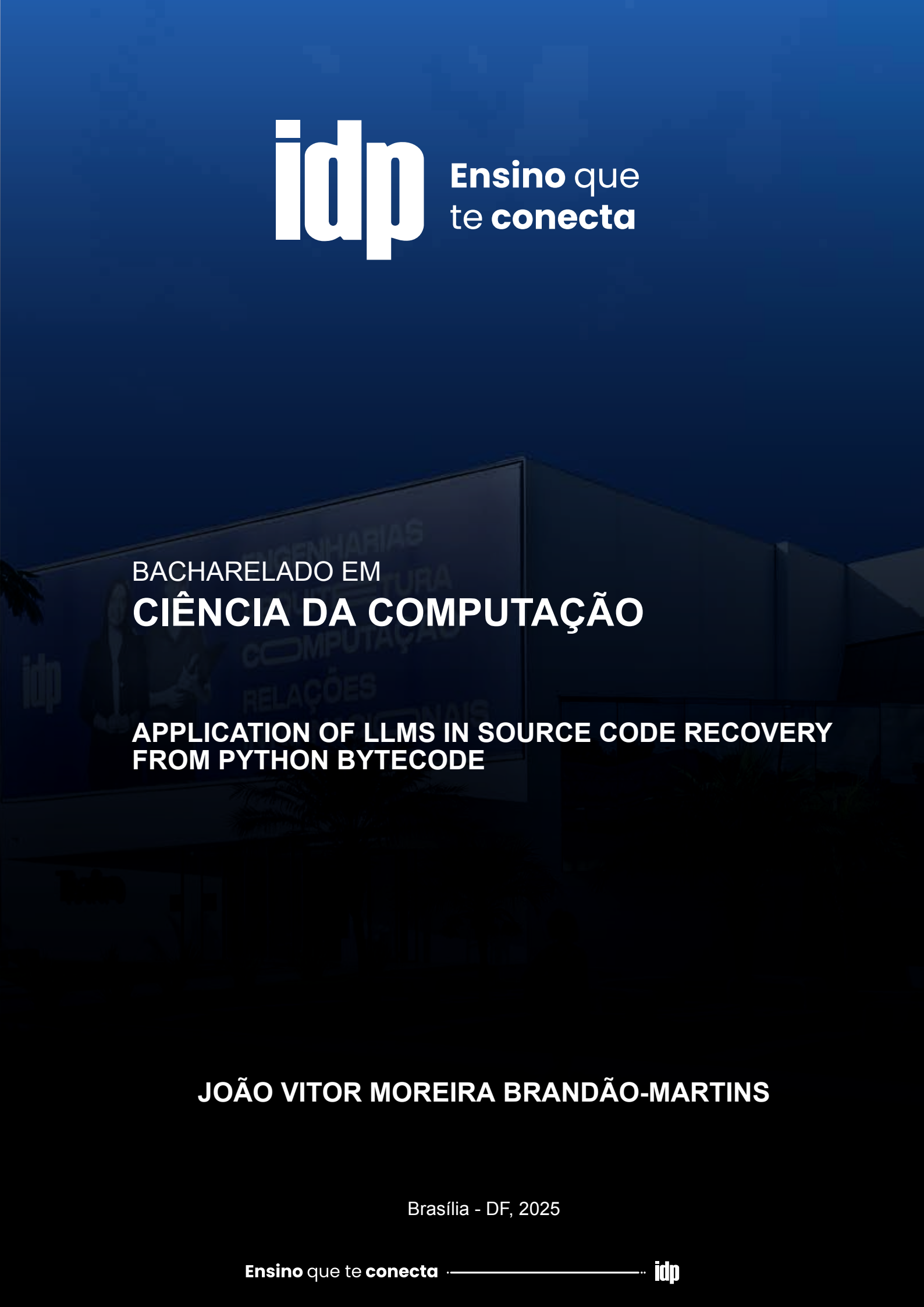


Ensino que te conecta

BACHARELADO EM

CIÊNCIA DA COMPUTAÇÃO

APPLICATION OF LLMS IN SOURCE CODE RECOVERY FROM PYTHON BYTECODE

JOÃO VITOR MOREIRA BRANDÃO-MARTINS

Brasília - DF, 2025

JOÃO VITOR MOREIRA BRANDÃO-MARTINS

APPLICATION OF LLMS IN SOURCE CODE RECOVERY FROM PYTHON BYTECODE

Undergraduate Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science at the Brazilian Institute of Education, Development and Research (IDP).

Advisor
Jeremias Moreira Gomes

Brasília - DF, 2025

Código de catalogação na publicação – CIP

B817a Brandão-Martins, João Vitor Moreira
Application of LLMS in source code recovery from Python bytecode / João Vitor Moreira Brandão-Martins. — Brasília: Instituto Brasileiro de Ensino, Desenvolvimento e Pesquisa, 2025.

75 f. : il.

Orientador: Prof. Ms. Jeremias Moreira Gomes

Monografia (Graduação em Ciência da Computação) — Instituto Brasileiro de Ensino, Desenvolvimento e Pesquisa – IDP, 2025.

1. Engenharia reversa de software. 2. Python - Linguagem de programação. 3. Geração de código - Ciência da Computação. I.Título

CDD 004

Elaborada por Natália Bianca Mascarenhas Puricelli – CRB 1/3439

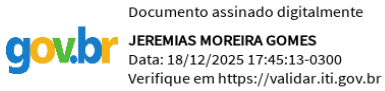
JOÃO VITOR MOREIRA BRANDÃO-MARTINS

APPLICATION OF LLMS IN SOURCE CODE RECOVERY FROM PYTHON BYTECODE

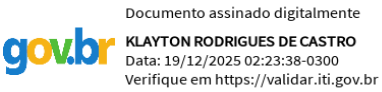
Undergraduate Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science at the Brazilian Institute of Education, Development and Research (IDP).

Approved on 08/05/2025

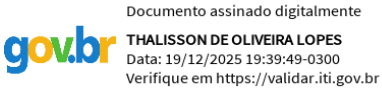
Examining Committee



Jeremias Moreira Gomes (Advisor) - IDP



Klayton Rodrigues de Castro (Internal Member) - IDP



Thálisson de Oliveira Lopes (Internal Member) - IDP

ABSTRACT

Source code recovery from intermediate representations, such as bytecode or binary code, plays a fundamental role in reverse engineering, especially in scenarios where the original source is unavailable. Although Python is typically referred to as an interpreted language, its execution involves compilation into bytecode, an intermediate form executed by the Python Virtual Machine (PVM). This process removes high-level information and introduces challenges for accurate decompilation. Traditional tools often produce code that is syntactically valid but semantically limited or difficult to interpret.

In recent years, Large Language Models (LLMs) based on transformer architectures have shown promising results in tasks involving source code understanding, generation, and even binary analysis. This study investigates the application of LLMs to the task of recovering Python source code from bytecode, an area still largely unexplored in the literature. Through a systematic review of related work, the research identifies a gap in the use of LLMs for Python bytecode decompilation.

This work proposes an approach centered on modern LLMs. The hypothesis is that such models can assist in both syntactic and semantic reconstruction of the original source code. The expected contributions include evaluating the success rate of this approach and offering new insights into the intersection between machine learning and reverse engineering.

Keywords: Decompilation, LLM, Python, Bytecode, Source Code Recovery.

CONTENTS

LIST OF FIGURES

LIST OF TABLES


1

1 INTRODUCTION

1.1 CONTEXT AND MOTIVATION

Decompilation, the process of recovering high-level source code from a low-level representation such as binary or bytecode, is one of the pillars of reverse engineering [1]. This process is indispensable for a multitude of software engineering tasks, including security analysis, malware detection, vulnerability discovery, and the maintenance of legacy systems where the original source code is unavailable [2, 3]. By translating machine-readable code into a human-readable format, decompilation empowers developers and security analysts to comprehend program behavior, identify potential threats, and facilitate software migration and interoperability [4].

While decompilation is a well-established field for compiled languages like C/C++, its application to dynamic languages such as Python presents unique challenges. Python's execution model relies on compiling source code into bytecode, a platform-independent intermediate representation that is then executed by the Python Virtual Machine (PVM). Although tools exist to decompile Python bytecode, they often face significant limitations. The compilation process inherently discards crucial semantic information, including original variable names, comments, and high-level syntactic structures [5, 6]. Consequently, the output of traditional decompilers is frequently syntactically valid but semantically obscure, producing code that is difficult to read, maintain, or verify for correctness. These outputs often contain what are termed "decompilation quirks", that is, differences from the original code that reduce readability and may even alter program behavior [5].

In recent years, the rapid advancement of Large Language Models (LLMs) has demonstrated remarkable capabilities in understanding, generating, and translating both natural and programming languages [7]. This progress has opened a new frontier for decompilation. By framing decompilation as a translation task, from the "language" of bytecode to the "language" of high-level source code, researchers have begun to explore the potential of neural networks and, more recently, LLMs to bridge this gap [4, 8]. Studies have shown that LLMs can significantly outperform traditional decompilers in generating more readable and executable code for languages like C [4] and WebAssembly [3], and can even be used to refine the imperfect output of existing

tools [5].

1.2 PROBLEM STATEMENT

The primary challenge in decompiling Python bytecode is the significant loss of semantic information during the initial compilation phase. Traditional decompilers can reconstruct control flow and basic operations, but they struggle to recover the abstractions that make source code comprehensible. Furthermore, they often lack long-term consistency, as changes to the bytecode in new Python releases can break existing decompilers, requiring significant effort to maintain compatibility [9]. The resulting code is often populated with generic variable names (e.g., `var_1`, `var_2`), convoluted control structures, and lacks the idiomatic expressions a human programmer would use. This "semantic gap" between the decompiled output and the original source code hinders effective program comprehension and subsequent analysis.

While recent research has demonstrated the potential of LLMs in binary code understanding [7, 10], the majority of these studies focus on low-level machine code (e.g., x86, ARM) rather than the bytecode of high-level dynamic languages. Furthermore, there is a need to systematically evaluate the performance of current state-of-the-art models. This work aims to address that gap by exploring how these advanced models can be leveraged to produce high-fidelity source code that is not only syntactically correct but also semantically close to the original.

1.3 OBJECTIVES

The primary objective of this research is to investigate and apply LLMs for recovering original Python 3.13 source code from bytecode, exploring the potential of these models as alternatives to conventional decompilers.

1.4 CONTRIBUTIONS

This work contributes to the field of software engineering and reverse engineering by:

- Proposing a novel, automated architecture for decompilation that integrates Large Language Models with the Model Context Protocol to create an iterative "generate-test-refine" loop.
- Establishing a scalable methodology for constructing a high-quality, ground-truth dataset of Python functions and their corresponding bytecode, enriched with AI-generated unit tests for rigorous functional validation.
- Providing an empirical evaluation of the capabilities of a state-of-the-art LLM in the specific domain of Python bytecode decompilation, addressing a gap in the current literature.

1.5 RESEARCH QUESTION AND HYPOTHESIS

1.5.1 Research Question

This undergraduate thesis seeks to answer the following central question:

> Can large language models (LLMs) effectively support the recovery of Python 3.13 source code from bytecode, generating syntactically valid and semantically coherent outputs that approximate the original source?

1.5.2 Hypothesis

This research is based on the hypothesis that LLMs can significantly contribute to the decompilation of Python bytecode by assisting in both syntactic reconstruction and semantic understanding of the original code. By exploring this hypothesis, we aim not only to evaluate the technical feasibility of the approach but also to offer a new perspective on the intersection between machine learning and software reverse engineering.

2

2 COMPILATION

This section presents the theoretical basis of the compilation process, focusing on the Python programming language.

2.1 DEFINITION

A compiler is a language processor which takes as input a code written in a programming language, called source code, and translates it into another language, usually a lower-level one, as shown in Figure 1 below, while also signaling to the user any errors detected during this process. Both the source code and the generated code may belong to different programming languages [11].

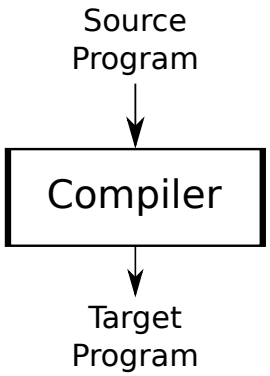


Figure 1: Compilation process. **Source:** [11].

The compiler can output several types of target programs, such as assembly code, intermediate code, machine code or even error messages to the user [11]. If the compiler's output is an executable program, the user can run it to process input and produce outputs, as shown in Figure 2.

Figure 2: Running the target program. **Source:** [11].

However, there is another type of language processor, called interpreters. These are the ones used by Python. What they do is, instead of generating a target program, they execute directly the operations in the source code on inputs provided by the user, as demonstrated in Figure 3 [11].

