



**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 Ribaudò

January, 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

<b>Chapter 1</b>	<b>Introduction</b>	<b>6</b>
1.1	Style and Language . . . . .	6
1.2	Motivation . . . . .	8
1.3	Content of the Thesis . . . . .	8
1.4	L <sup>A</sup> T <sub>E</sub> X Tricks . . . . .	8
<b>Chapter 2</b>	<b>Related Work</b>	<b>11</b>
2.1	Generative Refinement of Decompiler Output . . . . .	11
2.2	LLM-based Benchmarking . . . . .	12
2.3	Why Perplexity . . . . .	13
<b>Chapter 3</b>	<b>Background</b>	<b>14</b>
3.1	Ghidra . . . . .	14
3.2	SLEIGH and P-code . . . . .	15
3.2.1	P-code Semantics and Varnodes . . . . .	15
3.3	The Decompilation Pipeline . . . . .	16
3.3.1	Actions and Rules . . . . .	16
3.3.2	DefaultGroups . . . . .	17
3.4	Logic of Control Flow Structuring . . . . .	18
3.4.1	Basic Block Formulation . . . . .	18
3.4.2	The Structuring Algorithm . . . . .	22

3.4.3	The <i>for</i> special case . . . . .	25
3.4.4	The <i>Goto</i> Problem . . . . .	27
3.5	Code Emission . . . . .	27
3.6	Large Language Models . . . . .	28
3.6.1	Transformer . . . . .	28
3.6.2	Tokens . . . . .	30
3.6.3	Softmax . . . . .	31
3.6.4	Quantization . . . . .	32
3.7	Decoding . . . . .	32
3.7.1	Temperature . . . . .	32
3.7.2	Top-p and Top-k . . . . .	33
3.8	Perplexity . . . . .	34
3.9	Human-like . . . . .	35
<b>Chapter 4</b>	<b>Methodology</b>	<b>36</b>
<b>Chapter 5</b>	<b>Results</b>	<b>38</b>
5.1	Discussion . . . . .	39
<b>Chapter 6</b>	<b>Conclusion</b>	<b>40</b>

# Chapter 1

## Introduction

This is a template for the Computer Science Master’s Theses at *dipartimento di informatica, bioingegneria, robotica e ingegneria dei sistemi* (DIBRIS), University of Genoa.<sup>1</sup> It uses the L<sup>A</sup>T<sub>E</sub>X class `masterthesis.cls` by Davide Ancona. We will use this template to discuss what is generally expected in the structure of a Master’s Thesis. While reading academic documents, you will find that they generally follow a similar structure.

This structure is generally good for research documents, where your main goal is to do something new. In other cases (e.g., the goal of your thesis is to review a topic and provide an understanding of the field), a different structure may be more desirable. In any case, discuss the structure of your thesis with your advisor.

I have put hints and links through the document for tools you can use; some may become stale over time, and something better may appear. If you find some that are not covered here, consider sending me an email or a pull request to integrate them!

### 1.1 Style and Language

We are writing a scientific document, and we should consider that skepticism is at the core of the scientific method. We should be skeptical of our work, and we can see the thesis as an effort to persuade a skeptical reader that we have done a piece of work that is both useful and solid. Whenever you state something, you should give proof: either by referencing one or more authoritative documents or proving it yourself. Avoid marketing-speak (“our work is poised to revolutionize...”).<sup>2</sup>

---

<sup>1</sup><https://dibris.unige.it>

<sup>2</sup>Unless you can prove what you say. But these statements are mostly impossible to prove... By the way, notice that footnotes should be put after punctuation as in this case.

Do not try to reach a given number of pages. Instead, try to make it *complete*, *clear* and *concise*. Try to give all the information that will convince the reader that your work is solid and that you have done a good job; use simple language and consistent terminology. If you can say something more simply, do it. Avoid weasel words<sup>3</sup> and, when you can, the passive voice. “We consider this to be not a security risk” is much clearer than “This is not considered a security risk” because it makes it explicit who makes the claim, and takes the responsibility for it.

Consider that your reader will be a fellow computer scientist, but not necessarily an expert on your topic. Try to give them the information they need to understand your work, but do not lose your reader’s time and attention with unnecessary details.

Documents like this generally follow a structure like this one. It is not a strict requirement, but if you follow it readers will likely find it easier to navigate. Consider that in scientific documents some readers will not read the whole document, but will look for specific pieces of information. A clear structure will help them.

Chapters generally start with a short introduction to their topic. This is followed by the main content of the chapter, and then by a conclusion that summarizes the main points. Note that both chapter and section titles are capitalized.<sup>4</sup>

Introduce the expanded form of acronyms the first time you use them, excepting only those so common that you can reasonably take for granted for any reader (e.g., CPU or PDF).

