UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Master's Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fulfilment of the Requirements for the Degree of

Master of Science

Assessing Large Language Models for Type Inference in python on real-world dataset

by
Rashida Bharmal

Thesis Supervisor:
Prof. Eric Bodden
&
Jun Prof. Mohammed Soliman

Paderborn, March 27, 2025

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet. Zur sprachlichen Überarbeitung einzelner Textpassagen wurde das KI-Werkzeug ChatGPT unterstützend eingesetzt. Die inhaltliche Ausarbeitung, Analyse und Bewertung erfolgten eigenständig.

Ort, Datum	Unterschrift

Abstract.
Python's dynamic type system offers flexibility but often leads to runtime errors and reduced maintainability in large-scale software systems. While optional type annotations (PEP 484) help mitigate these issues, they are inconsistently adopted across real-world codebases. To address this gap, recent studies have explored the use of Large Language Models (LLMs) for type inference, showing promising results on micro-benchmarks. However, their performance on real-world codebases remains underexplored.

This thesis investigates the effectiveness of LLMs for Python type inference using a real-world dataset. We extend the TypeEvalPy framework by incorporating the ManyTypes4Py dataset, enabling a comprehensive evaluation of LLM performance across frequent, rare, and user-defined types. Two state-of-the-art LLMs, Codestral (22B) and Qwen2.5-Coder (7B), are evaluated using two prompting strategies on micro-benchmark: mask-based prompting and question-and-answer (QnA) prompting. Furthermore, we apply Parameter-Efficient Fine-Tuning (PEFT) using LoRA to adapt these models to the type inference task.

Our results show that QnA prompting significantly outperforms mask-based prompting on the TypeEvalPy micro-benchmark. Codestral achieves an overall exact match accuracy of 88.7% with QnA prompting, compared to 67.8% with mask-based prompting. Qwen2.5-Coder improves from 61.5% to 83.6% using the same strategy. Fine-tuning further boosts performance: Codestral improves from 86.4% to 96.9% , and Qwen2.5-Coder from 84.0% to 93.8%. Analysis of frequent and rare types shows that fine-tuning enhances structured type inference while occasionally misclassifying generic types. These findings suggest that LLMs provide a robust solution for type inference in real-world scenarios, though improvements are needed for rare and user-defined types.

Contents

1 Introduction **1**

2 Background **4**
2.1 Type Inference in Python . . . 4
2.1.1 Motivating Example . . . 4
2.1.2 Existing Tools for Type Inference in Python . . . 5
2.2 TypeEvalPy - A Micro-benchmarking Framework for Python Type Inference Tools 6
2.3 Datasets for Python Type Inference . . . 7
2.3.1 Typilus Dataset . . . 7
2.3.2 Py150 Dataset . . . 8
2.3.3 ManyTypes4Py Dataset . . . 8
2.4 Large Language Models . . . 10
2.4.1 What are Large Language Models? . . . 10
2.4.2 Applications of LLMs . . . 10
2.4.3 Prompting Strategies . . . 11
2.4.4 Fine-Tuning . . . 11

3 Related Work **13**
3.1 Related Work . . . 13

4 Methodology **15**
4.1 Dataset Preprocessing . . . 15
4.2 Model Selection . . . 17
4.2.1 Selected Models . . . 18
4.3 Prompt Engineering . . . 18
4.3.1 Mask-based Prompting . . . 18
4.3.2 Question-and-Answer (QnA) Prompting . . . 19
4.3.3 Comparison of Prompting Strategies . . . 19
4.4 Fine-tuning . . . 20
4.4.1 Parameter-Efficient Fine-Tuning (PEFT) with LoRA . . . 20
4.4.2 Dataset Preparation and Prompt Structuring . . . 20
4.4.3 Fine-Tuning Pipeline . . . 20
4.5 Experimental setup . . . 22
4.5.1 TypeEvalPy extension . . . 22
4.5.2 Model Execution . . . 25
4.5.3 Hardware configuration . . . 25

5 Results **27**
5.1 Metrics . . . 27
5.1.1 Limitations of Precision Measurement . . . 27
5.2 RQ1- Prompting strategy . . . 27
5.2.1 Performance of Mask-Based Prompting . . . 28
5.2.2 Performance of Question-and-Answer (QnA) Prompting . . . 28
5.2.3 Comparison of Strategies and Models . . . 29
5.3 RQ2 - LLMs type inference on real-world dataset . . . 29
5.3.1 Performance with `Any` in Dataset . . . 29
5.3.2 Performance without `Any` in Dataset . . . 30
5.3.3 Performance on Built-in Types . . . 30
5.3.4 Comparison Between Codestral and Qwen2.5-Coder . . . 31
5.4 RQ3 - Finetuned LLMs type inference on real-world dataset . . . 31
5.4.1 Performance with `Any` in Dataset . . . 31
5.4.2 Performance without `Any` in Dataset . . . 32
5.4.3 Performance on Built-in Types . . . 32
5.4.4 Comparison of Baseline vs. Finetuned Models . . . 33

6 Discussion **37**
6.1 Most Questions Asked . . . 37
6.2 Most Frequent Types . . . 38
6.3 Rare Types . . . 39
6.4 Non-ubiquitous Types . . . 41

7 Limitations **43**

8 Conclusion **45**

Bibliography **47**

1 Introduction

The rapid adoption of dynamically typed programming languages such as Python has significantly influenced modern software development. With its concise syntax, extensive ecosystem, and high readability, Python enables fast prototyping and development, making it particularly appealing to startups, researchers, and data scientists. However, this convenience introduces a critical trade-off: Python's dynamic typing system, while flexible, lacks static guarantees about variable types and program behavior, leading to potential runtime errors, maintenance difficulties, and tooling limitations as projects grow in scale and complexity [CLJ20].

To address this, Python introduced optional type annotations through PEP 484 [PEP25], allowing developers to specify types for variables, function parameters, and return values. These annotations enhance code clarity, support better refactoring tools, and improve static analysis. Nonetheless, type annotations are not mandatory, and in practice, many real-world Python codebases remain sparsely annotated—particularly legacy systems or those developed in rapid, iterative cycles. Manual type annotation is often seen as tedious and error-prone, especially in large codebases with complex or user-defined types. As a result, automatic type inference has become a crucial area of research to bridge this annotation gap.

Traditional type inference systems rely heavily on rule-based static analysis. Tools like `mypy`, `Pyright`, and `Pyre` attempt to infer missing type information by analyzing code syntax, control flow, and data flow. While effective in some contexts, these tools struggle with Python's dynamic features such as monkey patching, dynamic imports, and reassignments [CLJ20]. They are also limited in dealing with ambiguous or incomplete type information, often failing silently or falling back to the overly generic *Any* type. These challenges have motivated the development of machine learning (ML)–based approaches that aim to infer types using statistical patterns learned from large corpora of annotated code.

Models such as `Typilus` [ABDG20], `TypeWriter` [PGLC20], and `Type4Py` [MLPG22] introduced data-driven techniques to predict types using variable names, usage contexts, and code embeddings. Hybrid systems like `HiTyper` [PGL+22] combine neural predictions with static rules to refine type inference results. Despite measurable improvements, these systems often struggle with rare or user-defined types due to limited training data and inherent biases toward frequent types in their training distributions.

In parallel, the emergence of Large Language Models (LLMs) trained on massive amounts of code and natural language—such as OpenAI's Codex [CTJ+21a], Meta's LLaMA [TLI+23b], and Mistral's Codestral [Mis24]—has transformed the field of code intelligence. These models exhibit strong zero-shot and few-shot reasoning capabilities, demonstrating potential in a wide range of software engineering tasks including code generation, summarization, completion, and

recently, type inference. Several recent studies, including by Venkatesh et al. [VMVW24], have evaluated LLMs on micro-benchmark datasets like TypeEvalPy [SVSW+24], which consist of curated code snippets annotated with ground truth types. Results from these evaluations reveal that LLMs consistently outperform traditional ML-based and static tools on micro-benchmark. However, their effectiveness on real-world, large-scale codebases remains underexplored, raising questions about scalability, consistency, and generalizability.

Goal. The goal of this thesis is to evaluate the applicability, scalability, and effectiveness of Large Language Models for type inference in Python on real-world dataset. By bridging the gap between micro-benchmark experiments and practical usage, this study seeks to assess whether LLMs can serve as reliable tools for enhancing code quality and maintainability in large-scale, dynamic Python projects.

To achieve this, we extend the TypeEvalPy framework by incorporating the `ManyTypes4Py` dataset [MLG21], which consists of over 183K Python files and around 869K type annotations. ManyTypes4Py provides a long-tail distribution of type usage, capturing a wide variety of frequent and rare types, including complex user-defined and parameterized types, thereby serving as a rigorous benchmark for LLM evaluation.

Two state-of-the-art LLMs—**Codestral (22B)** [Mis24] and **Qwen2.5-Coder (7B)** [Ali24] are selected for this study due to their strong performance in code-related tasks and computational efficiency. We explore two distinct prompting strategies: *mask-based prompting*, where type annotations are masked out and inferred by the model; and *question-and-answer (QnA) prompting*, where the model is explicitly asked to infer types based on line and column context. These prompts are evaluated using the TypeEvalPy micro-benchmark to determine the most effective strategy.

Furthermore, we examine the effect of *Parameter-Efficient Fine-Tuning (PEFT)* using LoRA (Low-Rank Adaptation) [HWLL24] to adapt these models to the type inference task using the ManyTypes4Py training set. We analyze how fine-tuning impacts the models' ability to predict function return types, parameter types, and local variable types, especially under challenging conditions such as rare type occurrences or ambiguous code snippets.

This work is guided by the following research questions:

- **RQ1:** What is the most efficient prompting strategy for type inference in Python on micro-benchmark?
- **RQ2:** How do Large Language Models perform with the selected prompting strategy for type inference on a real-world dataset?
- **RQ3:** To what extent does fine-tuning improve the accuracy of LLMs in Python type inference?

By answering these questions, this thesis offers the first comprehensive evaluation of LLM-based type inference on real-world Python code, provides insights into the effectiveness of fine-tuning for structured code tasks, and highlights the opportunities and limitations of LLMs as scalable tools for improving developer productivity and code quality.

Structure: This thesis is structured as follows:

In Chapter 2, we provide background information, including: (a) an overview of Python's type system and the motivation for type annotations, (b) a discussion of static, ML-based, and hybrid type inference techniques, and (c) an introduction to Large Language Models (LLMs),

along with prompting and fine-tuning strategies. Chapter 3 reviews related work, focusing on exploring the applications of LLMs for software engineering and static analysis tasks.

Chapter 4 presents the methodology, covering: (a) data preprocessing (b) LLM model selection, (c) prompt engineering, (d) implementation setup, including extensions to the TypeEvalPy framework. The results are presented in Chapter 5, where we address each research question in detail. These findings are further analyzed in Chapter 6, which discusses trends, limitations, and the performance of LLMs across different type distributions.

Chapter 7 discloses potential threats to validity and discusses reproducibility considerations. Finally, Chapter 8 concludes the thesis by summarizing key findings and suggesting directions for future research in LLM-based type inference.

2 Background

2.1 Type Inference in Python

Python is a dynamically typed language, meaning variables do not require explicit type declarations. While this feature enhances flexibility and developer productivity, it also introduces challenges for static analysis, refactoring, and tooling support. As projects grow in complexity, the lack of type information can lead to runtime errors that are difficult to detect early. `Type inference` attempts to bridge this gap by automatically predicting the types of variables, function arguments, return values, and class attributes, without requiring the programmer to explicitly annotate them.

2.1.1 Motivating Example

Consider the following Python code snippet:

```
1  def parse(data):
2      return json.loads(data)
3
4  parser = parse()
5  value = parser('{"x": 10}')
6  result = value + 5        # Type mismatch!
7
8  parser = int
9  value = parser("42")
10 result = value + 5        # Works
```

Listing 2.1: Motivating example demonstrating dynamic type behavior in Python

In this example, the variable `parser` is first assigned to the function `parse`, which returns the result of `json.loads(data)`—typically a dictionary.. Thus, on line 4, `value` becomes a `dict`, and the operation `value + 5` on line 6 leads to a type error. Later, `parser` is reassigned to the built-in `int` function. This time, `parser("42")` returns an integer, and the addition is valid.

A static type inference system, when applied to this code, should be able to determine the type of the variable `value` at different points in the program. Specifically:

- On line 4, `parser` is assigned to the function `parse`, which returns the result of `json.loads(data)`, a dictionary. Therefore, on line 5, `value` is of type `dict`, and the expression `value + 5` on line 6 is invalid, as Python does not support addition between a dictionary and an integer.

- On line 8, `parser` is reassigned to the built-in `int` function, which returns an integer. As a result, `value` on line 9 is of type `int`, and the operation `value + 5` on line 10 is valid.

This example illustrates the core challenge that type inference aims to solve: deducing the correct types of variables and expressions in the absence of explicit type annotations. In dynamically typed languages like Python, variables can be reassigned to values of different types throughout a program's execution. Without type annotations, static analysis tools must rely on contextual information and inference to identify the expected types.

A robust type inference system would detect that the statement `result = value + 5` is only type-safe in certain contexts. By inferring the type of `value` based on its assignment and usage, such a system could issue a warning or error for line 6 during static analysis, well before the code is executed.

This capability is especially valuable in large or legacy codebases where manual type annotations are missing or incomplete. Automated type inference enhances code reliability, supports intelligent tooling (e.g., auto-completion, refactoring), and helps catch bugs early in the development process.

