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 `Application Programming Interface (API)s`. 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 APIs 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

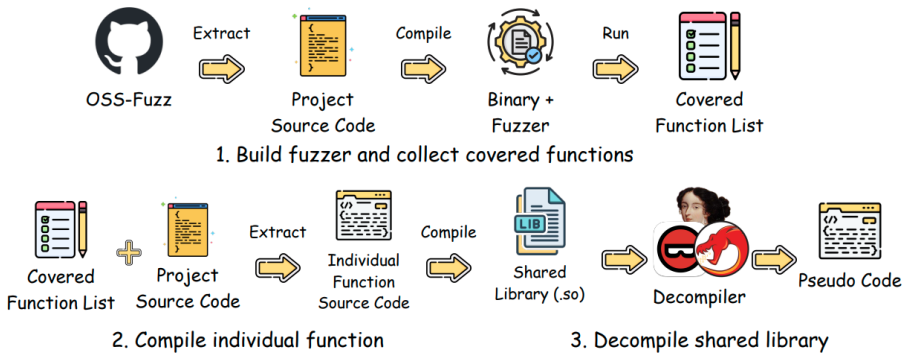


Figure 2.1: Creation of the dataset used in DecompileBench [Gao+25].

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, you can see some examples at Table 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.

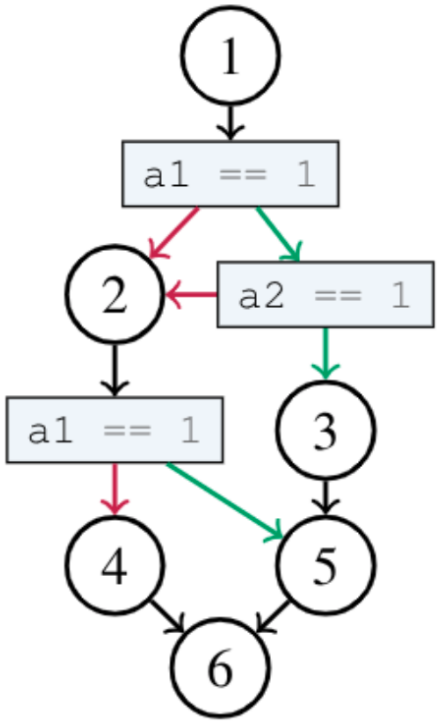


Figure 3.1: Control Flow Graph of a function [BEP25]

In this example we have the C code of the function described in figure 3.4.1 and its corre-

sponding 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(L'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

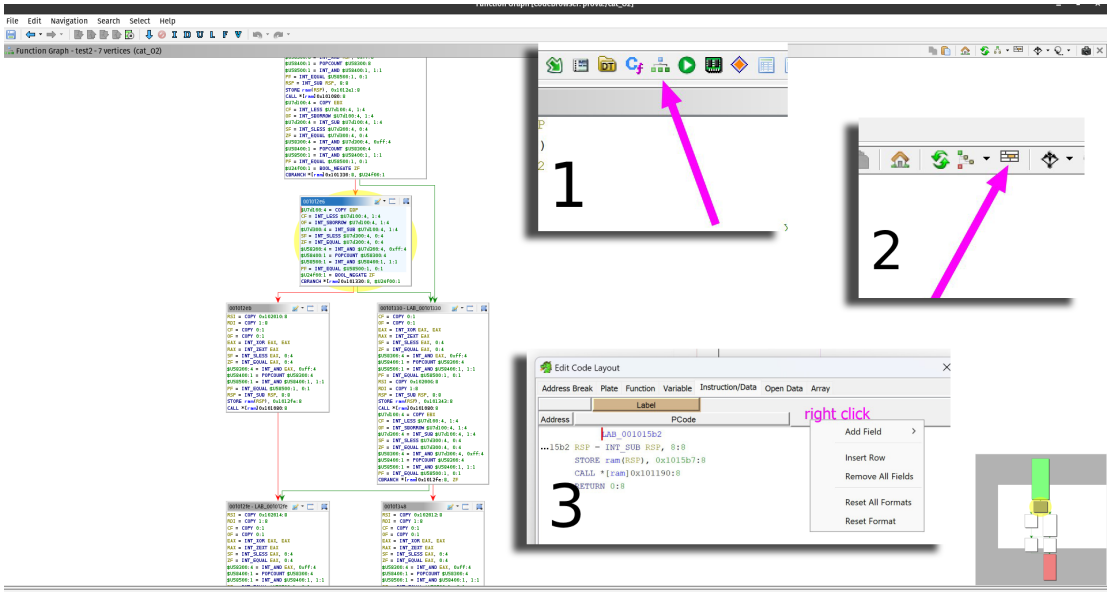


Figure 3.2: Control Flow Graph with P-code in Ghidra

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.

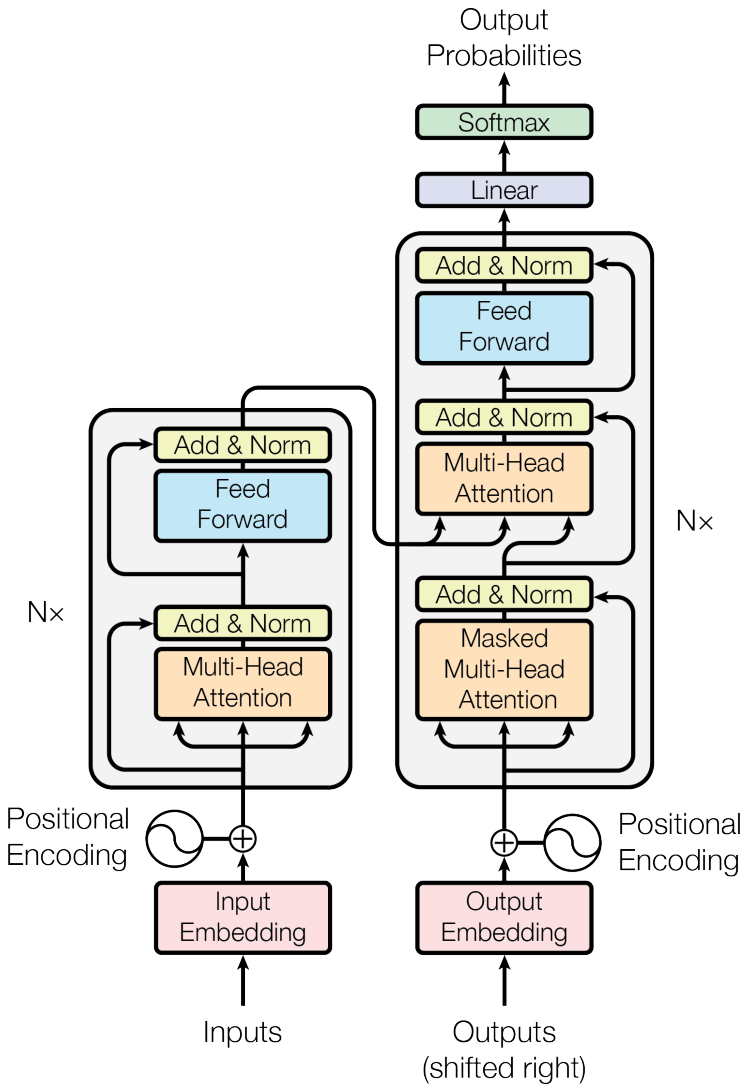


Figure 3.3: The Transformer model architecture

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/

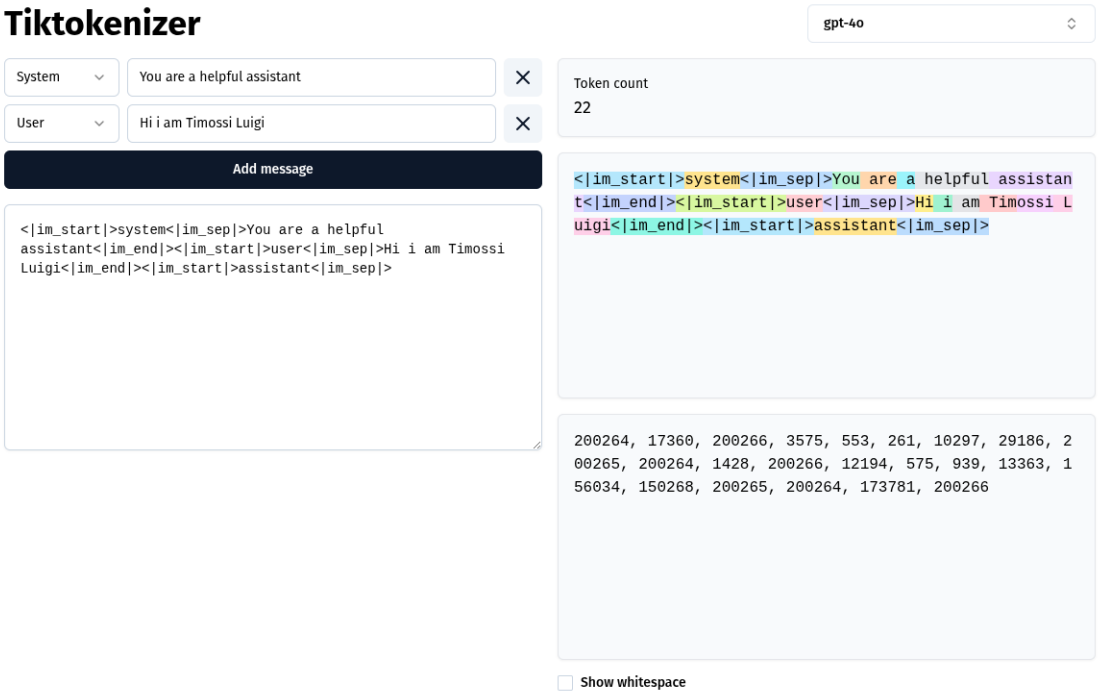


Figure 3.4: Tokenization example using tiktokenizer

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 in Section 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.

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 Graphics Processing Unit (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.

Initially we wanted to use modles in the range of 8B parameters, the canditates where Meta Llama 3.1 (8B), Qwen2.5-Coder-Instruct (7B), DeepSeek-R1-Distill-Qwen (7B) and Google Gemma 2 (9B) for various reasons, such as the availability of the model, the performance on code-related tasks, and the memory requirements, but after a lot of testing and variation of prompts, we decided to going up to use models in range of 14B parameters since the firsts were not able to provide consistent and meaningful scores for our evaluation, falling into the Position Bias problem, which is a common issue in LLM-based evaluation where the model's scoring is heavily influenced by the position of the input text rather than its content, leading to unreliable assessments (even when we tried to mitigate it with different prompt engineering techniques, such as randomizing the order of the inputs, or using different templates for the prompts).

Figure 4.2: Example of a function with only one difference between the BASE and the PR.