Figure 3: An interpreter. **Source:** [11].

The compiled target software is usually faster and more efficient than the interpreted program at mapping inputs to outputs. On the other hand, the process of interpretation is generally better at giving feedback to the user about errors in the source code, due to its ability to execute the code line by line [11].

To leverage the advantages of both approaches, a hybrid model was developed in which the source code is first translated into an intermediate representation called bytecode. This bytecode is then executed by virtual machines, as shown in Figure 4 [11].

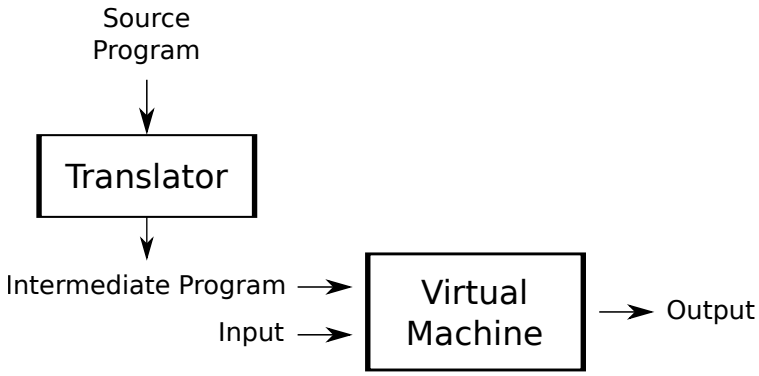


Figure 4: A hybrid compiler. **Source:** [11].

The compilation process is generally divided into two main stages: analysis and synthesis. The analysis stage is responsible for breaking the program down into smaller parts and generating an intermediate representation of the code, often in the form of an abstract syntax tree (AST) [11]. The synthesis stage, in turn, is responsible for building the final code from this intermediate representation, a task that involves more specialized techniques for code generation and optimization [11]. Both stages will be explored in more detail in the following paragraphs.

The analysis stage, also called *front end* of the compiler, as mentioned before, divides the source code into smaller pieces and checks its content, checking whether it follows the syntax and grammar rules, as well as its semantics. This phase also collects information from the source code, storing it in a table called symbol table, that

will be used in the synthesis stage, generating in the end both the symbol table and a intermediate representation of the source program [11].

The synthesis stage, also called *back end* of the compiler, however, is responsible for generating the final target program, using the outputs from the analysis stage, the intermediate code and the symbol table [11].

The whole process of compilation is constituted of several *phases*, which every phase transforms one representation of the original program into another. Figure 5 below shows the phases of a typical compiler [11].

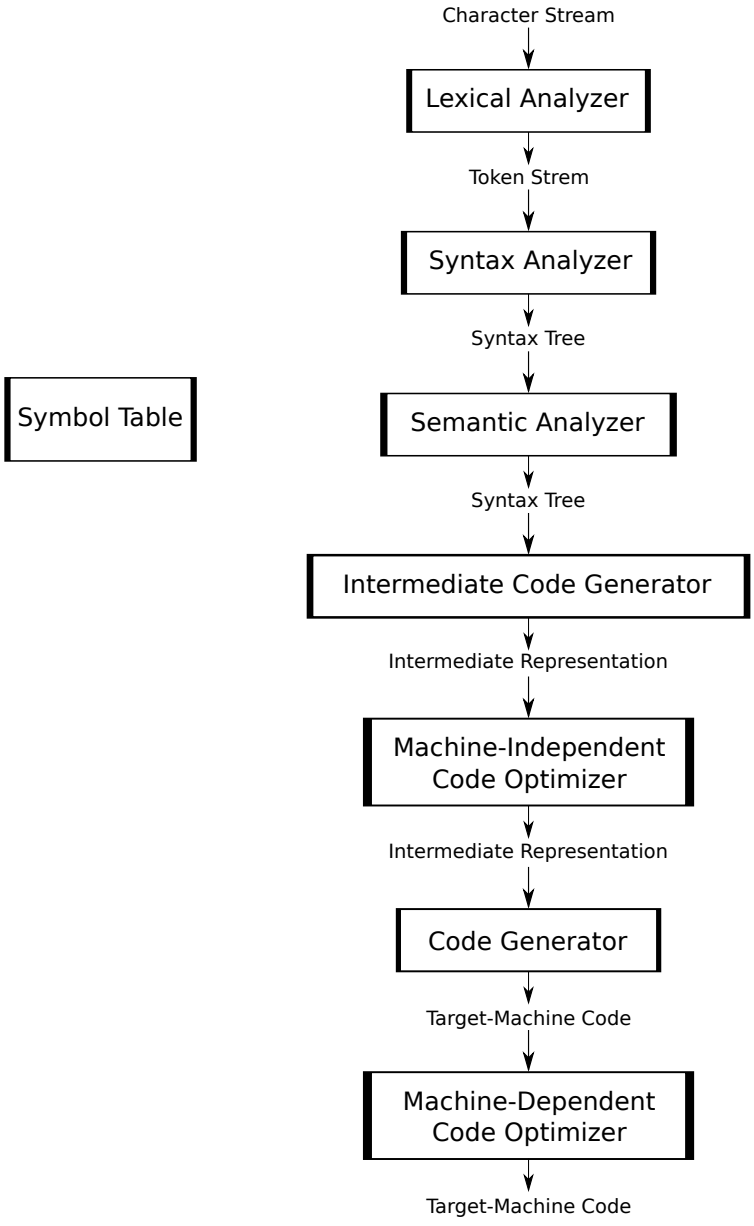


Figure 5: Phases of a compiler. **Source:** [11].

2.2 INTERPRETATION

While the fundamental concepts of compilation and interpretation apply broadly, the remainder of this chapter focuses on their specific implementation, exemplified using the Python programming language. This contextualization is essential to understand the mechanisms of bytecode generation and execution that this work aims to reverse.

Although Python is commonly referred to as an interpreted language, its execution model involves a preliminary compilation step [12]. Instead of generating a native binary executable, the Python source code is compiled into an intermediate representation known as bytecode [13]. This bytecode is then executed at runtime by the Python Virtual Machine (PVM), a core component of the language's runtime system.

The PVM operates according to a stack-based bytecode interpretation model [11], where an interpreter traverses the program's intermediate representation and directly executes its instructions. In this model, Python's bytecode acts as an abstract machine language tailored specifically for the PVM. While this design favors portability and simplicity, it generally results in lower performance compared to execution of native machine code. Even so, the PVM remains essential to Python's architecture, faithfully enacting the language's semantics during execution.

2.3 BYTECODE

Bytecode constitutes an intermediate form generated during Python's compilation process, designed to be efficient and portable for execution by the PVM [14]. Rather than executing the human-readable source code line by line, the interpreter works with this lower-level sequence of platform-independent instructions. These instructions, composed of operation codes (opcodes) and their arguments, can be stored in .pyc files, enabling faster startup times and reducing the need to recompile unchanged source files [15].

Opcodes represent the atomic operations carried out by the PVM, such as loading variables, performing arithmetic, or controlling program flow [16]. In CPython, each bytecode instruction typically consists of one byte for the opcode and one byte for its argument. Since Python 3.6, the interpreter uses a wordcode format where each instruction occupies exactly two bytes, a design choice that improves performance and simplifies the interpreter loop by standardizing the instruction size [17].

For instance, consider the following simple Python code:

```
1  def hello_world():
2      print("Hello World")
```

The corresponding bytecode, generated using the *dis* [18] library, for Python 3.13 might appear as:

```
4           RESUME                   0

5           LOAD_GLOBAL              1 (print + NULL)
            LOAD_CONST               1 ('Hello World')
            CALL                     1
            POP_TOP
            RETURN_CONST             0 (None)
```

In this output, each line includes the instruction offset, the opcode, and its argument. The opcodes, such as RESUME, LOAD_GLOBAL, and CALL, represent the concrete operations the PVM performs to execute the program.

It's important to note that bytecode is specific to the Python version that generated it. Different Python versions may produce different bytecode for the same source code. For example, the bytecode below was generated for Python 3.10:

```
5           0 LOAD_GLOBAL              0 (print)
            2 LOAD_CONST               1 ('Hello World')
            4 CALL_FUNCTION            1
            6 POP_TOP
            8 LOAD_CONST               0 (None)
            10 RETURN_VALUE
```

By comparing both of them, it is possible to see that they are not identical, even though they were generated from the same source code. This version dependency is a crucial consideration when working with Python bytecode, and consequently, when developing tools for its analysis or decompilation. Also, for this reason decompiling tools need to be updated frequently to keep up with the latest Python versions.

2.4 SYNTACTIC TREE

An Abstract Syntax Tree (AST) is a tree-shaped representation of the syntactic structure of source code, widely used in the design of compilers and interpreters for various programming languages. In the context of Python, the AST is built by the parser during the compilation process [16]. It captures the essential syntactic and structural elements of the code, using nodes to represent constructs such as operations and assignments [19]. This intermediate representation is crucial for the compiler to traverse the tree and subsequently generate the bytecode executed by the Python Virtual Machine (PVM). A successfully constructed AST also indicates that the source code has passed the initial syntax validation.

For example, consider the following simple Python code:

```
1 print("Hello World")
```

The corresponding AST, generated using Python's built-in AST module, can be seen in Appendix A.

Visually, this hierarchy can be represented as a tree where the `Module` is the root, containing an `Expr` (expression) node, which in turn contains a `Call` node representing the function invocation. This structure is illustrated in Figure 6.

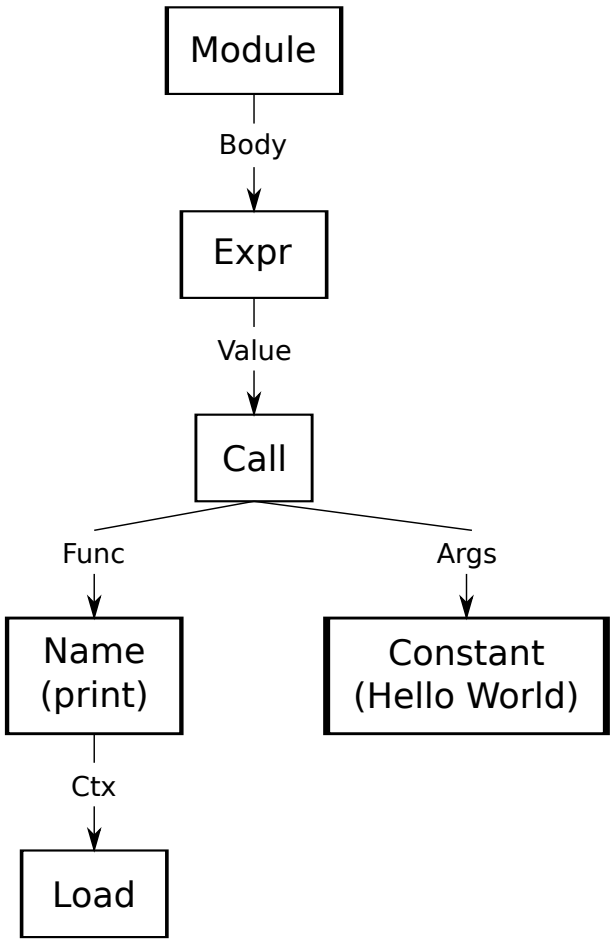


Figure 6: AST representation of the code `print("Hello World")`.

2.5 DECOMPILATION

Decompilation is, essentially, the reverse process of compilation: it takes binary code, or other compiled types of code, that can be executed by the machine and converts it back into a higher-level source code form that humans can read and understand [4, 8]. This process is fundamental for analyzing software when the original source code is unavailable, such as in proprietary systems, legacy code, firmware, or malware [8, 20]. Decompilation is challenging because, during compilation, much of the high-level structure and naming information is lost [4, 20].

Traditional decompilation tools operate by attempting to reverse the synthesis phase of compilation. This process typically involves constructing a Control Flow Graph (CFG) from the bytecode to identify basic blocks and execution paths. Algorithms then analyze this graph to detect patterns that correspond to high-level control structures, such as loops and conditionals, effectively structuring the code. Data flow analysis is also performed to track variable usage and lifespan.

In the specific context of Python, tools like *uncompyle6* [21, 22] translate Python bytecode back into equivalent Python source code. Instead of simply treating the bytecode as a linear stream, *uncompyle6* uses compiler technology to create a parse tree from the instructions, where nodes at the upper levels resemble a Python Abstract Syntax Tree (AST). This approach allows the tool to classify and understand the code structure effectively, reconstructing high-level constructs across a wide range of Python versions, from 1.0 to 3.8 [22].

In the following snippet is presented a Python source code that was compiled using Python 3.8 and then decompiled back to source code using *uncompyle6*.

