Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Using LLMs to Rank Decompiled Code Variants

Luigi Timossi

Master's Thesis

Università di Genova, DIBRIS
Via Dodecaneso, 35 16146 Genova, Italy
https://www.dibris.unige.it/

Computer Science MSc
Computer Security and Engineering Curriculum

Using LLMs to Rank Decompiled Code Variants

Luigi Timossi

Advisors: Matteo Dell'Amico, Giovanni Lagorio
Examiner: Marina Ribaudo

February, 2026

Abstract

This space will be occupied by the abstract, a summary of all the things that i have done in this thesis

Table of Contents

Chapter 1

Introduction

Reverse engineering is a critical process in software security, enabling analysts to understand, debug, and modify software without access to its source code [CC90]. Decompilation tools like `Ghidra` and `Hex-Rays` have long been the backbone of this process, translating binary executables back into high-level code [Eag11; Nat19]. However, the output from these tools often suffers from issues such as poor readability, non-idiomatic constructs, and a lack of meaningful variable names, which can significantly hinder the analyst's ability to comprehend and work with the decompiled code.

The advent of large language model (LLM) has opened new avenues for enhancing reverse engineering workflows. LLM have demonstrated remarkable capabilities in understanding and generating code, making them promising candidates for improving the quality of decompiled output. Recent research has explored using LLM to refine decompiler output, generate comments, and even act as judges to evaluate code quality. However, much of this work has focused on either generative refinement or broad benchmarking of decompilers, often relying on proprietary models and tools [Tan+24; HLC24]. In this thesis, we take a different approach by leveraging LLM to evaluate the 'humanness' of decompiled code using intrinsic model metrics like `perplexity` [Hin+12]. We investigate how well local LLM can distinguish between different versions of the same codebase, such as `pull requests`, without modifying the code itself. This fine-grained analysis is crucial for assessing incremental changes in code quality and readability, which is often more relevant in real-world reverse engineering tasks than wholesale comparisons of different decompilers. Our work also addresses the practical constraints of reverse engineering, such as privacy and cost, by exploring the feasibility of running these evaluations on local hardware, rather than relying on cloud-based `APIs`. This makes our approach more accessible and applicable in security-sensitive contexts where data privacy is paramount [Sta+24; Car+21].

Chapter 2

Related Work

The intersection of LLM and reverse engineering has rapidly evolved, transforming how analysts interact with decompiled code. While recent literature has indeed explored utilizing LLMs to assist in reverse engineering (using framework like Model-Compute-Pairing (MCP) servers), the vast majority of this work focuses on two distinct areas:

- Generative refinement of decompiler output
- LLM-based evaluation and benchmarking of decompilation tools

Unlike generative approaches that aim to produce better code, our work focuses on measuring the 'humanness' of existing code using intrinsic model metrics like perplexity and using those LLM as a Judge 2.3. Furthermore, unlike broad benchmarking frameworks that rely on proprietary Application Programming Interface (API)s to rank tools, our research investigates the granular utility of local LLM in distinguishing between specific versions (or pull requests) of the same codebase.

2.1 Generative Refinement of Decompiler Output

The most prominent use of LLMs in this field is the attempt to improve the readability of the raw output produced by traditional decompilers (like Ghidra or Hex-Rays). This body of work is complementary to ours; while we do not attempt to modify the code, understanding the deficits of raw decompiler output explains why our metrics (such as perplexity) are necessary to quantify 'humanness'.

Some example of this approach are LLM4Decompile [Tan+24] wich is an LLM model that was trained to decompile binary code into high-level language, acting as a decompiler

itself (LLM4Decompile-End is the model to decompile, LLM4Decompile-Ref is the model to refine another decompiler output); or DeGPT [HLC24], which introduces an end-to-end framework designed to optimize decompiler output directly employing a 'three-role mechanism' (Referee, Advisor, and Operator) to guide an LLM in renaming variables, appending comments, and simplifying structure.

Their work demonstrates that LLMs can significantly reduce the cognitive burden on analysts by rewriting code to be more idiomatic.

2.2 LLM-based Benchmarking

Closer to our specific problem domain is the emerging field of using LLMs to evaluate code quality, using LLMs to scale and automatize human evaluation, this technique is often referred to as 'LLM-as-a-Judge.'

DecompileBench [Gao+25] is the state-of-the-art in this area, It presents a comprehensive framework for evaluating decompilers, introducing the concept of using 'LLM-as-a-Judge' to rate code understandability across 12 specific dimensions (e.g., Variable Naming, Control Flow Clarity). Their work validates that LLMs can align well with human experts in ranking different decompilers (e.g., comparing Ghidra vs. Hex-Rays vs. LLM-based decompilers). While DecompileBench is similar to our work in its use of LLMs for assessment, differ from ours on the type of validation used: DecompileBench relies only on prompting the model to output a score based on Function Source Code, Decompiler A's Pseudo Code, and Decompiler B's Pseudo Code to calculate an ELO rating. In contrast, our work leverages also with intrinsic model metrics like `perplexity` to quantify the 'surprise' of a specific model. Perplexity provides a more objective, quantifiable measure of how natural the code appears to the model, rather than relying only with subjective ratings 2.4.

Our work also diverges from DecompileBench in its focus on evaluating code variants within the same codebase (e.g., different pull requests), rather than comparing entirely different decompilers. This fine-grained analysis is crucial on cases where its required to assess incremental changes rather than wholesale tool comparisons (es. GitHub Actions [Git]).

The last difference is that DecompileBench heavily utilizes proprietary, closed-source models (like GPT-4 or Claude-3.5) and licensed decompilers (like Hex-Rays or Bininja). Our work specifically explores the feasibility of running these evaluations on local hardware. This addresses the privacy and cost constraints often present in security-sensitive reverse engineering tasks, which large-scale benchmarks often overlook.

The creation of the Dataset that we use is a subset of the one used in DecompileBench, since the entire dataset was overly large (at least more than 1 Tb) for our local hardware, and we needed to focus on a smaller subset of code variants to test the utility of perplexity

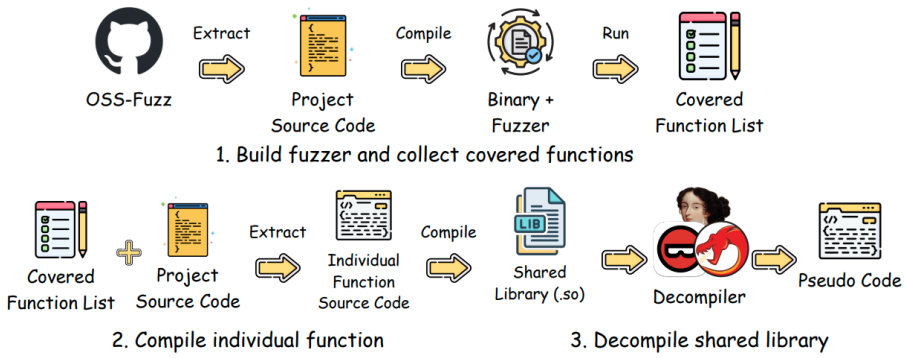

Figure 2.1: Creation of the dataset used in DecompileBench [Gao+25].

and LLM-as-a-Judge in a more controlled setting. we used the technique proposed by the DecompileBench team: extracting source code from OSS-Fuzz projects [Goo], identifying functions covered during execution using Clang's coverage sanitizer, and then extracting individual function implementations with their dependencies using `clang-extract`. These functions are compiled into standalone binaries (.so) with everything (binary position, funciton name, and source code) saved in a dataset object [pyt].

2.3 LLM-as-a-Judge

The 'LLM-as-a-Judge' [Li+25] is a technique that leverages the reasoning capabilities of LLMs to subsitute human evaluators in tasks that require subjective judgment. these aspects can be various (e.g., overall quality, logic, readability, etc.) and are often difficult to quantify with traditional metrics. In the context of reverse engineering, using an LLM as a judge allows us to evaluate the 'humanness' of decompiled code. Some problems with this approach are the potential for `bias` in the model's judgments, such as position bias (prefer always the first option), or lenght bias (prefer the longer option). These problems must be carefully managed in a study that relies on LLMs for evaluation.

2.4 Why Perplexity

If we accept that human source code is 'natural' and predictable, we can model it stochastically using neural (Transformer) language models. From this perspective, the decompiler acts as a noisy channel that introduces distortions into the original signal. The goal of LLM-based evaluation is to quantify how much the output signal (the decompiled code)

deviates from the expected statistical distribution of 'natural' human code [Hin+12]. A language model trained on human source code learns a probability distribution P over token sequences. When this model observes a sequence of decompiled code $S = t_1, t_2, \ldots, t_N$, it assigns a probability to each token based on the preceding context 3.8. If the decompiled code uses alien or 'unnatural' constructs, the model, expecting human patterns, will assign these tokens a very low probability. This statistical 'surprise' is the foundation of perplexity.

Chapter 3

Background

This chapter provides the necessary background knowledge to understand how Ghidra works, in particular the decompilation process, with his architecture, main components, and the decompilation pipeline. The Decompiler of Ghidra is enormous and complex software, (over 200k lines of C++) so it will focus only on the parts that are relevant to this thesis. then it will introduce the main concepts behind LLMs, their architecture, sampling, and metrics for evaluation.