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

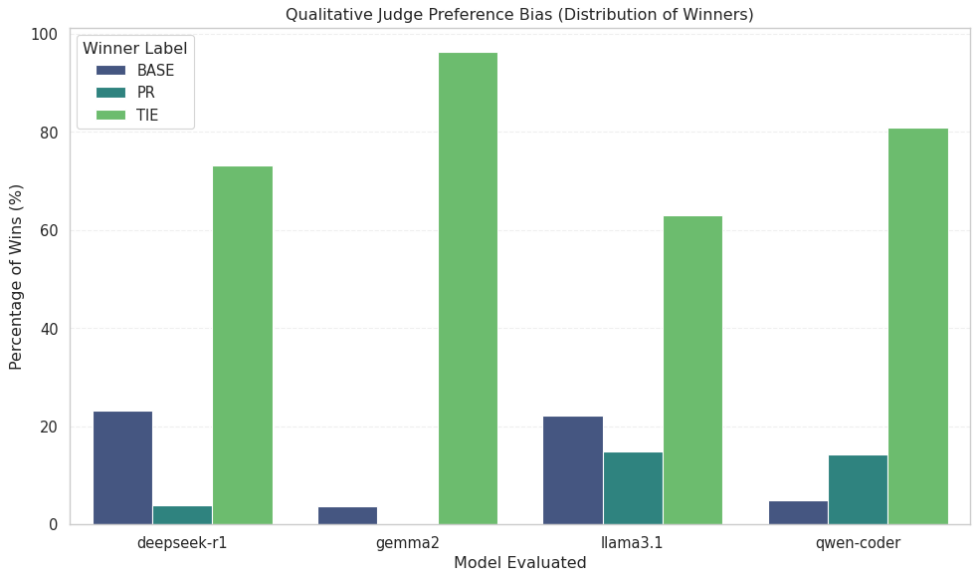
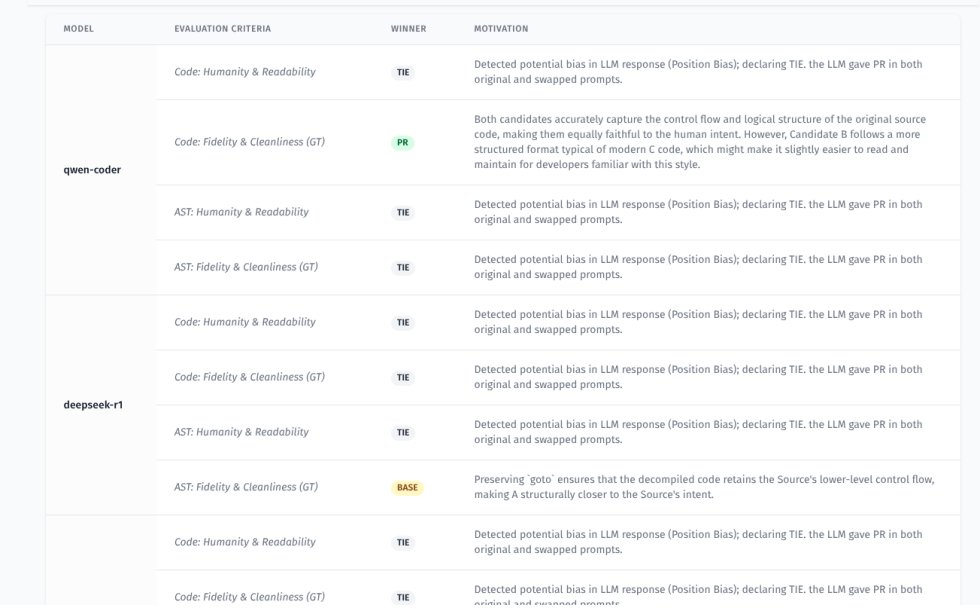


Figure 4.1: Distribution of winner bias, the TIE is when a model select always one choice (BASE, PR) even after swapping the input.

As example of this problem in Figure 4.1 when the majority of the scores are in the "TIE" category, meaning that the model is always selecting the same choice (either BASE or PR) as the winner, even after swapping the input order, which indicates a strong position bias and a lack of meaningful differentiation between the two inputs based on their content. Even after checking the non TIE cases, we found that the motivation for the choice was not based on the content of the decompilation and the difference beetween the two inputs. In this case whe have a function with only one difference between the BASE and the PR, that can be seen in Figure 4.2, and the corrisponding responses in Figure 4.3 from the model, Among the TIEs we have two cases where the model consistently selected a winner, in the case of PR the motivation was consistent with the content of the decompilation and the difference between the two inputs, while in the case of BASE the motivation was on some superficial aspect of the input ignoring the context.



MODEL	EVALUATION CRITERIA	WINNER	MOTIVATION
qwen-coder	*Code: Humanity & Readability*	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
	Code: Fidelity & Cleanliness (GT)	PR	Both candidates accurately capture the control flow and logical structure of the original source code, making them equally faithful to the human intent. However, Candidate B follows a more structured format typical of modern C code, which might make it slightly easier to read and maintain for developers familiar with this style.
	AST: Humanity & Readability	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
	AST: Fidelity & Cleanliness (GT)	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
deepseek-r1	*Code: Humanity & Readability*	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
	Code: Fidelity & Cleanliness (GT)	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
	AST: Humanity & Readability	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
	AST: Fidelity & Cleanliness (GT)	BASE	Preserving `goto` ensures that the decompiled code retains the Source's lower-level control flow, making A structurally closer to the Source's intent.
	Code: Humanity & Readability	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.
	Code: Fidelity & Cleanliness (GT)	TIE	Detected potential bias in LLM response (Position Bias); declaring TIE. the LLM gave PR in both original and swapped prompts.

Figure 4.3: Example of the motivation for the choice of the model, where the motivation is not based on the content of the decompilation and the difference beetween the two inputs, but rather on some superficial aspect of the input.

So for these reasons we decided to use bigger models, since with these results we were not able to draw any meaningful conclusion from the evaluation, and we wanted to have a more reliable and consistent evaluation for our study.

4.2.1.1 Model Selection

Based on these considerations, we wanted to select 4 models for our evaluation, the range of 14B parameters is the last range that can be run on our hardware with 4-bit quantization, and that can provide a good performance for our evaluation. Unfortunately, due to the scarcity of models in this range, and the tendency of relasing only very big or very small models, we did not have much choice in the selection (Exluding models done by Google, OpenAI or Meta), we found these state of art models:

- Qwen 3 (14B)
- DeepSeek-R1 Distill Qwen (14B)
- Phi 4 (14B)
- Mistral Nemo instruct 2407 (14B)

Unfortunatly after some testing we found out that Phi 4 and Mistral Nemo were not able to provide consistent and meaningful scores for our evaluation, falling into the Position Bias problem as can be seen in Figure 4.4, we also tried to replace them with Qwen2.5 Coder and Starcoder2 (both 15B) but even with those models we were not able to get consistent results, so we decided to use only Qwen3 and DeepSeek-R1 Distill Qwen, since they were able to provide more reliable and consistent scores.

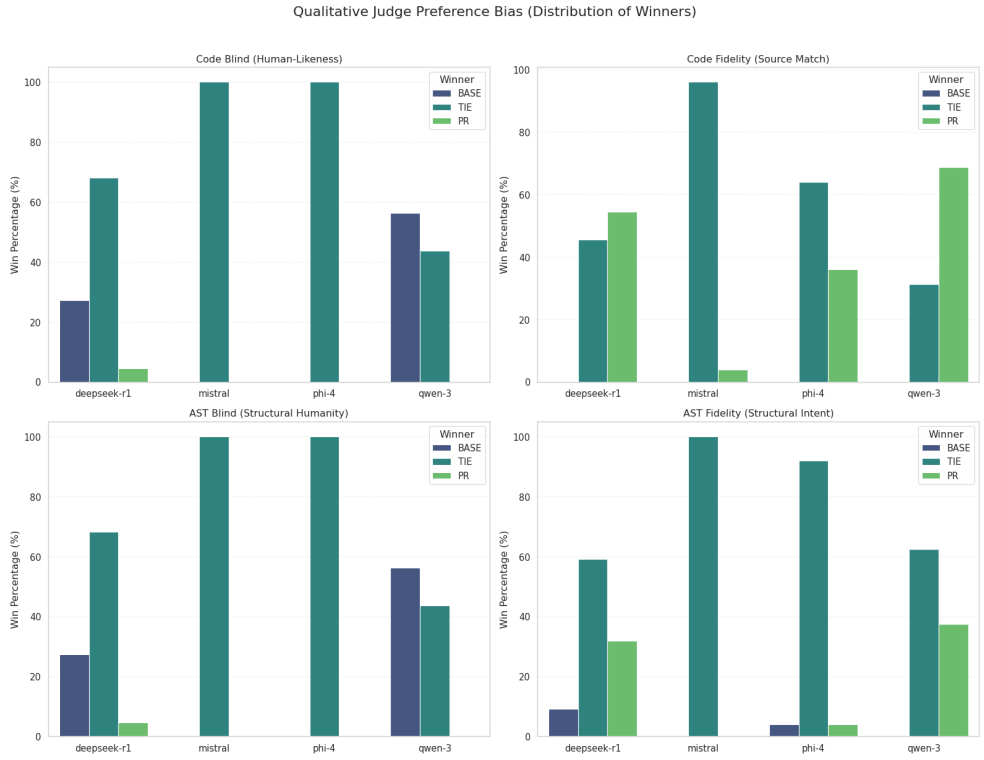


Figure 4.4: Distribution of winner bias for the new models, where we can see that Phi 4 and Mistral Nemo are still falling into the Position Bias problem, while Qwen3 and DeepSeek-R1 Distill Qwen are providing more reliable and consistent scores for our evaluation.

- **Qwen 3 (14B)**: is the latest generation in the Qwen family and comes as a suite of both dense and mixture-of-experts (MoE) models. A distinctive feature of the series is the ability to switch within the same model between a *thinking* mode (aimed at complex reasoning, mathematics, and coding) and a *non-thinking* mode (optimized for efficient, general-purpose dialogue), allowing the same backbone to adapt to different interaction styles and difficulty levels [Ten]

- **DeepSeek-R1 Distill Qwen (14B/15B[2])**: Part of the DeepSeek-R1 family, which

[2]In the name and the description it shown that is 14B parameters but in the tag of huggingface site its

introduces a reasoning-focused post-training pipeline combining cold-start supervised data and large-scale reinforcement learning to improve reasoning quality and readability over purely RL-trained variants (e.g., DeepSeek-R1-Zero). The *Distill* checkpoints transfer (distill) the reasoning patterns learned by the larger DeepSeek-R1 model into smaller dense backbones (including Qwen-based models), providing strong math/code/reasoning capability in a size that is practical to run locally on our hardware [Dee25].

We wanted to include different types of models but in the end we had to select models that are all based on the same architecture (Qwen), since they were the only ones that can provide consistent and meaningful scores for our evaluation, and that can fit in our hardware constraints, but we acknowledge that this is a limitation of our study and that it would be interesting to include models with different architectures and training paradigms in future work, to assess the generality of our findings across a wider range of LLM designs.