The source code before compilation is shown in Algorithm 1:

```
    def equated_monthly_installments(
    principal: float, rate_per_annum: float, years_to_repay: int
    ) -> float:
        if principal <= 0:
            raise Exception("Principal borrowed must be > 0")
        if rate_per_annum < 0:
            raise Exception("Rate of interest must be >= 0")
        if years_to_repay <= 0 or not isinstance(years_to_repay, int):
            raise Exception("Years to repay must be an integer > 0")
        rate_per_month = rate_per_annum / 12
        number_of_payments = years_to_repay * 12

        return (
            principal
            * rate_per_month
            * (1 + rate_per_month) ** number_of_payments
            / ((1 + rate_per_month) ** number_of_payments - 1)
        )

    if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

Algorithm 1: Source code before compilation.

The decompiled code after compilation is shown in Algorithm 2.

```
def equated_monthly_installments(principal, rate_per_annum, years_to_repay):
    if principal <= 0:
        raise Exception("Principal borrowed must be > 0")
    elif rate_per_annum < 0:
        raise Exception("Rate of interest must be >= 0")
    raise years_to_repay <= 0 or isinstance(years_to_repay, int) or Exception("Years to repay must be an integer > 0")
    rate_per_month = rate_per_annum / 12
    number_of_payments = years_to_repay * 12
    return principal * rate_per_month * (1 + rate_per_month) ** number_of_payments / ((1 + rate_per_month) ** number_of_payments - 1)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Algorithm 2: Decompiled code after compilation.

In this example, the decompiled output closely mirrors the structure of the original source code, illustrating the general effectiveness of the decompilation process. However, notable discrepancies exist. These include the substitution of multiple `if` statements with `elif` constructs and the complete omission of type hints in the function signature. More critically, the logic for raising exceptions is malformed; the decompiler incorrectly coalesced the conditional check and the exception instantiation into a single statement, rendering the code functionally incorrect. These deviations underscore the inherent challenges of decompilation, particularly in preserving high-level annotations and accurate control flow semantics. This illustrates that these deterministic approaches have limitations. They often generate code that, while functionally equivalent, is difficult to understand [4, 8]. They struggle to infer idiomatic constructs or meaningful variable names, often resorting to generic identifiers (e.g., `var_1`). Modern approaches are being developed to produce more readable and accurate decompiled code [3, 4, 8, 20]. There are no more deterministic and validated decompilers for Python specifically for versions beyond 3.8.


3

3 LARGE LANGUAGE MODEL

In recent years, Large Language Models (LLMs) have become central to the field of Natural Language Processing (NLP), enabling state-of-the-art performance in tasks such as translation, summarization, code generation, and even binary analysis. Their success is driven by a convergence of three main factors: large-scale datasets, powerful neural architectures, and specialized computing infrastructure.

3.1 HISTORICAL CONTEXT AND DEFINITION

The concept of language modeling has evolved significantly over the past decades. Early models were based on statistical methods such as n-grams, which estimate the probability of the next word based on a fixed-size window of previous tokens. These models suffered from data sparsity and lacked the ability to capture long-range dependencies.

The introduction of Recurrent Neural Networks (RNNs), and later Long Short-Term Memory networks (LSTMs) [23], addressed these limitations by maintaining a dynamic internal state that could, in principle, capture arbitrarily long sequences. However, these architectures struggled with parallelization and often encountered vanishing gradient problems.

A major breakthrough occurred with the introduction of the Transformer architecture [24], which replaced recurrence with a novel self-attention mechanism. This allowed for more efficient training and better modeling of global dependencies in text. These advances culminated in the emergence of LLMs such as BERT [25], GPT [26, 27], and LLaMA [28], ushering in the modern era of generative AI.

LLMs are deep neural networks, typically based on the Transformer architecture, and trained on massive datasets composed of natural language and source code. These models contain billions or even trillions of parameters [27, 28], which are adjusted through optimization processes to capture statistical relationships in text and code.

3.2 LANGUAGE MODELING AS A FORMAL PROBLEM

Language modeling is the task of assigning a probability to a sequence of tokens. A token represents the fundamental unit of text or code (e.g., a word, subword, or

character) that a language model processes. Formally, given a sequence of tokens $x = (x_1, x_2, \ldots, x_T)$, a language model estimates the joint probability of the sequence as the product of conditional probabilities:

$$P(x) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}) = \prod_{t=1}^{T} P(x_t \mid x_{<t}) \tag{1}$$

This formulation is known as *causal language modeling* and underlies autoregressive models such as GPT. The goal during training is to learn a function that, given a context $x_{<t}$, accurately estimates the probability of the next token x_t.

For instance, after reading "The cat sat on the", a well-trained model might assign high probability to "mat" and low probability to "banana". This predictive capability enables LLMs to generate coherent text, answer questions, and complete code based on context [29].

Tokens, the fundamental units of processing, are usually obtained through subword tokenization algorithms such as Byte Pair Encoding (BPE) [30]. These techniques strike a balance between vocabulary size and expressiveness, allowing LLMs to handle rare or unseen words effectively.

3.3 ARCHITECTURE AND ATTENTION MECHANISM

The Transformer architecture [24] underpins nearly all modern LLMs. It introduces a paradigm shift from recurrent processing to a fully attention-based mechanism, allowing the model to consider all positions in the input simultaneously, regardless of their distance. This not only improves performance but also enables efficient parallelization during training.

At its core, the Transformer relies on the *self-attention* mechanism, which computes how each token in a sequence relates to every other token. The central equation for scaled dot-product attention is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2}$$

where:

- Q (queries), K (keys), and V (values) are linear projections of the input embeddings,
- d_k is the dimensionality of the keys,
- softmax ensures that the resulting attention weights sum to 1.

This equation captures how much attention each token should pay to every other token, allowing the model to dynamically weigh contextual relevance during inference.

3.4 MODEL VARIANTS IN TRANSFORMER ARCHITECTURES

Transformers can be configured in three main architectural variants, depending on the nature of the task [31, 32]:

- **Encoder-only:** These models are ideal for classification or embedding tasks. A canonical example is BERT [25], which is trained using Masked Language Modeling (MLM) and performs well on sentence classification and question answering.
- **Decoder-only:** These models generate sequences in an autoregressive manner, predicting the next token based solely on previously generated ones. The GPT family [27] is the most prominent example. This setup is particularly well-suited for text generation, code completion, and chat-based applications.
- **Encoder-Decoder:** Also known as sequence-to-sequence models, this structure uses the encoder to digest the input and the decoder to generate output conditioned on the encoded context. Models such as T5 [31] and BART [32] fall into this category. They excel at tasks like translation, summarization, and style transfer.

Each architectural choice brings trade-offs in terms of efficiency, flexibility, and task suitability. Decoder-only models dominate in generative AI applications due to their simplicity and effectiveness in autoregressive tasks. Encoder-decoder models, on the other hand, remain the most effective when rich input-output mappings are needed, such as in translation and structured generation.

3.5 TRAINING: PRETRAINING AND OPTIMIZATION OBJECTIVE

The training of LLMs typically begins with a pretraining phase, in which the model is exposed to massive corpora of text and code. During this stage, learning is driven by *self-supervised learning*, where the model learns to predict parts of the input without requiring explicit human labels [25, 27].

A common example of this is next-token prediction. For instance, given the sequence:

```
The cat sat on the _
```

the model must infer that “mat” is the most likely missing token. Over time, this trains the network to internalize rich statistical representations of language.

The loss function used in this context is the cross-entropy loss, which measures the distance between the predicted probability distribution and the true (one-hot encoded) token. Mathematically, it is defined as:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t \mid x_{<t}) \tag{3}$$

where x_t is the ground-truth token at position t, and $P(x_t \mid x_{<t})$ is the probability assigned by the model.

Minimizing this loss encourages the model to assign higher probabilities to correct tokens, reinforcing its ability to model realistic sequences.

3.6 FINE-TUNING AND ADAPTATION

After pretraining, LLMs can be further adapted to specific tasks or domains using fine-tuning. This phase involves training on a narrower dataset, often with supervision, to improve performance on tasks like legal text processing, biomedical information extraction, or even bytecode analysis, or other specialized applications.

In many scenarios, retraining the entire model is computationally expensive. To mitigate this, *Parameter-Efficient Fine-Tuning* (PEFT) techniques have emerged, including:

- **LoRA** (Low-Rank Adaptation) [33]: Introduces trainable low-rank matrices into existing weight layers, significantly reducing the number of parameters that need to be updated.
- **QLoRA** [34]: Extends LoRA to quantized models (e.g., 4-bit weights), making fine-tuning feasible even on limited hardware.

These methods enable users to customize large models at low cost, without sacrificing quality.

3.7 REINFORCEMENT LEARNING FROM HUMAN FEEDBACK (RLHF)

An increasingly important step in aligning LLMs with human values and preferences is Reinforcement Learning from Human Feedback (RLHF) [35, 36].

The RLHF pipeline typically involves:

1. **Supervised Fine-tuning (SFT)**: The model is trained on human-labeled examples, such as preferred completions or dialogues.
2. **Reward Modeling**: A separate model is trained to predict human preferences by ranking alternative outputs, creating a reward signal that reflects human judgment.
3. **Reinforcement Learning**: The base model is fine-tuned using reinforcement learning to optimize its behavior, maximizing the reward signal derived from the reward model.

This process has been instrumental in the development of assistant-style models such as ChatGPT, where alignment with user intent, safety, and helpfulness is essential.

Unlike standard fine-tuning, which optimizes for task performance, RLHF focuses on *preference alignment*, enabling more nuanced control over LLM behavior.

3.8 PROMPTING AND CONTEXT MANAGEMENT

Once pretrained, LLMs operate in an inference mode where they generate outputs conditioned on a prompt, a sequence of tokens that serves as the input. Since LLMs are typically *decoder-only* architectures, they generate text auto-regressively: token by token, based solely on the provided sequence of preceding tokens [27].

The prompt directly determines the model's behavior. For example, providing the instruction:

```
Translate the following English sentence to French: "I love science."
```

leads the model to produce `"J'aime la science."`.

However, the model's ability to process this input is constrained by a fixed limit known as the *context window*, explained in detail in the following section.

3.9 CONTEXT WINDOW AND LIMITATIONS

Large Language Models (LLMs) operate within a maximum context length, measured in tokens (e.g., 2k, 8k, 32k), which constrains the volume of information the model can process in a single inference step. Furthermore, standard LLMs are stateless and lack persistent memory, meaning they cannot retain information across separate interactions once the context window is reset.

When an input exceeds this context limit, tokens are typically truncated (often from the beginning of the sequence) which can lead to loss of critical information and incoherent model behavior. Consequently, effective prompt design and context management are essential strategies in practical applications [37].

3.10 PROMPTING STRATEGIES

Several prompting techniques have emerged to improve LLM performance without additional training:

- **Zero-shot prompting**: The model is given only a task description.
  `"Classify: The movie was boring."` $\longrightarrow$ `"Negative"`
- **Few-shot prompting**: The model is given a few input-output examples before the query.
  `"Positive: I loved the film.`
  `Negative: The plot was terrible.`
  `Sentiment: It was amazing."` $\longrightarrow$ `"Positive"`

- **Chain-of-thought prompting** [38]: The prompt includes step-by-step reasoning to encourage intermediate steps in generation.

```
"Q: If there are 3 apples and you take away 2, how many are left?
A: There were 3 apples. You took 2. So, 1 is left."
```

These approaches enable better reasoning and task completion, even in zero-resource scenarios. Recent studies show that careful prompt engineering can rival or exceed fine-tuned models on specific benchmarks [39].

3.11 SYSTEM PROMPTS AND ROLE CONDITIONING

In chat-based systems, models also interpret a special system message (e.g., "You are a helpful assistant.") that conditions the behavior of the assistant. This is part of the context, and subtle changes can significantly affect responses.

3.12 RETRIEVAL-AUGMENTED GENERATION (RAG)

To overcome the limits of a model's static knowledge and its finite context window, a technique called *Retrieval-Augmented Generation* (RAG) was created. RAG improves LLMs by connecting them to an external, up-to-date knowledge source [40]. This method allows the model to give answers based on current or domain-specific information without needing to be fully retrained.

RAG works in a two-step process [40]:

1. **Retrieval:** First, when a query is made, a retriever model searches for relevant information in an external knowledge base (like website articles or technical documents). This retriever, such as the Dense Passage Retriever (DPR), turns the query and documents into vectors to find the best matches through a similarity search [40].

2. **Augmented Generation:** Next, the retrieved text pieces are added to the LLM's prompt. This gives the model extra context. A generator model, like BART, uses this expanded context to create a more accurate, factual, and detailed answer [40].

This hybrid approach combines the model's internal (parametric) knowledge with an external (non-parametric) knowledge base, providing key benefits:

- **Reduces Hallucinations:** By grounding answers in real data, RAG makes LLMs less likely to invent information [40].