3.1 Ghidra

Ghidra, released by the National Security Agency (NSA) in 2019, employs a bifurcated design that separates the user-facing interaction layer from the core analysis engine. This separation is not merely an implementation detail but a fundamental architectural constraint that dictates how data flows during the reverse engineering process.

The framework operates across two distinct memory spaces: a frontend implemented in Java and a backend analysis engine written in C++. The Java frontend is responsible for the Graphical User Interface (GUI), project database management, and plugin orchestration. It provides the high-level API exposed to users and scripts (e.g., Python or Java scripts via the GhidraScript framework). However, the computationally intensive tasks of data-flow analysis, variable inference, and control flow structuring are offloaded to a native C++ executable, typically named `decomp` or `decomp_dbg` (for debugging). These executables and the code are located at `Ghidra/Features/Decompiler/src/decompile/cpp`.

Communication is mediated by the `ghidra.app.decompiler.DecompInterface`. This interface manages a dedicated input/output stream to the native process, utilizing an XML-based protocol to exchange data. When a function decompilation is requested, the Java

client does not simply invoke a library function; it serializes the request into an XML command (e.g., `<decompile_at>`) and transmits it to the backend. The C++ process, holding its own representation of the function's data flow in `Funcdata` objects, performs the analysis and returns the results as a serialized XML stream describing the high-level code structure and syntax tokens.

3.2 SLEIGH and P-code

As written in the documentation created by running `<make doc>` [NSA] The decompiler provides its own Register Transfer Language (RTL), referred internally as p-code 3.1. The disassembly of processor specific machine-code languages, and subsequent translation into p-code, forms a major sub-system of the decompiler. There is a processor specification language, referred to as SLEIGH, which is dedicated to this translation task, this piece of the code can be built as a standalone binary translation library, for use by other applications.

3.2.1 P-code Semantics and Varnodes

Unlike intermediate languages in compilers, P-code is designed specifically for reverse engineering, prioritizing the explicit representation of memory and register modifications.

The fundamental unit of data in P-code is the *Varnode*. A Varnode is defined by the triple *(Space, Offset, Size)*, representing a contiguous sequence of bytes in a specific address space.

Table 3.1: Some P-code Operations and Semantics `opcodes.hh` [NSA; Pco]

Opcode	**Operands**	**Semantics**
`CPUI_COPY`	$in_0 \rightarrow out$	Copy one operand to another.
`CPUI_LOAD`	$space, ptr \rightarrow out$	Load from a pointer into a specific address.
`CPUI_STORE`	$space, ptr, val$	Store at a pointer into a specified address space.
`CPUI_INT_ADD`	$in_0, in_1 \rightarrow out$	Integer addition, signed or unsigned.
`CPUI_CBRANCH`	$dest, cond$	Conditional jump to *dest* if *cond* is non-zero.

We must distinguish between two forms of P-code used during analysis:

1. **Raw P-code:** The direct, unoptimized output of the SLEIGH translation. It is represented by the class *PcodeOpRaw* (or by unprocessed PcodeOp), and contains the bare essentials: an opcode, a sequence number (address), and the input/output Varnodes.

2. **High P-code:** The result of the analysis pipeline. In this form, the code has been converted to Static Single Assignment (SSA) form (a form where every varnode is defined exactly once for each function, if a variable is assigned multiple times, each assignment is given a new instance called low-level variable), dead code has been eliminated, and high-level concepts like function calls (replacing jump-and-link semantics) have been recovered. It is represented by the class *HighVariable*; this is an abstraction that groups multiple low-level Varnodes (which may reside in different registers or stack locations during execution) into a single logical variable, similar to a variable in C code.

The transformation from Raw to High P-code is where the majority of the decompilation logic resides. It is an inference process that attempts to raise the abstraction level of the code, often relying on heuristics that may fail in the presence of obfuscation or aggressive compiler optimizations.

3.3 The Decompilation Pipeline

The C++ decompiler engine processes a function at a time through a series of iterative passes. The architecture organizes these passes into *Actions* and *Rules*, managed by the `ActionDatabase`. inside the `ActionDatabase::universalAction` we have two main types of objects:

- `ActionGroup`: Represents a list of Actions that are applied sequentially. The group's properties (eg., rule_repeatapply) influence how the contained actions are executed.
- `ActionPool`: It is a pool of Rules that are applied simultaneously to every PcodeOp. Each Rule triggers on a specific localized data-flow configuration. The Rules are applied repeatedly until no Rule can make any additional transformations.

3.3.1 Actions and Rules

Actions represent large-scale transformations applied to the graph of varnodes and operations. They are the base class for objects that make modifications to a function's

(Funcdata) syntax tree. Their purpose is to manage complex stages of the workflow, such as recovering the control-flow structure or generating SSA form.

Rules, on the other hand, are a class designed to perform a single specific transformation on a PcodeOp or a Varnode. A Rule triggers when it recognizes a particular local configuration in the data flow and specifies a sequence of modification operations to transform it.

3.3.2 DefaultGroups

Actions and Rules are selected and activated according to the type of *DefaultGroup* they belong to. These groups represent standardized workflows for different analysis phases and are built by the method `ActionDatabase::buildDefaultGroups`. The main groups are:

- **decompile**: the standard workflow for full decompilation, composed of all of the phases.
- **jumptable**: optimized for analyzing jump tables.
- **normalize**: used for code normalization.
- **paramid**: for parameter identification.
- **register**: for register analysis.
- **firstpass**: a first fast analysis pass.

Each DefaultGroup is a list of names that refer to specific `ActionGroup`, `ActionPool` or individual `Action` to execute in that configuration. These lists define subsets of all the Actions.

The decompiler can be customized by selecting different DefaultGroups in java with the method `setSimplificationStyle` of the decompiler interface but Only the group named *decompile* return C code to ghidra, since in `ghidra_process.cc` we have:

Listing 3.1: ghidra_process.cc

```
[...]
  fd->encode(encoder,0,ghidra->getSendSyntaxTree());
  if (ghidra->getSendCCode()&&
 (ghidra->allacts.getCurrentName() == "decompile")) //HERE WE HAVE THE CHECK
    ghidra->print->docFunction(fd);
[...]
```

3.4 Logic of Control Flow Structuring

Recovering high-level control structures (loops, conditionals) from the unstructured Control Flow Graph (CFG) is arguably the most challenging phase of decompilation. It is effectively a pattern-matching problem on a directed graph, aimed at finding subgraphs that correspond to structured programming constructs.

3.4.1 Basic Block Formulation

The decompiler first aggregates P-code operations into *BasicBlocks* sequences of instructions with a single entry point and a single exit point (excluding internal calls). The CFG is formed by the edges representing jumps and branches between these blocks. Ghidra normalizes this graph to ensure a unique entry block, often inserting empty placeholder blocks to handle re-entrant loops or complex function entries.

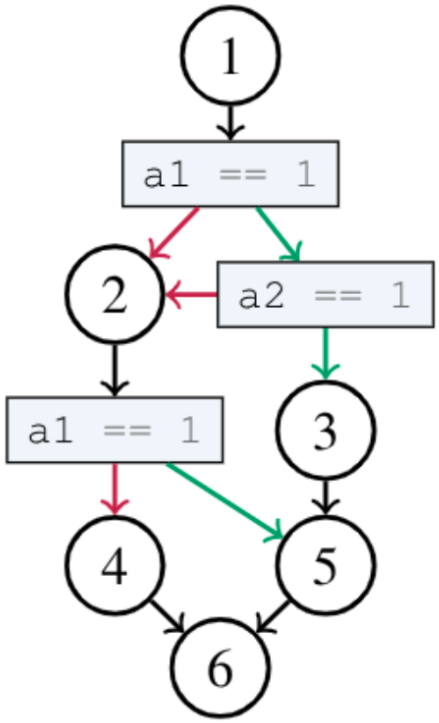

Figure 3.1: Control Flow Graph of a function

In this example we have the C code of the function described in 3.1 and its corresponding

P-code representation[1].