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 name. Concretely, a dictionary (`MODELS_CONFIG`) defines the available models 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
)
```

On startup, the server checks the availability of the GPU, CUDA and, to reduce cold-start delays, checks and optionally downloads all model snapshots before accepting requests, ensuring that evaluation runs are not affected by network variability. The greatest bottleneck in our setup is the model loading time and context switch (changing from one model to another), which can take various minutes per model due to the size and quantization overhead. We have organized the request pipeline in the Client to minimize the number of context switches but to mitigate further, the server implements a simple caching mechanism: At the start, when checking if the weights are in Huggingface cache folder we load

tagged as 15B

the model into the GPU for quantization, after that we move the quantized weights and his tokenizer to CPU memory (RAM) and unload the model from VRAM. When every model have been loaded the server opens up requests maintaining the quantized weights in RAM, so that every context switch can be fulfilled by quickly moving the quantized weights from RAM to VRAM, which is much faster than loading from disk and quantizing on the fly.

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=4096`: 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.

The evaluation stage is an end-to-end pipeline over a set of target Ghidra Pull Requests (PRs). At a high level, the client:

1. Ensures that the *base* version is built and has produced decompilations for all dataset binaries
2. Iterates over the selected PR, building each corresponding Ghidra revision and extracting the related decompilations
3. Selects a limited subset of binaries that actually exhibit decompilation differences, to focus the evaluation on meaningful cases and reduce noise
4. For each model, queries the LLM server to score and compare the outputs, producing per-PR and aggregate JSON reports for later analysis.

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 PRs 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 that 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. This incremental strategy prevents re-running expensive steps and makes the workflow resumable.

4.3.2 Ghidra Headless

For decompilation we use the headless mode of Ghidra, which allows us to run Ghidra in a command-line environment without the need for a GUI [Ghi]. This is particularly useful for automating the decompilation process and integrating it into our evaluation workflow. The headless mode is invoked via a command-line script; the entry point is `support/pyghidraRun` (preferred, when available) executed in headless mode (`--headless`), creating a temporary per-binary Ghidra project, importing the binary, and finally running a post-script (`extract.py`) that performs the actual decompilation and exports the results to JSON. We use parallel execution to speed up the decompilation of multiple binaries; To avoid race conditions and project-file locks during parallel execution, we create an isolated project directory for each binary and delete it at the end of the run.

The headless invocation follows this template:

```
$GHIDRA_HOME/support/pyghidraRun --headless <proj_dir> <proj_name> \
    -deleteProject \
    -import <binary_path> -overwrite \
    -scriptPath <scripts_dir> \
    -postScript extract.py
```

For older Ghidra versions where `pyghidraRun` is not present, we fall back to the standard headless launcher `support/analyzeHeadless` with the same arguments. The client passes configuration to the post-script via environment variables: the output directory,

the evaluated Ghidra version tag, and the comma-separated list of target function names to decompile (if not specified, all functions are decompiled, in our case we pass only the function name present in the database for that binary because in the compilation process some other functions are added in the binary for requirements). This allows the same `extract.py` script to be reused across runs and versions without hardcoding paths or dataset-specific information.

4.3.3 Evaluation

Not every decompilation difference is relevant for our study: superficial variations (e.g., whitespace, renaming, minor formatting) would introduce noise. For this reason, before invoking the LLM, the client performs a structural comparison between the baseline and PR outputs:

- it loads the JSON produced by the decompiler for both `base` and `pr_#ID`,
- it compares the abstracted representation (via our `tree-sitter`-based abstraction, see Section 4.3.4) of the two outputs,
- it retains only binaries for which the abstracted forms differ, i.e., where control-flow or structure plausibly changed.

For each retained binary we compute a complexity proxy (*cyclomatic complexity*) based on the control-flow graph of the decompilation, which is defined as:

$$M = E - N + 2P,$$

where:

- E is the number of edges in the control-flow graph,
- N is the number of nodes in the control-flow graph,
- P is the number of connected components (one function is considered a single connected component, since we decompile at function level this is always 1 in our case).

The cyclomatic complexity quantifies code complexity based on decision points (e.g., if, while, case), where higher scores indicate harder-to-test, maintain, and error-prone code [Son]. Then for sampling reasons, we sort candidates by decreasing complexity and keep only the top `MAX_SAMPLES`. This strategy ensures that our evaluation focuses on the most challenging and relevant cases, where improvements in decompilation quality are most impactful.

4.3.3.1 LLM-based scoring

For each selected binary and for each model exposed by the server, we score:

- the baseline decompilation,
- the PR decompilation,
- the original source function (dataset ground truth),

using the server `/score` endpoint, which returns token-level negative log-likelihood aggregated into a *perplexity* value. In addition, we compute the perplexity of the abstracted AST forms (base/PR/source), enabling an evaluation that is less sensitive to naming and formatting. To reduce redundant server calls across repeated evaluations, the client maintains a cache keyed by `(func_name, model_id, test_binary_name, ppl_type)` for perplexity of base, source and their abstracted forms.

4.3.3.2 Comparison metric

For each sample we derive a simple *relative* metric:

$$\Delta\text{PPL} = \text{PPL}_{PR} - \text{PPL}_{base},$$

(and analogously for the abstracted representation). A negative ΔPPL indicates that the PR output is more *likely* under the model than the baseline output, which we interpret as a proxy for improved fluency/regularity. Conversely, a positive ΔPPL suggests a regression in perceived readability.

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 vs̈tructured loops) rather than being biased by variable names, comments or formatting.

We can see from this example how the original code (left) is transformed into the abstracted version (right), where all identifiers and literals are replaced with placeholders, while the control flow and structure are maintained.

Source Code

```c
//This function does random stuff dont try to understand it
void complex(int a, char *b) {
    long *f;
    int h[10];
    if (a > 0) {
        while (a < 10) {
            printf("Value: %d\n", a);
            a++;
        }
    } else {
        goto end;//random comment
    }
    h[0] = 42;
    end:
    char c = b[0];
    f->g(h[i]);
    (*(int *)(p + 4)) = 5;
}
```

Abstracted Code

```c
type id(type id, type *id){
    type *id;
    type id[10];
    if(id > 0){
        while(id < 10){
        call(str, id);
        id++;
        }
    }else{
        goto lbl;
    }
    id[0] = 42;
    lbl:
    type id = id[0];
    call(id[id]);
    (*(type)(id + 4)) = 5;
}
```

This allows us to evaluate the “humanness” of the decompilation output based on its logical structure and flow, rather than being influenced by specific naming choices or formatting styles that may vary widely across different decompilers and Ghidra versions.

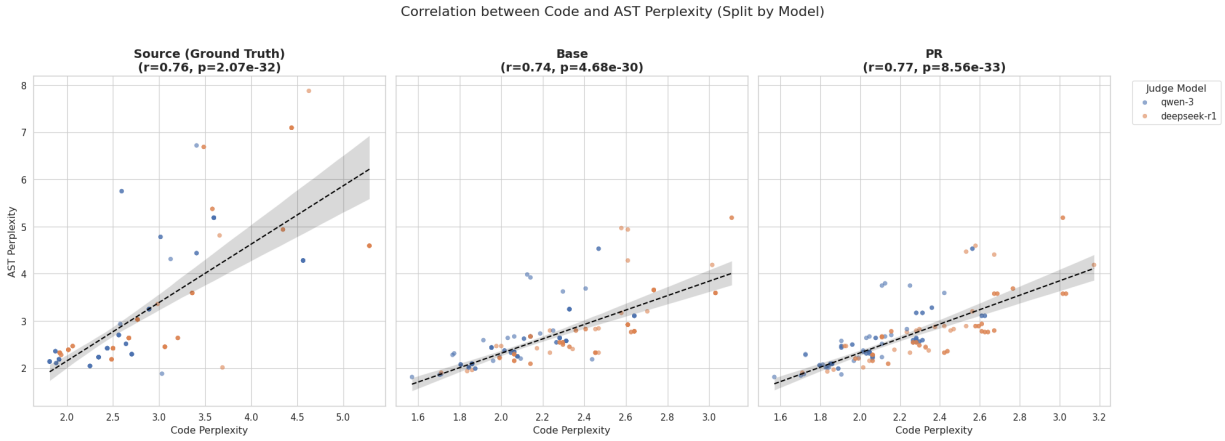


Figure 4.5: Pearson correlation between the abstracted AST perplexity and source code perplexity for the same decompilation output, across all evaluated samples.

This anonimation also allows us to compute a separate perplexity score for the abstracted AST representation, which can be compared against the perplexity of the original source code decompilation. By analyzing the correlation between these two scores, we can gain insights into how well the LLM is capturing the structural and logical aspects of the code, as opposed to just surface-level token patterns.

As shown in Figure 4.5, we observed a positive correlation (Pearson's $r > 0.7$, $p < 0.001$) between the perplexity of the fully abstracted AST and that of the original source code across all evaluated models. In other words, samples that were rated as more "surprising" or "incoherent" in their original form also tended to have higher perplexity in their abstracted AST representation, and vice versa.

4.3.5 Prompting

One key element of our evaluation framework is the design of the prompts used to query the LLM for scoring the decompilation outputs. We divide our analisys in two main types of prompts:

- **Human evaluation prompt**: This type of prompt is designed to elicit a qualitative assessment from the LLM about which version of the decompilation output is better in terms of human readability.
- **Closer to source prompt**: This type of prompt is designed to elicit a qualitative assessment from the LLM about which version of the decompilation output is closer

to the original source code.

for each type of prompt we have two version, the one with the code in the original form and the one with the code in the abstracted form, this is to check if the model prefers one version over the other and if the abstraction process is effective in making the model focus on the structural aspects of the code rather than being influenced by specific naming choices or formatting styles.

4.3.5.1 Biases and prompt design

We had to design the prompts carefully to avoid introducing bias and help the model to focus on the relevant aspects of the decompilation outputs. For avoid "update version" bias, we used a template that does not explicitly mention the concept of "base" and "PR", but rather presents the two decompilation outputs as "Version A" and "Version B", and then asks the model to compare them based on specific criteria. To help the model from possible hallucinations it is important to make it generate the motivation for the choice *before* revealing the correct answer, so we ask the model to first provide an explanation of which version it thinks is better and why, since it will use the token generated to fill his context window.

For avoid "position bias" we perform a consistency check by swapping the inputs (the one referred to as Version A and the one as Version B) and re-querying the model to see if it still prefers the same version, which can help us to identify and mitigate cases where the model's preference is influenced by the position of the input rather than its content. If the model's choice changes after swapping, it suggests that the original preference may have been due to position bias rather than a genuine assessment of the decompilation quality, so is flagged as *TIE*. To optimize the context window for the input, we designed the prompt to put version A and version B in a Diff format, where we include all the code of the first version but only the lines that differ in the second version, this way we can reduce the amount of tokens in the input and make it easier for the model to focus on the relevant differences between the two versions.

For avoiding the bias of the model where it always selects the version with `++` since it is a common pattern where the PR version or the "update"" version is better than the one with `--`, we decided to use different symbols for the versions, instead of using `++` and `--` we used `%` and `&`, this way we can avoid the bias of the model and make it focus on the content of the decompilation rather than being influenced by the symbols used to represent the versions (obviusly telling the legend for the new symbols inside the prompt).

Here we can see an example of the Diff design for the prompt, where version A and version B are represented with `%` and `&` respectively, and the prompt includes only the lines that differ between the two versions, while the rest of the code is included in full for both

versions (The function does random stuff dont try to understand it):

ver. A

```
void complex(int a, char
    *b) {
  long *f;
  int h[10];
  if (a > 0) {
    while (a < 10) {
      printf("Value:
          %d\n", a);
      a++;
    }
  } else {
    goto end;
  }
    h[0] = 42;
    end:
    char c = b[id];
    f->g(h[i]);
    (*(int *)(p + -4)) =
        5;
}
```

ver. B