2.1.2 Existing Tools for Type Inference in Python

To address the challenges posed by dynamic typing in Python, a range of tools and systems have been developed to support automatic type inference. These tools differ in their approaches, ranging from static analysis techniques to machine learning-based methods. This section provides an overview of the most prominent type inference tools and technologies.

mypy

`mypy` is one of the most widely adopted static type checkers for Python. It supports gradual typing based on the standard introduced in PEP 484 and performs limited type inference. `mypy` can infer simple types such as local variables, return types from return statements, and container types in simple contexts. However, its inference is constrained when annotations are missing, especially across function or module boundaries. Despite its limitations, `mypy` is useful in projects with partial or complete annotations, providing early error detection and integration with IDEs.

Pyright

`Pyright`, developed by Microsoft, is a fast and feature-rich static type checker. It performs deeper inference than `mypy`, particularly for union types, generics, and conditional type narrowing. Pyright supports strict and lenient typing modes and is well-integrated with development environments, especially in Visual Studio Code. Its performance and robust type analysis make it a popular choice in larger projects.

Pyre

`Pyre`, created by Meta, is designed for scalable type checking in large codebases. It features a modular analysis engine and supports both eager and lazy type evaluation. Pyre includes a tool called `infer`, which attempts to automatically infer missing type annotations and suggest them to developers. While effective for many patterns, Pyre still depends on static syntax and control flow, and may struggle with dynamically constructed types or runtime behavior.

HeaderGen

`HeaderGen` [VWLB23] is a static analysis driven tool designed to enhance the structure and readability of Python code in Jupyter Notebooks. Unlike conventional type inference tools, which focus solely on assigning types, HeaderGen aims to improve notebook comprehensibility by generating context-aware structural headers. It does this through a flow-sensitive call-graph analysis, which identifies the fully qualified names of all functions invoked throughout the notebook.

To perform this analysis, HeaderGen first constructs an assignment graph, capturing relationships between identifiers and their assigned values. It then enriches this graph with type information during fixed-point iterations, allowing it to infer the types of program identifiers. This iterative process enables HeaderGen to resolve types in a context-sensitive manner, even in the presence of reassignments and dynamic bindings.

Though its primary objective is notebook structuring, HeaderGen's type inference capabilities are a core part of its design. Its evaluation on micro-benchmark demonstrates strong performance, achieving a precision of 95.6% and recall of 95.3%, making it a reliable tool for both static analysis and notebook enhancement.

Type4Py

`Type4Py` [MLPG22] is a machine learning–based type inference system that uses deep similarity learning to infer types from unannotated Python code. Trained on a large, type-checked dataset, it embeds code and types in a high-dimensional vector space using a hierarchical neural network. Type4Py is capable of inferring variable, argument, and return types—even for user-defined and rare types. It significantly improves upon previous models like Typilus [ABDG20] and TypeWriter [PGLC20], and is available as a Visual Studio Code extension to support type auto-completion.

HiTyper

`HiTyper` [PGL+22] is a hybrid system that combines static analysis with deep learning. It uses a Type Dependency Graph (TDG) to encode relationships between variables and types. HiTyper iteratively refines type predictions by alternating between rule-based inference and neural network predictions, applying constraints to correct or discard invalid type guesses. This allows it to handle both static and dynamic patterns effectively. It has shown strong performance, particularly on rare and user-defined types, where traditional approaches often struggle.

2.2 TypeEvalPy - A Micro-benchmarking Framework for Python Type Inference Tools

Despite the growing interest in Python type inference, the field has lacked a standardized and comprehensive framework for evaluating the accuracy and effectiveness of inference tools. Most prior approaches relied on their own evaluation datasets and ad hoc scripts, making it difficult to compare different tools under consistent conditions. To address this gap, TypeEvalPy [SVSW+24] was introduced as a dedicated micro-benchmarking framework for Python type inference systems.

TypeEvalPy consists of a carefully curated set of 154 Python code snippets, annotated with 845 ground truth types, and organized into 18 distinct categories covering a broad range of Python features—such as function calls, class structures, control flow, exception handling, and data structures. Each category targets specific language behaviors, enabling fine-grained

evaluation of inference capabilities. The framework supports containerized execution of various type inference tools, providing a clean and reproducible testing environment. It also includes modules to standardize inferred type formats, normalize naming conventions, and compute a range of metrics such as exact match accuracy, category-wise scores, and summary statistics.

To enhance scalability and broaden its applicability, TypeEvalPy has been extended with an automatic benchmark generation capability, referred to as TypeEvalPy Autogen. This extension programmatically synthesizes new test cases by varying constructs found in the original benchmark. The autogen benchmark dramatically increases coverage, now consisting of:

- 7,121 Python files
- 78,373 ground truth type annotations

This expansion allows for more robust and statistically significant evaluations across tools, models, and categories of Python features. It also provides a platform to test generalization, rare type handling, and LLM performance at scale.

TypeEvalPy currently supports a wide range of type inference tools, including traditional analyzers like Jedi, Pyright, HeaderGen, Pyre; machine learning models such as Type4Py, HiTyper; and large language models such as Llama , mistral, qwen etc.

In summary, TypeEvalPy—together with its autogen extension—provides a rigorous, extensible, and tool-agnostic evaluation suite for benchmarking Python type inference methods. It plays a central role in this thesis by enabling consistent, large-scale comparisons across both traditional and modern approaches.

2.3 Datasets for Python Type Inference

Developing and evaluating type inference systems—particularly for dynamically typed languages like Python—requires access to high-quality datasets with reliable ground truth type annotations. Several such datasets have been proposed, varying in their scale, structure, and intended use. In this section, we focus on three widely used datasets: Typilus, Py150, and ManyTypes4Py.

2.3.1 Typilus Dataset

The Typilus dataset [ABDG20] is constructed from real-world Python projects that make use of type annotations, under the rationale that such projects are more likely to adopt type inference systems like Typilus. To collect this data, the authors selected 600 public GitHub repositories containing at least one type annotation, identified using regular expressions. These repositories were cloned, and static type information was further enriched using `pytype`, a static type inference tool. To enable `pytype` to analyze imported libraries effectively, the Python environment was enriched with the 175 most downloaded packages from `PyPI`, ensuring broader type coverage and realistic dependency resolution.

To address the issue of code duplication, which can significantly bias machine learning models, the dataset underwent deduplication using the approach of Allamanis et al. [All19]. This process removed over 133,000 (near) duplicate files, retaining only one file per cluster of similar files. The final dataset comprises 118,440 files containing approximately 5.99 million symbols, of which 252,470 have type annotations that are neither `Any` nor `None`.

The type annotations exhibit a heavy-tailed distribution: while the dataset includes around 24.7k distinct types, the top 10 most frequent types—such as `str`, `int`, and `bool`—make up nearly half of all annotations. In contrast, only 158 types appear more than 100 times, and the remaining 25,000+ types each occur fewer than 100 times, yet collectively constitute 32% of the dataset. These rarer types often include user-defined types and parameterized generics,

emphasizing the importance of inference models that perform well not just on common types but also across the long tail of less frequent, complex types.

For model development and evaluation, the dataset is split into training, validation, and test sets in a 70-10-20 ratio. A Docker container is also provided to reproduce the dataset construction process, along with the list of repositories and commit hashes used.

2.3.2 Py150 Dataset

The Py150 dataset [RBV16], introduced by Microsoft, consists of 150 Python projects drawn from open-source repositories. It is primarily composed of function-level code representations, including abstract syntax trees (ASTs), token-level information, and identifier usage. Although it lacks explicit type annotations, it can be augmented with tools like `pytype` or `mypy` to generate inferred types, making it suitable for training or evaluating type inference models.

The dataset's strength lies in its structured representation of code, which makes it well-suited for models that operate over syntax trees or code graphs, such as graph neural networks (GNNs). Py150 has been widely adopted for a range of machine learning tasks on code, including variable naming, code completion, and more recently, type prediction. Its compact yet rich structure makes it a foundational resource for learning from Python syntax.

2.3.3 ManyTypes4Py Dataset

To address the limitations of earlier datasets in capturing type diversity and real-world usage, **ManyTypes4Py** [MLG21] was introduced as a large-scale, long-tail benchmark for Python type inference. It comprises 5,382 open-source Python projects sourced from GitHub, containing over 183,000 source files and approximately 870,000 type annotations across 67,060 distinct types. These annotations span function arguments and return types, local and global variables, and class attributes.

Repositories were selected based on the presence of `mypy` in their declared dependencies, increasing the likelihood of consistent and well-maintained type annotations. To ensure data quality and avoid training/evaluation bias, an extensive deduplication process was performed using `CD4Py`, a custom tool developed by the authors. This tool uses TF-IDF vectorization and nearest-neighbor similarity detection to remove over 350,000 duplicate files across the corpus.

Each source file is processed using `LibSA4Py`, a lightweight static analysis pipeline that extracts type-relevant structural and contextual information from code. This includes abstract syntax tree (AST) features, docstrings, identifiers, and type hints. The dataset is formatted as structured JSON files and partitioned into training (70%), validation (10%), and test (20%) splits for use in machine learning pipelines.

A key strength of ManyTypes4Py lies in its representation of the long tail of type usage. While common types such as `str`, `int`, and `bool` dominate the corpus, the dataset also contains a substantial proportion of rare and user-defined types. This distribution challenges models to generalize beyond frequent annotations and motivates research into type inference for less common, semantically rich constructs.

ManyTypes4Py has become a widely adopted benchmark for evaluating and training type inference models. It supports a variety of downstream tasks, including type prediction, code completion, and representation learning for code. In this thesis, ManyTypes4Py serves as the primary dataset for model training and evaluation, integrated via the extended `TypeEvalPy` framework.

The JSON structure of the dataset is organized hierarchically across project, module, class, and function levels. Table 2.1 provides an overview of the key fields extracted and formatted by the `LibSA4Py` pipeline.

Table 2.1: Description of fields in the JSON files produced by the `LibSA4Py` pipeline in ManyTypes4Py [MLG21]

Field Name in JSON	Description
Project-level Fields	
`author/repo`	Project name and author from GitHub.
`src_files`	Paths to the project's source files.
`file_path`	Path to the specific source file.
Module-level Fields	
`untyped_seq`	Normalized seq2seq representation of the source code.
`typed_seq`	Type labels for tokens in `untyped_seq`, or 0 if missing.
`imports`	List of imported modules.
`variables`	Global variable names and their types.
`classes`	JSON object representing analyzed classes.
`funcs`	JSON object representing analyzed functions.
`set`	Data split label: `train`, `valid`, or `test`.
Class-level Fields	
`name`	Class name.
`variables`	Class variables and their types.
`funcs`	Methods defined in the class.
Function-level Fields	
`name`	Function name (in class or module).
`params`	Parameter names and their types, if available.
`ret_exprs`	Return expressions in the function.
`ret_type`	Return type, if annotated or inferred.
`variables`	Local variable names and their types.
`params_occur`	Locations of parameter usage in the function body.
`docstring`	Full docstring object (if available), with subfields:
`docstring.func`	One-line summary of the function.
`docstring.ret`	Description of the return value.
`docstring.long_descr`	Full function description.

These datasets offer complementary strengths: Typilus provides static-analysis-augmented real-world code, Py150 offers structured syntax representations for graph-based models, and ManyTypes4Py delivers a large-scale, type-rich corpus with strong generalizability. Table 2.2 summarizes their key characteristics.

Table 2.2: Comparison of Python Type Inference Datasets

Dataset	Scale	Type Source	Key Features
Typilus	118,440 files, 252k typed symbols	Developer + Pytype (inferred)	Real-world projects with explicit and inferred types; deduplicated using Allamanis' method; long-tail distribution analysis; supports reproducibility via Docker.
Py150	150 projects with function-level ASTs	No types (tool-augmented)	Structured code representations with ASTs and identifiers; used in variable naming, completion, and type prediction; can be augmented with type inference tools.
ManyTypes4Py	183k files, 870k annotations, 67k unique types	Developer (explicit)	Long-tail-focused benchmark; curated from 5,382 MyPy-typed projects; deduplicated with CD4Py; rich static context from LibSA4Py; JSON-formatted for ML.

2.4 Large Language Models

Large Language Models (LLMs) are a class of deep learning models trained to understand and generate human-like text by learning patterns from massive text corpora. Based primarily on the Transformer architecture [VSP+17], LLMs are typically trained using self-supervised objectives such as next-token prediction or masked language modeling. These models are capable of capturing rich linguistic, semantic, and even task-specific knowledge across a broad range of domains.

2.4.1 What are Large Language Models?

LLMs are characterized by their scale, both in terms of the number of parameters (often billions or more) and the size of the training data. Examples of prominent LLMs include OpenAI's GPT-3 [BMR+20], Google's PaLM [CND+23], Meta's LLaMA [TLI+23a], and Mistral [J+23]. These models are pretrained on diverse datasets such as books, websites, source code, and social media content, allowing them to generalize to a wide variety of language-related tasks.

The strength of LLMs lies in their emergent capabilities: with increased scale, they exhibit few-shot and zero-shot learning, where models perform tasks with little or no task-specific training. As a result, LLMs are no longer just tools for language modeling, but general-purpose reasoning engines that can be applied across a variety of domains.

2.4.2 Applications of LLMs

LLMs have shown impressive performance across a wide range of applications, including:

- **Natural Language Understanding:** Question answering, text classification, summarization, and sentiment analysis.

- **Natural Language Generation:** Dialogue systems, storytelling, content creation, and translation.
- **Programming and Code Understanding:** Code completion, generation, summarization, and type inference using models such as Codex [CTJ+21b], CodeGen [NPH+22], and StarCoder [LAZ+23].
- **Education and Assistance:** Intelligent tutoring systems, writing aids, and language translation services.
- **Scientific and Legal Domains:** Automated report generation, literature synthesis, and document analysis.

LLMs are increasingly being embedded in productivity tools, IDEs, and cloud APIs, enabling interaction through natural language prompts without explicit programming or configuration.

2.4.3 Prompting Strategies

Prompting refers to the technique of conditioning a language model to perform a desired task by providing it with a textual instruction or example. Prompting has become a powerful method to leverage pretrained LLMs without additional training.