Source Code (C)	P-Code / Basic Blocks

`int a2_local;`
`int a1_local;`
`putchar(L'1');`
`if ((a1 == 1)`
	Basic Block 0
	`0x0010118d:1: RSP(0x0010118d:1) = RSP(i) + #0xfffffffffffffff8`
	`0x0010118d:2: *(ram,RSP(0x0010118d:1)) = RBP(i)`
	`0x00101195:d: u0x00004780(0x00101195:d) = RSP(i) + #0xfffffffffffffff4`
	`0x00101195:f: *(ram,u0x00004780(0x00101195:d)) = EDI(i)`
	`0x00101198:10: u0x00004780(0x00101198:10) = RSP(i) +`
	`#0xfffffffffffffff0`
	`0x00101198:12: *(ram,u0x00004780(0x00101198:10)) = ESI(i)`
	`0x001011a0:14: RSP(0x001011a0:14) = RSP(i) + #0xfffffffffffffe0`
	`0x001011a0:15: *(ram,RSP(0x001011a0:14)) = #0x1011a5`
	`0x001011a0:67: u0x10000008:1(0x001011a0:67) =`
	`*(ram,RSP(0x001011a0:14))`
	`0x001011a0:16: call`
	`jputchar(free)(#0x31:4,u0x10000008:1(0x001011a0:67))`
	`0x001011a5:17: u0x00004780(0x001011a5:17) = RSP(i) +`
	`#0xfffffffffffffff4`
	`0x001011a5:18: u0x00011e80:4(0x001011a5:18) =`
	`*(ram,u0x00004780(0x001011a5:17))`
	`0x001011a5:1e: ZF(0x001011a5:1e) = u0x00011e80:4(0x001011a5:18) ==`
	`#0x1:4`
	`0x001011a9:23: goto Block_2:0x001011bd if (ZF(0x001011a5:1e) != 0)`
	`else Block_1:0x001011ab`

`|| (a2 != 2)){`	**Basic Block 1**
	`0x001011ab:24: u0x00004780(0x001011ab:24) = RSP(i) +`
	`#0xfffffffffffffff0`
	`0x001011ab:25: u0x00011e80:4(0x001011ab:25) =`
	`*(ram,u0x00004780(0x001011ab:24))`
	`0x001011ab:2b: ZF(0x001011ab:2b) = u0x00011e80:4(0x001011ab:25) !=`
	`#0x2:4`
	`0x001011af:31: goto Block_2:0x001011bd if (ZF(0x001011ab:2b) != 0)`
	`else Block_4:0x001011b1`

[1]P-codes varies during all phases of the decompilation process; due to optimization rules, dead code elimination, and other transformations, the P-code shown here are taken from the `collapseInternal` method using `printRaw` of the FlowBlock class. The BasicBlock order may not correspond directly to the original source code order

Source Code (C)	P-Code / Basic Blocks

`putchar(L'2');`
`if (a1 != a2) {`
	Basic Block 2
	`0x001011c2:46: RSP(0x001011c2:46) = RSP(i) + #0xfffffffffffffe0`
	`0x001011c2:47: *(ram,RSP(0x001011c2:46)) = #0x1011c7`
	`0x001011c2:69: u0x10000011:1(0x001011c2:69) =`
	`  *(ram,RSP(0x001011c2:46))`
	`0x001011c2:48: call`
	`  jputchar(free)(#0x32:4,u0x10000011:1(0x001011c2:69))`
	`0x001011c7:49: u0x00004780(0x001011c7:49) = RSP(i) +`
	`  #0xffffffffffffff4`
	`0x001011c7:4a: u0x00011e80:4(0x001011c7:4a) =`
	`  *(ram,u0x00004780(0x001011c7:49))`
	`0x001011ca:4d: u0x00004780(0x001011ca:4d) = RSP(i) +`
	`  #0xfffffffffffffff0`
	`0x001011ca:4e: u0x00006a00:4(0x001011ca:4e) =`
	`  *(ram,u0x00004780(0x001011ca:4d))`
	`0x001011ca:54: ZF(0x001011ca:54) = u0x00011e80:4(0x001011c7:4a) ==`
	`  u0x00006a00:4(0x001011ca:4e)`
	`0x001011cd:59: goto Block_3:0x001011cf if (ZF(0x001011ca:54) == 0)`
	`  else Block_5:0x001011db`

`putchar(L'4');`
`goto LAB_001011e5;`
	Basic Block 3
	`0x001011d4:5b: RSP(0x001011d4:5b) = RSP(i) + #0xfffffffffffffe0`
	`0x001011d4:5c: *(ram,RSP(0x001011d4:5b)) = #0x1011d9`
	`0x001011d4:6b: u0x1000001a:1(0x001011d4:6b) =`
	`  *(ram,RSP(0x001011d4:5b))`
	`0x001011d4:5d: call`
	`  jputchar(free)(#0x34:4,u0x1000001a:1(0x001011d4:6b))`
	`0x001011d9:5e: goto Block_6:0x001011e5`

`} else {`
`  putchar(L'3');`
`}`
	Basic Block 4
	`0x001011b6:33: RSP(0x001011b6:33) = RSP(i) + #0xfffffffffffffe0`
	`0x001011b6:34: *(ram,RSP(0x001011b6:33)) = #0x1011bb`
	`0x001011b6:6d: u0x10000023:1(0x001011b6:6d) =`
	`  *(ram,RSP(0x001011b6:33))`
	`0x001011b6:35: call`
	`  jputchar(free)(#0x33:4,u0x10000023:1(0x001011b6:6d))`
	`0x001011bb:36: goto Block_5:0x001011db`

`putchar(L'5');`
`}`
	Basic Block 5
	`0x001011e0:38: RSP(0x001011e0:38) = RSP(i) + #0xfffffffffffffe0`
	`0x001011e0:39: *(ram,RSP(0x001011e0:38)) = #0x1011e5`
	`0x001011e0:6f: u0x1000002c:1(0x001011e0:6f) =`
	`  *(ram,RSP(0x001011e0:38))`
	`0x001011e0:3a: call`
	`  jputchar(free)(#0x35:4,u0x1000002c:1(0x001011e0:6f))`

Source Code (C)	P-Code / Basic Blocks

`1 LAB_001011e5:`
`2 putchar('6');`
`3 return;`
`4 }`
	Basic Block 6
	`0x001011ea:3c: RSP(0x001011ea:3c) = RSP(i) + #0xfffffffffffffff0`
	`0x001011ea:3d: *(ram,RSP(0x001011ea:3c)) = #0x1011ef`
	`0x001011ea:71: u0x10000035:1(0x001011ea:71) =`
	`*(ram,RSP(0x001011ea:3c))`
	`0x001011ea:3e: call`
	`jputchar(free)(#0x36:4,u0x10000035:1(0x001011ea:71))`
	`0x001011f1:44: return(#0x0)`

Basicblocks are created in `flow.cc` by the method `FlowInfo::splitBasic`. The routine partitions the P-code instruction stream at control-flow boundaries: conditional and unconditional jumps, call sites that alter control flow, and return instructions. Each such instruction ends the current block and/or starts a new one (targets of jumps also begin blocks).

The CFG with the BasicBlocks can also be seen in ghidra by entering the `Display Function Graph` window and enabling the P-code field in the layout of Instruction/Data (These Pcode are the final high-level Pcode). See figure 3.2

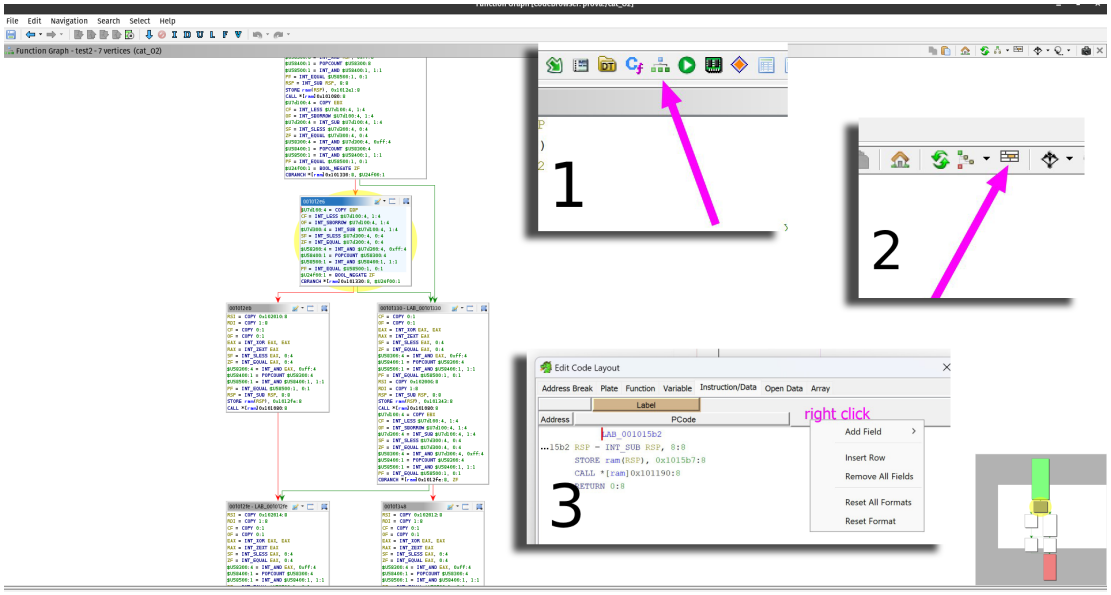

Figure 3.2: Control Flow Graph with P-code in Ghidra

3.4.2 The Structuring Algorithm

To transform the CFG into C statements, Ghidra employs a structuring algorithm implemented in the `ActionBlockStructure` class (an Action 3.3.1). The process involves identifying regions of the graph that match known schemas (or patterns) of control flow:

inside the `apply` method of `ActionBlockStructure` we have a call to `collapseAll` that is the main loop of the algorithm:

```
void CollapseStructure::collapseAll(void)
{
  int4 isolated_count;

  finaltrace = false;
  graph.clearVisitCount();
  orderLoopBodies();

  collapseConditions();

  isolated_count = collapseInternal((FlowBlock *)0);
  while(isolated_count < graph.getSize()) {
    FlowBlock *targetbl = selectGoto();
    isolated_count = collapseInternal(targetbl);
  }
}
```

The method implements a deterministic sequence of passes that progressively transform the BasicBlocks into structured FlowBlocks and performs the following steps:

1. **Preparation**
   The algorithm first clears previous visitation state (`graph.clearVisitCount`) and invokes `orderLoopBodies`. This pass discovers loop headers and back-edges, establishing a partial ordering among loop bodies. Detecting loops early is essential to prevent later structuring passes from erroneously breaking loop semantics.