```
void complex(int a, char
    *b) {
  long *f;
  int h[10];
  if (a > 0) {
    while (a < 10) {
      printf("Value:
          %d\n", a);
      a++;
    }
  } else {
    goto end;
  }
  h[0] = "*";
  end:
  char c = b->id;
  f->g(h[i]);
  (*(int *)(p - 4)) = 5;
}
```

ver. prompt

```
void complex(int a, char
    *b) {
  long *f;
  int h[10];
  if (a > 0) {
    while (a < 10) {
      printf("Value:
          %d\n", a);
      a++;
    }
  } else {
    goto end;
  }
% h[0] = 42;
& h[0] = "*";
  end:
% char c = b[id];
& char c = b->id;
  f->g(h[i]);
% (*(int *)(p + -4)) = 5;
& (*(int *)(p - 4)) = 5;
}
```

4.4 Pull requests

Ghidra's development is organized around PRs on GitHub, where contributors propose changes to the codebase that can include bug fixes, new features, or improvements to existing functionality. Each PR represents a specific set of changes that can affect the decompilation output in different ways. Fortunatly in the GitHub repository the PRs have tags (in our case label:"Feature: Decompiler"), since a lot of them does not modify decompiler logic, so we can easily select only those that are relevant for our evaluation, and that can provide meaningful insights into the impact of specific changes on decompilation quality.

For our evaluation, we select a subset of PRs:

- **PR #8628**: improves the cleanup rule `Rule2Comp2Sub` so that it also handles *constant* subtractions that are rewritten during decompilation as additions with negative immediates (e.g., `x + -0x1a` → `x - 0x1a`), improving readability and bringing the output closer to typical C source style.

- **PR #8587**: extends the `constantptr` rule to automatically detect and correct *one-based* indexing patterns for spacebase constants (e.g., *(undefined *) ((long)i * 0x30 + 0xaddr + (long)j * 4) → globalArray[(long)i + -1].field[j]).

- **PR #8161**: fixes an issue in `BlockWhileDo` where the decision to use overflow syntax could be made using unoptimized pcode and then treated as permanent: after pcode optimization the exit-condition block may no longer be "complex," but the stale overflow flag would still block for-loop recovery; the patch re-checks that overflow-marked `BlockWhileDo` blocks are still complex at the time of for-loop recovery and clears the overflow decision when it is no longer justified.

- **PR #7253**: sorts `switch case` entries by their target address (i.e., the destination basic-block address) rather than leaving them in an arbitrary/disassembler-dependent order.

- **PR #6722**: fixes cases where the decompiler fails to recover submember array access due to `RulePtrArith` distributing an addition through an `INT_MULT` (e.g., `(idx + -0x30) * 4 + 4` → `idx * 4 + (-0x30 * 4 + 4)`), which prevents matching the struct member; the rule now undoes this distribution and retries, allowing clean output such as `PTR_0041a1b8->ar[local_18]` instead of raw pointer casts and offsets.

Ghidra is a large and complex codebase, PRs are modifications that can affect some specific aspect of the decompilation output, it will never be the case that a PR will improve all the decompilations or change the decompilation core aspects in a drastic way, but rather it will improve some specific cases and make them more readable, while in other cases it can make them worse, so we want to evaluate the impact of these specific changes on the decompilation quality and readability, and see if the LLM-based evaluation can capture these improvements or regressions in a meaningful way.

4.5 Reporting

For each PR the client writes a dedicated report file (`reports/#PR.json`) containing aggregate statistics such as mean ΔPPL across samples, mean baseline/PR/source perplexities, and their abstracted counterparts, and then a list of results for each model: *(i)* the list of evaluated samples with their individual perplexities and ΔPPL values, and *(ii)* results of qualitative analysis from the LLM. At the end of the full run, a `final_report.json`

aggregates all per-PR summaries. If a report already exists, the client loads it and skips re-evaluation, enabling robust recovery after interruptions.

Finally, to avoid leaving a large model resident in GPU memory, the client calls the server cleanup endpoint at the end of the run, explicitly unloading the active model.

4.6 Dogbolt

Our analysis is focused on evaluating the impact of specific PRs on decompilation readability and humanness for Ghidra, but it is also interesting to see how the decompilation quality has evolved over time across different decompilers, and to have a more general overview of the trends and patterns in the decompilation output horizzontally (not vertical on only Ghidra). For this reason, we also created a framework of three different decompilers (Ghidra 12.0.1, Binary Ninja 5.2, and Hex-Rays 9.2) using Dogbolt[3], an interactive online *Decompiler Explorer* that displays outputs from multiple decompilers for the same input binary, as can be seen in Figure 4.6. Dogbolt is community-maintained and open source but intentionally does not provide a public automation API, we used it in a restricted version of our Dataset (only the project "file"), while all large-scale and reproducible measurements in this thesis were executed in our own local pipeline.

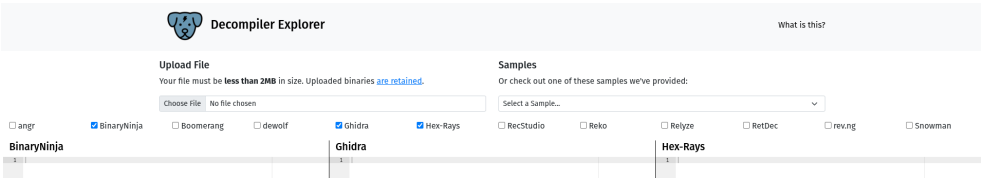


Figure 4.6: Example of the Dogbolt interface, where we can see the decompilation outputs from three different decompilers

Other differences with our main evaluation framework are the fact that the outputs of decompilers are not directly comparable using Diff since they generate different code, names, and formatting, so we rely, in this case, on giving all the code to the LLM. For the evaluation we use the same prompting strategy described in Section 4.3.5, but we used a pipeline when for every function we create three different prompts, one for each pair of decompilers (Binary Ninja vs Ghidra, Ghidra vs Hex-Rays, Binary Ninja vs Hex-Rays), and then we aggregate the results to have a more general overview of the trends and patterns in the decompilation output across different decompilers. We also had to parse the results for extracting the functions from the dogbolt results since this is external to the Dataset maker framework in Section 4.1.

[3] https://dogbolt.org/

Chapter 5

Results

5.1 LLM performance