- **Provides Current Knowledge:** The external knowledge base can be updated easily, so the LLM can answer questions about recent events.

- **Improves Transparency:** Responses can include their sources, allowing users to check the facts and trust the answers more.

3.13 MODEL CONTEXT PROTOCOL (MCP)

The Model Context Protocol (MCP) is an recently proposed standard designed to standardize the definition, discovery, and invocation of external tools and resources for AI applications [41, 42]. Unlike traditional methods such as manual API wiring or platform-specific plugins, which often lead to fragmented and fragile integrations, MCP provides a unified protocol that decouples tool implementation from usage. This allows AI models to interact with a diverse ecosystem of tools and data sources through a consistent interface, enhancing interoperability and scalability [42].

The architecture of MCP is built around three primary components that collaborate to facilitate secure and efficient communication [42]:

- **MCP Host:** The AI application (e.g., an IDE, a chat interface, or an autonomous agent) that initiates tasks and provides the execution environment. It integrates the MCP client to communicate with external services.
- **MCP Client:** Acting as an intermediary within the host, the client maintains a one-to-one communication link with an MCP server. It manages requests, processes notifications, and handles capability negotiation on behalf of the host.
- **MCP Server:** A standalone service that exposes specific capabilities to the AI model. These capabilities are categorized into three main primitives:
  - *Tools:* Executable functions that allow the model to perform actions (e.g., querying an API, executing code) and receive results.
  - *Resources:* Structured or unstructured data sources (e.g., logs, files, database records) that the model can read to gain context.
  - *Prompts:* Reusable templates and workflows that help standardizing interactions and optimizing model performance for specific tasks.

This modular design, illustrated in Figure 7, shifts the paradigm from hardcoded tool bindings to a dynamic discovery model. Clients can list available tools at runtime and negotiate schemas, allowing for a flexible “supply-and-consume” ecosystem where tools can be developed independently of the models that use them [42].

By treating the prompt as a contract, MCP turns “prompt engineering” into a more systematic process of “context programming”. This approach helps developers build more robust applications where the LLM acts like a “natural language CPU”, executing instructions consistently within a larger system [41]. MCP shifts the focus from fine-tuning the model to intelligently orchestrating the context that is fed to it.

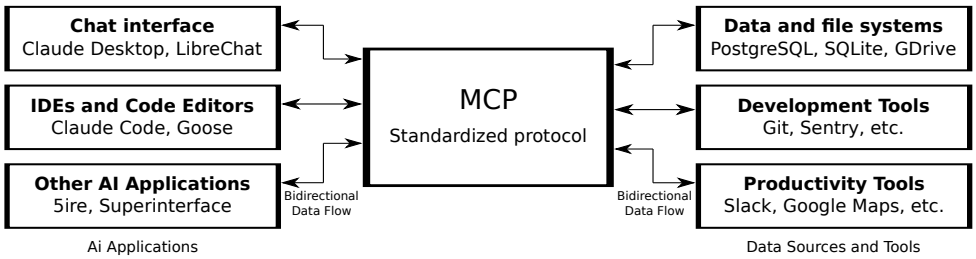


Figure 7: Model Context Protocol (MCP) architecture [41].

3.14 SUMMARY

In summary, Large Language Models (LLMs) are the result of decades of progress in neural architectures, computing power, and data availability. The Transformer architecture, with its self-attention mechanism, was a major step forward, allowing models to understand long-range connections in text and overcoming the limitations of older recurrent models [23, 24]. The modern approach of large-scale pre-training, followed by efficient adaptation methods like fine-tuning, has created a strong framework for building versatile AI systems. The Model Context Protocol introduced a structured way to interact with LLMs, enhancing their reliability and usability in complex applications [41]. The future of the field points toward even larger models with expanded context windows and new ways of integrating knowledge, promising to further transform the interaction between humans and machines.

4

4 LITERATURE REVIEW

4.1 METHODOLOGY

The literature review was conducted to identify the state of the art in applying large language models (LLMs) to decompile Python bytecode and recover source code from intermediate or binary representations. The main goal was to analyze approaches using deep learning, transformers, or advanced prompting strategies in reverse engineering.

Searches were performed in the ACM Digital Library, Scopus, Web of Science, and IEEE Xplore databases. The primary search query used was:

```
("large language models" OR "LLMs")
AND ("bytecode" OR "binary code")
AND ("code recovery" OR "decompile"
OR "decompilation" OR "decompiler")
```

No strict time frame was set, as most relevant publications are recent. However, priority was given to works from the last ten years, with a special focus on the last five.

The selection process involved a rigorous application of both inclusion and exclusion criteria to ensure relevance and quality. Inclusion criteria prioritized studies that addressed Large Language Models (LLMs), deep learning, or transformer architectures applied to reverse engineering, decompilation, or source code recovery. Furthermore, selected articles focused on tasks involving the generation or reconstruction of source code from various low-level representations such as bytecode, binaries, or Intermediate Representations (IRs), and explored diverse prompting strategies for LLMs. Preference was given to peer-reviewed publications within the last 10 years, with a strong emphasis on research from the past 5 years. Conversely, articles were excluded if their theme was non-technical (e.g., educational or philosophical applications of LLMs unrelated to code), if they did not utilize LLMs or machine learning for reverse engineering tasks. In total, 33 papers were found, and 7 were selected for a detailed analysis.

The Table 1 shows the distribution of articles found across the different research databases. Its sum exceeds the total number of unique articles due to overlaps between databases.

Database	Found
ACM	6
IEEE	14
Scopus	29
Web of Science	23

Table 1: Articles found in each research database

4.2 RELATED WORKS

Recent studies have increasingly applied language models to reverse engineering, especially for recovering source code from low-level representations like binaries and bytecode. This section reviews seven key studies that show the state of the art in this field, focusing on deep learning and Large Language Models (LLMs).

A central theme in this research is treating decompilation as a sequence-to-sequence translation task. This approach, first explored with Recurrent Neural Networks (RNNs) [8], has become much more powerful with the Transformer architecture used in modern LLMs. By viewing low-level instructions as a "source language" and high-level code as the "target language", researchers have used these models to generate code that is often more readable and semantically correct than the output from traditional, rule-based decompilers.

Two main strategies have emerged for applying LLMs to decompilation:

1. End-to-End Decompilation: This approach trains an LLM to directly translate a low-level format into high-level source code. A leading example is **LLM4Decompile** [4], which created a series of open-source LLMs specifically for this task. Their models significantly outperform traditional tools like Ghidra and even general-purpose models like GPT-4o, producing C code from binaries with much higher rates of re-executability. Similarly, **WaDec** [3] demonstrates the power of a fine-tuned LLM for a specific target, WebAssembly. It produces highly readable and recompilable C code by training the model on a specialized dataset of code snippets.

2. Decompiler Output Refinement: This is a practical, hybrid approach where an LLM acts as a post-processor to improve the output of an existing decompiler. It focuses on fixing "quirks", which are unnatural or non-idiomatic patterns that traditional tools often produce. The work on **Automatic Fixation of Decompilation Quirks** [5] shows this effectively, using a pre-trained model to treat these quirks like grammatical errors and "correct" them in decompiled Java code. The LLM4Decompile project also explores this with its "Ref" models, which refine Ghidra's output, proving that combining traditional tools with LLMs can lead to superior results.

Beyond these two strategies, other studies have explored unified models and broader

applications. **HexT5** [10] presents a model pre-trained on pseudo-code to recover different types of semantic information at once, such as variable names, function names, and comments from stripped binaries. This shows a move towards more complete, holistic solutions. A systematic benchmark presented in **How Far Have We Gone in Binary Code Understanding Using Large Language Models** [43] evaluates various LLMs on tasks like function name recovery and code summarization. It confirms their strong potential while also highlighting their current limitations, reinforcing the idea that specialized models perform best.

Finally, the practical benefits of this technology are shown in studies like **Can Neural Decompilation Assist Vulnerability Prediction on Binary Code?** [20]. This work uses neural decompilation as a first step to enable deep learning models to find security vulnerabilities in binary files. It shows that decompiled code is a more effective input for this task than raw assembly, outperforming other state-of-the-art methods.

4.3 REVIEW SUMMARY

In summary, the examined works highlight a clear trend: LLMs are powerful tools for code recovery and binary analysis. However, their success depends heavily on domain-specific fine-tuning, or other techniques, with large, high-quality datasets. While there has been significant progress for languages like C, C++, and WebAssembly, a clear gap exists in the literature: none of the analyzed studies directly address the decompilation of Python bytecode.

Furthermore, while previous studies have explored a variety of model architectures, this work focuses on evaluating the applicability of modern large language models to the task of Python bytecode decompilation. The goal is to assess whether the reasoning abilities, long-context handling capabilities and generalization properties reported in recent state-of-the-art models can help mitigate the semantic gap inherent in recovering high-level source code from bytecode. The challenges addressed in other domains, such as reconstructing meaningful identifiers, recovering structural abstractions and producing idiomatic code, remain equally relevant in the Python ecosystem. In this context, the existing literature provides a solid foundation and a clear motivation for extending these techniques to the specific problem of Python bytecode decompilation.


5

5 PROPOSED ARCHITECTURE

This chapter details the proposed architecture for the decompilation of Python bytecode using Large Language Models (LLMs).

5.1 OVERVIEW

The proposed architecture is designed to address the loss of semantic information in decompiled code by integrating Large Language Models (LLMs) with a rigorous feedback loop based on software testing. It establishes a controlled environment where input bytecode is processed, and the resulting source code is systematically validated to ensure functional correctness.

The architecture defined a data flow to ensure the reliability of the decompilation process:

- **Input Stage:** The process begins with the ingestion of Python bytecode from a self-contained environment and executable code.
- **Processing Stage:** The Large Language Model acts as the core transformation engine, interpreting the bytecode context to generate a candidate source code.
- **Validation Stage:** A dedicated test runner executes the generated code against a suite of unit tests to verify functional equivalence.
- **Output Stage:** The cycle concludes by outputting the successfully decompiled code or logging the results of the attempt.

The architecture consists of a set of components that are organized into a workflow, within which a context-retrieval mechanism supports the decompilation process.

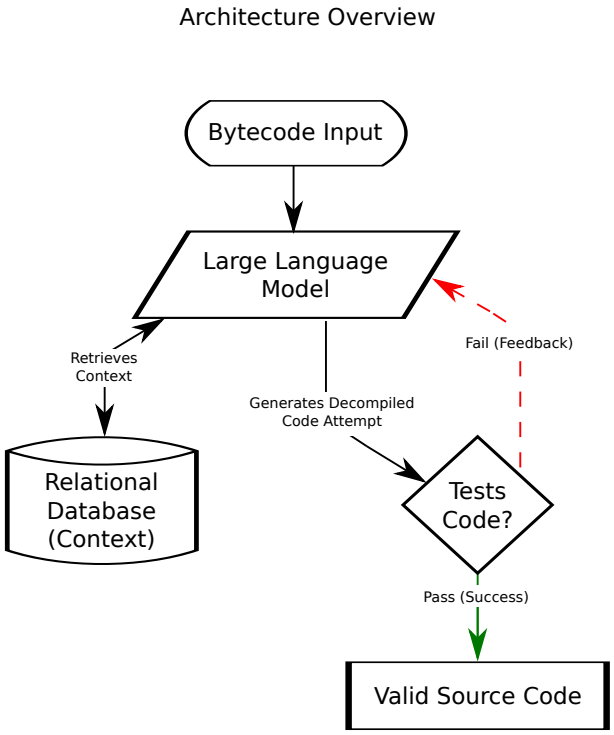


Figure 8: Simplified flowchart of the proposed architecture.

As illustrated in Figure 8, the MCP allows the model to search for context in the database, using relevant labels, retrieving context information that may assist in the process of decompilation.

5.2 COMPONENTS AND RESPONSABILITIES

The proposed architecture consists of the following key components that will be detailed in the next sections:

- **Model Interface**
- **Context Store**
- **Test Runner**
- **Orchestrator**

5.2.1 Model Interface

The Model Interface serves as the abstraction layer responsible for mediating the interaction between the Large Language Model (LLM) and the system's external resources. Implemented via the Model Context Protocol (MCP), this component acts as an MCP Server, exposing the project's relational database as a set of standardized tools and context resources.

Instead of relying on rigid, hardcoded queries, the Model Interface defines a schema of capabilities that the LLM can discover and invoke dynamically. This design effectively

decouples the inference engine from the data storage details, allowing the model to actively query for semantic context, such as reference implementations or similar bytecode patterns, based on its own reasoning process. By facilitating this bi-directional and protocol-driven communication, the Model Interface transforms the static dataset into an interactive knowledge base, essential for bridging the semantic gap in decompilation.