There are several key prompting strategies:

- **Zero-Shot Prompting:** The model is asked to perform a task without any examples, relying entirely on the instruction provided in the prompt.
- **Few-Shot Prompting:** The prompt includes a few input-output examples to demonstrate the desired behavior. This technique was popularized by GPT-3 [BMR+20].
- **Chain-of-Thought Prompting:** The model is encouraged to generate intermediate reasoning steps before producing an answer, improving performance on complex reasoning and arithmetic tasks [WWS+22b].
- **Self-Consistency:** An ensemble of multiple chain-of-thought responses is sampled, and the most consistent answer is selected [WWS+22a].
- **System Prompting:** Used in chat-based models (e.g., ChatGPT), where a system-level prompt defines the model's persona, style, or behavior.

Prompting is a low-resource alternative to model modification. However, it may be sensitive to phrasing, formatting, and model size, leading to inconsistent results.

2.4.4 Fine-Tuning

While prompting allows task adaptation without model updates, fine-tuning refers to the process of continuing model training on a specific dataset to specialize it for a downstream task. This can significantly improve performance, especially when large amounts of task-specific data are available.

There are two main approaches:

- **Full Fine-Tuning:** All model parameters are updated. This method is resource-intensive and typically applied to smaller LLMs or in research contexts.

- **Parameter-Efficient Fine-Tuning (PEFT):** Only a small subset of parameters are trained, using methods such as LoRA [HSW+22], adapters [HGJ+19], or prefix tuning [LL21]. These techniques reduce memory requirements and training cost while achieving comparable results to full fine-tuning.

Fine-tuning enables LLMs to adapt to domain-specific data, handle structured inputs (such as code, tables, or APIs), and align with user intent or ethical constraints. In this thesis, fine-tuned models are used to improve type inference performance on Python code, using the ManyTypes4Py dataset.

3 Related Work

3.1 Related Work

In this section, we review recent developments in leveraging large language models (LLMs) for software engineering tasks. These tools and approaches span a wide spectrum of software engineering applications, including autonomous coding, debugging, test generation, vulnerability detection, and static analysis.

LLMs for Software Engineering

Recent advances in LLMs have enabled the development of intelligent agents capable of performing complex software engineering tasks. **SWE-Agent** [YJW+24] demonstrates the feasibility of LLMs acting autonomously across the software development lifecycle by breaking down tasks into subtasks, using external tools such as version control systems, and iteratively refining outputs.

Tools like **CodiumAI** [Cod23] and **Pythagora** [AI23] enhance developer productivity by generating tests and ensuring functional coverage through LLM-backed reasoning. CodiumAI focuses on generating meaningful tests for untested code paths, while Pythagora combines LLMs with test-driven development (TDD) to automatically build backend services from API specifications. **Cody** by Sourcegraph [Sou23] further extends LLM usage into IDEs by leveraging code intelligence and repository context to support real-time development and navigation.

Multi-agent frameworks such as **Agent4SE** [LWC+24] and **OpenHands** [WLS+24] represent a shift toward modular, extendable architectures where LLM agents can coordinate tasks such as writing code, executing commands, and managing planning steps. These systems highlight the growing role of LLMs as collaborative, tool-using entities within modern software pipelines.

LLMs for Vulnerability Detection

LLMs are also being investigated for their ability to detect and explain security vulnerabilities in source code. **GPTScan** [SWX+24] combines LLMs with static analysis tools to detect vulnerabilities in smart contracts. The authors report that GPTScan improves detection performance by leveraging LLMs' capacity for reasoning over code logic, outperforming traditional symbolic analyzers in detecting logic-based vulnerabilities such as re-entrancy and authorization bypass.

More recently, **LProtector** [SWZ+24] was proposed as an LLM-driven vulnerability detection system capable of scaling to large codebases. LProtector uses a multi-step pipeline that includes code understanding, vulnerability reasoning, and explanation generation. It formulates vulnerability detection as a code-to-vulnerability mapping task and leverages pre-trained LLMs to both detect and explain vulnerabilities in natural language. The system supports various types of vulnerabilities and programming languages, and initial evaluations indicate improved precision and interpretability compared to traditional static analyzers.

These tools highlight the advantages of LLMs in capturing semantic patterns and security smells that might not be explicitly defined in rule-based systems. However, they also face limitations in deeper program analysis, which often requires interprocedural reasoning and symbolic verification.

LLMs for Static Analysis and Debugging

LLMs are increasingly being applied to static analysis and debugging tasks, where they help interpret source code, identify bugs, and reason about program behavior. These tools offer a more semantic and context-aware approach than traditional rule-based static analyzers.

SkipAnalyser [MAH+23] enhances static code analysis with LLMs to provide more insightful and context-sensitive results. Built on Meta's Skip framework, it utilizes LLMs to generate summaries and trace bugs across call graphs, helping developers comprehend large and complex systems more effectively than with traditional rule-based static analysis tools.

CodePal [Cod24] and **Tabnine** [Tab23] integrate LLMs directly into the developer's environment, assisting with refactoring, linting, and offering code suggestions. These tools blur the line between interactive IDE support and static analysis by embedding intelligent assistance within the coding workflow.

Venkatesh et al. [VSM+24] conducted a comprehensive evaluation of 26 LLMs for both type inference and call graph analysis. Their findings show that while LLMs often outperform traditional methods in type inference, they struggle with deeper program analysis tasks like accurate call graph resolution. This highlights the current limitations of LLMs in reasoning over long-range code dependencies and the need for further fine-tuning and tool augmentation. Their work motivates the need for broader cross-language evaluations (e.g., Python, Java, JavaScript) and benchmarking LLM-based tools against existing static analyzers in real-world settings.

4 Methodology

In this section, we describe the methodology used to evaluate LLMs for type inference in real-world Python codebases. Our approach involves: (1) preparing the dataset, (2) selecting the LLMs, (3) designing effective prompts for type inference, (4) fine-tuning models using PEFT technique, and (5) setting up and running the experiment pipeline. We discuss each of these aspects in detail in the following sections.

4.1 Dataset Preprocessing

The ManyTypes4Py dataset, which serves as the foundation for our experiments, required extensive preprocessing to reconstruct and standardize the data for compatibility with our evaluation framework. This section outlines the major steps undertaken in that pipeline.

The original dataset does not directly include raw source code; instead, it provides metadata linking to GitHub repositories and commit hashes, along with tokenized annotations. To enable precise type evaluation and downstream analysis, we reconstructed the dataset by mapping each annotation to its corresponding source code file from the specified commit in the original repository.

We began by cloning all listed repositories in the dataset following the official data preparation instructions. During this process, we encountered several repositories that were either unavailable, missing commits, or structurally altered. Projects with unresolved source files were excluded to maintain dataset integrity.

Once the repositories were downloaded, we processed the project-level JSON files, each of which contains metadata for one or more Python source files. For every '.py' file object, we attempted to locate the corresponding file in the local repository using the relative file path. If the source file existed, its content was read and embedded back into the JSON object under the `source_code` key. Files without matching source code were skipped. If none of the files in a project had resolvable source code, the entire project-level JSON was discarded to avoid retaining incomplete data.

Each successfully processed '.py' file was categorized into one of the predefined splits—`train`, `test`, or `valid`—as indicated in the dataset metadata. These groupings were preserved, and the files were stored into split-specific JSON files: `train.json`, `test.json`, and `valid.json`.

Following this split, we performed a translation step to convert the files into the format required by the TypeEvalPy benchmark. This involved extracting type facts for function return types, parameter types, and local variable types. Each extracted type was normalized by (1)

removing redundant module prefixes such as `builtins` and `typing`, and (2) replacing excessively nested types (depth > 2) with `Any`, following Type4Py's preprocessing standards [MLPG22]. The final outputs were saved as `train_gt.json`, `test_gt.json`, and `valid_gt.json`.

Additionally, to enable both practical and strict evaluation settings, we created two versions of each split: one with all type annotations (including `Any`) and one with all `Any` annotations removed. The inclusion of `Any` reflects realistic usage in dynamic Python code and evaluates models under practical conditions, while the cleaned version focuses on the model's ability to infer concrete types. This dual setup allows for a more balanced assessment of type inference performance.

An overview of dataset statistics across all versions and splits is presented in Table 4.1, and the complete preprocessing pipeline is visualized in Figure 4.4.

Table 4.1: Dataset Statistics

Category	**Dataset**		
	Train	**Test**	**Valid**
Files			
All	46,416	12,771	5,086
Without "Any"	40,194	11,093	4,456
Function Return Type (FRT)			
All	41,844	11,422	4,527
Without "Any"	41,249	11,224	4,473
Function Parameter Type (FPT)			
All	55,263	14,577	5,960
Without "Any"	53,408	14,082	5,750
Local Variable Types (LVT)			
All	248,072	67,576	27,602
Without "Any"	142,156	38,457	15,759
Total Facts			
All	345,179	93,575	38,089
Without "Any"	236,813	63,763	25,982

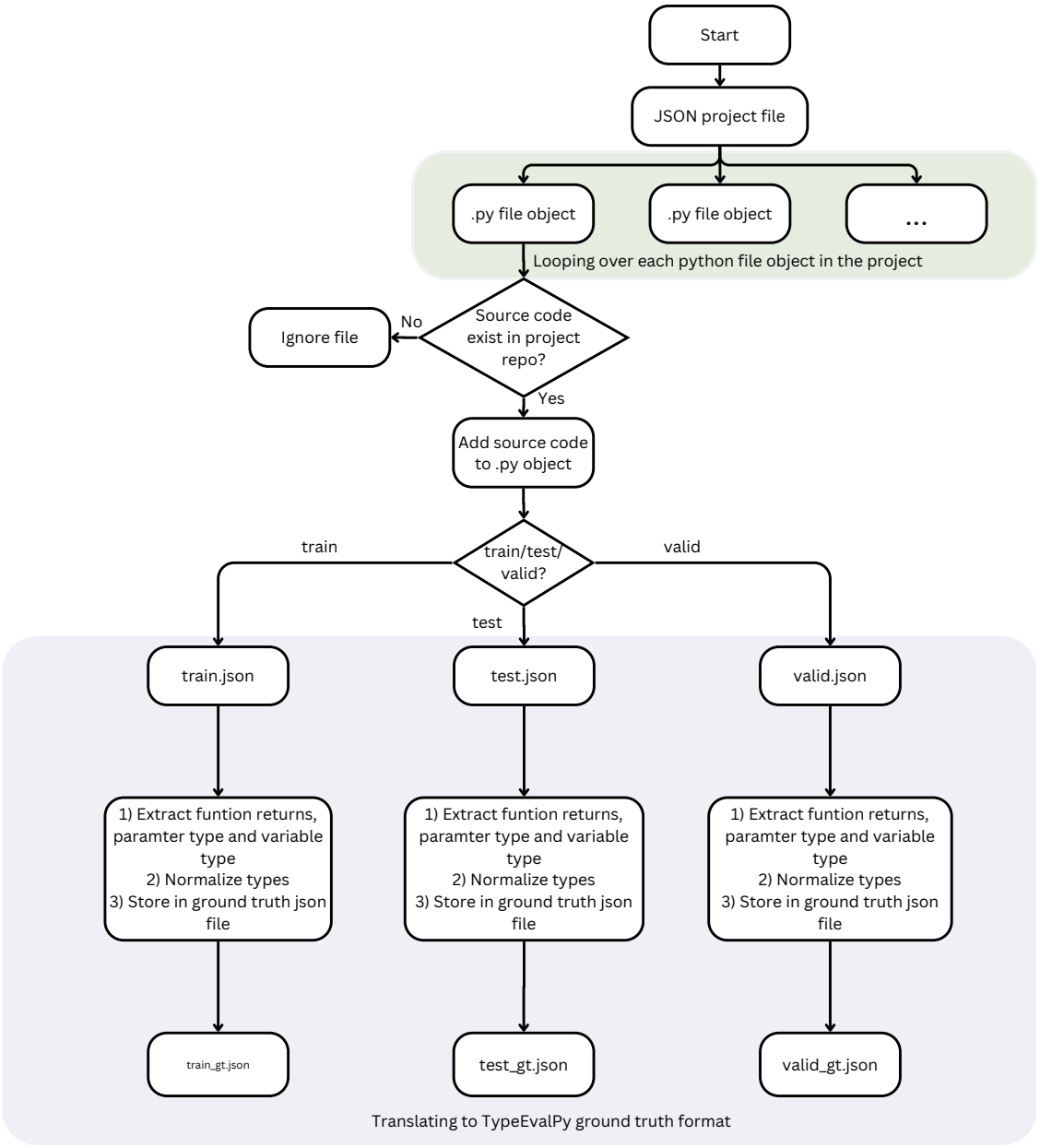

Figure 4.1: Data preprocessing pipeline for ManyType4Py dataset.

4.2 Model Selection

Selecting appropriate large language models (LLMs) for type inference in Python is a crucial step in ensuring the validity and effectiveness of our methodology. Given the goal of leveraging

real-world datasets—specifically ManyTypes4Py—we prioritized models that exhibit strong performance in code understanding and type prediction, while also being computationally efficient and accessible for broader experimentation.

4.2.1 Selected Models

For this study, we selected two state-of-the-art LLMs that strike a balance between high performance and resource efficiency. These models are not only popular in the code intelligence community but also compact enough to be deployed on modest hardware, making them suitable for reproducible research.

Codestral (22B)

Codestral is a 22-billion parameter model optimized for programming-related tasks such as code generation and understanding. It has demonstrated strong performance in recent research on Python type inference, particularly on micro-benchmark datasets like TypeEvalPy [SVSW+24]. These evaluations suggest that Codestral is well-suited for structured code analysis tasks, including type prediction in controlled scenarios.