An empirical evaluation of the *qwen-3* and *deepseek-r1* models was conducted across two distinct tasks: evaluating the "Humanity" of decompiled code via generation (LLM Judge) and calculating code perplexity (Score). The performance was measured in terms of execution time and peak VRAM usage.

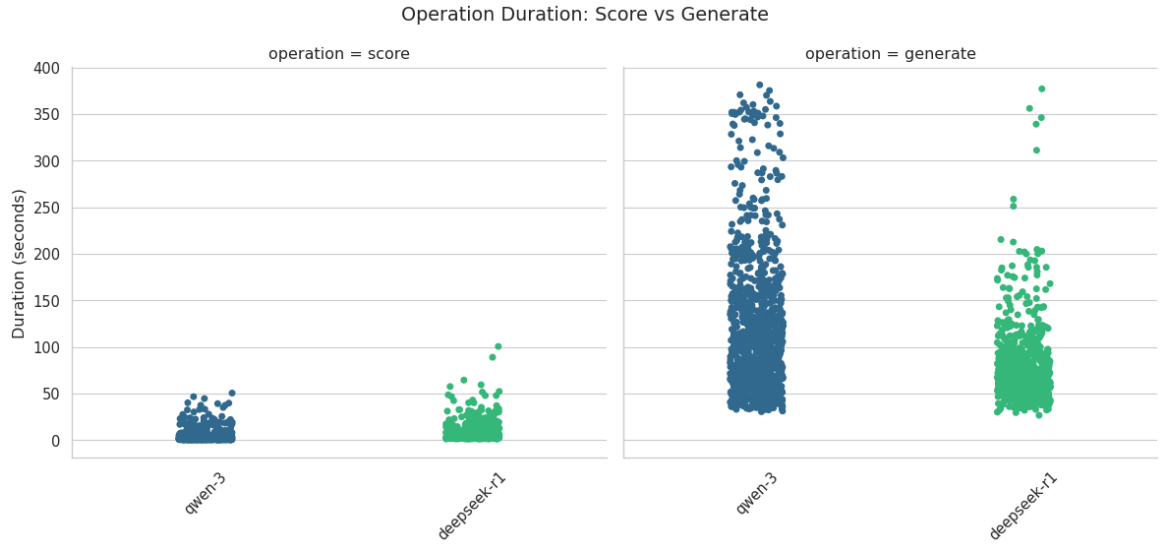


Figure 5.1: Execution time for each evaluated model.

As shown in Figure 5.1, there is a stark dichotomy in execution times between the two

operations. The `score` operation is inherently fast, with both models completing most passes in under 50 seconds. However, the `generate` operation exhibits significantly higher and more erratic execution times, particularly for *qwen-3*, which frequently exceeds 200 seconds and reaches up to nearly 400 seconds.

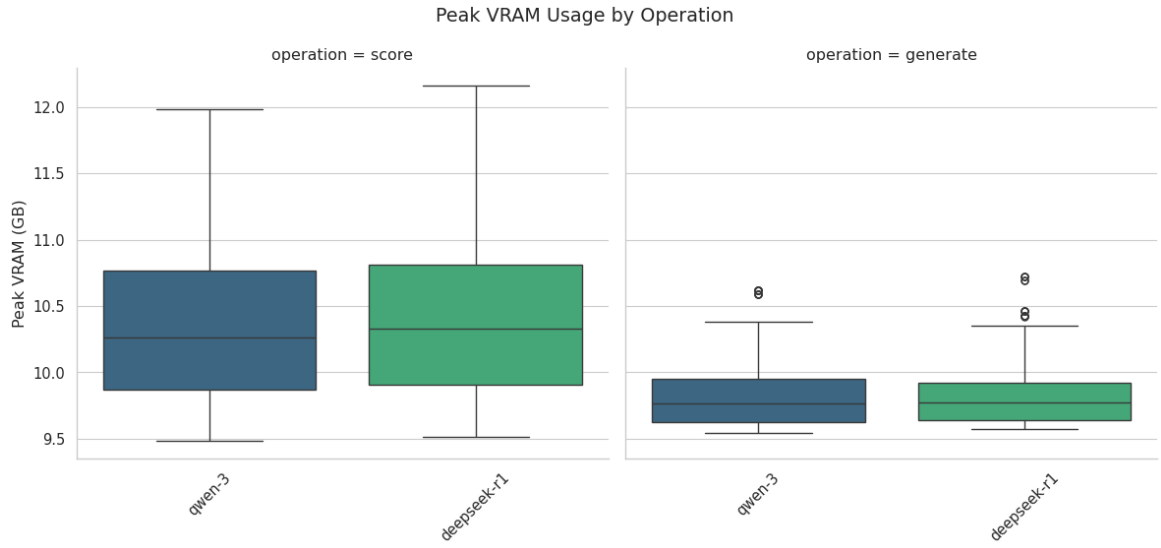


Figure 5.2: Peak VRAM for each evaluated model.

Figure 5.2 presents a counter-intuitive finding: calculating perplexity (`score`) requires a higher median and maximum Peak VRAM than generating text (*generate*). While generation heavily utilizes the KV cache over time, perplexity calculations typically require processing the entire sequence in a single forward pass to compute logits, leading to massive activation memory spikes [Atm+25].

5.1.1 Generation

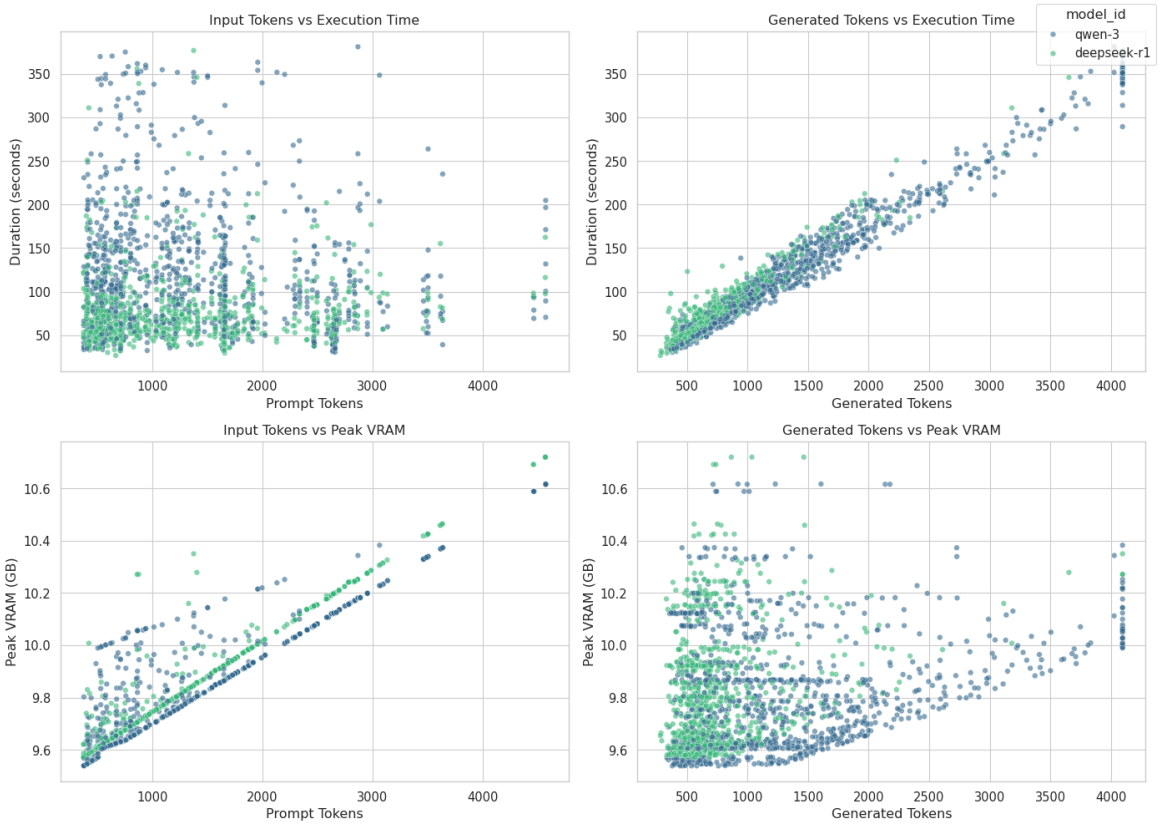
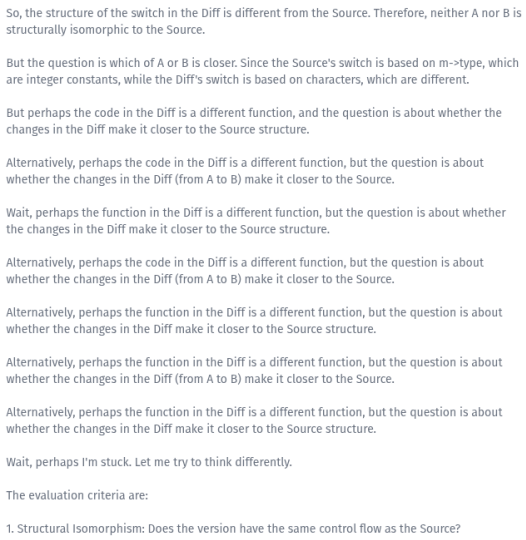


Figure 5.3: Generated tokens vs execution time and peak VRAM for each evaluated model.

A closer inspection of Figure 5.3 reveals the root cause of the discrepancy between the two models. The scatter plot for "Generated Tokens vs Execution Time" shows a near-perfect linear correlation, confirming that generation time is bottlenecked entirely by the length of the model's output rather than the input context. Critically, *qwen-3* routinely generates between 2,000 and 4,000+ tokens during the evaluation task and often reaches our limit of 4096 new tokens.

Given that the task is an LLM Judge producing an evaluation of code "humanity," a rationale exceeding 4,000 tokens is highly suspicious. This suggests that *qwen-3* may be suffering from severe verbosity, repeating the input code, or failing to trigger stop tokens appropriately. In contrast, *deepseek-r1* is significantly more concise, rarely exceeding 2,500 tokens, which translates directly to more predictable and efficient execution times.

In Figure 5.4 we can see an example of the output of *qwen-3* for a specific function, we can see that the model is repeating the same pattern of thoughts, which is a common symptom of verbosity and lack of proper stop token triggering. This behavior not only leads to unnecessarily long outputs but also significantly increases execution time and VRAM usage, as the model continues to generate tokens without producing meaningful content. This verbosity issue is a critical flaw for the LLM Judge task, as it undermines the model's ability to provide concise and relevant evaluations of code "humanity," and it also creates significant computational inefficiencies that could be prohibitive in larger-scale evaluations or with longer input contexts, maybe could be mitigated by implementing stricter stop conditions or by fine-tuning the model to better understand the task requirements and avoid unnecessary verbosity but it will need further investigation.


So, the structure of the switch in the Diff is different from the Source. Therefore, neither A nor B is structurally isomorphic to the Source.

But the question is which of A or B is closer. Since the Source's switch is based on m->type, which are integer constants, while the Diff's switch is based on characters, which are different.

But perhaps the code in the Diff is a different function, and the question is about whether the changes in the Diff make it closer to the Source structure.

Alternatively, perhaps the code in the Diff is a different function, but the question is about whether the changes in the Diff (from A to B) make it closer to the Source.

Wait, perhaps the function in the Diff is a different function, but the question is about whether the changes in the Diff make it closer to the Source structure.

Alternatively, perhaps the code in the Diff is a different function, but the question is about whether the changes in the Diff (from A to B) make it closer to the Source.

Alternatively, perhaps the function in the Diff is a different function, but the question is about whether the changes in the Diff make it closer to the Source structure.

Alternatively, perhaps the function in the Diff is a different function, but the question is about whether the changes in the Diff (from A to B) make it closer to the Source.

Alternatively, perhaps the function in the Diff is a different function, but the question is about whether the changes in the Diff make it closer to the Source structure.

Wait, perhaps I'm stuck. Let me try to think differently.

The evaluation criteria are:

1. Structural Isomorphism: Does the version have the same control flow as the Source?

Figure 5.4: Example of repetitions in *qwen-3* output

5.1.2 Score

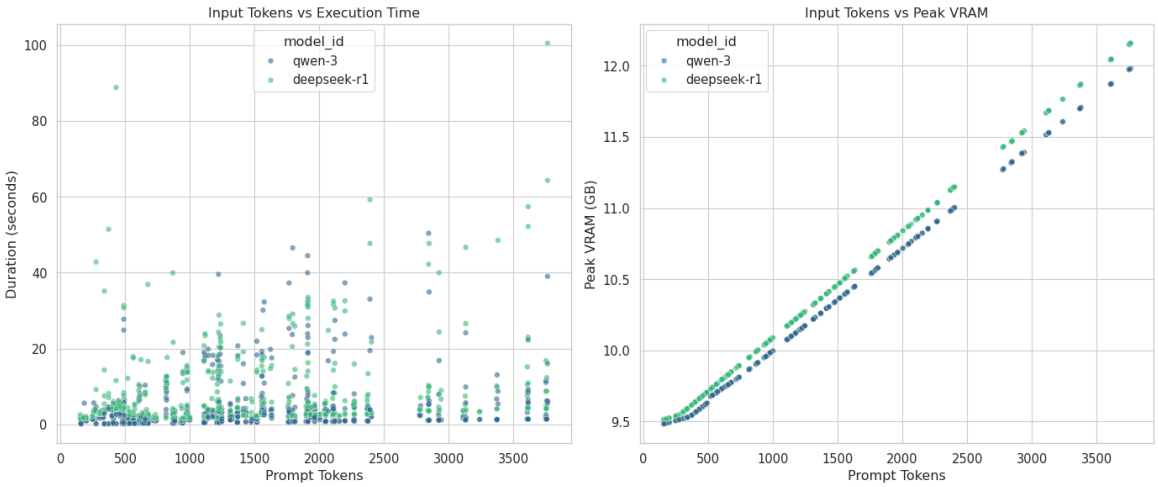


Figure 5.5: Score vs execution time and peak VRAM for each evaluated model.

The linear scaling of VRAM during the *score* operation (Figure 5.5, right panel) is a major scalability risk. At roughly 3,800 input tokens, both models push past 12 GB of VRAM. If the dataset contains an enormous larger input (e.g., 8k to 32k tokens), the current perplexity scoring methodology will inevitably result in Out-Of-Memory (OOM) failures on standard consumer GPUs.

Fortunately, the *generate* operation does not exhibit this issue, as it processes tokens sequentially and can leverage the KV cache to manage memory more efficiently.

The duration is mostly flat (near 0–10 seconds), but spikes occur at highly specific input token lengths for both models. This implies either batch-processing artifacts or that the evaluation dataset contains many decompiled functions of the exact same token length that trigger specific internal computational bottlenecks. A possible explanation is that runtime is influenced not only by input length, but also by the distribution of per-token loss values. This hypothesis requires additional targeted experiments, and further studies will be necessary to confirm it.