2. **Conditional simplification**
   Next, `collapseConditions` attempts to simplify complex boolean logic and fold adjacent blocks that form logical AND/OR patterns (for example, transforming sequences that represent `if (A && B)` or `if (A || B)` into single conditional constructs). This phase applies local rules such as `ruleBlockOr` to reduce predicate complexity before higher-level structuring.

3. **Initial collapse**

The engine then calls `collapseInternal ((FlowBlock *)0)`, which scans the graph and applies standard structuring rules (e.g. `ruleBlockIfElse`, `ruleBlockWhileDo`, `ruleBlockSwitch`) to collapse perfectly structured regions. The routine returns an `isolated_count` indicating how many blocks have been fully resolved without introducing gotos.

4. **Unstructured flow handling**
   If the graph is not fully collapsed (`isolated_count < graph.getSize`), the method iterates: it selects a problematic edge with `selectGoto` and marks that edge as unstructured (to be emitted as a `goto`/`break`/`continue` in the final code). The selection is driven by heuristics to minimize disruption to surrounding structure. After marking the edge, `collapseInternal (targetbl)` is invoked again (often passing the target block of the newly created goto) so the structuring engine can resume collapsing other regions. This loop repeats until every block is resolved.

in the `collapseInternal` method we have the main pattern recognition method, some patterns have precedence over others, since it may occur that a region matches multiple schemas. For example, a `switch` may also match an `if-else` pattern.

These are the preferred patterns, in order:

- `goto`
- `cat` (block concatenation)
- `proper if` (if without else)
- `if-else`
- `while-do`
- `do-while`
- `infinite loop`
- `switch`

These 'rules' are implemented inside a loop that tries every pattern till no more changes are possible.

in the `ruleBlockWhileDo` method we can see how the pattern matching is done:

```
bool CollapseStructure::ruleBlockWhileDo(FlowBlock *bl)

{
  FlowBlock *clauseblock;
  int4 i;

  if (bl->sizeOut() != 2) return false; // Must be binary condition
  if (bl->isSwitchOut()) return false;
  if (bl->getOut(0) == bl) return false; // No loops at this point
  if (bl->getOut(1) == bl) return false;
  if (bl->isInteriorGotoTarget()) return false;
  if (bl->isGotoOut(0)) return false;
  if (bl->isGotoOut(1)) return false;
  for(i=0;i<2;++i) {
    clauseblock = bl->getOut(i);
    if (clauseblock->sizeIn() != 1) continue; // Nothing else must hit clause
    if (clauseblock->sizeOut() != 1) continue; // Only one way out of clause
    if (clauseblock->isSwitchOut()) continue;
    if (clauseblock->getOut(0) != bl) continue; // Clause must loop back to bl

    bool overflow = bl->isComplex(); // Check if we need to use overflow syntax
    if ((i==0)!=overflow) {      // clause must be true out of bl unless we use
        overflow syntax
      if (bl->negateCondition(true))
   dataflow_changecount += 1;
    }
    BlockWhileDo *newbl = graph.newBlockWhileDo(bl,clauseblock);
    if (overflow)
      newbl->setOverflowSyntax();
    return true;
  }
  return false;
}
```

Firstly is checked that the block has exactly two outgoing edges (a binary condition) and is not already part of a switch or a loop. Then, for each outgoing edge, it checks if the clauseblock (the potential loop body) has exactly one incoming edge (from the condition block) and one outgoing edge (back to the condition block). If these conditions are met, it confirms the presence of a while-do loop structure.

A condition is considered complex when the basic block that computes it contains too

many instructions to be cleanly represented within a single conditional expression. The method `BlockBasic::isComplex` performs this check.

Criteria: the algorithm counts the number of *statements* in the block:

- A conditional jump (branch) counts as 1 statement.
- `CALL` instructions count as 1.
- Operations that produce outputs used only inside the block or marker instructions do not count, but if a variable is used many times or is tied to memory, it contributes to the count.

If the total number of statements in the block exceeds 2, the block is considered complex.

The overflow syntax (`f_whiledo_overflow`) is a specific state assigned to a `BlockWhileDo` when its loop control condition is determined to be complex. It indicates that, although a logical `while` structure exists, the conditional block is too long or complicated to be emitted as a single boolean expression `while(condition){}`. Instead of printing `while(<complex condition>){}`, the decompiler emits an alternative form, typically an infinite loop with an internal `break` to preserve semantics.

After identifying a structure in the next iteration of the main loop in `collapseInternal`, a single FlowBlock representing the high-level construct (e.g., a `BlockWhileDo` for a while loop) is created. This new block encapsulates the original matched block, maintaining their internal P-code operations while providing a structured interface for further processing and eventual emission.

3.4.3 The *for* special case

As can be seen in section number 3.4.2, the Ghidra decompiler does not have an explicit rule to recognize `for` loops. Indeed, `for` loops in Ghidra are treated as special cases of `while-do` loops[2]: The check is performed in the method `BlockWhileDo::finalTransform`, this method proceeds only if the block is not marked with overflow syntax.

1. **Loop variable identification:** `findLoopVariable` is called to search for a variable controlling the iteration (e.g., `i` in `i < 10`). This variable must appear in the exit condition and be modified within the loop body.
2. **Initializer identification:** `findInitializer` searches for the instruction that initializes the variable (e.g., `i = 0`) in the block immediately preceding the loop.

[2]The transformation is triggered only if the architecture option `analyze_for_loops` is enabled.

3. **Relocation:** If both an iterator (`iterateOp`) and an initializer (`initializeOp`) are found, the decompiler physically moves the P-code operations (using `opUninsert` / `opInsertAfter`) so they lie adjacent to the loop boundaries, preparing them for syntactic emission.

4. **Non-printing marking:** In `finalizePrinting` these operations are marked with `opMarkNonPrinting`. This instructs the emitter not to print them as separate statements inside the body or before the loop, but to include them in the `for (...)` header.

```
void BlockWhileDo::finalTransform(Funcdata &data)
{
  // Simplification style
  BlockGraph::finalTransform(data);
  if (!data.getArch()->analyze_for_loops) return;
  if (hasOverflowSyntax())
      return; // Still too complex
  FlowBlock *copyBl = getFrontLeaf();
  if (copyBl == (FlowBlock *)0) return;
  BlockBasic *head = (BlockBasic *)copyBl->subBlock(0);
  if (head->getType() != t_basic) return;
  PcodeOp *lastOp = getBlock(1)->lastOp(); // There must be a last op in body,
      for there to be an iterator statement
  if (lastOp == (PcodeOp *)0) return;
  BlockBasic *tail = lastOp->getParent();
  if (tail->sizeOut() != 1) return;
  if (tail->getOut(0) != head) return;
  PcodeOp *cbranch = getBlock(0)->lastOp();
  if (cbranch == (PcodeOp *)0 || cbranch->code() != CPUI_CBRANCH) return;
  if (lastOp->isBranch()) {      // Convert lastOp to -point- iterateOp must
      appear after
    lastOp = lastOp->previousOp();
    if (lastOp == (PcodeOp *)0) return;
  }

  findLoopVariable(cbranch, head, tail, lastOp);
  if (iterateOp == (PcodeOp *)0) return;

  if (iterateOp != lastOp) {
    data.opUninsert(iterateOp);
    data.opInsertAfter(iterateOp, lastOp);
  }
```

```
  // Try to set up initializer statement
  lastOp = findInitializer(head, tail->getOutRevIndex(0));
  if (lastOp == (PcodeOp *)0) return;
  if (!initializeOp->isMoveable(lastOp)) {
    initializeOp = (PcodeOp *)0;  // Turn it off
    return;
  }
  if (initializeOp != lastOp) {
    data.opUninsert(initializeOp);
    data.opInsertAfter(initializeOp, lastOp);
  }
}
```

If all conditions are met, the decompiler effectively transforms the `while-do` structure into a `for` loop by relocating and marking the relevant P-code operations.

3.4.4 The *Goto* Problem

A significant limitation of this approach arises when the CFG contains irreducible control flow that does not match any predefined schema. (This is common in binaries optimized with aggressive compiler techniques or those containing manual assembly optimizations).

When `ActionBlockStructure` fails to find a matching pattern, the jump inside the FlowBlock remains and it will be represented as a *goto*[3]. statement to preserve semantic correctness, this phenomenon significantly degrades the readability of the output.

3.5 Code Emission

The final phase of the pipeline is the translation of the structured High P-code into C syntax. This is not a simple text dump but a structured generation of an Abstract Syntax Tree (AST) represented by `ClangToken` objects.

Before emission, the *ActionNameVars* pass attempts to assign meaningful names to the recovered `HighVariable` objects. If debug symbols (DWARF, PDB) are available, they are utilized. In their absence, Ghidra relies on heuristics based on variable usage (e.g., loop counters named `i`, `j`) or storage location (e.g., `iVar1`, `uVar2`). This process is highly stochastic and often results in generic, non-descriptive identifiers.