Given its performance on curated benchmarks and its accessibly open model weights, Codestral serves as a strong candidate for further empirical evaluation on more challenging, real-world datasets such as ManyTypes4Py.

Qwen2.5-Coder (7B)

Qwen2.5-Coder is a 7-billion parameter model fine-tuned for code generation and multi-language understanding. Although primarily designed for general code tasks, its moderate size and efficient performance make it a viable option for large-scale type inference. Evaluating this model helps explore how well mid-sized LLMs perform in structured code understanding tasks such as type prediction.

4.3 Prompt Engineering

In this study, we explore two distinct prompting strategies for type inference in Python: a question-and-answer (QnA) prompt based on the TypeEvalPy framework and a mask-based prompt.

We begin by evaluating both prompts using the TypeEvalPy micro-benchmark, which offers a controlled environment for systematically assessing their effectiveness. The selected prompting strategy is then used for large-scale evaluation on the ManyTypes4Py dataset.

4.3.1 Mask-based Prompting

The Mask-based prompting strategy removes explicit type annotations and requires the model to infer missing type information from the remaining code context. In this approach, all type hints in Python source code are replaced with the placeholder [MASK], prompting the model to predict the most appropriate type for each masked element.

To implement this transformation, we use `LibCST`, a Python library for non-destructive code modification. The tool systematically traverses function signatures, parameter lists, return types, and variable assignments, replacing any explicit type annotation with [MASK]. This produces a prompt where all type annotations are missing, requiring the model to infer and replace them based on context. Listing 4.1 provides an example of this strategy.

```
**Python Code Provided**:
```main.py
def param_func(x: [MASK]) -> [MASK]:
    return x

c: [MASK] = param_func("Hello")
```

Task Description
Your task is to replace '[MASK]' with the most appropriate Python type annotations for
    all function return types, variable annotations, and function parameters.
Strict Requirements:
1. Maintain the exact same structure, formatting, and indentation as in the input code.
2. Do not alter the line numbers or remove existing blank lines.
3. Do not add any additional blank lines or comments.
4. Do not add any explanations or extra information in the output.
5. Only return the annotated version of the code.
6. Ensure proper and consistent type annotations wherever applicable.
```

Listing 4.1: Section of masked prompt for type inference in instruction format used in the study

4.3.2 Question-and-Answer (QnA) Prompting

The QnA-based prompting strategy formulates type inference as a structured question-answering task. Instead of relying on implicit inference, the model is explicitly asked about the types of function return values, function parameters, and local variables within a given code snippet.

Each query follows a structured format, referencing the exact line and column numbers of the target element within the provided Python code. This structured approach ensures that the model directly focuses on specific elements of the code, reducing ambiguity in type inference. Listing 4.2 provides an example of this prompting approach.

```
**Python Code Provided**:
```main.py
def param_func(x):
    return x

c = param_func("Hello")
```

Questions:
1. What is the return type of the function 'param_func' at line 1, column 5?
2. What is the type of the parameter 'x' at line 1, column 15, in the function 'param_func
    '?
3. What is the type of the variable 'c' at line 4, column 1?
```

Listing 4.2: Section of question and answer prompt for type inference in question-answer format used in the study

4.3.3 Comparison of Prompting Strategies

We compare both prompting strategies using an exact match metric on the TypeEvalPy micro-benchmark, assessing performance across three categories: Function Return Type (FRT), Function Parameter Type (FPT), and Local Variable Type (LVT).

For each strategy, we measure exact match performance, where a prediction is considered correct only if it exactly matches the ground truth annotation. Empirical results showed that

QnA prompting was more effective than Mask-based prompting, which led us to adopt QnA prompting for subsequent experiments on the ManyTypes4Py dataset.

A detailed comparison of the two prompting strategies, along with evaluation results, is presented in Section 5.2.

4.4 Fine-tuning

4.4.1 Parameter-Efficient Fine-Tuning (PEFT) with LoRA

To specialize Large Language Models (LLMs) for the task of Python type inference, we employed Parameter-Efficient Fine-Tuning (PEFT), specifically using LoRA (Low-Rank Adaptation). Unlike full fine-tuning, which updates all model weights, PEFT only modifies a small subset of parameters by inserting lightweight, trainable adapters into specific layers of the model—namely, the query, key, value, and projection layers.

This approach offers several key benefits:

- **Low memory usage:** Only a fraction of the model is updated, making training feasible on consumer GPUs.
- **Faster training:** Reduced parameter count leads to quicker convergence.
- **Preserved knowledge:** Since the majority of the model remains frozen, it retains its original pre-trained capabilities.

We applied this fine-tuning strategy to both Codestral and Qwen2.5-Coder (7B) models, enabling them to adapt to real-world type inference tasks efficiently.

4.4.2 Dataset Preparation and Prompt Structuring

We fine-tune both Codestral and Qwen2.5-Coder models using the ManyTypes4Py training dataset, applying the QnA-based prompt structure discussed earlier (see Listing 4.2). The dataset is converted into a structured format adhering to the `{instruction, input, questions, output}` schema, following the methodology outlined by Wang et al. [WKM+23].

We created two training sets:

- One including `Any` types: 345,179 QnA pairs across 46416 files.
- One excluding `Any` types: 236,813 QnA pairs across 40194 files.

This allowed us to assess how ambiguous or underspecified types affect model learning.

4.4.3 Fine-Tuning Pipeline

To run fine-tuning efficiently, we used the `Unsloth` library—an optimized wrapper for training LLMs on limited hardware. It supports advanced features such as:

- **Low VRAM training:** Enables fine-tuning large models in 4-bit precision
- **RoPE (Rotary Positional Embedding) scaling:** Allows longer input sequences
- **Gradient checkpointing:** Reduces memory usage by recomputing intermediate steps

The fine-tuning configuration is designed to balance performance and resource usage. We used:

- LoRA rank: 16
- LoRA alpha: 16
- Dropout: 0 (disabled for optimization)
- Targeted modules: Attention and feedforward layers (`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`)

The training process was managed using HuggingFace's `SFTTrainer`. Below is the configuration we used in code, with key arguments explained:

```
trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="messages",  # Field containing QnA-style prompts
    max_seq_length=4000,            # Allows long-context training
    args=TrainingArguments(
        per_device_train_batch_size=2,       # Micro-batch size
        gradient_accumulation_steps=4,       # Effective batch size: 8
        num_train_epochs=2,                  # Fast fine-tuning
        learning_rate=2e-4,                  # Tuned for LoRA efficiency
        fp16=True or bf16=True,              # Mixed precision for speed
        optim="adamw_8bit",                  # Memory-efficient optimizer
        output_dir="outputs",                # Where to save results
    )
)
```

Listing 4.3: Finetuning configuration

The `SFTTrainer` is responsible for managing the fine-tuning process. Below, we explain the key configuration parameters used in Listing 4.3:

- `model`, `tokenizer`: The pre-loaded language model and tokenizer, optionally with LoRA adapters applied.
- `train_dataset`: The dataset loaded using the HuggingFace `datasets` library, consisting of QnA-formatted examples derived from ManyTypes4Py.
- `dataset_text_field="messages"`: Specifies the field within each dataset entry that contains the formatted QnA-style prompt for training.
- `max_seq_length=4000`: Defines the maximum input token length for each training example. Extended context is crucial for modeling long functions or class bodies.

TrainingArguments:

- `per_device_train_batch_size=2`: The number of samples processed per device per training step. Lower values reduce GPU memory usage.
- `gradient_accumulation_steps=4`: Accumulates gradients across four steps to simulate a larger effective batch size ($2 \times 4 = 8$).
- `num_train_epochs=2`: Number of full passes over the training dataset. A low value helps avoid overfitting and speeds up training.

- `learning_rate=2e-4`: Step size used by the optimizer to update weights. Tuned for stable LoRA training.
- `fp16=True` or `bf16=True`: Enables mixed-precision training to improve throughput and reduce memory consumption. `bf16` is preferred on supported hardware (e.g., Ampere/Hopper GPUs).
- `optim="adamw_8bit"`: Utilizes a memory-efficient 8-bit version of the AdamW optimizer, ideal for quantized or large-scale models.
- `output_dir="outputs"`: Directory to save training checkpoints and logs.

This configuration strikes a balance between resource efficiency and performance, enabling effective fine-tuning of large-scale models on modest hardware setups.

The impact of this fine-tuning configuration on model performance is discussed in Section 5.4.

4.5 Experimental setup

4.5.1 TypeEvalPy extension

Adapter: For real-world dataset

To extend TypeEvalPy for large-scale inference on real-world Python code, we implemented a new adapter that facilitates interaction with Large Language Models (LLMs) via the Transformers module. While TypeEvalPy previously supported LLM-based inference, the existing adapters were primarily designed for micro-benchmark. The newly introduced adapter allows for the seamless processing of large-scale datasets, ensuring compatibility with real-world codebases such as ManyTypes4Py.

This extension is responsible for managing prompt generation, inference execution, and result formatting. It processes the ManyTypes4Py test set by systematically extracting function return types, parameter types, and variable assignments. The adapter also enables compatibility with multiple model backends, including Codestral, Qwen2.5-Coder (7B), and OpenAI models, making it possible to evaluate different LLM architectures under a unified framework. Additionally, GPU memory management and batch processing mechanisms are integrated to prevent resource exhaustion and optimize inference speed. Once the models generate predictions, the adapter ensures that all outputs are reformatted into the TypeEvalPy ground truth structure, allowing for direct performance comparisons across models.

By implementing this adapter, we extend TypeEvalPy's evaluation capabilities beyond micro-benchmarking, enabling a scalable and reproducible method for assessing LLM-based type inference in real-world Python projects.

Prompt Generation

For evaluation, we generated structured prompts from the preprocessed ManyTypes4Py test set, ensuring that all test cases were consistently formatted for inference across models.

The prompt generation process follows a systematic sequence. First, our evaluation script parses the JSON-formatted test files, extracting relevant type annotations from ManyTypes4Py. Each Python file is paired with its corresponding ground truth type information. The system then maps each test instance to a structured prompt format, aligning model inputs with the reference annotations.

Once all type-related information is extracted, we apply the QnA-based prompting strategy described in Section 4.3, converting each type query into a model-friendly question. This ensures that inference inputs maintain semantic consistency with the training and fine-tuning phases.

The final test set contains 93,575 QnA pairs in the version including Any types, and 63,763 QnA pairs in the version excluding them. These two test sets allow us to evaluate model performance under different assumptions about type specificity and ambiguity.

Transformer pipeline

Once the prompts are generated, they are passed through a Transformer-based inference pipeline, which enables efficient model execution and response retrieval. The pipeline is designed to handle large-scale inference requests while optimizing batch sizes and computational resources to accommodate different model architectures.

The inference process begins by sorting and batching the generated prompts, ensuring that input sizes are optimized for memory efficiency. Since different test cases have varying token lengths, the script dynamically adjusts batch sizes to balance processing speed while preventing out-of-memory errors. Once the batches are prepared, they are sent to the selected LLM backends. These models process the structured prompts, generating type predictions for each test case.

Following inference, the model-generated outputs are parsed and reformatted into the TypeEvalPy evaluation structure, ensuring that predictions can be directly compared with ground truth type annotations. The script performs post-processing to clean the outputs, validate their structure, and store them in the results directory for further evaluation.

By integrating batch management, inference execution, and structured result parsing, the Transformer pipeline ensures that the ManyTypes4Py evaluation is both scalable and reproducible.

Batching Strategy and GPU Memory Utilization

To ensure efficient inference across large-scale datasets while preventing memory overflow, we implemented a dynamic batching strategy based on prompt token length and available GPU memory. The goal was to maximize batch size while staying within the 78 GB memory constraint of the NVIDIA H100 GPU.

Figures 4.2 and 4.3 illustrate the memory utilization patterns of two different LLM backends—Codestral and Qwen2.5-Coder—under varying batch sizes and token limits. Each bubble represents a unique configuration of token limit and batch size, where: (a) The x-axis corresponds to the token limit per prompt. (b) The y-axis shows the batch size. (c) The color and size of each bubble indicate the memory usage in MB (darker and larger means higher memory consumption).

From the visualizations, we observe the following patterns:

- Inverse relationship: As token limits increase, the maximum feasible batch size decreases for both models. This is expected, as higher token counts per prompt require more GPU memory.
- Model-specific trends: Codestral appears to reach peak memory usage (55 GB) at lower batch sizes compared to Qwen2.5-Coder. This indicates that Codestral has a higher memory footprint per token.
- Optimal points: The optimal batch size for a given token limit is often where memory usage approaches, but does not exceed, 70–72 GB. This threshold was chosen to maintain stability and avoid OOM (Out-of-Memory) errors.

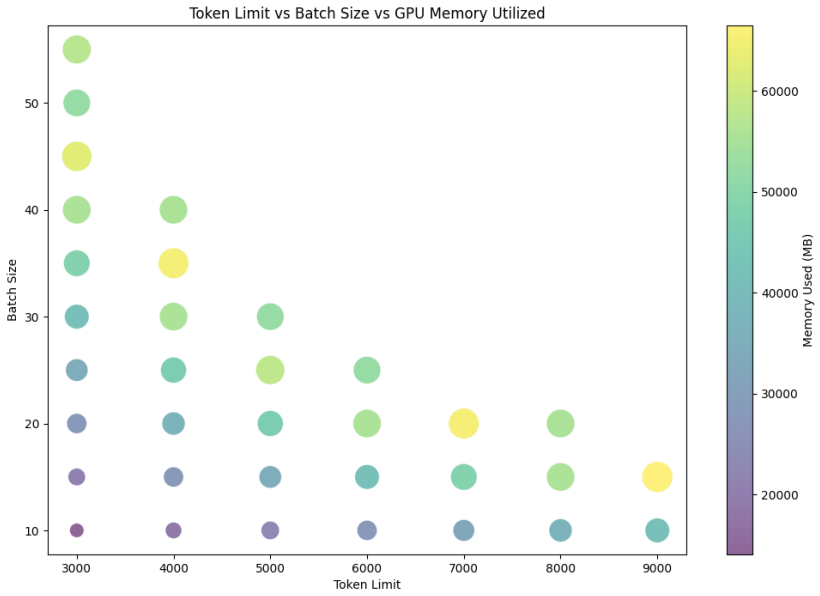

Figure 4.2: GPU Memory Utilization for `Codestral` under varying token limits and batch sizes.