While both models are capable of performing the required tasks, *deepseek-r1* is objectively better suited for the LLM Judge (*generate*) role due to its restraint and conciseness, avoiding the computationally expensive verbosity traps that plague *qwen-3*. However, for the perplexity (*score*) task, the architecture of the operation itself poses a severe hardware bottleneck that scales poorly with larger decompiler outputs.

5.2 Perplexity as a Metric for "Humanness"

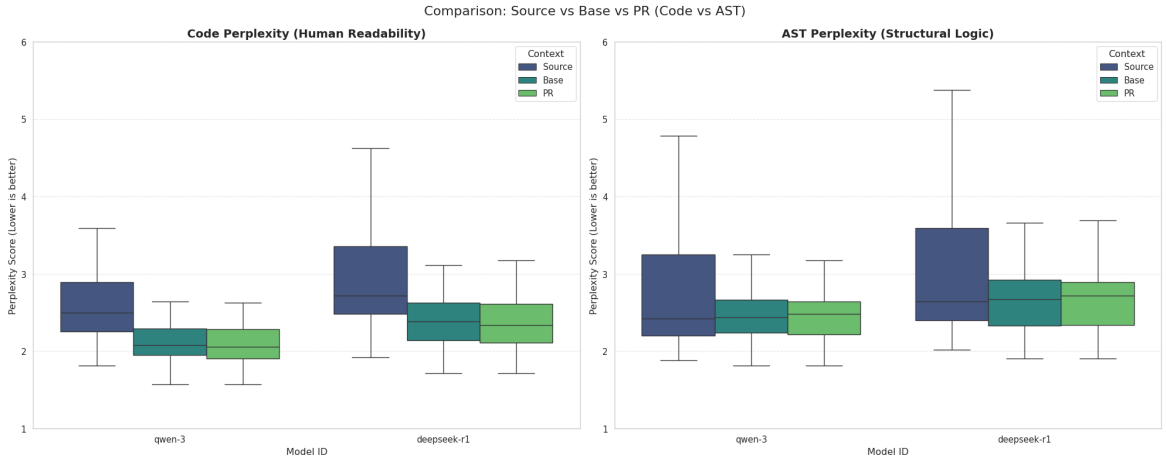


Figure 5.6: On the left perplexity values across original source code, base code, and pr code. On the right perplexity values across abstracted representations of the same functions.

In Figure 5.6 we can see the distribution of perplexity values for the original source code, the base code, and the pr code, as well as their abstracted representations. We can observe that the original source code has generally higher perplexity values compared to the decompiled versions, which is unexpected since the original source code should be more "natural" and predictable than the decompiled output. This suggests that the decompilation process may introduce certain patterns or structures that are more familiar to the language model, leading to lower perplexity scores, while the original source code may contain more variability and less predictable constructs that result in higher perplexity. We can observe that the perplexity distribution calculated with *deepseek-r1* is generally higher than the one calculated with *qwen-3*, this is consistent with the previous observation that *qwen-3*.

Another observation is that the abstracted representations of the code (right side of the figure) tend to have higher perplexity values compared to their original counterparts (left side of the figure). This is likely because the abstraction process removes specific identifiers and literals, which can make the code less predictable and more "surprising" to the language model, but even in this case, the original source code still has higher perplexity than the decompiled versions, reinforcing the idea that the decompilation process may be introducing more predictable patterns into the code.

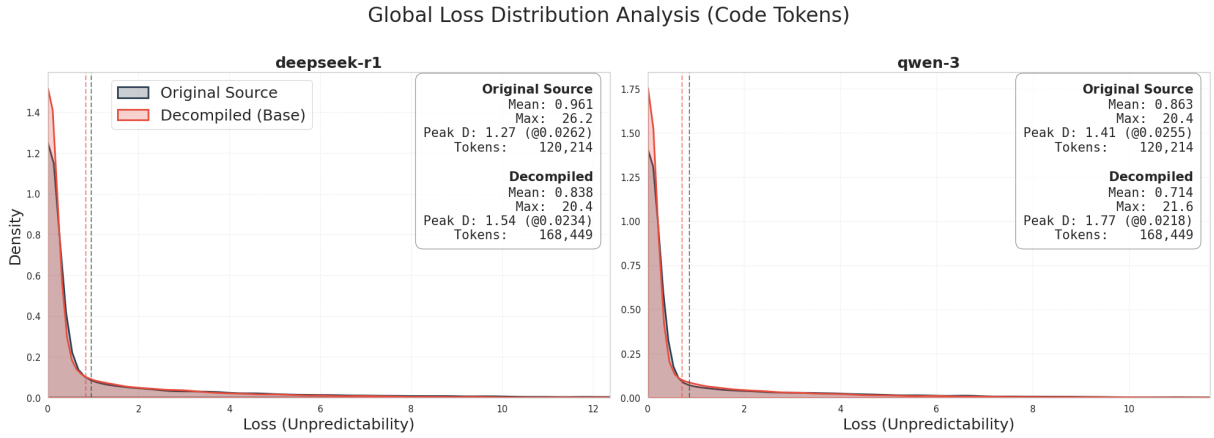


Figure 5.7: Density distribution of cross-entropy loss values for all tokens in the original source code and the decompiled versions.

In Figure 5.7 we can observe that the original code exhibits a wider, slightly flatter distribution with a higher mean loss compared to the decompiled output, which is characterized by a sharper peak shifted towards zero. This visualization highlights a counter-intuitive phenomenon: despite the original source code being the "human ground truth", the language model finds the decompiled code significantly more predictable.

We attribute this behavior to Token Inflation and Loss Dilution: As indicated by the to-

ken counts in the figure (e.g., ∼168k tokens for decompiled vs ∼120k for original source), the decompilation process introduces a substantial *token inflation*. Ghidra generate verbose, explicit code full of boilerplate structures (e.g., redundant casts, standard control flow patterns, explicit initializations, and restricted vocabulary). These pattern tokens are syntactically rigid and in a context where are used, they are easy to predict for the model, leading to a large number of tokens with very low loss values (close to zero). Their sheer volume effectively dilutes the mean loss, artificially lowering the overall perplexity score compared to the denser, more information-rich human code. In contrast, the original source code reflects human authorship, which includes domain-specific naming conventions, creative syntactic choices, and stylistic variability. This "human entropy" flattens the density curve and shifts the mean loss to the right, as the model is more frequently "surprised" by the programmer's unique choices compared to the machine's standardized output.

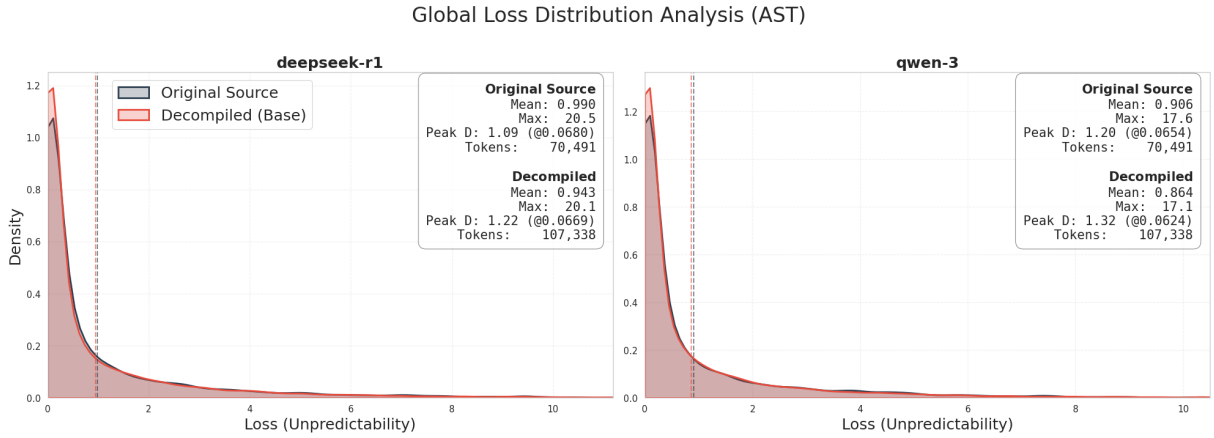


Figure 5.8: Density distribution of cross-entropy loss values for all tokens in the original source code and the decompiled versions.

In Figure 5.8 we can see the AST version of the previous analysis, where we abstracted away variable names and literals to focus on the structural aspects of the code. Comparing these results with the previous analysis on raw code tokens, we observe two critical phenomena:

1. **Persistence of structural inflation:** Even in the anonymized form, the *token inflation* remains significant. The decompiled AST contains ∼107k tokens compared to ∼70k for the original source (∼+52%). This confirms that the verbosity of the decompiled code is not merely lexical (e.g., long variable names) but syntactical. The decompiler introduces explicit casts, redundant blocks, and verbose control flow structures that persist even after anonymization, continuing to dilute the mean loss with predictable tokens.

2. **The closer entropy:** Unlike the raw code analysis, where the gap between the distributions was pronounced, the AST distributions for Source and Decompiled code are more similar in shape. The difference in Mean Loss has narrowed (e.g., for *deepseek-r1*, the gap represents only ~ 0.047, compared to larger margins in the raw code).

   This convergence suggests that the *lexical entropy* was a discriminator in the previous analysis.

   - In the raw code, human-written names provided high variability (surprisal), while decompiled names were generic.
   - In the AST version, the anonymization process effectively "standardizes" the two codes; Consequently, the source code becomes more predictable by the model.

The token-level analisys allow us to verify the tokens that contribute the most to the loss, and consequently to the perplexity, in a specific function. We can observe that the token itself is not the only factor that contributes to the loss, but especially the context in which it is used. For example, in Figure 5.9 we can see the loss values for the token "`LAB`", which is a common label used in the decompiled code to indicate jump targets. This token has a low loss value (1.664) when it appears after the `dowhile loop`, but it has a much higher loss value (11.812) when it appears inside the `if` allowing a flow branch to ignore the condition and entering the scope without checks.