[3]Or a `break`/`continue` if it jumps out of/into a loop structure

The C++ backend generates a stream of `ClangToken` objects representing the code structure. This tokenized representation is sent to the Java frontend via the XML protocol. This structured data allows the Ghidra GUI to provide interactive features—such as cross-referencing and dynamic renaming—since the UI elements remain linked to the underlying `Varnode` and `HighVariable` objects.

3.6 Large Language Models

The advent of LLM marks a fundamental discontinuity in the history of artificial intelligence and Natural Language Processing (NLP). It is not merely an increase in computational capacity, but an ontological redefinition of how machines process, represent, and generate semantic information. At the heart of this revolution lies the Transformer architecture, introduced in 2017 by Vaswani [Vas+23], which enabled overcoming the sequential limitations of previous Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) architectures.

3.6.1 Transformer

The shift from recurrent architectures to the Transformer was motivated by the need for parallelization and the handling of long-range dependencies. Whereas RNNs processed tokens sequentially $(t_1, t_2, \ldots, t_n)$, accumulating error and dispersing the gradient over long sequences, the Transformer processes the entire sequence simultaneously, relying entirely on the attention mechanism.

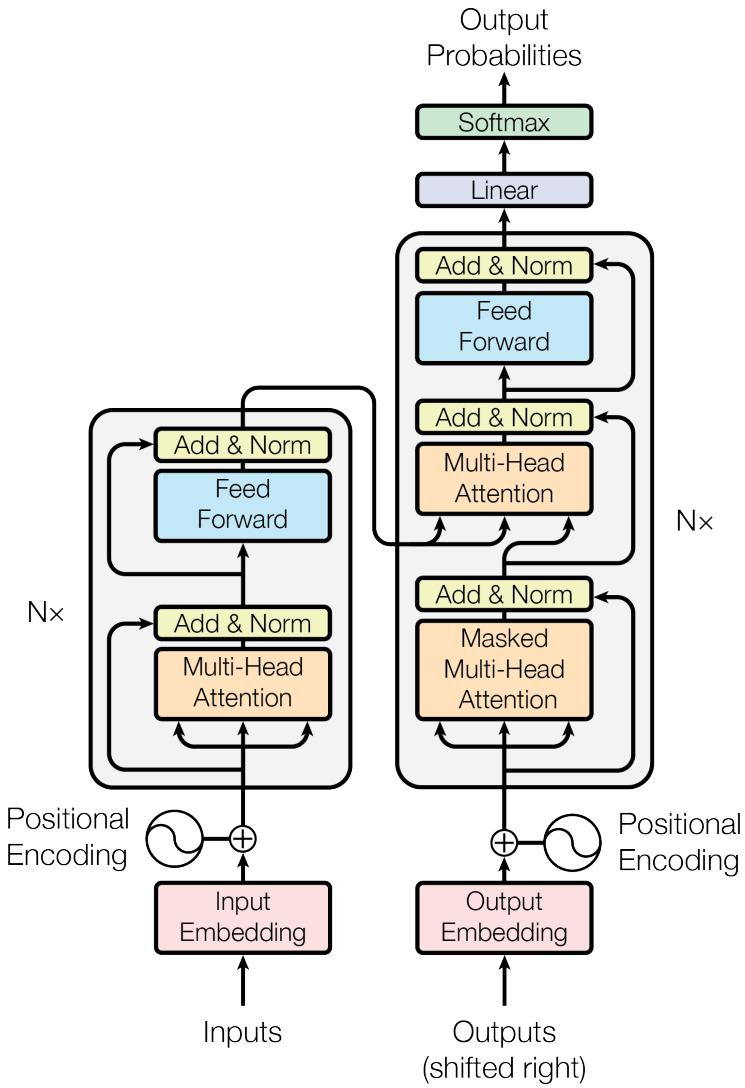

Figure 3.3: The Transformer model architecture

As shown in figure 3.3, the Transformer architecture consists of an encoder-decoder structure, where both components are composed of multiple layers of self-attention mechanisms and feed-forward neural networks.

The **encoder** (left) ingests a token sequence $x = (x_1, \ldots, x_n)$ and produces continuous representations $z = (z_1, \ldots, z_n)$. The **decoder** (right) conditions on z and generates the output tokens $y = (y_1, \ldots, y_m)$ autoregressively, emitting one token per step. Both sides are built by stacking identical blocks composed of Multi-Head Attention and position-wise Feed-Forward layers, typically wrapped with residual connections and normalization. Inputs and targets are first embedded into a n-dimensional space, and a positional encoding

is added to each embedding to encode token order [Vas+23].

At the core there is Multi-Head Attention, which allows the model to jointly attend to information from different representation subspaces at different positions. Each attention head computes scaled dot-product attention, enabling the model to focus on relevant parts of the input sequence when generating each output token.

Most of the state of art LLMs use an architecture with only the decoder part, omitting the encoder entirely [Bai]. This design choice is preferred for its simplicity, its good zero-shot generalization, and cheaper training cost to attain a reasonable performance.

3.6.2 Tokens

In LLMs, text is processed in chunks called **tokens**. A token can represent a word, a subword, or even a single character, depending on the tokenization scheme used. The choice of tokenization method significantly impacts the model's performance, as it affects how the model interprets and generates text. [Mul+18]

We can see an example using tiktokenizer[4], a webtool for visualizing tokenization for differents models, to tokenize a sentence:

[4] https://tiktokenizer.vercel.app/

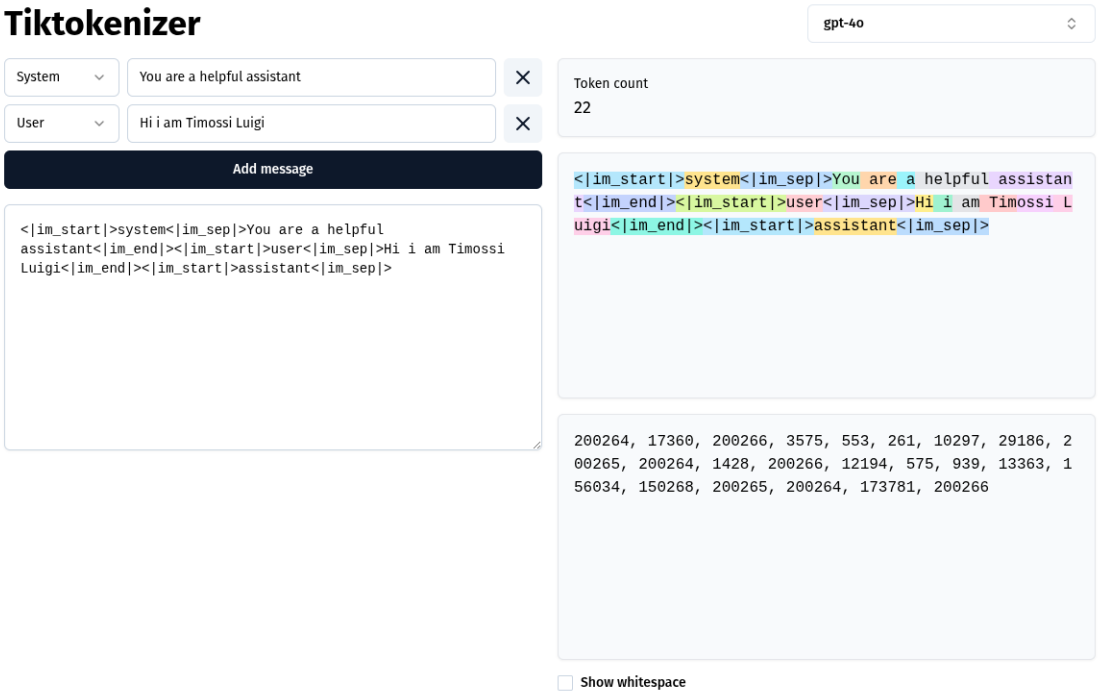

Figure 3.4: Tokenization example using tiktokenizer

As shown in figure 3.4, the sentence 'Hi i am Timossi Luigi' is tokenized into a sequence of tokens. Each token corresponds to a specific integer ID in the model's vocabulary.

For this example, for the model `gpt-4o` the token ' i' corresponds to the ID 575. The tokenization process is crucial for LLMs, as it transforms raw text into a format that the model can process. Different models use different tokenization values or schemes like Byte Pair Encoding (BPE).

In the tokens we count also the special tokens like $< |im\_start| >$ that indicates boundaries for roles messages like system, assistant, or user. in this case the token $< |im\_start| >$ corresponds to 'input message start' after that we have the role then the token 'input message separator' the message and finally the token 'input message end'. different models use different special tokens or does not use them at all.

3.6.3 Softmax

The `Softmax` function is used as an output function in the last layer of a neural networks to transform a vector of real values into a probability distribution. (see figure 3.3) This

function for each component of the vector it computes the exponential normalized by the sum of the exponentials of all components, producing in output a vector of the same dimension with values in the interval [0,1] whose sum is 1. [Zvo]

$$softmax(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}$$

where $y = (y_1, y_2, .., y_n)$ is an input vector and values $y_i, (i = \overline{1, n})$ are in range from $-\infty$ to $+\infty$.