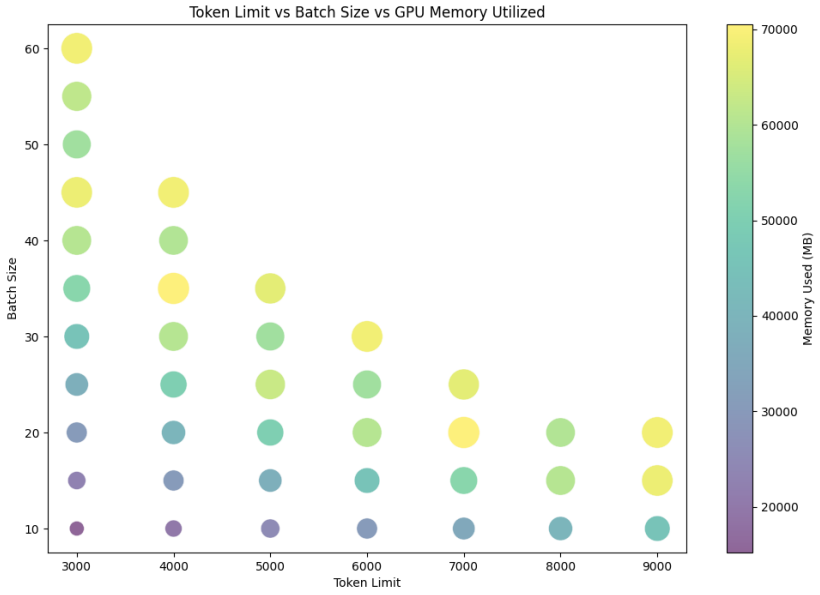

Figure 4.3: GPU Memory Utilization for `Qwen2.5-Coder` under varying token limits and batch sizes.

- Scaling trade-offs: While larger batches offer better throughput, the token limit must be restricted accordingly. In contrast, smaller batches can accommodate longer inputs, which is crucial for complex or nested Python functions.

These insights guided our dynamic batching system, which adjusts batch size in real-time based on the token length distribution of incoming prompts. This adaptive approach allowed us to achieve consistent throughput without manual tuning for each model or test case.

4.5.2 Model Execution

To accommodate hardware constraints, all models are executed in 4-bit quantization mode, allowing efficient inference and fine-tuning while minimizing memory usage. This quantization setting enables the deployment of large language models (LLMs) on limited-resource systems without significantly impacting performance.

To ensure fair comparison, both the baseline and fine-tuned models are evaluated using the same data processing and evaluation pipeline. This consistency eliminates variations in preprocessing or metric computation, ensuring that performance differences are attributable solely to the model configuration.

Figure 4.4 provides a visual overview of the complete type inference pipeline described in this section, including data preprocessing, prompt generation, model execution, and evaluation.

4.5.3 Hardware configuration

All methodologys are conducted on the following hardware configuration: 1×NVIDIA H100-80C GPU, 16×Intel(R) Xeon(R) Platinum 8462Y+ CPU, 78 GB RAM

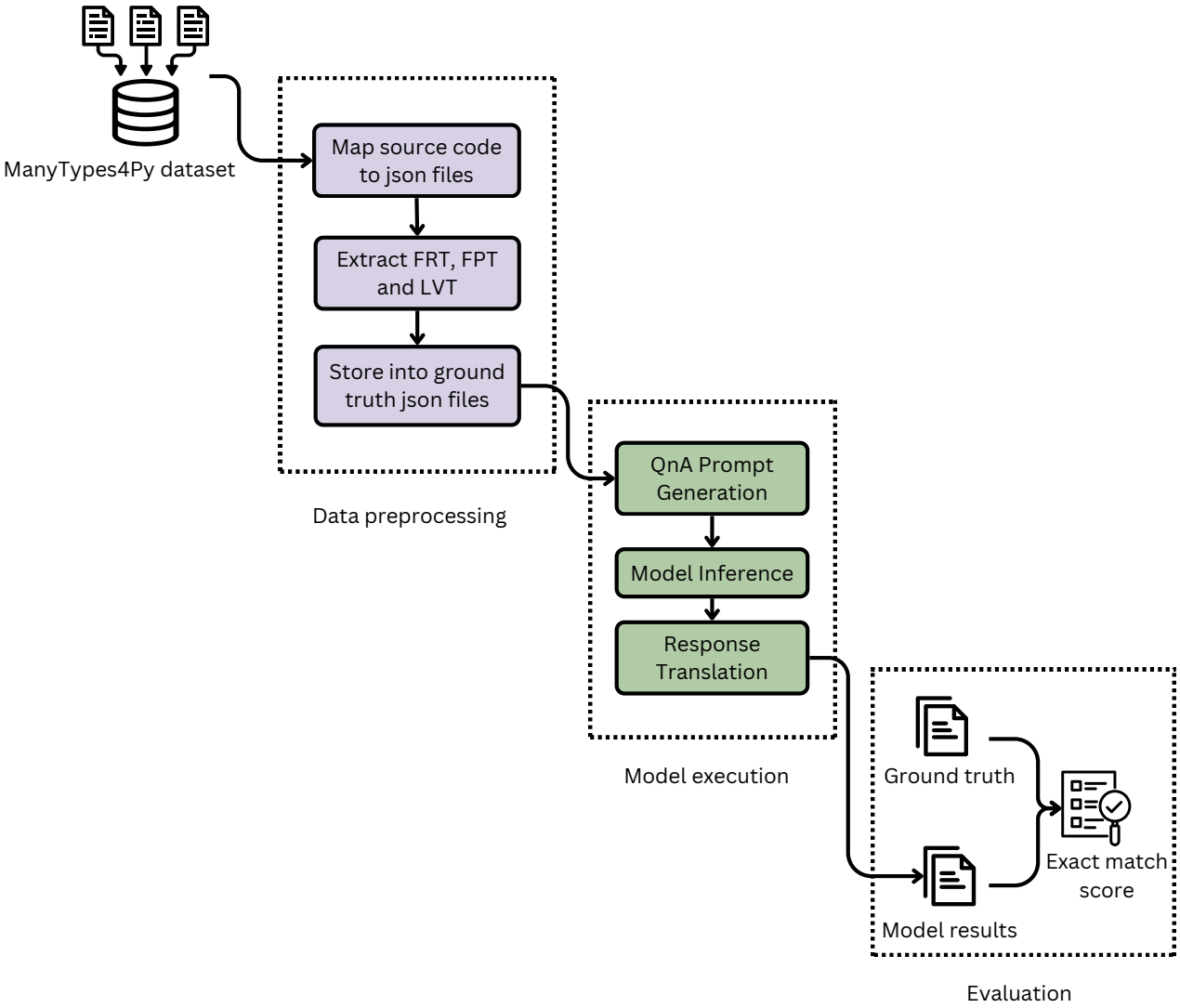

Figure 4.4: Overview of the type inference pipeline used in this study.

5 Results

5.1 Metrics

We evaluate model performance using exact matches, a widely used metric in type inference research. A prediction is considered correct if the predicted type annotation is identical to the ground truth annotation. Given a predicted type T_p and a ground truth type T_g, an exact match occurs when:

$$T_p = T_g \tag{5.1}$$

This metric ensures a strict evaluation where only completely correct predictions contribute to the accuracy score, providing a robust measure of model performance. We evaluate LLMs across three type categories: (1) Function Return Type (FRT), (2) Function Parameter Type (FPT), and (3) Local Variable Type (LVT).

5.1.1 Limitations of Precision Measurement

We do not report precision values due to the nature of how ManyTypes4Py constructs its ground truth annotations. The dataset is based on developer-annotated codebases rather than a fully labeled dataset with exhaustive type coverage. ManyTypes4Py collects type annotations from real-world Python projects using a static analysis pipeline, but these projects do not annotate all possible elements. Consequently, missing annotations cannot always be treated as incorrect, making precision measurement unreliable in this context.

5.2 RQ1- Prompting strategy

To determine the most effective prompting strategy for type inference in Python, we evaluated two approaches: Mask-based Prompting and Question-and-Answer (QnA) Prompting. These strategies were applied to the large language models Codestral and Qwen2.5-Coder, using the TypeEvalPy micro-benchmark. The evaluation measured exact match performance across different type inference scenarios, and the results provide insight into the effectiveness of each prompting method. The results of this evaluation are presented in Table 5.1.

Table 5.1: Exact match comparison of LLMs for type inference on the TypeEvalPy micro-benchmark.

Model	**FRT**	**FPT**	**LVT**	**Total**
Total instance	**239**	**88**	**533**	**860**
Mask-based Prompt				
Codestral	83.2%	75.0%	59.6%	67.8%
Qwen2.5	82.8%	76.1%	49.5%	61.5%
Question-and-Answer Prompt				
Codestral	**87.9%**	**92.0%**	**88.6%**	**88.7%**
Qwen2.5	79.9%	85.2%	85.0%	83.6%

5.2.1 Performance of Mask-Based Prompting

The Masked-based Prompting strategy resulted in moderate exact match performance across all type inference categories. Using this approach, Codestral achieved an overall exact match of 67.8%, while Qwen2.5-Coder scored 61.5%. The highest performance within this strategy was observed in Function Return Type (FRT) inference, where Codestral reached 83.2% and Qwen2.5-Coder reached 82.8%. However, performance declined in Function Parameter Type (FPT) inference, with Codestral at 75.0% and Qwen2.5 at 76.1%.

The most substantial limitation of this strategy was in Local Variable Type (LVT) inference, where Codestral and Qwen2.5-Coder achieved only 59.6% and 49.5% exact match, respectively. This suggests that Masked-based Prompting may provide insufficient contextual information, particularly for local variables, where type inference relies more on surrounding code rather than explicit type hints. The relatively lower performance in LVT inference indicates that this strategy does not effectively help the models in determining variable types within function bodies.

5.2.2 Performance of Question-and-Answer (QnA) Prompting

The Question-and-Answer Prompting strategy led to a significant increase in exact matches across all type inference categories. With this approach, Codestral's overall exact match increased from 67.8% to 88.7% (a gain of 20.9 percentage points), while Qwen2.5-Coder improved from 61.5% to 83.6% (a gain of 22.1 percentage points).

For Function Return Type (FRT) inference, Codestral improved from 83.2% to 87.9%, while Qwen2.5-Coder saw a slight decrease from 82.8% to 79.9%. Although Qwen2.5-Coder exhibited a small decline in this category, its improvements in the other type inference tasks suggest that the QnA approach still provides an overall advantage.

For Function Parameter Type (FPT) inference, Codestral's exact match increased from 75.0% to 92.0% (a gain of 17.0 percentage points), while Qwen2.5-Coder improved from 76.1% to 85.2% (a gain of 9.1 percentage points). This suggests that explicitly asking for function parameter types provides clearer guidance, likely because function signatures contain more explicit type information that can be leveraged by a well-structured prompt.

The largest improvement was observed in Local Variable Type (LVT) inference, where Codestral's exact match increased from 59.6% to 88.6% (a gain of 29.0 percentage points), and Qwen2.5-Coder improved from 49.5% to 85.0% (a gain of 35.5 percentage points). These re-

sults suggest that structured questions provide critical context for type inference within function bodies, which is especially beneficial for predicting local variable types.

5.2.3 Comparison of Strategies and Models

The results demonstrate that the QnA Prompting strategy consistently outperforms the Masked-based Prompting strategy across all type inference categories. The structured question format led to a 17–35 percentage point increase in exact match performance, with the largest improvement in Local Variable Type (LVT) inference. This suggests that explicitly structuring type inference queries as questions provides additional clarity, leading to better predictions.

When comparing models, Codestral consistently outperformed Qwen2.5-Coder, achieving higher exact match scores in both prompting strategies. The performance gap between the two models widened with QnA Prompting, where Codestral maintained a lead of approximately 5 percentage points overall. The greater improvements observed in Codestral's Function Parameter Type (FPT) and Local Variable Type (LVT) inference suggest that it is better optimized for structured prompt-based type inference.

Answer to RQ1: The findings indicate that the QnA Prompting strategy is the more effective approach for type inference in Python on micro-benchmark. Given its superior performance, the QnA Prompting strategy will be used for further analysis to evaluate how Large Language Models perform for type inference on a real-world dataset.

5.3 RQ2 - LLMs type inference on real-world dataset

This section presents the results of evaluating Codestral and Qwen2.5-Coder on the ManyTypes4Py dataset, focusing on their exact match performance in type inference. The evaluation considers two key dataset variations: 1. With `Any`: where `Any` types are included in the ground truth. 2. Without `Any`: where `Any` types are excluded to assess the model's precision on concrete types.

In addition to these two variations, we also report the exact match performance specifically on built-in types (e.g., int, str, list), providing further insight into the models' ability to correctly infer commonly used types.

The following subsections analyze the models' performance across these conditions, followed by a direct comparison between Codestral and Qwen2.5-Coder.

5.3.1 Performance with `Any` in Dataset

The evaluation of Codestral and Qwen2.5-Coder on the ManyTypes4Py dataset with `Any` included reveals high accuracy in function return type (FRT) and function parameter type (FPT) inference, but notable challenges in local variable type (LVT) prediction. Codestral achieves exact match rates of 94.2% in FRT and 93.2% in FPT, while Qwen2.5-Coder closely follows with 93.4% in FRT and 90.8% in FPT.