For this document, you should use formal writing. Do not use contractions<sup>5</sup> such as *don’t* or *it’s*.

Be extremely wary of large language models (LLMs) like Gemini, ChatGPT, or Copilot.<sup>6</sup> You take responsibility for the content of your thesis and declare you have written it yourself. If you want a tool that helps you to write in better English, consider using something like Grammarly<sup>7</sup>; it is also supported by the Overleaf online L<sup>A</sup>T<sub>E</sub>X editor.<sup>8</sup> Note that Overleaf now has a generative AI option: do not use it without declaring it, as it would violate its terms of service<sup>9</sup> which, among other things, forbids “to mislead any third party that any output from Grammarly’s generative AI was solely human generated”.

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Weasel\\_word](https://en.wikipedia.org/wiki/Weasel_word)

<sup>4</sup><https://www.grammarly.com/blog/capitalization-in-the-titles/>

<sup>5</sup><https://www.grammarly.com/blog/contractions/>

<sup>6</sup>Note that we used the serial comma ([https://en.wikipedia.org/wiki/Serial\\_comma](https://en.wikipedia.org/wiki/Serial_comma)) here. It is a good idea to use it as it can remove ambiguity from your text.

<sup>7</sup><https://grammarly.com>

<sup>8</sup>[https://www.overleaf.com/learn/how-to/Use\\_Grammarly\\_with\\_Overleaf](https://www.overleaf.com/learn/how-to/Use_Grammarly_with_Overleaf)

<sup>9</sup><https://www.grammarly.com/terms>

## 1.2 Motivation

This is the first important thing you should write in your Introduction. You can see it as the effort to convince your reader that the problem you are addressing *matters*. Mention that the problem is unsolved, but you will discuss that at you’re facing a skeptical reader asking “is this useful?”.

Sometimes it will be obvious why the problem matters, but in other cases, this will be one of the most delicate parts of your work. Of course, your advisor will help you with this.

## 1.3 Content of the Thesis

The other main goal of the Introduction is to give the reader an overview of what they will find in the document. There is no concept of “spoiler” in scientific documents: summarize what the reader will find in every chapter. Avoid obvious statements that help nobody such as, for example, “*In the Related Work chapter we discuss relevant pieces of work in the state of the art*”; provide synthetic but useful information instead (e.g., “*In Chapter 2 we discuss other density-based clustering algorithms and highlight the differences to our approach*”). This will help them to navigate it (and to find what they’re interested in, if they are reading the whole document).

I am not a fan of those “*This document is structured as follows*” paragraphs. My advice is to expose the content of the thesis and give references, organically, to the parts of it that treat them while you are at it.

The Introduction is arguably the most important chapter of an academic text because, after reading it, one should understand the key concepts of your work. The rest of the document “only” explains the details; you can expect somebody to stop at the Introduction alone.<sup>10</sup>

## 1.4 L<sup>A</sup>T<sub>E</sub>X Tricks

If you’re using L<sup>A</sup>T<sub>E</sub>X to write your document, you can use some tricks to make your life easier. This document uses the `\Cref`<sup>11</sup> command to conveniently reference sections, figures, and tables, and `\textcite` and `\authorcite` to cite paper authors. You can also

---

<sup>10</sup>Likewise, while you are going through somebody else’s work, it makes sense to stop at the Introduction (and maybe the Conclusion) if you’re interested only in a high-level overview.

<sup>11</sup>I also created a `\fullref` to generate text like “**Introduction** (Chapter 1)”. Check the source if you are interested.



use `\xspace` to define your macros and avoid spacing problems. Check the source code of this document to see how they are used.

`rubber` is a good tool to compile  $\text{\LaTeX}$  documents. You can run the command `rubber -d main` to compile this document to PDF, while running `pdflatex` and `biber` (or `bibtex`) the right number of times.

The Overleaf<sup>12</sup> online editor is a good tool to synchronize with your advisor; it exports a git repository so you can edit your work online and offline.

To handle acronyms, the `acronym`<sup>13</sup> package is a good resource to automatically introduce an acronym's expanded form the first time you use it. It is also possible to create an acronyms table; consider using it if you use many acronyms in your paper and you think it may help the reader. Once again, you can have a look at how it's used in the source of this document.

You are likely going to cite several websites in your bibliography. Here is documentation<sup>14</sup> on how to do that in both BibTeX and BibLaTeX. You may want to use a reference management software like Zotero<sup>15</sup> or Mendeley<sup>16</sup> to handle your bibliography database.