```
    } while (uStack_38._4_2_ != 10);
LAB_00101b6a:
    if (__ptr != (ushort *)0x0) {
LAB_00101bb4:
      free(__ptr);
    }
  }
```

Figure 5.9: Loss values for `xls_parseWorkBook` in the decompiled base code

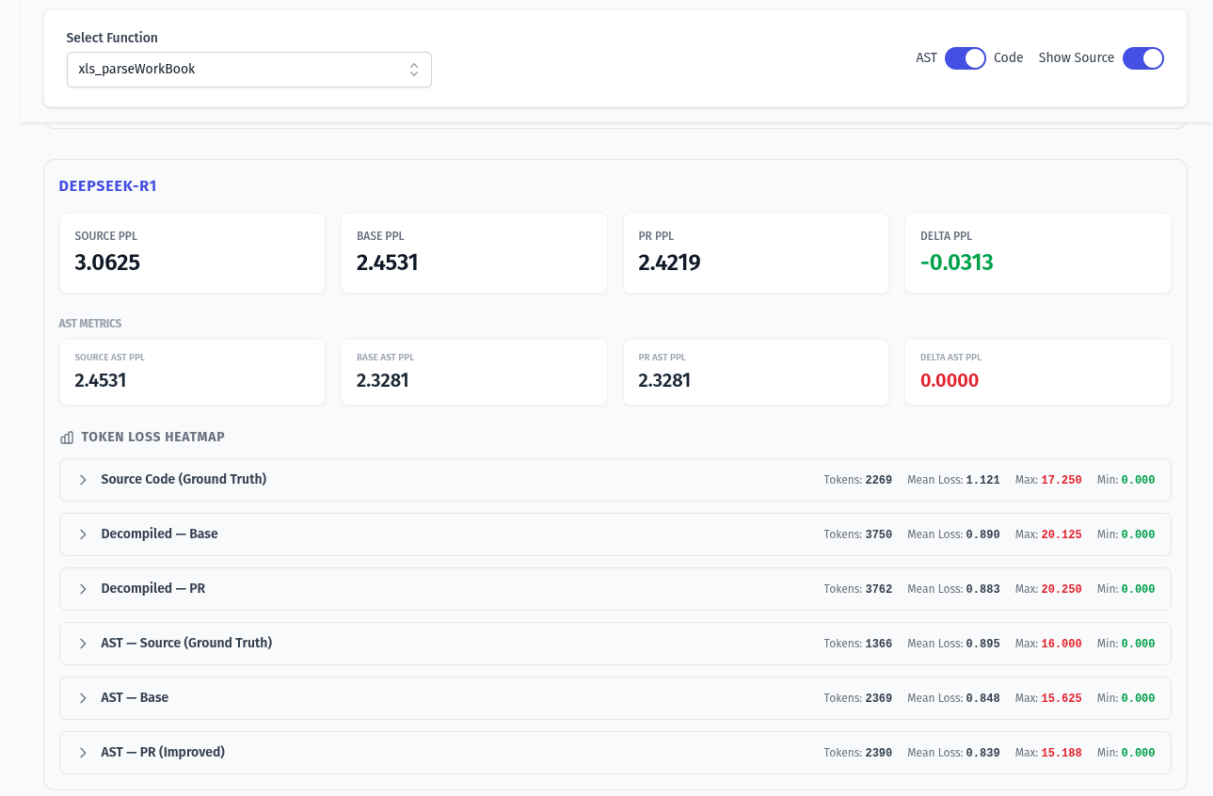


Figure 5.10: Example of loss values for a function in the decompiled code.

Obviously every function has a different distribution of loss values, and some functions may be more "natural" than others. In Figure 5.10 we can see an example of the loss values for `xls_parseWorkBook` function from the DeepSeek analisys, remembering that the perplexity is calculated as the exponential of the mean loss 3.8, we can see that the original source code has a perplexity of ~ 3.06 (mean loss ~ 1.12), while the decompiled base version has a perplexity of ~ 2.4 (mean loss ~ 0.89). We can see that the max values for the loss are higher for the decompiled version, in contrary to the global distribution where the decompiled code had a sharper peak towards zero, but then when we look at the anonymized version of the same function, the original source code became the one with a higher max loss value than the decompiled version. However, the anonymization process manages to bring the two versions significantly closer, both in terms of loss values and, consequently, perplexity.

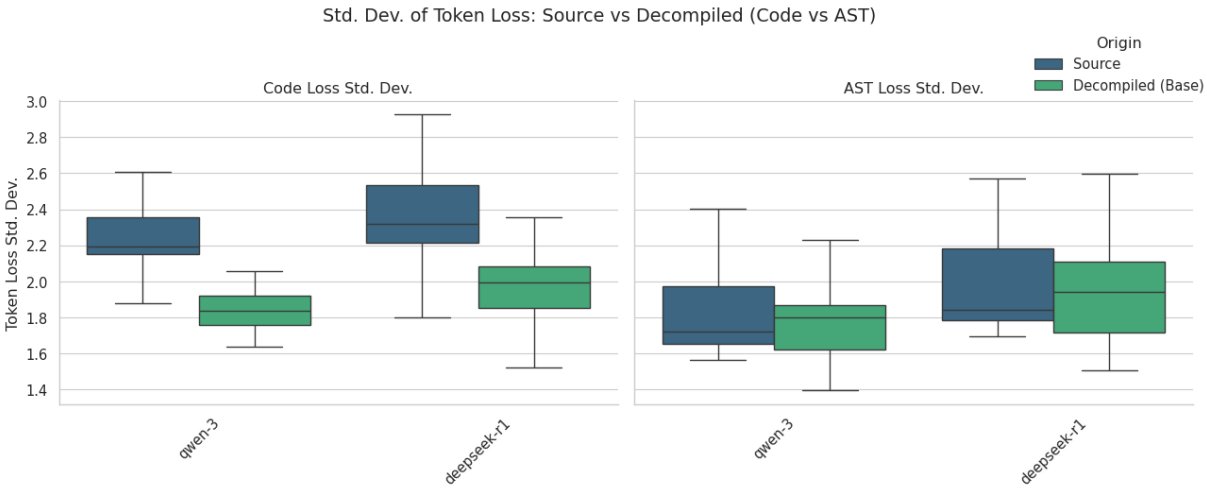


Figure 5.11: Standard deviation of loss values across all functions in the normal version and anonymized version.

In Figure 5.11 we investigate the *Entropy Variance* of the code by analyzing the standard deviation of the token loss. While the mean loss indicates the average predictability, the standard deviation reveals the *dynamic range* of the code complexity.

We can observe a clear trend across both models:

- **Difference in Variance:** The original source code consistently exhibits higher standard deviation compared to the decompiled versions. This confirms that human-written code is characterized by *burstiness*: it alternates between low-entropy boilerplate and high-entropy domain-specific logic. The language model struggles to predict this rhythm, leading to fluctuating loss values. The decompiled code shows significantly lower variance. This reflects the *monotonicity* of machine-generated code. In our case Ghidra applies consistent transformation rules throughout the binary, resulting in more predictabile results.

- **The Lexical Factor:** Comparing the code panel with the AST one, we notice that the gap between Source and Decompiled shrinks significantly in the AST representation. This implies that a substantial portion of the entropy variance in human code is driven by *lexical choices* (variable naming and literals) as we predicted. Once these are removed, the structural variability of human code is only marginally higher than that of the decompiled code.

This result reinforces our conclusion: in our case (Ghidra vs Source) Human-Likeness is not defined by raw predictability (where the machine wins), but by the **variance of**

unpredictability. A "natural" code signature is one that surprises the model in inconsistent, context-dependent bursts, rather than being uniformly predictable. Meanwhile for the anonymized code, the gap in variance is smaller but it still exists, suggesting that even at the structural level, human code retains a degree of unpredictability that machine-generated code lacks.

5.2.1 Other decompilers

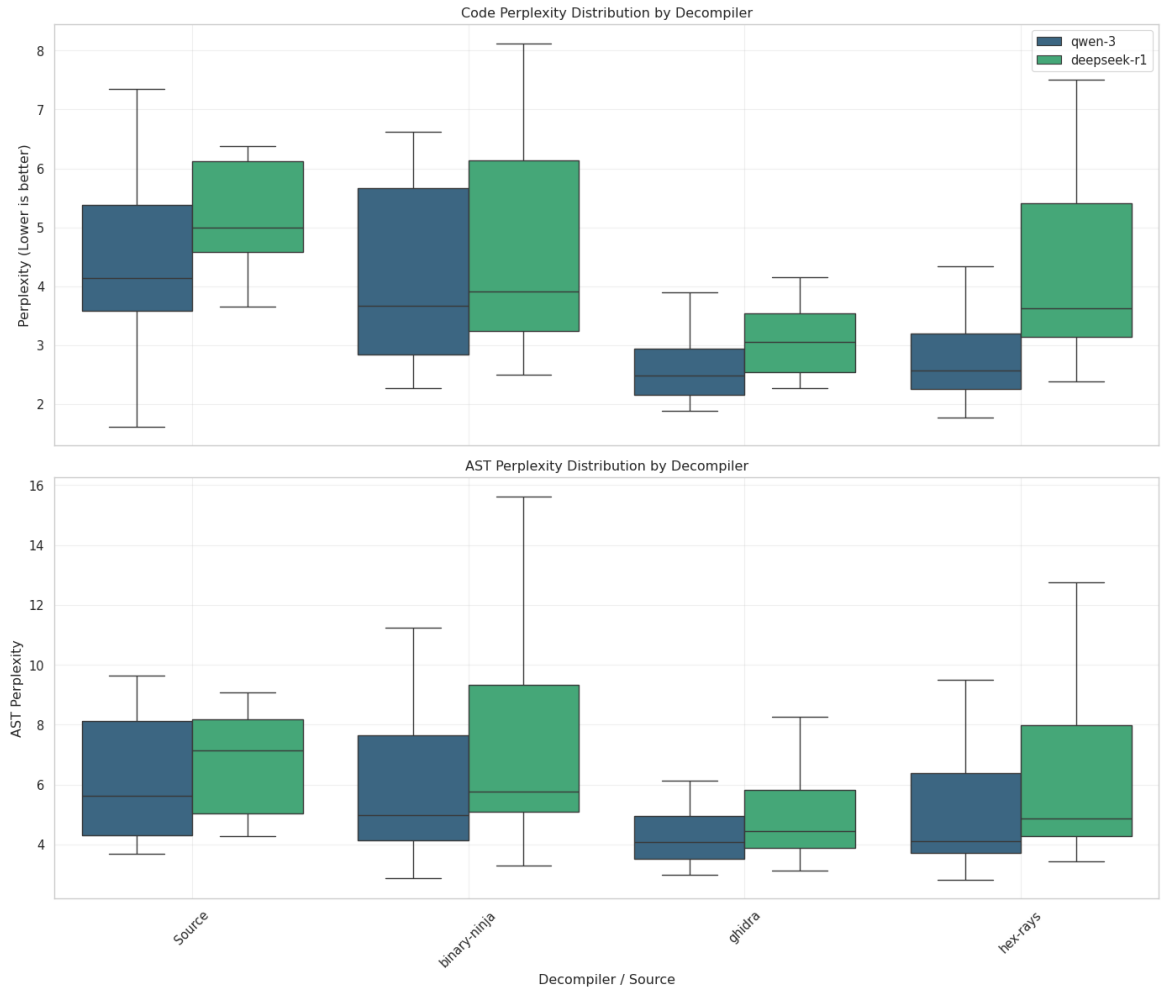


Figure 5.12: Perplexity values for the original source code and the decompiled versions from other decompilers.

The dogbolt analysis allows us to compare the perplexity values of the decompiled versions from other decompilers (Hex-Rays and Binary Ninja) with the original source code and the Ghidra decompilation, with a subset of the original database (we cannot compare in an absolute way the plot in Figure 5.6 with these but only the relative ordering of the distributions). We can see in Figure 5.12 that the perplexity values for the decompiled versions resulted by the dogbolt analysis, are generally in line with the Ghidra ones. we can see that the original source code still has higher perplexity values compared to the decompiled versions, and the abstracted representations of the code still tend to have higher perplexity values compared to their original counterparts. Another key observation is that the binary ninja has a perplexity distribution that is more similar to the original source code compared to the Ghidra decompilation, meanwhile Hex-Rays has a distribution in the middle between the other two decompilers, then we can observe that every distribution with *deepseek-r1* has a higher mean perplexity than the same distribution with *qwen-3* like we seen previously.

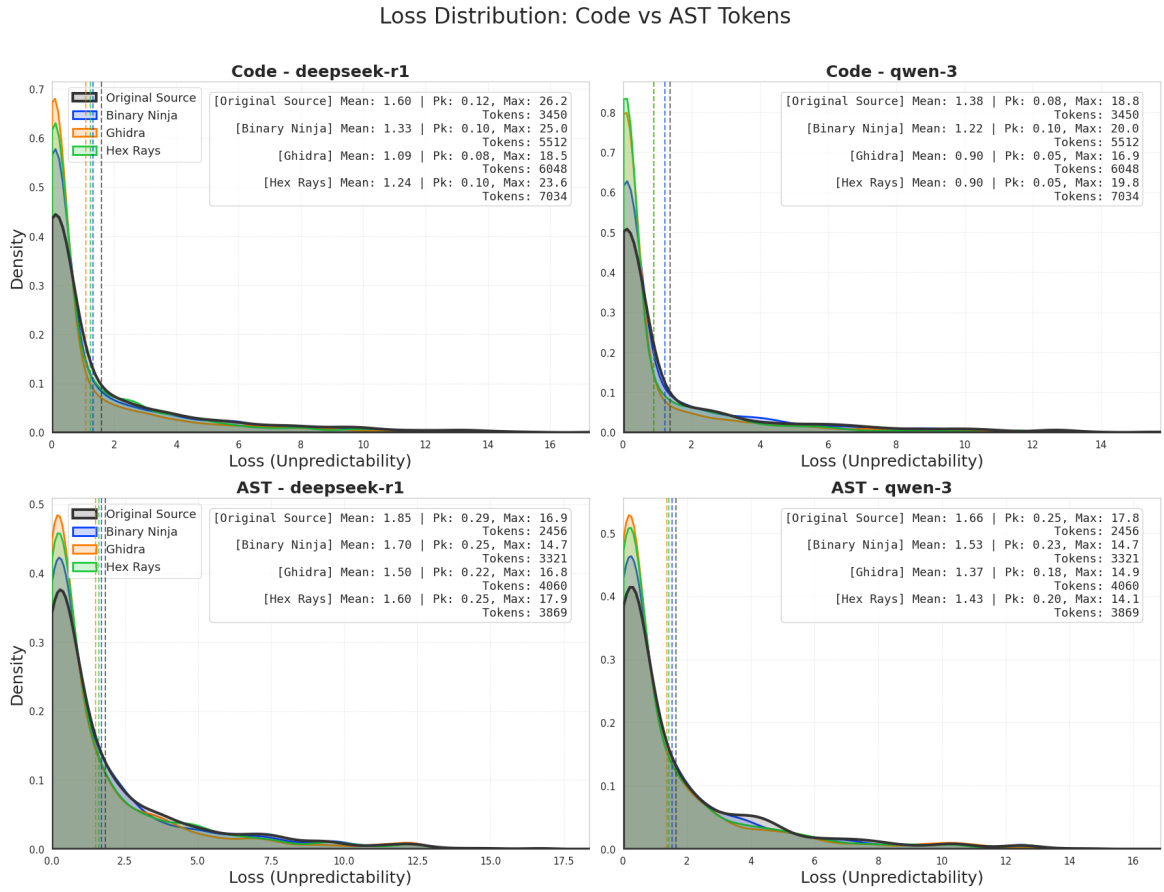


Figure 5.13: Loss values for the original source code and the decompiled versions from other decompilers. Code and Abstract

The density plots in Figure 5.13 confirm the phenomena observed in the previous sections across both LLM judges. Focusing on the raw *Code Tokens* (top row), we can observe a distinct hierarchy in predictability:

- **Ghidra** exhibits the sharpest peak near zero and the lowest mean loss ($\sim$ 1.09 for DeepSeek-R1 and $\sim$ 0.90 for Qwen-3). This reaffirms its tendency to generate highly rigid and predictable code.
- **Hex-Rays** follows closely, showing a similar sharp peak but with a slightly higher mean loss and a much larger token count (over 7,000 tokens in this sample), in particular with the *deepseek-r1* while with *qwen-3* we can see that the mean is the same as Ghidra but with a density at peak higher than the others (as the Max loss is much higher than Ghidra). The massive token inflation in Hex-Rays suggests extremely verbose syntactic and lexical choices that artificially dilute the cross-entropy loss.