However, both models struggle with LVT inference, where Codestral attains an exact match rate of 48.2% and Qwen2.5-Coder reaches 49.1%. This suggests that the presence of `Any` increases ambiguity in local variable type prediction, making it more challenging for the models to generalize.

Table 5.2: Exact match comparison of LLMs for type inference on the ManyTypes4Py dataset.

Model	FRT(%)	FPT(%)	LVT(%)	Total(%)
With `Any`				
Total Instances	11422	14577	67576	93575
Codestral	94.2%	93.2%	48.2%	60.8%
Qwen2.5-Coder	93.4%	90.8%	49.1%	61.0%
Without `Any`				
Total Instances	11224	14082	38457	63763
Codestral	94.3%	93.1%	81.7%	86.4%
Qwen2.5-Coder	93.5%	90.7%	78.7%	84.0%
Bult-in Types				
Total Instances	8205	7918	24140	40263
Codestral	**96.7%**	**95.1%**	**93.4%**	**94.4%**
Qwen2.5-Coder	96.3%	93.1%	91.7%	92.9%

Overall, the total exact match performance is 60.8% for Codestral and 61.0% for Qwen2.5-Coder, indicating nearly identical results when `Any` is present in the dataset.

5.3.2 Performance without `Any` in Dataset

Removing `Any` from the dataset significantly improves exact match performance, particularly in LVT inference. Codestral's exact match rate for LVT increases from 48.2% to 81.7%, while Qwen2.5-Coder improves from 49.1% to 78.7%. This highlights that the removal of `Any` allows both models to infer types with greater confidence.

For function return type (FRT) and function parameter type (FPT), Codestral maintains strong performance, achieving 94.3% in FRT and 93.1% in FPT. Qwen2.5-Coder follows closely with 93.5% in FRT and 90.7% in FPT.

The total exact match rate increases from 60.8% (with `Any`) to 86.4% (without `Any`) for Codestral, and from 61.0% to 84.0% for Qwen2.5-Coder. This improvement underscores the impact of `Any` on type inference accuracy, with both models benefiting from its removal.

5.3.3 Performance on Built-in Types

When evaluating built-in types, both models achieve high exact match rates across all categories. Codestral attains 96.7% in FRT, 95.1% in FPT, and 93.4% in LVT. Qwen2.5-Coder follows with 96.3% in FRT, 93.1% in FPT, and 91.7% in LVT.

The overall exact match rate for built-in types is 94.4% for Codestral and 92.9% for Qwen2.5-Coder, indicating that both models effectively infer fundamental Python types. Codestral maintains a slight advantage across all three categories, suggesting a marginally better generalization ability for built-in types.

5.3.4 Comparison Between Codestral and Qwen2.5-Coder

Both models demonstrate strong performance in function return type (FRT) and function parameter type (FPT) inference, consistently exceeding 90% across all dataset conditions. Codestral outperforms Qwen2.5-Coder by 1-2 percentage points in these categories, suggesting a slight but consistent advantage.

The most pronounced difference emerges in local variable type (LVT) inference, where Codestral outperforms Qwen2.5-Coder by 3 percentage points when `Any` is removed (81.7% vs. 78.7%). However, both models struggle with LVT prediction when `Any` is included, reinforcing the inherent difficulty of inferring local variable types in ambiguous contexts.

In terms of overall exact match rates, Codestral surpasses Qwen2.5-Coder by 2.4 percentage points when `Any` is removed (86.4% vs. 84.0%), while their performance remains nearly identical when `Any` is included (60.8% vs. 61.0%). This suggests that while both models significantly benefit from the removal of `Any`, Codestral maintains a measurable lead over Qwen2.5-Coder.

For built-in types, both models achieve exceptionally high performance, exceeding 90% in all categories. However, Codestral retains a slight edge, particularly in function parameter type (FPT) and local variable type (LVT) inference, solidifying its advantage in structured type inference tasks.

Answer to RQ2: The findings indicate that both baseline models struggle with type inference when `Any` is included in the dataset, achieving total exact match rates of 60.8% for Codestral and 61.0% for Qwen2.5-Coder. However, their performance significantly improves when `Any` is removed, with Codestral reaching 81.7% and Qwen2.5-Coder at 78.7%. For built-in types, both models demonstrate strong performance, with Codestral achieving 94.4% and Qwen2.5-Coder reaching 92.9%, indicating that they generalize well for fundamental Python types.

5.4 RQ3 - Finetuned LLMs type inference on real-world dataset

Building on the previous analysis, this section presents the exact match performance of the finetuned Codestral and Qwen2.5 models on type inference using the ManyTypes4Py dataset. As before, the evaluation is conducted across the three key type inference categories described in Section 5.1. The models are tested under two main dataset conditions: 1. With `Any`: where `Any` types are included in the ground truth. 2. Without `Any`: where `Any` types are excluded to focus on specific type predictions.

In addition to these dataset variations, we also include a focused analysis of the models' performance on built-in types (e.g., int, str, list), providing insight into their ability to infer common Python types.

The following subsections present the exact match rates of finetuned models across these conditions, including a comparison with baseline models in the Comparison of Baseline vs. Finetuned Models section.

5.4.1 Performance with `Any` in Dataset

Table 5.3 presents the exact match rates for finetuned Codestral and Qwen2.5-Coder when `Any` is included in the dataset. Finetuned Codestral achieves 98.3% exact match in FRT and 93.0% in FPT, while Finetuned Qwen2.5-Coder attains 96.8% in FRT and 95.7% in FPT. These

Table 5.3: Exact match comparison of Finetuned LLMs for type inference on the ManyTypes4Py dataset.

Model	FRT(%)	FPT(%)	LVT(%)	Total(%)
With `Any`				
Total Instances	11422	14577	67576	93575
Codestral	98.3%	93.0%	92.9%	94.2%
Qwen2.5-Coder	96.8%	95.7%	85.9%	88.8%
Without `Any`				
Total Instances	11224	14082	38457	63763
Codestral	98.1%	97.9%	96.2%	96.9%
Qwen2.5-Coder	96.7%	95.8%	92.2%	93.8%
Bult-in Types				
Total Instances	8205	7918	24140	40263
Codestral	**99.2%**	**99.0%**	**98.1%**	**98.5%**
Qwen2.5-Coder	98.5%	98.0%	94.6%	96.1%

results confirm that both models now exhibit strong performance in function-level type inference, showing the effectiveness of finetuning.

Local variable type (LVT) inference remains more challenging but demonstrates substantial improvement. Finetuned Codestral reaches 92.9% exact match accuracy, while Finetuned Qwen2.5-Coder achieves 85.9%. The total exact match rate across all instances is 94.2% for Finetuned Codestral and 88.8% for Finetuned Qwen2.5-Coder, indicating that Codestral consistently outperforms Qwen2.5-Coder across all categories.

5.4.2 Performance without `Any` in Dataset

When `Any` is removed from the dataset, exact match performance further improves for both models. As shown in Table 5.3, Finetuned Codestral attains 98.1% in FRT and 97.9% in FPT, while Finetuned Qwen2.5-Coder achieves 96.7% in FRT and 95.8% in FPT. The removal of `Any` allows the models to work with clearer type annotations, leading to improved inference accuracy.

Local variable type (LVT) inference sees the most substantial gains, where Finetuned Codestral achieves 96.2% exact match accuracy, compared to 92.2% for Finetuned Qwen2.5-Coder. The total exact match rate increases to 96.9% for Finetuned Codestral and 93.8% for Finetuned Qwen2.5-Coder, demonstrating that removing `Any` results in improved predictability of type inference.

5.4.3 Performance on Built-in Types

Table 5.3 presents the exact match rates for built-in types, which evaluate how well the models infer fundamental Python types. Both finetuned models exhibit exceptionally strong performance in this category. Finetuned Codestral achieves 99.2% in FRT, 99.0% in FPT, and 98.1% in LVT, while Finetuned Qwen2.5-Coder reaches 98.5% in FRT, 98.0% in FPT, and 94.6% in

LVT.

The total exact match rate for built-in types is 98.5% for Finetuned Codestral and 96.1% for Finetuned Qwen2.5-Coder, indicating that both models have developed a strong understanding of fundamental Python types. However, Codestral maintains a 2.4 percentage point lead over Qwen2.5-Coder, suggesting that it generalizes slightly better in built-in type prediction.

5.4.4 Comparison of Baseline vs. Finetuned Models

The improvements observed in the finetuned models are best understood when compared to their baseline versions. The most substantial improvement is observed in local variable type (LVT) inference, where Finetuned Codestral improves by 44.7 percentage points (from 48.3% to 92.9%) and Finetuned Qwen2.5-Coder improves by 37 percentage points (from 49.0% to 85.9%) when `Any` is included. Even when `Any` is removed, both models improve significantly, confirming that finetuning is particularly effective for enhancing local variable type inference.

For function return type (FRT) and function parameter type (FPT) inference, the improvements are moderate but consistent. Codestral improves by approximately 5 percentage points, while Qwen2.5-Coder gains 4-6 percentage points. Since both models already performed well in FRT and FPT inference in their baseline versions, finetuning serves to refine their accuracy rather than introducing drastic improvements.

Figure 5.1 visually illustrates these trends, highlighting how overall performance improves by 33.7 percentage points for Codestral and 28.3 percentage points for Qwen2.5-Coder when `Any` is included. Without `Any`, the improvements are 10.9 and 10.7 percentage points, respectively.

Figures 5.2, 5.3, and 5.4 highlight these improvements, showing how both models now achieve near-perfect results in function-level type inference. Figure 5.4 reinforces these findings by showcasing the notable improvements in LVT inference, which was previously the weakest area for both models. The gap between baseline and finetuned models is the widest in LVT inference, confirming that finetuning is particularly beneficial for local variable type prediction.

Answer to RQ3: The findings indicate that finetuning significantly improves type inference performance across all dataset conditions. With `Any` included, Codestral achieves 94.2% and Qwen2.5-Coder reaches 88.8%, demonstrating substantial gains over baseline models. When `Any` is removed, performance further improves, with Codestral attaining 96.9% and Qwen2.5-Coder reaching 93.8%. For built-in types, both models exhibit near-perfect accuracy, with Codestral at 98.5% and Qwen2.5-Coder at 96.1%. These results confirm that finetuning enhances LLMs' ability to infer Python types, particularly for local variable type inference.