3.6.4 Quantization

Quantization is a technique used to reduce the memory footprint and computational requirements of neural networks by representing weights and activations with lower precision. In the context of LLMs, quantization can be applied to the model's parameters (weights) and activations (intermediate outputs) to enable faster inference and reduce memory usage, especially on resource-constrained devices. [Dev; Fac] There are different quantization schemes, such as:

- **Post-training quantization**: This method quantizes a pre-trained model without requiring additional training. It can be applied to both weights and activations, but it may lead to a drop in model accuracy if not done carefully.
- **Quantization-aware training**: This method incorporates quantization into the training process, allowing the model to learn to compensate for the reduced precision. This approach typically results in better accuracy compared to post-training quantization.

Quantization can significantly reduce the computational requirements of LLMs, enabling faster inference and making it feasible to deploy large models on edge devices or in real-time applications. However, it is important to carefully evaluate the impact of quantization on model performance, as aggressive quantization can lead to a significant drop in accuracy.

3.7 Decoding

When the model generates text, it produces a vector of raw scores, called *logits*, for each token in the vocabulary at each timestep. These logits represent the unnormalized likelihood of each token being the next token in the sequence. By applying the `Softmax` function

to these logits, the model obtains a probability distribution over the vocabulary; Once the probability distribution is computed, the model must select the next token. This process, called decoding, can be influenced by different strategies such as:[IBM]

- **Greedy decoding**: always select the token with the highest probability, produces output that closely matches the most common language in the model's pretraining data and in your prompt text, which is desirable in less creative or fact-based use cases. This can cause the model to produce repetitive or generic output.
- **Sampling decoding**: the model chooses a subset of tokens, and then one token is chosen randomly from this subset to be added to the output text. Sampling adds variability and randomness to the decoding process, which can be desirable in creative use cases. This can cause the model to produce unexpected or incorrect output

3.7.1 Temperature

Temperature (T) is a hyperparameter that acts directly on the `Softmax` function. The `softmax` function with temperature becomes:

$$softmax(y_i) = \frac{e^{(\frac{y_i}{T})}}{\sum_{j=1}^{n} e^{(\frac{y_j}{T})}}$$

where $T > 0$ is the temperature parameter. The temperature modifies the distribution of probabilities over the tokens:

- $T < 1$ (Cooling): Differences between logits are amplified. The token with the highest logit receives a probability close to 1. The distribution becomes 'peaked', reducing variety and increasing determinism. Useful for logical or mathematical tasks.
- $T > 1$ (Heating): Differences are flattened. The distribution tends toward uniformity. Tokens with lower logits gain probability mass, increasing 'creativity' but also the risk of incoherence (hallucinations).
- $T \to 0$: Equivalent to the `Greedy decoding`, where always the single most probable token is chosen.

Miklos and Rebeka (both Junior Research) [Mik], in their study on the impact of temperature on text generation have tested different temperature settings using OpenAI GPT-4.1 with the prompt `Why do researchers use control groups in experiments?`; They send two times the same prompt with different temperature settings and analyzed the outputs:

- At $T = 0.1$, the output was identical for both runs and both answers offered a clear, textbook-style explanation
- At $T = 1.4$, the outputs varied between runs, providing more expressive, creative answers with illustrative examples.
- At $T = 2$, the outputs became incoherent and nonsensical with grammatical errors, illogical statements, and gibberish characters.

These results highlight how much temperature settings influence the balance between coherence and creativity in LLM outputs.

Even with an optimal temperature, the long tail of the distribution (thousands of tokens with infinitesimal but nonzero probability) can introduce errors if sampled. To mitigate this, techniques like `Top-k` and `Top-p` sampling are employed.

3.7.2 Top-p and Top-k

Top-p (nucleus) sampling is a stochastic decoding strategy that at each generation step restricts sampling to the smallest subset of tokens whose cumulative probability is at least a threshold p. [wikb].

Formally, given vocabulary V and context $x_{<t}$, the nucleus $V^{(p)}$ is the minimal subset satisfying

$$V^{(p)} \subseteq V, \qquad \sum_{x \in V^{(p)}} P(x \mid x_{<t}) \geq p.$$

Tokens outside $V^{(p)}$ are assigned zero probability; probabilities inside the nucleus are renormalized

Top-k sampling limits the candidate tokens to the k most probable ones at each step, nucleus sampling dynamically adjusts the candidate set based on the cumulative probability threshold p.

The combined use of Temperature (to model the shape of the curve) and Top-p (to intelligently truncate the tail) represents the current industry standard for high-quality text generation.

3.8 Perplexity

Evaluating the quality of an LLM is intrinsically challenging because language judgments are often subjective. Nonetheless, there are rigorous quantitative metrics. `Perplexity` is the standard measure used during LLM pre-training; it stems from information theory and quantifies the model's uncertainty when predicting the next token. [wika]

Mathematically, `perplexity` is the exponential of the average negative log-likelihood (i.e., the exponentiated cross-entropy) of the predicted tokens. Exponentiating the cross-entropy restores the measure to probability-like units, yielding an intuitive 'effective branching factor' the average number of plausible next-token choices the model considers. [Mor]

$$\mathrm{PPL}(X) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(x_i \mid x_{<i})\right)$$

A `perplexity` of K indicates that, on average, the model behaves as if it were choosing among K equally likely alternatives at each prediction step. [Sax]

- Relation to entropy: $\mathrm{PPL} = 2^{H(P)}$, where $H(P)$ denotes the Shannon entropy of the distribution.
- Interpretation: Lower `perplexity` means the model assigns higher probability mass to the true tokens from the test set.

Note that a low `perplexity` reflects statistical predictability relative to the training corpus and does not guarantee factual accuracy or correctness.

3.9 Human-like

LLM are designed to generate text that closely mimics human language, capturing nuances, context, and stylistic elements. This capability is achieved through extensive training on vast corpora of text, enabling the models to learn patterns and structures inherent in human communication. This mimic of human-like text generation has profound implications across various domains, including customer service, content creation, and education. since the generated text is often indistinguishable from that written by humans, LLM we can use them to measure how 'human-like' is a piece of text. If we have two C functions that perform the same task but generated with different decompilers (or different settings/version of the same decompiler), we can probably assume that using an LLM to measure how 'human-like' is the generated code can be a good proxy for code quality/readability.

Here we have two main path to measure the human-likeness of a piece of code:

- Ask the LLM directly via prompt to rate the human-likeness of the code. This approach can lead to subjective and inconsistent results, as the model's responses may vary based on the prompt phrasing, the context, and hallucinations.
- Use `perplexity` as a quantitative metric to evaluate the human-likeness of the code. This approach leverages the statistical properties of the language model to assess how well the generated code aligns with patterns learned from human-written code.

As shown in section number 3.8, `perplexity` measures how well a language model predicts a sequence of tokens. A lower `perplexity` indicates that the model finds the sequence more predictable, which often correlates with human-like text. By calculating the `perplexity` of code snippets generated by different decompilers, we can objectively compare their human-likeness. For this reason, `perplexity` is used only to evaluate the Human-like quality of the decompiled code, it **does not** evaluate its functional correctness.

Chapter 4

Methodology

The framework of our work is based on a client-server architecture, where the server hosts the LLM and provides an API for interacting with it, while the client is responsible for building specific Ghidra version, preparing the code samples and prompts, invoking the server, and collecting results in JavaScript Object Notation (JSON) format. Every service is designed to be modular, allowing for the integration of different LLM models and evaluation metrics, for reproducibility we used *Docker Compose* to containerize both the server and client components, ensuring consistent environments across different machines and operating systems. The dataset creation is also a containerized process, and the result are mounted as volumes to the client container, allowing for easy access and manipulation of the data without the need for complex data transfer mechanisms.

4.1 Dataset Maker

As written on related works 2.2, we decided to use a subsect of the complete OSS Fuzz dataset used by DecompileBench [Gao+25] for our evaluation, specifically the four Open Source projects: `file, libxls, readstat, xz`, which are written in C and have a rich history of commits and pull requests on GitHub. These choice was motivated by the need to have a manageable dataset size for local evaluation, while still covering a set of real world code and functions to evaluate our approach. for every project the dataset maker extracts all the functions and recompiles them into standalone binaries; this process create different optimization levels of the binary, specifically `-O0` and `-O2`, and `-O3` which are the most common optimization levels used in real world scenarios, and which can have a significant impact on the decompilation output and its readability.

For obtaining this we had to fork the original dataset maker script and modify it to fit

our edits; such as the specifically optimization levels (in the original repo they were using all the optimization levels) and some bug fix as pointed out by one pull request on the original repo [edm]. so we clone our fork into a container (wich will also run docker inside for building the projects), edited with the patches as shown in the `README` file of DecompileBench, selected just our four projects and then run the dataset maker script.

4.1.1 Dataset Collection

The result of the dataset maker is a folder named `Dataset` wich contains other three subfolders:

- **binary**: contains the compiled binary of the functions, every file is named with the format `task_project_functionName-OX.so`, and can be used for decompilation and evaluation.
- **compiled_ds**: contains a file structure of the dataset format used by the `Datasets` library [pyt], which is a Python library for handling large datasets in a efficient way, and which we use for loading the dataset in our client code. This structure have a file '.arrow' that store data and two JSON files for the metadata such as field names and types. In our case we are interested only in three fields `file`, which contain the name of the function, `path` wich contains the path 'binary/namefile', and `func` wich contains the source code of the function.
- **eval**: contains also a dataset structure, but we will not use it since is used for recompile success and other metrics that we are not interested in, since we want to focus on the evaluation of the decompilation output rather than the compilation process.