5.2.2 Context Store

The Context Store is implemented as a relational database that functions as the central persistence layer for the architecture, maintaining the system's state and enabling structured data retrieval. It is responsible for storing the entire lifecycle of the decompilation process, from the initial ingestion of source code and bytecode to the final validation results.

5.2.3 Test Runner

The Test Runner constitutes the validation layer of the proposed architecture,tasked with verifying the semantic correctness of the reconstructed source code. Unlike static analysis tools, this component performs dynamic verification by executing the generated code against a comprehensive suite of unit tests, ensuring functional equivalence with the original logic.

From an architectural perspective, the Test Runner encapsulates the execution environment, abstracting the complexity of execution management, including resource allocation and runtime monitoring, ensuring stability regardless of the input code's behavior.

The module operates as a deterministic oracle within the feedback loop. It evaluates the candidate solution by comparing its execution outputs against established ground-truth data. The outcome of this validation, whether a complete success or a failure with specific error diagnostics, determines the subsequent state of the workflow.

5.2.4 Orchestrator

The Orchestrator serves as the control unit of the proposed architecture, responsible for automating and synchronizing the end-to-end decompilation workflow. Acting as the system's backbone, it manages the lifecycle of each decompilation task, from the initial retrieval of bytecode to the final storage of results.

Its primary function is to coordinate the iterative "generate-test-refine" loop. The Orchestrator sequentially triggers the Model Interface to generate source code, dispatches this code to the Test Runner for validation, and interprets the execution feedback. Based on the test results, it dynamically decides the next course of action: either finalizing the process upon success or initiating a refinement cycle with updated context if errors are detected.

Furthermore, the Orchestrator enforces execution policies, such as managing the maximum number of retry attempts and handling timeout conditions. It also ensures data integrity by logging all intermediate states, generated artifacts, and validation metrics into the Context Store, thereby enabling comprehensive traceability of the decompilation process.

5.2.5 Logging and Metrics

There are a few pieces of information captured during the decompilation process to evaluate the performance of the architecture:

- **LLM Response Error flag:** A boolean flag indicating whether the LLM's response was not successful (e.g., API error, timeout).
- **Limit of attempts exceeded:** A boolean flag indicating whether the maximum number of decompilation attempts was reached without success.
- **Number of times the model tried to decompile:** The number of generated code snippets that are syntactically valid Python code.
- **Execution Time:** The total time taken for the decompilation process, measured from the start of the bytecode ingestion to the completion of validation.

5.3 IMPLEMENTATION DETAILS

The following sections provide specific implementation details for each component of the proposed architecture. The tools and technologies selected are not mandatory and can be replaced by alternatives that fulfill the same roles, once they follow the defined architectural principles.

5.3.1 Workflow Engine Implementation

The workflow engine serves as the backbone for orchestrating the disparate components of the architecture, database interactions, LLM inference, and test execution. For this implementation, n8n was selected as the orchestration platform [44]. n8n is a workflow automation tool that employs a node-based visual interface to define execution logic.

The choice of n8n is motivated by some factors, such as its visual abstraction, which facilitates the mapping and monitoring of the "generate-test-refine" loop. Its extensibility through custom nodes and HTTP requests enables facilitating integration with the Model Context Protocol (MCP) server and the external test runner. Additionally, the availability of a self-hosted version [45] allows for data privacy and control over the execution environment, satisfying the security requirements for handling untrusted bytecode.

However, the use of such a high-level tool introduces certain limitations. It may incur a resource overhead and slight latency compared to a purely code-based orchestration script, particularly regarding data serialization between nodes. Additionally, while visual workflows aid initial understanding, managing extremely complex loops with extensive error handling can become visually cluttered, potentially increasing maintenance effort compared to standard modular code.

5.3.2 Model Implementation

The decompilation process utilizes the Gemini 2.5-flash model, selected for its optimization in high-frequency, low-latency tasks [46]. This characteristic is essential for the architecture's iterative workflow, which often requires multiple inference cycles per function.

The model offers significant advantages regarding scalability and efficiency. Its reduced cost per token compared to larger models enables extensive experimental testing without prohibitive expense [47]. Furthermore, its extended context window allows for the processing of complete bytecode listings and retrieved database context.

However, the model's lightweight architecture entails a trade-off in reasoning depth compared to larger parameter-dense alternatives. This limitation is structurally mitigated by the system's feedback loop, which relies on execution-based validation and retrieval of context to verify and refine the generated code.

5.3.3 Storage Implementation

The Context Store is implemented using PostgreSQL, an open-source object-relational database system [48]. The selection of a relational database ensures data integrity through full ACID (Atomicity, Consistency, Isolation, Durability) compliance and provides robust support for complex SQL queries. These capabilities are essential for the system's retrieval of data, enabling the Model Interface to execute precise filtering and joins to retrieve relevant code examples based on bytecode patterns and classification labels. The implementated database schema is detailed in Appendix B. For an alternative implementation, other storage alternatives can be considered, for more scalable or NoSQL-based solutions, as long as they provide similar querying capabilities.

5.3.4 Test Runner Implementation

A dedicated execution environment was developed to validate the functional correctness of the decompiled code. This component, implemented in Python, orchestrates the execution of the generated functions against the predefined suite of unit tests. It is engineered to capture standard output streams and runtime exceptions, giving the feedback mechanism its integral role in the architecture's iterative refinement loop, providing the Large Language Model with specific error diagnostics to guide subsequent decompilation attempts.

5.4 EXPERIMENTAL PIPELINE

The experimental validation of the proposed architecture is structured into two distinct and sequential phases: a preparatory phase focused on dataset construction and knowledge base formation, followed by an execution phase dedicated to the iterative decompilation process. This bipartite division ensures that the system is evaluated against a rigorous, pre-validated standard.

Phase 1 serves as the foundational stage, where the ground truth is established. It involves the extraction of functions, the generation of comprehensive test suites, and the semantic enrichment of the dataset through AI-driven classification. The artifacts produced in this phase create the necessary infrastructure for the subsequent experiments. Phase 2 then leverages this foundation to execute the decompilation workflow. It utilizes the prepared dataset to prompt the LLM, retrieves relevant context based on the established classifications, and validates the generated code against the pre-verified test suites. This separation of concerns ensures that the evaluation in Phase 2 is both robust and reproducible, relying on the high-quality data and testing standards defined in Phase 1.

The entire experiment is orchestrated around a central relational database that stores source code, bytecode, generated tests, and classification labels, acting as the single source of truth for the experiment.

5.4.1 Phase 1: Dataset and Testbed Construction

The first phase focuses on preparing a high-quality dataset and a comprehensive testbed. This process leverages the capabilities of the LLM to enrich the initial data collected from open-source repositories.

1. **Data Extraction:** The process begins by extracting Python functions from a curated open-source repository. These functions, along with their original source code and metadata, are stored in a database.

2. **AI-driven Data Enrichment:** A Large Language Model (LLM) is employed to analyze and enrich the dataset. This model performs two main tasks:

    - **Function Classification:** It generates descriptive labels for each function, classifying them based on their purpose or algorithmic category (see Appendix C.2 for the complete prompt).
    - **Automated Test Generation:** The model was instructed to generate a pool tests for each function, based on its signature, source code, and initial doctests (the full prompt used for this task can be found in Appendix C), outputting tests that will be executed by the Test Runner.

    All generated labels and tests are stored in the central database.

3. **Ground-Truth Test Generation:** For each function, a ground-truth test answer key is established, executing the generated tests on the original source code. The results serve as a baseline for verifying the functional correctness of the decompiled code later on. This ensures that the decompiled output matches the expected behavior defined by the original implementation.

The entire Phase 1 is illustrated in Figure 9.

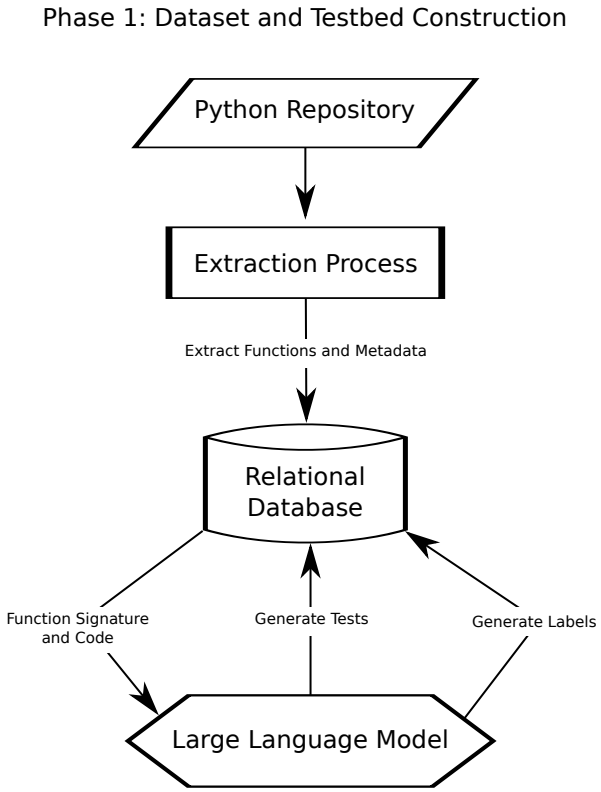


Figure 9: Flowchart of Phase 1: Dataset and Testbed Construction.

5.4.2 Phase 2: Orchestration of the Iterative Decompilation Flow

The second phase represents the core of the decompilation process, where bytecode is translated into source code and iteratively validated. The entire orchestration is shown in Figure 10.

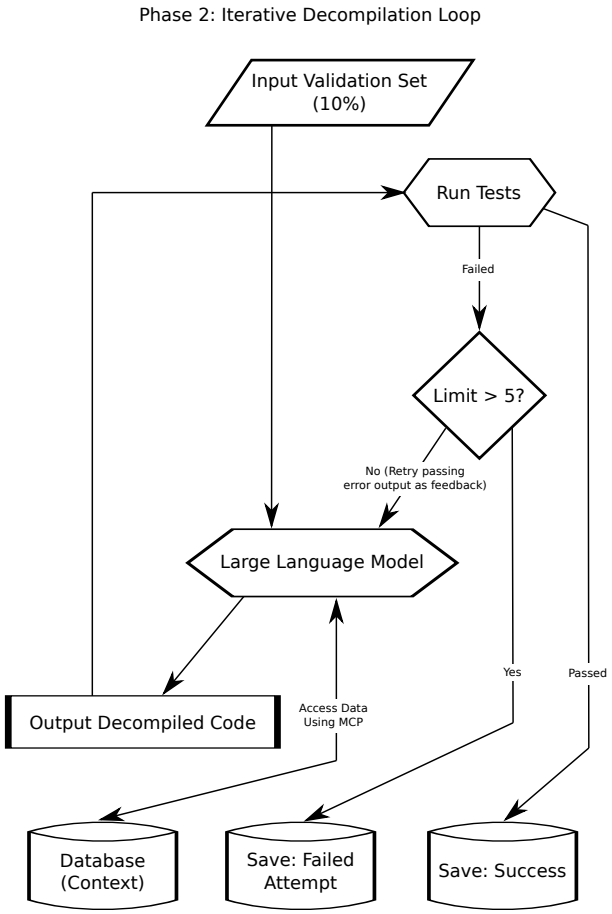


Figure 10: Flowchart of Phase 2: Iterative Decompilation and Validation.

The Figure 10 illustrates the iterative process of decompilation:

- **Dataset Splitting:** Before initiating the steps in Phase 2, the dataset created in Phase 1 is divided into two segments: 10% is set aside as a hold-out validation set for evaluating the final decompilation results, while the remaining 90% is used to provide context during the decompilation process. The validation set remains untouched throughout the decompilation workflow, serving solely as a benchmark for assessing the architecture's performance.
- **Bytecode:** Each function's bytecode is sent to the LLM to generate the initial decompiled source code (see Appendix C.3 for the full prompt).
- **LLM:** The LLM consults the database based on classification labels to retrieve relevant context that may assist in the process of decompiling.
- **Runner:** A Python script executes the generated code against the established test suite, capturing the output and any errors and comparing them against the ground truth.

- **Feedback Loop:** If the generated code fails any test, the error output is appended to the original prompt, and a new decompilation attempt is made. This cycle continues until the code passes all tests or a maximum of five attempts is reached.
- **Storage:** The attempts, whether successful or exceeded the limit, are recorded in the database for further analysis.

5.4.3 Full Workflow Execution

The complete workflow is illustrated in Figure 11, showcasing the integration of both phases and the iterative decompilation process.