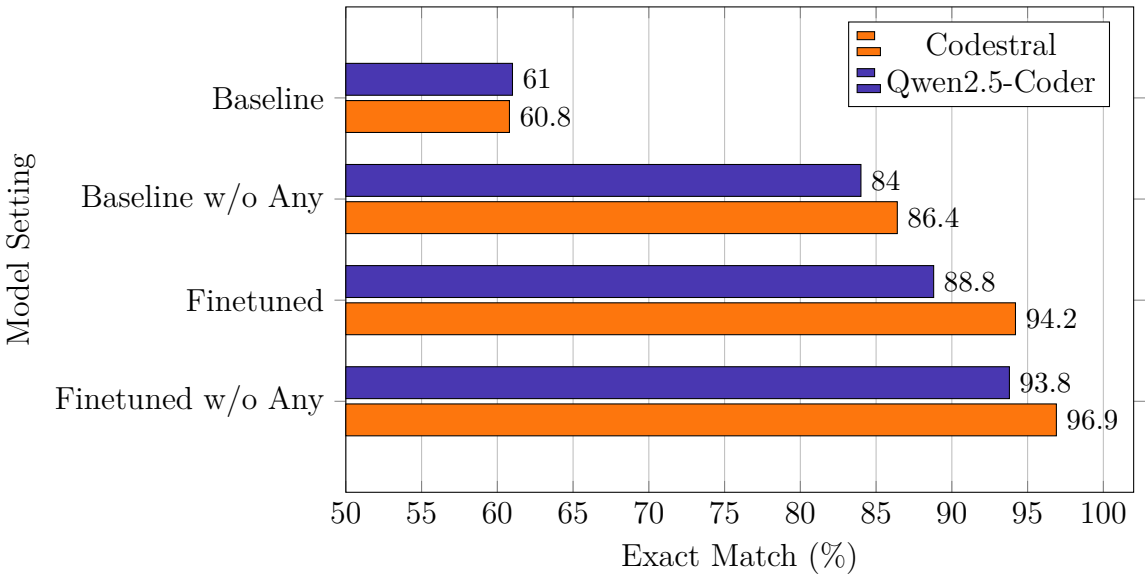

Figure 5.1: Overall Performance of Baseline and Finetuned Models

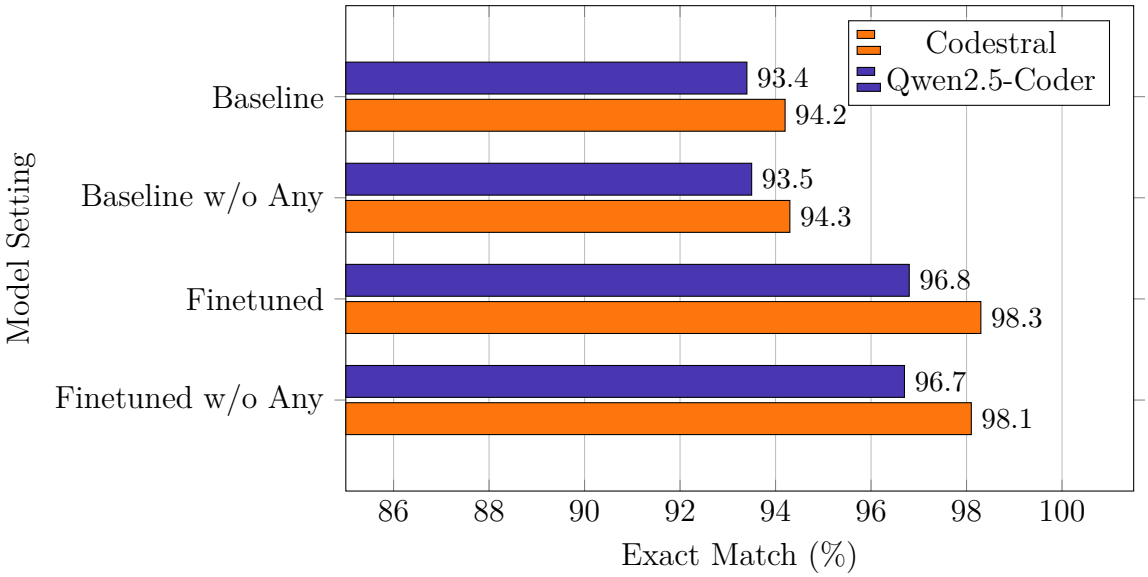

Figure 5.2: Comparison of Function Return Type (FRT) Performance

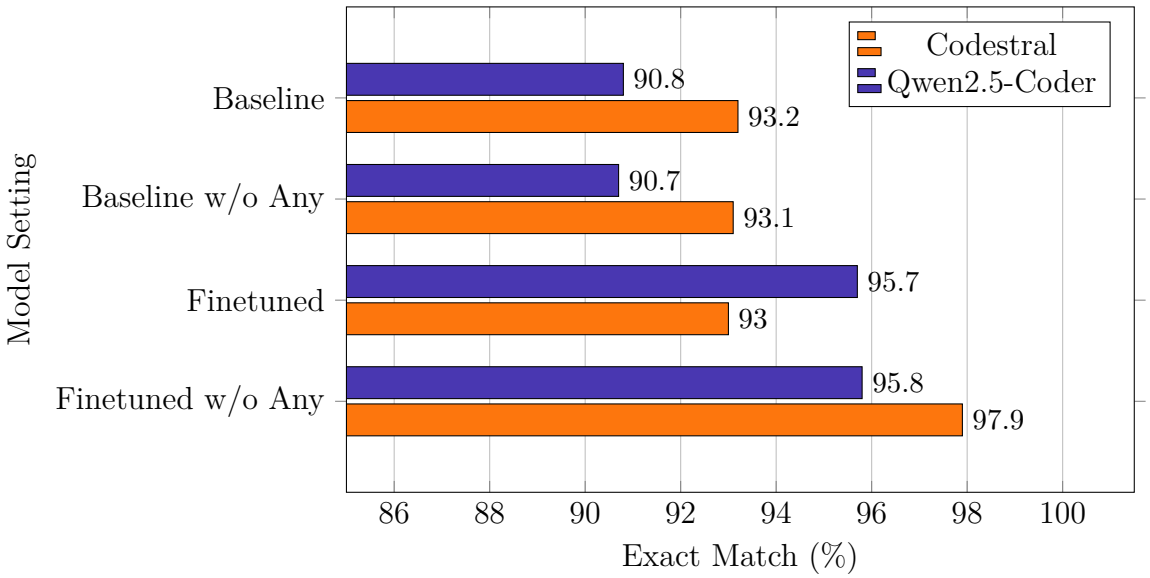

Figure 5.3: Comparison of Function Parameter Type (FPT) Performance

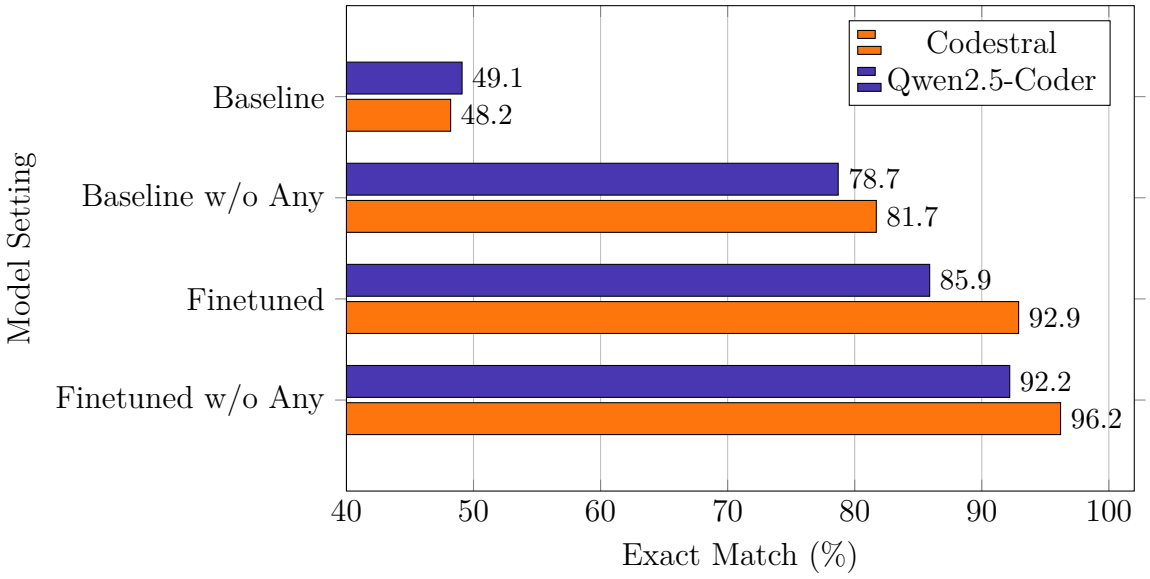

Figure 5.4: Comparison of Local Variable Type (LVT) Performance

6 Discussion

We analyze the performance of LLM-based type inference to understand key insights. We have considered the dataset without `Any` for the analysis. First, we examine files with the highest number of *questions* to assess how LLMs handle large-scale inference tasks. Aligning with the literature [MLPG22], we further analyze model performance across three type distributions to evaluate their effectiveness:

1. Frequent types, which dominate the training dataset, provide insight into how well LLMs generalize to common annotations and
2. Rare types, appearing in fewer than 0.1% of instances, test LLMs' ability to infer under-represented annotations.
3. Non-ubiquitous types, i.e excluding the built-ins and typings.

6.1 Most Questions Asked

The results in Table 6.1 highlight a substantial difference in exact matches between the baseline and fine-tuned models. The baseline models struggle to align with expected type annotations, achieving only 57% (Codestral) and 56% (Qwen2.5-Coder) exact matches, while the fine-tuned models reach 100% accuracy across all cases. This improvement is particularly evident in structured codebases where correctly distinguishing between general class types and specific field or function types is crucial.

One of the primary reasons for the baseline's lower performance is its tendency to infer overly specific types instead of recognizing broader class types. This issue is most apparent in form-related files, such as *andrew-it/forms.py*, where the expected types in the ground truth classify form fields using high-level form class types (e.g., *Type[RegisterForm]*, *Type[LoginForm]*). However, the baseline models frequently predict field-level types such as *StringField, PasswordField,* and *BooleanField*. For example, instead of classifying RegisterForm.email as *Type[RegisterForm]*, the baseline models assign it *StringField*. While this prediction is technically correct at the field level, it does not align with the expected broader classification of the form, leading to a high number of mismatches in the exact match evaluation.

In *andrew-it/forms.py*, the baseline model produces only 1 correct match out of 78 queries, whereas the fine-tuned model correctly generalizes and achieves 78/78 exact matches. A similar issue arises in *tjcsl/orm.py*, where the baseline model fails entirely, scoring 0/64 exact matches,

Table 6.1: Exact matches on top-5 prompts with most questions

● 80-100%, ● 60-80%, ● 40-60%, ● 20-40%, ● 0-20%

Project (#Q.)	Codestral		Qwen2.5-Coder	
	B.	Ft	B.	Ft
andrew-it/forms.py (78)	**1** ●	**78** ●	**1** ●	**78** ●
Galbar/base.py (65)	65 ●	65 ●	65 ●	65 ●
tjcsl/orm.py (64)	**0** ●	**64** ●	**0** ●	**64** ●
mwhittaker/bexpr.py (63)	61 ●	63 ●	61 ●	63 ●
ShadowTemplate/cdm.py (61)	60 ●	61 ●	58 ●	61 ●
Total (331)	187 ●	331 ●	185 ●	331 ●

while the fine-tuned model achieves 100% accuracy. The fine-tuned model significantly improves performance by learning to generalize individual field types into broader form classifications, thereby aligning more closely with expected annotations.

A related issue is observed in function input/output type inference, particularly in *mwhittaker/bexpr.py*. In this case, the expected types are general classifications such as *Input* and *Output*, but the baseline model instead infers overly detailed union types. For example, rather than recognizing *Input* as a broad category, the baseline model outputs *Union[BexprEvalRequest, BexprSetRequest, BexprUnsetRequest]*, explicitly listing possible input types. Similarly, instead of classifying *Output* as a general return type, the baseline model predicts *Union[BexprEvalReply, BexprSetReply, BexprUnsetReply]*. While these predictions capture the specific components of input and output operations, they fail to match the expected abstraction, leading to lower exact match accuracy. The fine-tuned model, in contrast, correctly aligns with *Input* and *Output*, demonstrating a better understanding of when to generalize.

These findings emphasize the importance of fine-tuning in achieving the right balance between specificity and abstraction in type inference tasks. The baseline models tend to overfit to specific details, leading to unnecessary mismatches when a higher-level classification is expected. The fine-tuned models, however, learn to abstract appropriately, ensuring their predictions align more closely with ground truth expectations. This ability to generalize effectively explains the 100% exact match performance of the fine-tuned models, particularly in form-based and function-based type inference tasks.

6.2 Most Frequent Types

Table 6.2 presents the exact match accuracy for the ten most frequent types: *str, int, None, bool, List[Any], float, List[str], Dict[str, Any], Path,* and *Optional[str]*. These types collectively account for 34,652 occurrences in the dataset, making them essential benchmarks for evaluating the models' ability to generalize across standard type annotations.

The results indicate that baseline LLMs already perform well on frequent types, with Codestral achieving 94.0% accuracy and Qwen2.5-Coder reaching 93.0% before fine-tuning. Fine-tuning further enhances accuracy, particularly for *None* and *List[Any],* where both models achieve nearly perfect exact match rates post-training. For example, Codestral improves from 91% to 99% on None, while Qwen2.5-Coder sees a similar jump from 90% to 98%.

Primitive types such as *str, int,* and *bool* see modest but consistent improvements, with fine-tuning pushing their accuracy above 97% across both models. The largest relative gains

Table 6.2: Exact matches on top-10 most frequent types

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Type (Total)	Codestral		Qwen2.5-Coder	
	B.	F.	B.	F.
str (16,006)	15,218 ●	15,832 ●	15,025 ●	15,263 ●
int (6,023)	5,739 ●	5,974 ●	5,632 ●	5,856 ●
None (4,342)	3,969 ●	4,282 ●	3,922 ●	4,250 ●
bool (1,973)	1,902 ●	1,953 ●	1,923 ●	1,933 ●
List[Any] (1,561)	1,472 ●	1,539 ●	1,470 ●	1,504 ●
float (1,381)	1,311 ●	1,337 ●	1,308 ●	1,293 ●
List[str] (1,191)	1,100 ●	1,171 ●	1,050 ●	1,110 ●
Dict[str, Any] (812)	747 ●	790 ●	733 ●	777 ●
Path (746)	705 ●	722 ●	691 ●	719 ●
Optional[str] (617)	376 ◕	555 ●	381 ◕	555 ●
Total (34,652)	32,539 ●	34,155 ●	32,135 ●	33260 ●

are observed for *Optional[str],* where the baseline models struggle significantly. Codestral and Qwen2.5-Coder achieve only 61-62% accuracy before fine-tuning, indicating frequent misclassifications—likely defaulting to str instead of recognizing optional values. After fine-tuning, however, accuracy increases dramatically to 90%, suggesting that the models develop a stronger understanding of nullable types and their proper representation.

Fine-tuning also leads to notable improvements in structured container types like *List[str]* and *Dict[str, Any],* where exact match rates increase by 4-6 percentage points, indicating a stronger ability to correctly infer complex type structures. Overall, the fine-tuned models achieve near-perfect accuracy on the most common types, reinforcing their ability to generalize effectively across frequent type annotations while addressing key weaknesses in optional and container types.

6.3 Rare Types

Rare types, which appear in less than 0.1% of the dataset, pose a challenge for LLM-based type inference due to their infrequent occurrence and domain-specific nature. Table 6.3 presents the exact match accuracy for the top-10 rarest types, including specialized types such as *HttpRequest, Tuple[str, str], Config, AbstractEventLoop, subprocess.Popen[Any],* and *TextIOWrapper.* These types are commonly associated with networking, configuration management, subprocess handling, and asynchronous programming, requiring models to generalize effectively despite limited training examples.

The results indicate that fine-tuning significantly enhances model performance on rare types, addressing one of the key weaknesses of baseline LLMs. Before fine-tuning, Codestral achieves 72% accuracy and Qwen2.5-Coder scores 58%, highlighting the difficulty of predicting uncommon types. After fine-tuning, both models improve substantially, with Codestral reaching 93% accuracy and Qwen2.5-Coder improving to 89%, demonstrating that exposure to additional domain-specific data helps models better infer less frequent type annotations.