4.2 LLM Server

The server is responsible for hosting the LLM and providing an API for interacting with it, specifically for receiving code samples and prompts from the client, processing them with the LLM, and returning the results. The server is designed to be modular, allowing for the integration of different LLM models and evaluation metrics, and it is containerized using Docker for reproducibility and ease of deployment.

It uses Gnunicorn as the WSGI HTTP server for handling incoming requests, and it is built on top of a Python web framework (Flask) to define the API endpoints and handle the logic for processing requests and interacting with the LLM. The server is configured

with a single worker and thread to manage sequential requests with an extended timeout of 800 seconds to accommodate longer inference times (but more importantly the model loading and the context switch when changing models). On startup, it performs Graphics Processing Unit (GPU) availability checks and pre-downloads all required model weights to the container's cache directory using the Hugging Face libraries.

4.2.1 Models

The heavy part of the framework is without doubt the server, and the models that runs on it. In our case the local enviroment is a single GPU machine with 16 GB of Video Random Access Memory (VRAM), so we had to select models that can run on this hardware, and that can provide a good performance for our evaluation. We also used the VRAM Calculator[1] to estimate the memory requirements of different models and ensure they fit within our hardware constraints, inside the VRAM have to cohesist different areas, such as:

- **Base Model Weights**: The trained parameters of the model, the 'weights' with their precision (could be quantizated for reduced memory usage).
- **Activations**: Intermediate computation results during forward passes through the layers. This grows with batch size and input length, and is critical for stability during inference.
- **KV Cache**: Key-Value cache used to avoid recomputing attention for previously processed tokens. Given the lengthy decompilation prompts containing source code, this cache grows proportionally with input length.
- **Framework Overhead**: Fixed memory cost from PyTorch, CUDA drivers, and buffer management. This overhead exists regardless of model size.

Based on these considerations, we selected the following models for our evaluation:

- **Meta Llama 3.1 (8B)**:
- **Qwen2.5-Coder-Instruct (7B)**:
- **DeepSeek-R1-Distill-Qwen (7B)**:
- **Google Gemma 2 (9B)**:

  ...

[1] https://apxml.com/tools/VRAM-calculator

These models were chosen for their balance between performance and memory requirements and for their close number of parameters, allowing us to run them on our local hardware while still providing meaningful insights into the evaluation of decompilation output.

4.2.2 Configuration

The server supports multiple local LLMs through a simple configuration layer that maps a short, client-facing identifier to the corresponding Hugging Face repository ID. Concretely, a dictionary (`MODELS_CONFIG`) defines the available models (e.g., `qwen-coder`, `deepseek-r1`, `llama3.1`, `gemma2`) and is the single source of truth for both the `/models` endpoint and for request-time model switching.

For ensure lightness, all models are loaded using 4-bit quantization via `bitsandbytes`.

```
bnb_config = BitsAndBytesConfig(
  load_in_4bit=True,
  bnb_4bit_quant_type="nf4",
  bnb_4bit_use_double_quant=True,
  bnb_4bit_compute_dtype=torch.bfloat16
)
```

To reduce cold-start delays, the container optionally pre-downloads all model snapshots at startup using `snapshot_download`, ensuring that evaluation runs are not affected by network variability. Finally, since only one model can reside in GPU memory at a time, the server unloads the currently active model (explicit `del` + garbage collection + CUDA cache cleanup) before loading a new one.

4.2.3 Decoding strategy (temperature and top-p)

For the `/generate` endpoint, we configured the decoding parameters through a dedicated function that returns a dictionary of `transformers` generation arguments. In our experiments we rely on *nucleus sampling* (`top_p`) with a low `temperature`, to balance determinism (useful for fair comparisons across decompilers) and the ability to escape repetitive or low-quality completions.

Concretely, the default configuration is:

- `temperature=0.4`: reduces randomness by sharpening the token distribution. Lower values make outputs more stable across runs, which is desirable for evaluation.

- `top_p=0.9`: nucleus sampling, i.e., tokens are sampled only from the smallest set whose cumulative probability mass is p. This prevents the model from selecting very unlikely tokens while still allowing variation.
- `max_new_tokens=2048`: upper bound on completion length, used as a safety and latency-control measure.

4.2.4 Routes

The server exposes a minimal Representational State Transfer (REST) API, All endpoints exchange JSON payloads and are intentionally kept coarse-grained (a lot of work in a single request) to decouple the client implementation from model-specific details. The available routes are:

- **GET /**: health check endpoint. It returns the server readiness status, whether CUDA is available, and the currently loaded model identifier (if any). This is used by docker compose to ensure healthcheck status for required services.
- **GET /models**: returns the list of supported model keys (the abstract identifiers used by the client), mapped server-side to Hugging Face repository IDs.
- **POST /generate**: main inference endpoint. The request body includes `model_id` and a `prompt`. The server loads (or switches to) the requested model, wraps the prompt into a chat-style template via the tokenizer, runs text generation, and returns the generated completion.
- **POST /score**: scoring endpoint used to compute a language-model based score for a given text. The request body includes `model_id` and `text`. The server computes the token-level negative log-likelihood and returns the derived perplexity.
- **POST /free**: explicit cleanup endpoint to unload the currently resident model and aggressively release GPU memory.

Since different models cannot fit simultaneously in GPU memory, model switching is handled server-side: each request triggers a check on the currently loaded model and, if needed, a full unload/load cycle. To avoid concurrent access to GPU state, all inference and scoring operations are protected by a global lock, enforcing sequential execution.

4.2.5 Metrics

To make the evaluation reproducible and to quantify server-side overhead, the server logs per-request performance metrics to a CSV file (`llm_metrics.csv`). Each entry includes:

- **Model and operation**: `model_id` and `operation` (`generate` or `score`).
- **Latency**: wall-clock duration (seconds) measured around the full operation, including tokenization and GPU synchronization.
- **Peak GPU memory**: peak VRAM allocated during the operation, obtained via CUDA peak memory statistics.
- **Tokens**: number of prompt/input tokens and number of generated output tokens; these are also used to derive an approximate throughput (tokens per second).

Metric collection is implemented via a dedicated monitoring context manager that resets CUDA peak counters before execution and synchronizes the device before reading final statistics. This design provides a uniform measurement procedure across both generation and perplexity scoring, and enables later analysis of the impact of model switching, prompt length, and decoding configuration on runtime and memory usage.

4.3 Client

The client is responsible for orchestrating the entire evaluation workflow, including building specific Ghidra versions, preparing code samples and prompts, invoking the server for decompilation and scoring, and collecting results in JSON format for analysis. It is designed to be modular and flexible, allowing for easy integration of different evaluation strategies and metrics, and it is containerized using Docker for reproducibility and ease of deployment.

4.3.1 Building Ghidra

The build process is automated via Python scripts that interact with Git and Gradle (we use an ubuntu image for the container). Firstly we clone and build the Ghidra repository from GitHub, this version is used as the base for all our evaluations. after building base and extracted the functions from binary, we get the Pull Request (PR)s number that we want to evaluate against base from a function that calls github API and returns the list of all PRs of Ghidra. Then for each PR we have to checkout the specific version of Ghidra, for doing this we have a script that takes as input the PR number, and then it fetches the

specific head reference from the GitHub repository (`pull/ID/head:pr-ID`) and checks it out.

For building Ghidra are necessary two prerequisites: `Java 11` and `Gradle` (optionally), the first one is required for running the build scripts and the second one is used for managing dependencies and building the project, but since Ghidra in newer versions includes a wrapper for Gradle (`gradlew`), you can use it without installing Gradle globally.

One problem is that every version of Ghidra need a specific version of Java, so we have to check the `application.properties` file inside the repo for the required minimal Java version, and then install it in the container before building Ghidra. So inside the container we manage more than one version of Java, and we switch between them based on the requirements of the Ghidra version we are building. Another problem is that some PRs are based on older versions of Ghidra wich does not have the gradle wrapper, so for building those versions we have to do the same thing we have done with Java, but for Gradle, we have to install more than one version of Gradle and switch between them based on the requirements of the Ghidra version we are building; This only if `gradlew` is not available since is more preferible running it instead. This is the main reason for using an Ubuntu image for the container, since it allows us to easily manage multiple versions of Java and Gradle using the package manager and environment variables.

After building a specific version of Ghidra (Base or PR), for every binary found in the dataset folder, we check if it is not already decompiled by that specific version of Ghidra (i.e., if the corresponding JSON file with the decompilation output does not exist), after creating the list of the files not yet decompiled, we start the decompilation process.

4.3.2 Ghidra Headless

[Ghi]

4.3.3 Evaluation

...

4.3.4 Abstraction and Anonymization

To evaluate the structural quality of the decompilation independently of variable naming and formatting, we implemented an abstraction mechanism using `tree-sitter` a Parser used by `ATOM` [Wik], specifically the language for C with `tree-sitter-c`. The Python

client parses the decompiled C code into an AST and traverses it to generate a 'skeletal' representation of the code.