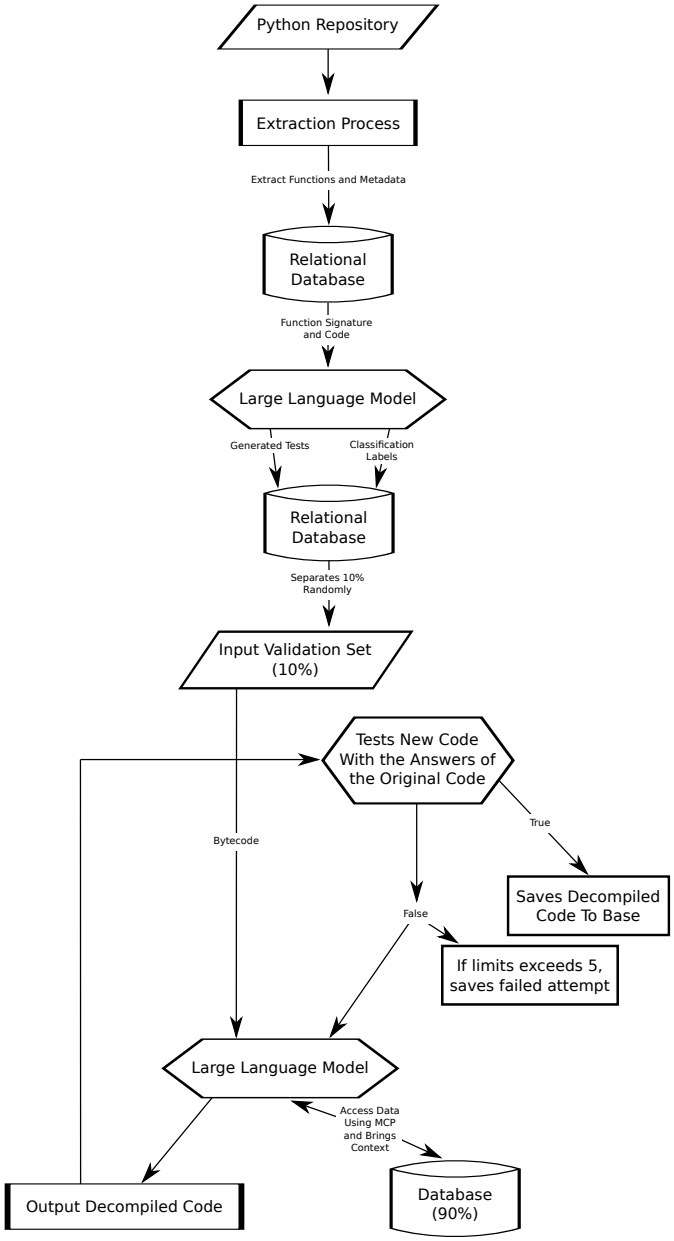


Figure 11: Comprehensive flowchart of complete workflow.

5.5 EVALUATION METRICS AND SUCCESS CRITERIA

The metric used to evaluate the decompilation process is the **Success Rate**, defined as the percentage of decompiled functions that pass all 25 unit tests in the validation suite. A function is considered successfully decompiled if the generated source code produces outputs identical to the original code for the given test inputs, demonstrating functional equivalence.

5.6 LIMITATIONS AND THREATS TO VALIDITY

Several limitations and potential threats to validity are acknowledged in this experimental design and evaluation process:

- **Internal Validity:** There is a risk that the model may have memorized code for lack of internal control of the modal and stochastic variation of prompts. This could lead to different results, affecting reproducibility.
- **External Validity:** The dataset is restricted to self-contained algorithmic functions. Consequently, the findings may not generalize to large-scale, enterprise-level applications involving complex dependencies and external libraries.
- **Construct Validity:** Success is measured by passing unit tests, which guarantees functional approximation but not the exact reconstruction of the original algorithm.
- **Test Coverage Risks:** Since the test suites are generated automatically, they may not cover all edge cases. Incomplete coverage could lead to false positives, where incorrect decompilations pass validation.
- **Bytecode Semantic Gap:** The compilation process strips essential semantic information, such as variable names and comments. This inherently limits the model's ability to fully restore the original readability and intent of the source code.
- **Model Specificity:** The results are dependent on the specific architecture and training. Different models may yield significantly different performance profiles.


6

6 RESULTS AND DISCUSSION

This chapter presents the results obtained from the experimental evaluation of the proposed architecture. We analyze the effectiveness of the approach using the metrics defined in the previous chapter and discuss the implications of the findings.

6.1 EXPERIMENTAL SETUP

As mentioned before, the experiments were conducted using the Gemini-2.5-flash model accessed via the N8N orchestration platform. The dataset approximately consisted of 1,600 Python functions, which were carefully selected to represent a diverse range of algorithmic challenges, ensuring comprehensive evaluation and were saved in a PostgreSQL database.

6.1.1 Execution Conditions

The experiments were performed on a laptop machine with the following specifications:

- CPU: AMD Ryzen 7 5800h
- GPU: NVIDIA GeForce GTX 1650 Mobile
- RAM: 24GB
- Storage: 256GB SSD NVMe
- OS: Pop_OS 22.04 LTS

The number of maximum attempts for each function was set to 5, meaning that if the model failed to generate a function that passed all unit tests within 5 tries, the attempt was considered a failure.

If the code attempt lasted more than 10 seconds without returning a result, it was automatically aborted and counted as a failed attempt.

6.1.2 Model Parameter Control

For executing the decompilation tasks, the Gemini-2.5-flash model was utilized without any additional configurations. On the other hand, for the test and labels generation, the model was set to use a temperature of 0.3 and a maximum output token of 8192.

6.1.3 Construction of Dataset

The entire dataset constructed in the collection phase is composed of 1,600 Python functions, each paired with its corresponding bytecode representation, random unit tests, and classification labels. The functions span a wide range of algorithmic categories, including sorting algorithms, mathematical computations, data structure manipulations, and string processing tasks. From this consistent dataset, a validation subset composed of 10% of the functions was extracted to evaluate the decompilation performance of the proposed architecture.

The validation dataset was constructed from a larger pool of Python functions collected during the data collection phase, being collected by randomly sampling functions. Its unique labels count was analyzed to ensure a balanced representation of different categories. The final validation set consisted of 160 diverse Python functions with 166 unique classification labels, covering a wide range of algorithmic challenges.

6.2 EXPERIMENTAL RESULTS

The validation was conducted on a held-out set of 160 Python functions, representing 10% of the total processed dataset. These functions were never included in the experiment's dataset or context retrieval phase, ensuring an unbiased evaluation.

6.2.1 Quantitative Analysis

The primary objective was to determine if the model could recover functionally equivalent source code from bytecode. The results of success rate are summarized in Table 2.

Result	**Count**	**%**
Successfully Decompiled	147	91.875
Failed	13	8.125
Total	**160**	

Table 2: Decompilation Results Summary

Out of the 160 bytecode samples, the system successfully recovered 147 functions. Success was defined as the generated code passing all 25 unit tests generated and validated in the data collection phase. This yields a high success rate of approximately **91.875%**. 13 functions failed to pass the test suite within the maximum allowed

attempts.

These results indicate that the approach shows potential in decompiling bytecode for the types of algorithmic functions present in the dataset. The feedback loop and its ability to read the error message from the test execution played a crucial role, since 30 decompiled codes were successful not at first try. The distribution of attempts for the successful functions is shown in Figure 12.

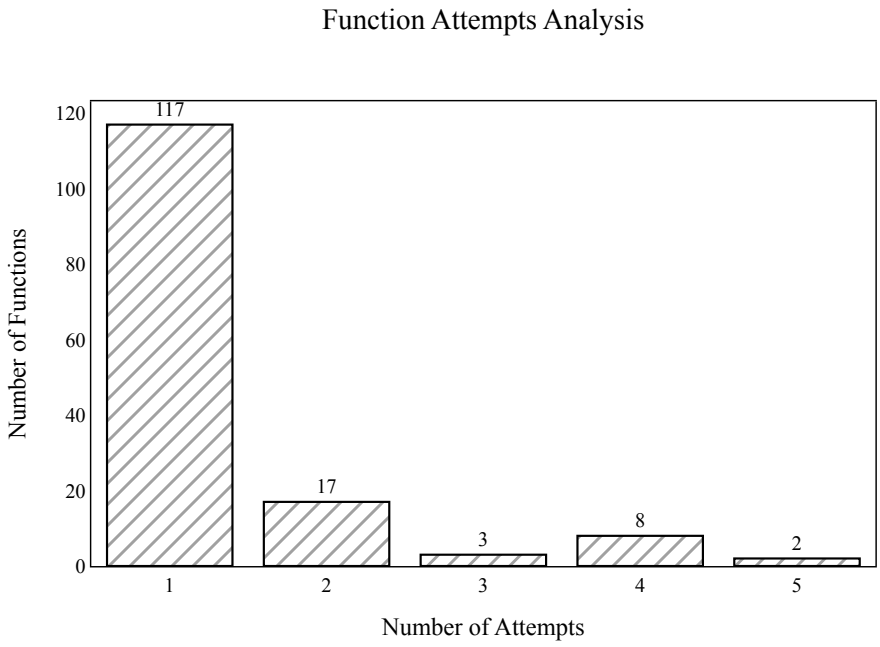


Figure 12: Distribution of Attempts for Successful Functions

- **Total Orchestration Run Time:** 1 hour 35 minutes 42 seconds
- **Average Time per Function:** Approximately 32.82 seconds
- **Average Time per Successful Function:** Approximately 24.8 seconds
- **Average Time per Failed Function:** Approximately 122.8 seconds
- **Average Attempts per Function:** Approximately 1.66 attempts

6.2.2 Qualitative Analysis

An analysis of the classification labels associated with the 13 failed samples provides further insight into the model's limitations. The failures were not uniformly distributed but rather clustered around specific algorithmic categories. A significant portion of the failures involved the model not outputing any code in the last try. Other failures involved complex mathematical and scientific calculations (e.g., `hubble_calculation`,

linear_equation_solving), suggesting challenges in reconstructing precise numerical logic or handling floating-point precision from bytecode. Another notable cluster was related to data structures, specifically tree_traversal and tree_manipulation, which often involve recursive patterns that can be ambiguous at the bytecode level.

In Algorithm 3, we present an example of an original function that was successfully decompiled by the model.

```
import numpy as np

def runge_kutta(f, y0, x0, h, x_end):
    n = int(np.ceil((x_end - x0) / h))
    y = np.zeros((n + 1,))
    y[0] = y0
    x = x0

    for k in range(n):
        k1 = f(x, y[k])
        k2 = f(x + 0.5 * h, y[k] + 0.5 * h * k1)
        k3 = f(x + 0.5 * h, y[k] + 0.5 * h * k2)
        k4 = f(x + h, y[k] + h * k3)
        y[k + 1] = y[k] + (1 / 6) * h * (k1 + 2 * k2 + 2 * k3 + k4)
        x += h

    return y
```

Algorithm 3: Original Function Example

And here, in Algorithm 4, is the corresponding decompiled output generated by the model.

```
import numpy as np

def runge_kutta(f, y0, x0, h, x_end):
    n = int(np.ceil((x_end - x0) / h))
    y = np.zeros(n + 1)
    y[0] = y0
    x = x0

    for k in range(n):
        k1 = f(x, y[k])
        k2 = f(x + 0.5 * h, y[k] + 0.5 * h * k1)
        k3 = f(x + 0.5 * h, y[k] + 0.5 * h * k2)
        k4 = f(x + h, y[k] + h * k3)
        y[k + 1] = y[k] + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
        x += h
    return y
```

Algorithm 4: Decompiled Function Example

It is evident that the decompiled code closely mirrors the original function, with only minor syntactic variations that do not affect functionality, since the multiplication occurs before the addition due to operator precedence rules, the expression `1/6 * h` is equivalent to `(h / 6)`.

In the Appendix is presented one of the failed decompilation cases. The original function can be seen entirely in Appendix D, but here is an snippet of it for reference:

```
    useable_form = data_set.copy()
    simplified = simplify(useable_form)
    simplified = simplified[::-1]
```

The entire decompiled source code is presented in Appendix E. The equivalent snippet from the decompiled code is:

```
    useable_form = data_set.copy()
    simplified = simplify(useable_form)
    simplified = simplified[::-1]
```

The comparison between the original source and the decompiled reveals that while the reconstruction achieved high structural fidelity, preserving variable nomenclature and control flow, it failed to maintain semantic integrity in critical operations. The most significant deviation occurred in the translation of Python's slicing syntax, where list inversion operations `([::-1])`, essential for the back-substitution phase of the Gaussian elimination algorithm, were erroneously decompiled as truncation operations `([:-1])`. This subtle syntactic hallucination fundamentally alters the algorithm's behavior, causing it to discard data rather than reorder it for processing.