One of the most notable improvements occurs in structured and complex types. The most dramatic gains are seen in *subprocess.Popen[Any]*, where the baseline models perform extremely

Table 6.3: Exact matches on top-10 rare types

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Type (Total)	Codestral		Qwen2.5-Coder	
	B.	F.	B.	F.
HttpRequest (67)	65	65	65	65
object (102)	**87**	**81**	**76**	**63**
Tuple[str, str] (79)	78	77	37	74
Config (62)	56	59	53	57
AbstractEventLoop (91)	61	91	58	89
Session (79)	74	79	69	79
subprocess.Popen[Any] (104)	1	102	0	99
TextIOWrapper (99)	84	98	43	92
Optional[Dict[Any]] (101)	34	75	29	71
Result (80)	78	80	71	80
Total (864)	618	807	501	769

poorly. Codestral achieves only 1% accuracy, while Qwen2.5-Coder fails entirely with 0% accuracy. After fine-tuning, both models dramatically improve, reaching 98% and 95% accuracy, respectively, indicating a significant enhancement in handling subprocess-related types. Similarly, file-handling types such as *TextIOWrapper* see a major boost, with Codestral improving from 85% to 99% and Qwen2.5 from 43% to 93%, reinforcing that fine-tuning refines the models' ability to infer rare but critical types.

Another area of improvement is in asynchronous and session-related types, such as *AbstractEventLoop* and *Session*. Both models achieve 100% accuracy post-finetuning, compared to significantly lower baseline performance. Codestral's accuracy on *AbstractEventLoop*, for example, increases from 67% to 100%, showing that fine-tuning strengthens its understanding of event loop structures.

Exception

A notable exception is the *object* type, where fine-tuning leads to a decline in accuracy. Codestral drops from 85% to 79%, while Qwen2.5-Coder sees an even sharper decrease from 75% to 62%. This suggests that fine-tuning causes the models to shift away from using object as a generic fallback and instead favors more specific class-based predictions.

For example, in *Levitanus/pyksp/pyksp/compiler/dev_tools.py*, the variable instead is explicitly assigned *object()* within the class *WrapProp*. The baseline models may have defaulted to predicting *object*, but after fine-tuning, they likely inferred a more concrete type based on contextual information.

```
class WrapProp:
    instead = object()
```

This behavior suggests that fine-tuning improves confidence in assigning concrete types, reducing reliance on generic placeholders. While this may lower exact match accuracy for object, it reflects an overall shift towards more precise and meaningful type predictions.

6.4 Non-ubiquitous Types

Non-ubiquitous types refer to types that are neither built-in Python types nor part of the standard typing module. These types typically originate from external libraries or frameworks and are essential in domains such as file handling, web development, machine learning, and scientific computing. Table 6.4 presents the exact match accuracy for the top-10 non-ubiquitous types, including Path, Response, Logger, ArgumentParser, Tensor, datetime, Namespace, Mock, ndarray, and DataFrame.

Table 6.4: Exact matches on top-10 non-ubiquitous types

● 80-100%, ● 60-80%, ● 40-60%, ● 20-40%, ● 0-20%

Type (Total)	**Codestral**		**Qwen2.5-Coder**	
	B.	**F.**	**B.**	**F.**
Path (746)	705 ●	722 ●	691 ●	719 ●
Response (482)	461 ●	477 ●	438 ●	476 ●
Logger (464)	446 ●	463 ●	448 ●	460 ●
ArgumentParser (331)	302 ●	310 ●	273 ●	309 ●
Tensor (260)	256 ●	260 ●	257 ●	260 ●
datetime (255)	246 ●	249 ●	241 ●	244 ●
Namespace (240)	233 ●	240 ●	225 ●	237 ●
Mock (210)	202 ●	209 ●	186 ●	206 ●
ndarray (155)	146 ●	153 ●	150 ●	153 ●
DataFrame (155)	148 ●	154 ●	145 ●	154 ●
Total (3,278)	3,145 ●	3,237 ●	3,054 ●	3,218 ●

The results show that both baseline and fine-tuned LLMs perform well on non-ubiquitous types, with fine-tuning further improving exact match accuracy. Before fine-tuning, Codestral achieves 96% accuracy and Qwen2.5-Coder reaches 93%. After fine-tuning, these scores improve to 99% and 98%, respectively, across all types. These improvements indicate that fine-tuning enhances the models' ability to recognize library-specific types and framework-dependent annotations.

Fine-tuning provides notable gains for types commonly used in machine learning and data science, such as Tensor, ndarray, and DataFrame. These types, frequently encountered in PyTorch, NumPy, and Pandas, already have high baseline accuracy, with Tensor at 98-99% before fine-tuning and reaching 100% exact match after fine-tuning for both models. Similarly, ndarray and DataFrame improve by 4-5 percentage points, demonstrating that fine-tuning helps reinforce correct type inference for structured data representations.

Another highly improved category is types related to system utilities and command-line interfaces, such as ArgumentParser and Namespace. The fine-tuned models achieve 100% accuracy on both types for Codestral and 99% for Qwen2.5-Coder, indicating an improved understanding of types commonly used in command-line parsing and script automation.

In logging and debugging contexts, Logger and Mock see substantial improvements. Codestral reaches 100% accuracy after fine-tuning for both types, while Qwen2.5-Coder improves from 89% to 98% for Mock and from 97% to 99% for Logger. This suggests that fine-tuning helps LLMs better recognize standard debugging and logging structures, leading to fewer misclassifications.

Types related to web development and API interactions, such as Response, also benefit from fine-tuning. Accuracy increases from 96% to 99% for Codestral and from 91% to 99% for Qwen2.5-Coder, showing that fine-tuning strengthens the models' recognition of response objects, likely improving type inference in web-based projects.

Overall, the results highlight that fine-tuning is particularly beneficial for non-ubiquitous types, especially those tied to specific libraries and frameworks. While baseline models already perform well on common external types, fine-tuning reinforces library-specific type mappings, reduces ambiguity, and improves exact match accuracy across various domains, including machine learning, data science, system utilities, and web development.

7 Limitations

While this thesis presents a comprehensive evaluation of large language models (LLMs) for Python type inference, several limitations must be acknowledged that may impact reproducibility, accuracy, and scalability of the findings.

Dataset Availability

The ManyTypes4Py dataset was built from open-source GitHub repositories collected in 2020. Since then, many of these projects have been deleted, archived, or made private. As a result, exact file references in the dataset may no longer be accessible. This poses a reproducibility challenge for future researchers aiming to replicate or extend this study using the same source material.

Incomplete Ground Truth

The dataset relies on developer-provided type annotations, which are often incomplete or selective. Not all variables, return types, or attributes are explicitly annotated, and some annotations may be imprecise or incorrect. As such, the ground truth is not exhaustive, and model performance may vary if evaluated on a more comprehensive or fully type-annotated dataset. However, evaluations using the TypeEvalPy framework—which includes auto-generated and exhaustive annotations—suggest that LLMs maintain strong performance even under more rigorous conditions, reinforcing confidence in their generalizability.

Token Limitations

Due to hardware and memory constraints, prompt inputs were limited to a maximum of 4,000 tokens. This truncation restricts the amount of code context the model can consider during inference, potentially omitting relevant type information defined earlier in the file or across different functions. While LLMs can still infer types effectively in local contexts, full-program understanding is limited under these constraints.

Computational Requirements

Training and evaluating large language models requires significant compute resources, even when using 4-bit quantization and parameter-efficient fine-tuning methods like LoRA. Running fine-tuning experiments and performing inference over tens of thousands of examples necessitates access to high-memory GPUs and prolonged training time, making the approach less accessible to researchers with limited infrastructure.

Potential Data Leakage

LLMs are trained on large-scale public code repositories, which may include some of the same projects used in the ManyTypes4Py benchmark. This introduces the possibility of data leakage, where models have prior exposure to test files, thereby inflating performance. While this is a valid concern, it is important to note that LLMs must still demonstrate contextual understanding and correct type reasoning beyond simple memorization. Furthermore, the type inference task itself requires structural and semantic understanding, suggesting that high accuracy is not solely attributable to overlap with training data.

8 Conclusion

This thesis explored the applicability of Large Language Models (LLMs) for type inference in Python, focusing on both micro-benchmark tasks and real-world dataset. While prior studies have shown promising results for LLMs on micro-benchmark, their scalability, generalization, and effectiveness in practical settings had remained largely unexplored.

To address this gap, we extended the TypeEvalPy framework to incorporate the ManyTypes4Py dataset, enabling a more realistic evaluation of type inference performance across a diverse range of Python projects. We selected two state-of-the-art LLMs—Codestral and Qwen2.5-Coder—and assessed their type prediction accuracy using two prompting strategies: mask-based and question-and-answer (QnA) prompting. Furthermore, we applied Parameter-Efficient Fine-Tuning (PEFT) using LoRA to adapt these models to the type inference task.

The results demonstrated that:

- **(RQ1)** QnA prompting consistently outperformed mask-based prompting across micro-benchmark scenarios, providing clearer context to the LLMs and improving accuracy.
- **(RQ2)** LLMs, particularly Codestral, showed strong generalization capabilities on real-world dataset.
- **(RQ3)** Fine-tuning using PEFT further improved prediction accuracy, particularly for rare and user-defined types, although some trade-offs were observed in handling generic types.

These findings highlight the potential of LLMs as powerful tools for scalable type inference in Python. By learning from surrounding context and usage patterns, LLMs can bridge the annotation gap in dynamic codebases, enabling improved static analysis, tooling, and code reliability without requiring exhaustive manual annotations.

Future Work

While the results are promising, several avenues remain open for further exploration. First, improving type inference for composite types (e.g., generics, nested types) remains a challenge and could benefit from hierarchical prompting or multi-stage inference pipelines. Second, expanding the benchmark to include multi-language datasets (e.g., TypeScript, Java) could test cross-lingual transfer capabilities of LLMs. Third, integrating uncertainty estimation and confidence scoring into LLM predictions could make the outputs more useful in production settings.

Lastly, optimizing fine-tuning strategies with larger code corpora or domain-specific subsets may further enhance model specialization.

Closing Remarks

In conclusion, this thesis demonstrates that LLMs, especially when fine-tuned and properly prompted, are effective for large-scale Python type inference. As codebases grow in size and complexity, the need for intelligent automation tools becomes more critical. The findings presented here provide a foundation for future advancements in LLM-driven developer assistance, static analysis, and automated code understanding.

Bibliography

[ABDG20] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

[AI23] Pythagora AI. Pythagora - autonomous backend development with llms. `https://pythagora.ai`, 2023. Accessed: 2025-03-18.

[Ali24] Alibaba Cloud Intelligence. Qwen2.5: A powerful and scalable code model series. `https://huggingface.co/Qwen`, 2024. Accessed: 2025-03-19.

[All19] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN international symposium on new ideas, new paradigms, and reflections on programming and software*, pages 143–153, 2019.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[CLJ20] Tianyin Chen, Yizhou Liu, and Guoqing Jin. An empirical study of python bugs: Types, root causes, and fix patterns. *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1398–1409, 2020.

[CND+23] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[Cod23] CodiumAI. Codiumai – meaningful tests for busy developers. `https://www.codium.ai`, 2023. Accessed: 2025-03-18.

[Cod24] CodePal. Codepal - your ai pair programmer. `https://codepal.ai`, 2024. Accessed: 2025-03-18.

[CTJ+21a] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. `https://arxiv.org/abs/2107.03374`, 2021. arXiv:2107.03374.

[CTJ+21b] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[HGJ+19] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR, 2019.

[HSW+22] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[HWLL24] Xia Han, Shizhu Wu, Hui Li, and Ming Li. Parameter-efficient fine-tuning of large language models: A survey. *arXiv preprint arXiv:2401.00011*, 2024.

[J+23] Albert Q Jiang et al. Mistral 7b. *Mistral.ai*, 2023. `https://mistral.ai/news/introducing-mistral-7b/`.

[LAZ+23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[LL21] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[LWC+24] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey, 2024.

[MAH+23] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: A tool for static code analysis with large language models. *arXiv preprint arXiv:2310.18532*, 2023.

[Mis24] Mistral AI. Codestral: A Code Language Model by Mistral. `https://mistral.ai/news/codestral/`, 2024. Accessed: 2025-03-19.

[MLG21] Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 585–589. IEEE, 2021.

[MLPG22] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.

[NPH+22] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[PEP25] PEP 484 – Type Hints | peps.python.org. https://peps.python.org/pep-0484/, February 2025.

[PGL+22] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2019–2030, 2022.

[PGLC20] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 209–220, 2020.

[RBV16] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *OOPSLA*, 2016.

[Sou23] Sourcegraph. Cody ai by sourcegraph. `https://sourcegraph.com/cody`, 2023. Accessed: 2025-03-18.

[SVSW+24] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. Typeevalpy: A micro-benchmarking framework for python type inference tools. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 49–53, 2024.

[SWX+24] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[SWZ+24] Ze Sheng, Fenghua Wu, Xiangwu Zuo, Chao Li, Yuxin Qiao, and Lei Hang. Lprotector: An llm-driven vulnerability detection system. *arXiv preprint arXiv:2411.06493*, 2024.

[Tab23] Tabnine. Tabnine - ai code completion assistant. `https://www.tabnine.com`, 2023. Accessed: 2025-03-18.

[TLI+23a] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[TLI+23b] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Myle Ott Goyal, Sergey Edunov, et al. Llama: Open and efficient foundation language models. `https://arxiv.org/abs/2302.13971`, 2023. arXiv:2302.13971.

[VMVW24] Sweta Venkatesh, Naman Mehta, Anand Vasudevan, and Hao Wang. The emergence of large language models for code analysis: Opportunities and challenges. *arXiv preprint arXiv:2402.12345*, 2024.

[VSM+24] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Amir M Mir, Sofia Reis, and Eric Bodden. The emergence of large language models in static analysis: A first look through micro-benchmarks. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 35–39, 2024.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[VWLB23] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. Enhancing comprehension and navigation in jupyter notebooks with static analysis. In *2023 IEEE international conference on software analysis, evolution and reengineering (SANER)*, pages 391–401. IEEE, 2023.

[WKM+23] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning Language Models with Self-Generated Instructions, May 2023.

[WLS+24] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents, 2024.

[WWS+22a] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[WWS+22b] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[YJW+24] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.