In this representation, specific identifiers, literals, and types are replaced with generic placeholders (e.g., `id`, `num`, `type`), while control flow keywords (`if`, `while`, `for`, `switch`, `goto`) and block structures are preserved. This process effectively anonymizes the code and create a filter/standard for identation and formatting, cleaning out the possible noise and forcing the LLM to focus purely on the control flow logic and structural complexity (e.g., the presence of 'goto' statements vs̈structured loops) rather than being biased by variable names, comments or formatting.

...

4.3.5 Prompting

Chapter 5

Results

Here you will present the results of your work. Your skeptical reader is now asking themself "is this working?". You will show here to what extent it does. Be honest about the limitations: if your system doesn't work in some cases, you should say so (and explain where and why, if you can). Like the Methodology chapter, this one may end up being split into multiple chapters.

Keep in mind that this shouldn't be a mere dump of experimental results; you're rather *teaching* your reader how and to what extent you managed to solve the problem you described in the Introduction (Chapter 1). If you have additional results that may be useful but are not necessary to understand the points you're making (e.g., you evaluated your system on multiple datasets and the results all tell the same story), the place for them is in an appendix.

This chapter should have a lot of links to the methodology chapter(s), because you're evaluating the choices you made there. If you developed a system made of multiple parts, make sure that you test them separately and together, so that the reader can understand how important each part is.

This chapter is likely to be quite full of figures and tables. Try to make them as informative as possible (e.g., use multiple lines in the same plot if possible). Showing graphs effectively is a complex art; try to spend some time on it and ask for guidance from your advisor. Whenever you have a figure or a table, make sure that you refer to it in the text (after all, if it's not referred to in the text it means it has no part in the story you're telling, so it has no place in this Chapter). Always use the text (and the caption, if you can) to explain what you want the reader to understand from the figure.

If you are using LaTeX, it will automatically place figures, tables, algorithms, etc. somewhere in the text. These parts of the documents are called *floats* because their position

Column 1	Column 2	Column 3
aaa	2	3.42
bbb	505	6.00
ccc	8	901.02

Table 5.1: A table using the LaTeX `booktabs` package. Note there are no vertical rules. In case you want to do more fancy stuff (e.g., merging cells), check also the `multirow` and `multicol` packages. It's generally a good idea for readability to align text to the left and numbers to the right (use the same number of significant digits).

floats around depending on LaTeX's will. There are options to advise LaTeX about where floats should be placed. My advice is not to waste too much time on this and trust that they will end up in a good place. Generally, write the floats' code before you refer to them: they will never be placed before the place they appear in.

For Figures, generally use vectorial formats (e.g., PDF, SVG) where it makes sense if you can. They will result in higher-quality images that are in most cases also smaller, leading to shorter compiling times and a smaller resulting PDF.

Tables look better without vertical rules: an example is in Table 5.1.

5.1 Discussion

At the end of this chapter, take a step back and summarize all the results so that the reader can understand the big picture. Sometimes, this becomes large and important enough to be worth a chapter on its own.

Chapter 6

Conclusion

You are finally done! Here you will summarize for the reader what you have taught them through the document, and the main takeaways of your work you would like them to remember.

It is also a good idea to discuss the limitations of your work and your views about what can be possible future work.

Bibliography

[Bai] Yumo Bai. *Why are most LLMs decoder-only? — yumo-bai*. https://medium.com/@yumo-bai/why-are-most-llms-decoder-only-590c903e4789. [Accessed 25-01-2026].

[Car+21] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. "Extracting Training Data from Large Language Models". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2633–2650.

[CC90] Elliot J Chikofsky and James H Cross. "Reverse engineering and design recovery: A taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17. DOI: 10.1109/52.43044.

[Dev] Nithin Devanand. *What is Quantization in LLM — techresearchspace*. https://medium.com/@techresearchspace/what-is-quantization-in-llm-01ba61968a51. [Accessed 06-02-2026].

[Eag11] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. 2nd. No Starch Press, 2011.

[edm] edmcman. *Fix a variety of problems by edmcman · Pull Request #4 · vul337/DecompileBench — github.com*. https://github.com/vul337/{D}ecompile{B}ench/pull/4. [Accessed 09-02-2026].

[Fac] Hugging Face. *Quantization — huggingface.co*. https://huggingface.co/docs/optimum/concept_guides/quantization. [Accessed 06-02-2026].

[Gao+25] Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Bolun Zhang, and Chao Zhang. *DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios*. 2025. arXiv: 2505.11340 [cs.SE]. URL: https://arxiv.org/abs/2505.11340.

[Ghi] Ghidra. *ghidra/Ghidra/RuntimeScripts/Common/support/analyzeHeadlessREADME.md at master · NationalSecurityAgency/ghidra — github.com*. https://github.com/{N}ational{S}ecurity{A}gency/ghidra/blob/master/{G}hidra/{R}untime{S}cripts/{C}ommon/support/analyze{H}eadless{R}{E}{A}{D}{M}{E}.md. [Accessed 09-02-2026].

[Git] GitHub. *GitHub Actions — github.com*. https://github.com/features/actions. [Accessed 26-01-2026].

[Goo] Google. *GitHub - google/oss-fuzz: OSS-Fuzz - continuous fuzzing for open source software. — github.com*. https://github.com/google/oss-fuzz. [Accessed 06-02-2026].

[Hin+12] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. "On the naturalness of software". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 837–847.

[HLC24] Peiwei Hu, Ruigang Liang, and Kai Chen. "DeGPT: Optimizing Decompiler Output with LLM". In: *Proceedings 2024 Network and Distributed System Security Symposium* (2024). URL: https://api.semanticscholar.org/CorpusID:267622140.

[IBM] IBM. *Foundation model parameters: decoding and stopping criteria — ibm.com*. https://www.ibm.com/docs/en/watsonx/saas?topic=prompts-model-parameters-prompting. [Accessed 23-01-2026].

[Li+25] Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. *From Generation to Judgment: Opportunities and Challenges of LLM-as-a-judge*. 2025. arXiv: 2411.16594 [cs.AI]. URL: https://arxiv.org/abs/2411.16594.

[Mik] Rebeka Kiss Miklós Sebők. *LLM Parameters Explained: A Practical, Research-Oriented Guide with Examples — promptrevolution.poltextlab.com*. https://promptrevolution.poltextlab.com/llm-parameters-explained-a-practical-research-oriented-guide-with-examples/. [Accessed 23-01-2026].

[Mor] Abby Morgan. *Perplexity for LLM Evaluation — comet.com*. https://www.comet.com/site/blog/perplexity-for-llm-evaluation/. [Accessed 22-01-2026].

[Mul+18] Lincoln A. Mullen, Kenneth Benoit, Os Keyes, Dmitry Selivanov, and Jeffrey Arnold. "Fast, Consistent Tokenization of Natural Language Text". In: *Journal of Open Source Software* 3.23 (2018), p. 655. DOI: 10.21105/joss.00655. URL: https://doi.org/10.21105/joss.00655.

[Nat19] National Security Agency. *Ghidra Software Reverse Engineering Framework*. https://ghidra-sre.org/. Accessed: 2024-05-01. 2019.

[NSA] NSA. *Ghidra Decompiler Analysis Engine: Decompiler Analysis Engine — share.google*. https://share.google/Q9gHjuuTY3ZlFlm4n. [Accessed 16-01-2026].

[Pco] Pcode-doc. *P-Code Reference Manual — spinsel.dev*. https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html. [Accessed 19-01-2026].

[pyt] Hugging Face python. *datasets — pypi.org*. https://pypi.org/project/datasets/. [Accessed 06-02-2026].

[Sax] Shubham Saxena. *Understanding Perplexity in Language Models: A Detailed Exploration — shubhamsd100*. https://medium.com/@shubhamsd100/understanding-perplexity-in-language-models-a-detailed-exploration-2108b6ab85af. [Accessed 23-01-2026].

[Sta+24] Robin Staab, Mark Vero, Mislav Balunović, and Martin Vechev. "Beyond Memorization: Violating Privacy via Inference with Large Language Models". In: *The Twelfth International Conference on Learning Representations (ICLR)*. 2024.

[Tan+24] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. "LLM4Decompile: Decompiling Binary Code with Large Language Models". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024, pp. 3473–3487. DOI: 10.18653/v1/2024.emnlp-main.203. URL: http://dx.doi.org/10.18653/v1/2024.emnlp-main.203.

[Vas+23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[Wik] Wikipedia. *Atom (text editor) - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/{A}tom_(text_editor)#:~:text={A}tom%20uses%20{T}ree%2{D}sitter%20to%20provide%20syntax%20highlighting%20for%20multiple%20programming%20languages%20and%20file%20formats.%5{B}17%5{D}. [Accessed 09-02-2026].

[wika] wikipedia. *Perplexity - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Perplexity. [Accessed 22-01-2026].

[wikb] wikipedia. *Top-p sampling - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Top-p_sampling. [Accessed 23-01-2026].

[Zvo] Enes Zvornicanin. *What Is and Why Use Temperature in Softmax? — Baeldung on Computer Science — baeldung.com*. https://www.baeldung.com/cs/softmax-temperature. [Accessed 23-01-2026].