fine-tuning SLMs on domain-specific data will increase model performance compared to using SLMs without fine-tuning.

## 5.3 Contributions to Body of Knowledge

1. Validated a tiered cleaning pipeline: quantifies how simple fence‐extraction alone recovers ~70 pp of executability over raw output, and how a full cleaning pass yields a further ~5 pp gain.
2. Established “full cleaning” as the de facto standard for automated code evaluation.
3. Clarified prompt-complexity trade-offs: showed that, on average, a succinct “basic” prompt outperforms more elaborate “instructional” or “full” prompts—saving tokens, time, and developer effort.
4. Identified a 7 B-parameter sweet spot: empirically ranked CodeQwen1.5-7B-Chat, Nxcode-CQ-7B-orpo, and Codestral Mamba at the top of the four diverse benchmarks.
5. Exposed data-leakage effects by comparing performance on public (HumanEval/MBPP) vs. uncontaminated (LBPP/ BigCodeBench) datasets.
6. Measured agentic workflow gains
7. Discovered the error-mode distributions by aggregating thousands of failure cases
8. Characterized latency vs. task complexity
9. Compared cost/throughput trade-offs
10. Provided temperature and top\_p tuning insights
11. Demonstrated lightweight fine-tuning gains validating that QLoRA on a single GPU can yield +3–5 pp improvements - a viable path for domain-specific code-generation enhancements.

These findings create a reproducible framework for evaluating, deploying, and improving SLMs on the code generation task based on practical, cost-aware system design.

## 5.4 Recommendations for Future Research

Due to the initial exploration of the 4 datasets and running thousands of experiments on baseline and agentic performance, there was too little time to explore all other different options:

1. Repeat the temperature and top\_p experiments on the other three datasets and maybe modifying other inference-time parameters of the model.
2. Repeat the fine-tuning experiment with other SLMs and using other training datasets.

Improve the agentic performance of the SLM models by implementing additional steps, such as:

1. Improvements in prompt engineering using the conclusions based on the current results.
2. Additional error analysis leading to improved post-processing of generated code to achieve better code executability.
3. For each of the two agentic workflows, a “plain” agentic setup was utilized using a sequence of consecutive prompts or direct API calls to an LLM. However, an improved continuation of this work would be to attempt a LangChain-based implementation with additional built-in capabilities (LangChain 2025). LangChain is a development framework that simplifies building language model applications by chaining prompts, managing context and memory, and integrating external tools. In addition, LangGraph is a framework that organizes multi-agent workflows using graph-based structures to manage complex, branched interactions and robust error handling (LangGraph 2025). 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.

Other recommendations include implementation of various steps that can provide more insights on model performance with the purpose of improving the overall results:

1. Adjusting SLM inference parameters or fine-tuning SLM as part of agentic workflows.
2. Fine-tuning SLM using the Mistral AI API (Mistral AI (2)) instead of Google Colab with HuggingFace will lead to lower latency.
3. Analysis of performance stability across multiple evaluation runs, including an analysis of variance in stochastic code generation and its impact on benchmarking.
4. Comparison of results obtained from the same models hosted on different platforms, addressing potential implementation differences in API services.
5. Analysis of errors at a deeper level than datasets (per model and/or per prompt) to draw and leverage further insights.