Furthermore, the logical discrepancies extend to the pivot selection mechanism used to mitigate division-by-zero errors. The decompiled code inverted the conditional logic regarding row selection, prioritizing unstable configurations that the original code explicitly sought to avoid. Consequently, although the decompiled output remains syntactically parsable and visually similar to the original, the accumulation of these precise logic errors renders the code functionally inoperable. This outcome highlights a specific limitation in the decompilation process: the difficulty in accurately recovering precise mathematical syntax when distinct operators share high visual or statistical similarity.

6.3 DISCUSSION

The high success rate indicates that the hypothesis that modern LLMs can often reconstruct functionally correct Python code from bytecode, at least for the types of algorithmic functions evaluated in this study.

6.3.1 Analysis of Successes

A qualitative analysis of the successfully decompiled samples reveals compelling evidence of the model's ability to reconstruct high-level abstractions. By examining the output code in comparison to standard Pythonic practices, several key strengths emerge:

- **Idiomatic Code Structure:** The model consistently favored idiomatic Python constructs over direct, low-level translations of bytecode instructions. For instance, in functions like "`is_pangram`", the model correctly generated code using sets ("`set()`") and list comprehensions or string methods like "`replace`" and "`lower`". This demonstrates a clear understanding of functional composition and built-in methods, contrasting with the explicit loops that a traditional decompiler might infer from the bytecode's jump instructions.

- **Type Hinting and Modern Standards:** A notable observation is the consistent inclusion of type hints, such as in "`binary_search_by_recursion`", where arguments were typed (e.g., "`list[int]`") and return types specified ("`-> int`"). This suggests that the model draws upon its vast training on modern codebases to produce code that adheres to contemporary quality standards, enhancing the maintainability of the decompiled output.

- **Semantic Variable Naming:** In many cases, the model successfully inferred meaningful variable names based on the context of operations. In geometric functions like "`surface_area_cylinder`", variables were correctly named "`radius`" and "`height`", and mathematical constants were imported from the "`math`" module ("`pi`", "`sqrt`"), rather than being hardcoded or assigned generic identifiers (e.g., "`v1`", "`v2`").

- **Complex Control Flow:** The model successfully recovered complex algorithmic structures. In sorting algorithms like "`quick_select`", it accurately reconstructed recursive logic and partition helper functions. The ability to distinguish between recursive calls and iterative loops from bytecode highlights the model's capacity for higher-level pattern matching.

6.3.2 Error Patterns

A detailed examination of the failed decompilation attempts reveals a distinct failure mode that transcends simple syntactic inaccuracy. Contrary to the expectation of generating syntactically invalid or logically flawed code, the primary source of failure was the model's inability to produce any output in the final iteration of the refinement loop. In these instances, the system did not register an explicit API error or timeout; rather, the model returned an empty response, which was subsequently recorded as the decompiled artifact. This behavior remains to be fully understood and may indicate limitations in the model's capacity to handle certain bytecode patterns, complexities or token limits.

6.3.3 Practical Implications

The findings of this study have significant implications for the field of software reverse engineering and security analysis. The demonstrated ability of LLMs to effectively decompile bytecode into high-level source code opens new avenues for automated code recovery, vulnerability assessment, and legacy system maintenance. However, the identified limitations also underscore the need for cautious application of these models, particularly in scenarios where precision and correctness are paramount. Future work should focus on addressing the failure modes identified, potentially through enhanced context provisioning, model fine-tuning, or hybrid approaches that combine LLMs with traditional decompilation techniques. Additionally, exploring the integration of user feedback mechanisms could further refine the decompilation process and improve overall accuracy.

Hybrid approaches are a promising direction for future research, because they can leverage the strengths of both LLMs and traditional decompilation techniques. Solo LLM-based decompilation may struggle with certain bytecode patterns and are not deterministic, which can lead to inconsistent results. By combining LLMs with established decompilation algorithms, it may be possible to achieve higher accuracy and reliability.

6.3.4 Final Considerations

While the results demonstrate the promising potential of LLMs in decompilation tasks, they also underscore significant areas for improvement. Addressing these limitations is crucial for advancing the state of the art in automated code recovery and reverse engineering. Gathering additional metrics, such as token consumption per attempt and the frequency of database context queries, could provide deeper insights into the model's behavior and success rates. Future work should explore hybrid approaches, evaluate alternative models, and refine context retrieval mechanisms to enhance overall performance and reliability


7

7 FINAL CONSIDERATIONS

This work investigated the application of Large Language Models (LLMs) to the challenge of decompiling Python 3.13 bytecode, a task traditionally hampered by the loss of semantic information during compilation. By proposing an architecture that integrates open-source tools with a modern LLM, this research sought to determine if AI could not only restore syntax but also recover the functional semantics of the original code.

7.1 SYNTHESIS OF RESULTS

The experimental results strongly support the feasibility of this approach. With a success rate of nearly 92% (147 out of 160 samples), the system demonstrated that it is possible to automatically recover functionally equivalent source code for a wide range of algorithms. The use of a rigorous validation process, involving 25 unit tests per function, ensures that the success metric is meaningful and not just a measure of syntactic correctness.

The study confirms the hypothesis that LLMs can effectively support source code recovery. The average processing time of approximately 33 seconds per function highlights the potential for efficiency in automated workflows. This adaptability, combined with the ability to handle version-agnostic decompilation without rigid rule-sets, underscores the advantage of generative AI in software engineering, overall when compared to traditional decompilation tools that required continuous manual updates to handle new language features.

7.2 CONTRIBUTIONS

The main contributions of this work include:

- **A Novel Decompilation Architecture:** We presented a robust architecture that implements an iterative “generate-test-refine” loop, significantly improving the reliability of the output.
- **Methodology for Dataset Creation:** We established a scalable method for creating ground-truth datasets for decompilation research, utilizing LLMs to generate and validate comprehensive test suites.
- **Evaluation of Modern LLMs:** We provided empirical evidence of the capabilities

of modern LLMs in low-level code understanding tasks, contributing to the growing body of knowledge on AI for Software Engineering.

7.3 LIMITATIONS

Despite the promising results, this study has limitations:

- **Dataset Scope:** The evaluation was primarily focused on algorithmic functions from the "The Algorithms" repository. However, even within this scope, the model presented difficulties in handling complex mathematical calculations and recursive data structure manipulations. Additionally, the performance on large-scale codebases remains to be tested.
- **Failure Modes:** A distinct failure mode was observed where the model sometimes produced empty responses in the final refinement iteration, rather than explicit errors, indicating potential instability in handling certain edge cases.
- **Specific Python Version:** The study focused on Python 3.13. Bytecode changes between versions, and the model's adaptability to older or future bytecode formats was not evaluated.
- **LLMs:** The decompilation was performed using only one model. The performance of other models, including open-source alternatives, should be further explored.

7.4 FUTURE WORK

Future research could expand on this foundation by:

- **Fine-tuning Models:** Training or fine-tuning a smaller, open-source model specifically on pairs of (bytecode, source code) to reduce reliance on general-purpose models.
- **Cross-Version Compatibility:** Investigating the model's ability to decompile bytecode from different Python versions.
- **Complex System Decompilation:** Extending the architecture to handle entire modules or classes rather than isolated functions, addressing the challenge of inter-procedural dependencies.
- **Hybrid Approaches:** Investigating the combination of LLMs with traditional decompilation techniques to mitigate specific weaknesses, such as mathematical precision and structural consistency.

- **Try different models:** Evaluating the performance of various LLMs, including open-source models, to identify the most effective architectures for code decompilation tasks.

In conclusion, this work represents a significant step forward in reverse engineering, demonstrating that LLMs can potentially bridge the semantic gap in decompilation.

REFERENCES

References

[1] C. Cifuentes and K. J. Gough, “Decompilation of binary programs,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[2] D. Cotroneo, F. C. Grasso, R. Natella, and V. Orbinato, “Can neural decompilation assist vulnerability prediction on binary code?” in *The 18th European Workshop on Systems Security (EuroSec 25)*, 2025, pp. 1–8.

[3] X. She, Y. Zhao, and H. Wang, “Wadec: Decompiling webassembly using large language model,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 481–492.

[4] H. Tan, Q. Luo, J. Li, and Y. Zhang, “Llm4decompile: Decompiling binary code with large language models,” pp. 3473–3487, 2024.

[5] R. Kaichi, S. Matsumoto, and S. Kusumoto, “Automatic fixation of decompilation quirks using pre-trained language model,” in *International Conference on Product-Focused Software Process Improvement*. Springer, 2023, pp. 259–266.

[6] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, “DIRE: A neural approach to decompiled identifier naming,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[7] X. Shang, S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang, and N. Yu, “How far have we gone in binary code understanding using large language models,” *arXiv preprint arXiv:2404.09836*, 2024.

[8] D. S. Katz, J. Ruchti, and E. Schulte, “Using recurrent neural networks for decompilation,” in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 346–356.

[9] R. Bernstein *et al.*, “uncompyle6: A cross-version python bytecode decompiler,” GitHub repository, 2020. [Online]. Available: https://github.com/rocky/python-uncompyle6

[10] J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, “Hext5: Unified pre-training for stripped binary code information inference,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 774–786.

[11] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers principles, techniques & tools*. pearson Education, 2007.

[12] Python Software Foundation, “Glossary — python 3.12.3 documentation,” https://docs.python.org/3/glossary.html, 2025, acesso em: 22 maio 2025.

[13] B. Stalder, “Demystifying the cpython source code structure,” https://realpython.com/cpython-source-code-guide/, 2021, real Python, 2021. Acesso em: 22 maio 2025.

[14] Yale University, Department of Computer Science, “Python virtual machine (pvm),” n.d., acessado em 5 de junho de 2025. [Online]. Available: https://zoo.cs.yale.edu/classes/cs200/lectures/PVM.html

[15] B. D. Software, “Understanding python bytecode,” n.d., acessado em 5 de junho de 2025. [Online]. Available: https://www.blackduck.com/blog/understanding-python-bytecode.html

[16] C. Developers, “Cpython interpreter internals,” n.d., acessado em 5 de junho de 2025. [Online]. Available: https://github.com/python/cpython/blob/main/InternalDocs/interpreter.md

[17] Python Software Foundation, “What’s new in python 3.6,” 2016, acessado em: 06 dez. 2025. [Online]. Available: https://docs.python.org/3/whatsnew/3.6.html

[18] ——, “dis — disassembler for python bytecode,” 2025, acessado em: 06 dez. 2025. [Online]. Available: https://docs.python.org/3.13/library/dis.html

[19] R. S. Gunashree, “Guide to understanding python’s (ast) abstract syntax trees,” https://www.devzery.com/post/guide-to-understanding-python-s-ast-abstract-syntax-trees, Sep. 2024, Último acesso em 7 de junho de 2025.

[20] D. Cotroneo, F. C. Grasso, R. Natella, and V. Orbinato, “Can neural decompilation assist vulnerability prediction on binary code?” in *Proceedings of the 18th European Workshop on Systems Security*, 2025, pp. 26–32.

[21] R. Bernstein, “uncompyle6 3.9.2,” PyPI - The Python Package Index, 2024, acessado em: 8 de dezembro de 2025. [Online]. Available: https://pypi.org/project/uncompyle6/

[22] R. Bernstein, H. Goebel, and J. Aycock, “python-uncompyle6: A cross-version python bytecode decompiler,” https://github.com/rocky/python-uncompyle6, 2025, accessed: 2025-12-08.

[23] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2019.

[26] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, “Improving language understanding by generative pre-training,” 2018.

[27] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[28] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.

[29] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, “A neural probabilistic language model,” *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[30] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[31] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[32] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” *arXiv preprint arXiv:1910.13461*, 2019.

[33] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2022.

[34] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient finetuning of quantized llms,” *arXiv preprint arXiv:2305.14314*, 2023.

[35] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[36] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” *Advances in neural information processing systems*, vol. 30, 2017.

[37] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” *arXiv preprint arXiv:2307.03172*, 2023.

[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[39] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[40] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[41] Model Context Protocol Community, “Introduction to the model context protocol,” https://modelcontextprotocol.io/docs/getting-started/intro, 2024, accessed on September 22, 2025.

[42] X. Hou, Y. Zhao, S. Wang, and H. Wang, “Model context protocol (mcp): Landscape, security threats, and future research directions,” *arXiv preprint arXiv:2503.23278*, 2025.

[43] X. Shang, S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang, and N. Yu, “How far have we gone in binary code understanding using large language models,” in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 1–12.

[44] n8n, “n8n documentation,” https://docs.n8n.io/, n.d., accessed: 2025-12-08.

[45] ——, “n8n - fair-code workflow automation,” https://github.com/n8n-io/n8n, 2025, accessed: 2025-12-08.

[46] Google Cloud, “Gemini 2.5 flash,” https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash, 2025, accessed: 2025-12-08.