- **Binary Ninja** stands out as the decompiler that most closely approximates the original source code. Its loss distribution is flatter, and its mean loss ($\sim$ 1.33 for DeepSeek-R1) is significantly closer to the original source ($\sim$ 1.60) than the other tools. It also exhibits less token inflation compared to Ghidra and Hex-Rays with a Max loss closer to the original source.

Meanwhile in the *AST* plots, where identifiers and literals are abstracted, we observe the expected flattening of all distributions. By stripping away lexical entropy, the gap between human-written code and machine-generated code narrows. Notably, the massive token inflation seen in Hex-Rays raw code drops significantly during AST abstraction (from 7,034 down to 3,869 tokens), indicating that a vast majority of its predictability stems from highly repetitive lexical tokens, types, and literal declarations rather than purely structural blocks. Even in the abstracted form, Binary Ninja maintains the distribution shape most similar to the original source code and with Hex-Rays having less tokens, the density at peak now is less than the Ghidra one for *qwen-3*. Finally, comparing the two models across all subplots, we note that while *qwen-3* consistently produces lower absolute loss values (higher confidence/predictability) than *deepseek-r1*, both models almost perfectly agree (except for the code tokens for *qwen-3* where Hex-Rays have a higher peak density than Ghidra) on the relative ordering of the distributions. Original source code always retains the highest mean loss and widest variance, followed by Binary Ninja, Hex-Rays, and finally Ghidra.

Figure 5.14: Heatmap of loss values for Hex-Rays decompilation with model *qwen-3*

Figure 5.15: Heatmap of loss values for Hex-Rays decompilation with model *deepseek-r1*

A token-level analysis of the Hex-Rays decompilation (e.g., Figure 5.14) for the model *qwen-3*, reveals that Hex-Rays generates comments and other repetitive tokens that are highly predictable. The most predictable tokens are often those associated with those artefacts of the decompilation process. These tokens create a dense cluster of low-loss values that significantly lower the overall perplexity score, despite the presence of more complex and less predictable tokens elsewhere in the code.

When we look at the same decompilation with the model *deepseek-r1* (Figure 5.15), we can see that the loss values are generally higher, but the distribution of low-loss tokens is still present, confirming that the anonimization process is a key factor in the analysis of the perplexity, as it standardizes the code through the removal of personalized identifiers, making the predictability of the code more dependent on its structural patterns rather than on specific lexical choices.

5.3 LLM-as-a-Judge Evaluation

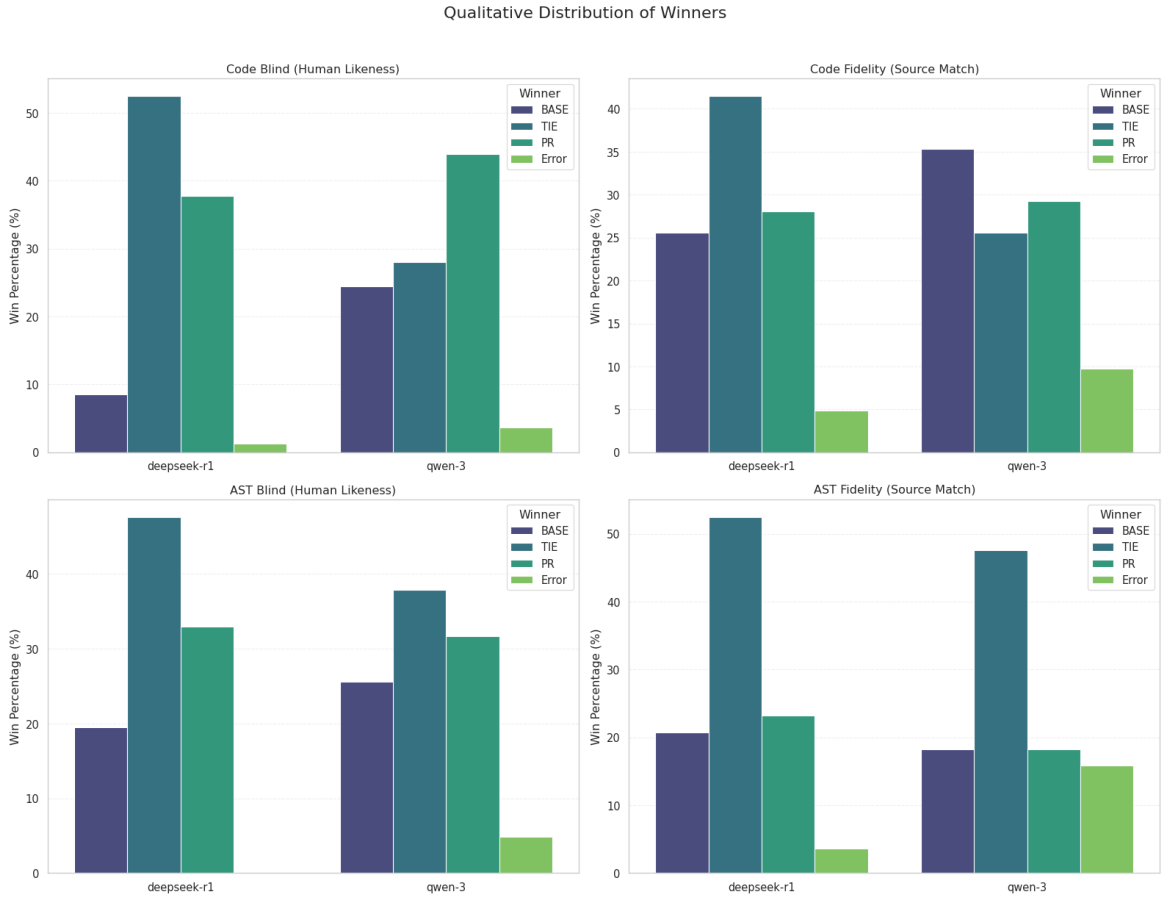


Figure 5.16: Distribution of winners across all evaluated models.

In Figure 5.16 we can see the distribution of winners across all evaluated models, based on the qualitative judgments of the LLM as a judge. The two models, Qwen3 and DeepSeek, have a significant number of Ties, but they also show a balanced distribution of wins between the base code and the pr code. We can see a bin for "Error" winners as well, this happens primarly when the model exceed the token limit of 4096 tokens (reasonable limit set by us), since they are reasoning models (they create a context with the generate tokens inside $< /think >$ tags) sometimes the context becomes too large (often because they start to repeating thoughts) and they finish the limit without giving inside the response a clear winner (e.g., "Winner":"X").

We previously said that a "Tie" is when the model judges always the same result regardless the switch of the base and pr code. We have a significant number of Ties for all models,

watching the result we can observe a ratio of 8.7 (618/71) beetween the number of times that the LLM prefers the PR version regardless the content. This suggests a strong bias towards the "newer" code, even without telling in the prompt that the Diff code is the newer one, this bias could be due to the fact that Diff code is often more recent and may contain improvements or bug fixes that make it more appealing to the model, or it could be a bias in the model itself towards preferring changes. Unfortunatly for highlight changes through versions and save on the context window the Diff method is the most convenient, forcing us to Do not count tie-break results in our analysis, giving up almost half of the results.

5.3.1 PR #8628

5.3.2 PR #8587

5.3.3 PR #8161

5.3.4 PR #7253

5.3.5 PR #6722

5.4 Correlation Perplexity & LLM

5.5 Vs Human Evaluation

5.6 Discussion

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

[Atm+25] Hannah Atmer, Yuan Yao, Thiemo Voigt, and Stefanos Kaxiras. *Prefill vs. Decode Bottlenecks: SRAM-Frequency Tradeoffs and the Memory-Bandwidth Ceiling*. 2025. arXiv: 2512.22066 [cs.AR]. URL: https://arxiv.org/abs/2512.22066.

[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].

[BEP25] Eva-Maria C. Behner, Steffen Enders, and Elmar Padilla. "SoK: No Goto, No Cry? The Fairy Tale of Flawless Control-Flow Structuring". In: (2025), pp. 411–431. DOI: 10.1109/EuroSP63326.2025.00032.

[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.

[Dee25] DeepSeek-AI. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[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].

[Son] SonarSource. *Cyclomatic Complexity Guide — How To Calculate & Test — sonarsource.com*. https://www.sonarsource.com/resources/library/cyclomatic-complexity/. [Accessed 10-02-2026].

[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.

[Ten] Tencent. *Qwen/Qwen3-14B · Hugging Face — huggingface.co*. https://huggingface.co/{Q}wen/{Q}wen3-14{B}. [Accessed 13-02-2026].

[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].