[47] Google AI for Developers, “Gemini api pricing,” https://ai.google.dev/gemini-api/docs/pricing, 2025, accessed: 2025-12-08.

[48] The PostgreSQL Global Development Group, “What is PostgreSQL?” https://www.postgresql.org/docs/current/intro-whatis.html, 2025, acesso em: 08 dez. 2025. [Online]. Available: https://www.postgresql.org/docs/current/intro-whatis.html

Appendix A - AST Text Representation

The following snippet shows the representation of the Abstract Syntax Tree (AST) for the code `print("Hello World")`, as generated by Python's `ast.parse` function.

```
<ast.Module object at 0x70c532ea7390>
{'body': [<ast.Expr object at 0x70c532ea6c10>],
'type_ignores': []}
children: [<ast.Expr object at 0x70c532ea6c10>]
<ast.Expr object at 0x70c532ea6c10>
{'value': <ast.Call object at 0x70c532ea4b90>,
'lineno': 1, 'col_offset': 0, 'end_lineno': 1,
'end_col_offset': 20}
children: [<ast.Call object at 0x70c532ea4b90>]
<ast.Call object at 0x70c532ea4b90>
{'func': <ast.Name object at 0x70c532ea6c50>,
'args': [<ast.Constant object at 0x70c532db5650>],
'keywords': [], 'lineno': 1, 'col_offset': 0, 'end_lineno': 1,
'end_col_offset': 20}
children: [<ast.Name object at 0x70c532ea6c50>,
<ast.Constant object at 0x70c532db5650>]
<ast.Name object at 0x70c532ea6c50>
{'id': 'print', 'ctx': <ast.Load object at 0x70c532d90410>,
'lineno': 1, 'col_offset': 0, 'end_lineno': 1, 'end_col_offset': 5}
children: [<ast.Load object at 0x70c532d90410>]
<ast.Constant object at 0x70c532db5650>
{'value': 'Hello World', 'kind': None, 'lineno': 1, 'col_offset': 6,
'end_lineno': 1, 'end_col_offset': 19}
children: []
<ast.Load object at 0x70c532d90410>
{}
children: []
```

Appendix B - Database Schema

The following snippet presents the Data Definition Language (DDL) SQL statements used to create the relational database schema for the proposed architecture. This schema supports the storage of code samples, analysis results, generated tests, and decompilation metrics.

```
-- Table to store random samples for testing
CREATE TABLE public.amostra_testes (
    id int8 NOT NULL PRIMARY KEY,
    caminho_arquivo text NULL,
    nome_funcao text NULL,
    codigo_fonte text NULL,
    bytecode text NULL,
    input_doctest text NULL,
    rotulos_llm jsonb NULL,
    testes_gerados_llm _text NULL
);

-- Main table storing the analyzed functions and metadata
CREATE TABLE public.analise_funcoes (
    id serial4 NOT NULL PRIMARY KEY,
    caminho_arquivo text NOT NULL,
    nome_funcao text NOT NULL,
    codigo_fonte text NOT NULL,
    bytecode text NOT NULL,
    input_doctest text NULL,
    doctest_args_str text NULL,
    assinatura_funcao text NULL,
    rotulos_llm text NULL
);

-- Table storing the results of decompilation attempts
CREATE TABLE public.decompilados (
    id bigserial NOT NULL PRIMARY KEY,
    modelo text NULL,
    contador int8 DEFAULT 0,
    limite_de_tentativas_atingido bool DEFAULT false,
    erro_na_resposta_do_modelo bool DEFAULT false,
    codigo_fonte_gerado text NULL,
    codigo_fonte_original text NULL,
    bytecode text NULL,
    testes_executados varchar NULL,
    tempo_de_execucao text NULL,
    gabarito text NULL,
    observacao text NULL
);
```

```
41
42 -- Table for storing ground truth from doctests
43 CREATE TABLE public.gabarito_doctest (
44     id serial4 NOT NULL PRIMARY KEY,
45     id_funcao int4 NOT NULL,
46     teste_doctest text NOT NULL,
47     gabarito_doctest text NULL,
48     gabarito_gerado text NULL,
49     gabaritos_iguais bool NULL,
50     erro_execucao text NULL
51 );
52
53 -- Table for storing generated unit tests
54 CREATE TABLE public.testes (
55     id serial4 NOT NULL PRIMARY KEY,
56     id_funcao int4 NOT NULL,
57     teste_gerado text NOT NULL,
58     resultado_execucao text NOT NULL,
59     modelo varchar(100) NULL,
60     saida_real text NULL,
61     log_erro text NULL
62 );
```

Appendix C - Experimental Prompts

C.1 Test Generation Prompt

```
Role: You are a Senior Python QA Engineer. Your goal is to generate a diverse
set of input arguments to stress-test a specific Python function.
Task: Analyze the provided Python function (source code and signature) and
generate a list of argument strings that can be passed directly to the
function.
Guidelines:
1. **Format**: Output strictly JSON with a single key "tests" containing a
list of strings.
2. **Content**: Each string must represent valid Python arguments (e.g.,
"1, 2", "'text', True", "key='value'").
3. **Efficiency**: Output ONLY valid JSON. No comments, no whitespace
padding.
4. **Coverage**:
- Generate **{num_testes_gerar} new, unique** test cases.
- Include: Happy path, Edge cases (empty strings, empty lists, 0,
negative numbers), Type boundaries (None).
5. **Restrictions**:
- Do NOT repeat the function name.
- Use only Python built-in types and literals (int, float, str, list,
dict, bool, None).
- Do NOT assume external variables exist.
- Do NOT repeat any arguments from the 'Existing Tests' section below.
- Do NOT include variables atributions in the arguments.
Target Function Information:
Name: `{nome_funcao}`
Signature: `{assinatura_funcao}`
{secao_testes_existentes}
Code:
```python
{codigo_da_funcao}
```

Generate the JSON object now:
```

C.2 Label Generation Prompt

```
You are an expert in Python code analysis. Analyze the function and output JSON
with a single key 'labels'.
The 'labels' should be a list of 1-5 concise categories (snake_case) describing
the function's purpose (e.g., "data_processing", "database_io",
"math_calculation").

Function: `{nome_funcao}`
Signature: `{assinatura_funcao}`
Code:
```python
{codigo_da_funcao}
```

Output JSON format: {{ "labels": ["label_one", "label_two"] }}
```

C.3 Decompilation Prompt

```
Subject: Python 3.13 Bytecode to Source Code Decompilation

Persona: You are an expert in Python bytecode reverse engineering,
with deep knowledge of the CPython VM internals, specifically for version
    3.13.

Core Task: Your task is to analyze the provided Python 3.13 bytecode below
and decompile it back into the original, human-readable, and semantically
equivalent Python source code.

Detailed Instructions:

1. Opcode Analysis: Examine the sequence of opcodes (bytecode instructions)
and their arguments. Reconstruct the program's logic by identifying:

- Control flow structures (for/while loops, if/else conditionals).
- Function (def) and class (class) definitions, including their scopes.
- Variable operations (assignment, loading, deletion).
- Function calls and attribute handling.
- Exception handling (try/except/finally).

2. Mandatory Knowledge Base Use: You *must* use your internal knowledge
    base,
using the tool "functions to get context" to select the PostgreSQL data,
which functions as a database of Python 3.13 code samples and their
    respective bytecodes,
to enhance the decompilation. Actively search this database for similar
    bytecode patterns
to infer the most probable and idiomatic source code structure. The tool
    must be used
by all meanings.

2.1. The columns in the table are:
  - id (bigserial)
  - caminho_arquivo (text)
  - nome_funcao (text)
  - codigo_fonte (text)
  - bytecode (bytea)
  - input_doctest (text)
  - rotulos_llm (text)
2.2. The table name is "analise_funcoes"
2.3. Use the following query to check the unique values from "rotulos_llm":
``` SQL
SELECT DISTINCT TRIM(label) AS label
```
```

```
FROM (
    SELECT unnest(string_to_array(rotulos_llm, ',')) AS label
    FROM analise_funcoes
) AS todas
WHERE TRIM(label) ILIKE '%data%'
ORDER BY label limit 15;
```
Where "data" is the term you want to search for the labels.
2.4. After you get the existing labels, search for the bytecode and source code
(codigo_fonte) in the table, for more context, using the following query:
``` SQL
SELECT codigo_fonte
FROM analise_funcoes
WHERE rotulos_llm ILIKE '%data%'
limit 15;
```
Where "data" is the label returned by the previous query.

3. Version Specificity: The decompilation logic must be strictly for Python 3.13,
accounting for any changes in opcodes or stack structure introduced in this version.

4. Output Format (Strict): Your response must contain *only* and exclusively the
decompiled Python source code. Do not include any explanations, greetings, or any text
that is not part of the source code. Do not wrap the code in anything, such as "``` python"
or things like that.

Bytecode to Decompile:
```
{{ $('amostra_testes').first().json.bytecode }}
```

{{ $if($('Run tests on decompiled source code').isExecuted, 'The result of the tests was:\n'+
$('Run tests on decompiled source code').first().json.stdout +
'\nAnd the tests were:\n' + $('cat testes').item.json.stdout || "", "")}}

Appendix D - Original Source Code Error Case

```
  def solve_simultaneous(equations: list[list]) -> list:
if len(equations) == 0:
    raise IndexError("solve_simultaneous() requires n lists of length n+1
 ")
_length = len(equations) + 1
if any(len(item) != _length for item in equations):
    raise IndexError("solve_simultaneous() requires n lists of length n+1
 ")
for row in equations:
    if any(not isinstance(column, (int, float)) for column in row):
        raise ValueError("solve_simultaneous() requires lists of integers
 ")
if len(equations) == 1:
    return [equations[0][-1] / equations[0][0]]
data_set = equations.copy()
if any(0 in row for row in data_set):
    temp_data = data_set.copy()
    full_row = []
    for row_index, row in enumerate(temp_data):
        if 0 not in row:
            full_row = data_set.pop(row_index)
            break
    if not full_row:
        raise ValueError("solve_simultaneous() requires at least 1 full
 equation")
    data_set.insert(0, full_row)
useable_form = data_set.copy()
simplified = simplify(useable_form)
simplified = simplified[::-1]
solutions: list = []
for row in simplified:
    current_solution = row[-1]
    if not solutions:
        if row[-2] == 0:
            solutions.append(0)
            continue
        solutions.append(current_solution / row[-2])
        continue
    temp_row = row.copy()[: len(row) - 1 :]
    while temp_row[0] == 0:
        temp_row.pop(0)
    if len(temp_row) == 0:
        solutions.append(0)
        continue
    temp_row = temp_row[1::]
```

```
        temp_row = temp_row[::-1]
        for column_index, column in enumerate(temp_row):
            current_solution -= column * solutions[column_index]
        solutions.append(current_solution)
    final = []
    for item in solutions:
        final.append(float(round(item, 5)))
    return final[::-1]
```

Algorithm 5: Original Function that was decompiled with Error

Appendix E - Decompiled Source Code Error Case

```
def solve_simultaneous(equations: list[list]) -> list:
    _length = None
    if len(equations) == 0:
        raise IndexError('solve_simultaneous() requires n lists of length n
    +1')

    _length = len(equations) + 1
    if any(len(item) != _length for item in equations):
        raise IndexError('solve_simultaneous() requires n lists of length n
    +1')

    for row in equations:
        if any(not isinstance(column, (int, float)) for column in row):
            raise ValueError('solve_simultaneous() requires lists of
    integers')

    if len(equations) == 1:
        return [equations[0][-1] / equations[0][0]]

    data_set = equations.copy()

    if any(0 not in row for row in data_set):
        temp_data = data_set.copy()
        full_row = []
        for row_index, row in enumerate(temp_data):
            if 0 in row:
                full_row = data_set.pop(row_index)
                break

        if not full_row:
            raise ValueError('solve_simultaneous() requires at least 1 full
     equation')
        data_set.insert(0, full_row)

    useable_form = data_set.copy()
    simplified = simplify(useable_form)
    simplified = simplified[:-1]

    solutions = []
    for row in simplified:
        current_solution = row[-1]
        if not solutions:
            if row[-2] == 0:
                solutions.append(0)
            else:
```

```
                solutions.append(current_solution / row[-2])
        else:
            temp_row = row.copy()[:-1]
            while temp_row[0] == 0 and len(temp_row) != 0:
                temp_row.pop(0)

            if len(temp_row) == 0:
                solutions.append(0)
            else:
                temp_row = temp_row[1:]
                temp_row = temp_row[:-1]
                for column_index, column in enumerate(temp_row):
                    current_solution -= column * solutions[column_index]
                solutions.append(current_solution)

    final = []
    for item in solutions:
        final.append(float(round(item, 5)))

    return final[::-1]
```

Algorithm 6: Decompiled Code with Error