Your supervisors and reviewers will likely go over your document in several revisions; I'm sure they would really be thankful to you if you used a way to clearly show what changed since their previous pass. `Latexdiff`<sup>17</sup> is a tool to do just that; you can also use it on Overleaf<sup>18</sup>. Another possibility is tracking the changes on the PDF output, using tools like Draftable<sup>19</sup>.

Advisors may have very different preferences in how to share feedback. A possibility is the `todonotes`<sup>20</sup> package. This document's `preamble.tex` source file shows an example of using it to include comments; note that they can be removed from the PDF output by passing the `final` option to the document class (i.e., `\documentclass[final]{masterthesis}` in the main `tex` file.

MD: This is an example of a note using `todonotes`.

**IMPORTANT:** Make sure the curriculum on the cover pages and the department address matches your own. Currently, you'll have to edit the `masterthesis.cls`

---

<sup>12</sup><https://www.overleaf.com>

<sup>13</sup><https://ctan.org/pkg/acronym>

<sup>14</sup><https://bibtex.eu/faq/how-can-i-use-bibtex-to-cite-a-website/>

<sup>15</sup><https://www.zotero.org/>

<sup>16</sup><https://www.mendeley.com/>

<sup>17</sup><https://ctan.org/pkg/latexdiff>

<sup>18</sup>[https://www.overleaf.com/learn/latex/Articles/How\\_to\\_use\\_latexdiff\\_on\\_Overleaf](https://www.overleaf.com/learn/latex/Articles/How_to_use_latexdiff_on_Overleaf)

<sup>19</sup><https://www.draftable.com/>

<sup>20</sup><https://tug.ctan.org/macros/latex/contrib/todonotes/todonotes.pdf>

file accordingly; I plan to eventually provide a class option to facilitate that without the need to edit the `.cls` file. If you feel like you want to help me with that, please let me know! You'll get gratitude and an acknowledgment here.

# 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 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. 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] which 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." [Li+25]

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 code "humanness". Perplexity provides a more objective, quantifiable measure of how "natural" the code appears to the model, rather than relying on subjective ratings.[3.8][2.3]

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.

## 2.3 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.

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  $S = t_1, t_2, \dots, t_N$ , it assigns a probability to each token based on the preceding context. 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 its 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. 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 <i>dest</i> if <i>cond</i> is non-zero.

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

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

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

### 3.3 The Decompilation Pipeline

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

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

#### 3.3.1 Actions and Rules

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



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

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

### 3.3.2 DefaultGroups

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

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

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

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

Listing 3.1: ghidra\_process.cc

---

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

---

## 3.4 Logic of Control Flow Structuring

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

### 3.4.1 Basic Block Formulation

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

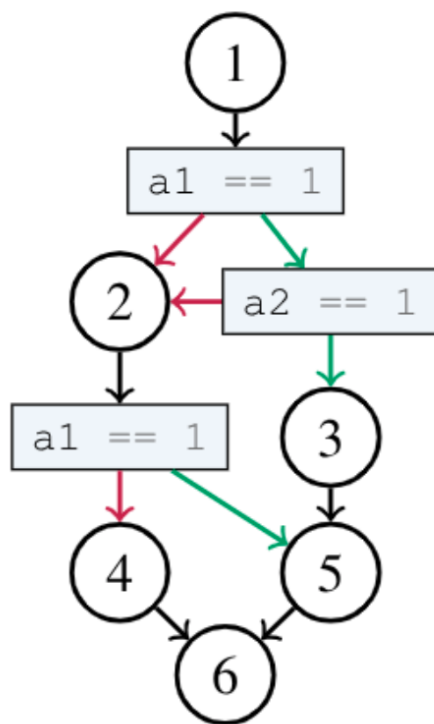


Figure 3.1: Control Flow Graph of a function

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

P-code representation <sup>1</sup>.

Source Code (C)	P-Code / Basic Blocks
<pre> 1  int a2_local; 2  int a1_local; 3  putchar(L'1'); 4  if ((a1 == 1) </pre>	<p><b>Basic Block 0</b></p> <pre> 0x0010118d:1: RSP(0x0010118d:1) = RSP(i) + #0xffffffffffffff8 0x0010118d:2: *(ram,RSP(0x0010118d:1)) = RBP(i) 0x00101195:d: u0x00004780(0x00101195:d) = RSP(i) + #0xffffffffffffff4 0x00101195:f: *(ram,u0x00004780(0x00101195:d)) = EDI(i) 0x00101198:10: u0x00004780(0x00101198:10) = RSP(i) +                 #0xfffffffffffff0 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) +                 #0xffffffffffffff4 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 </pre>
<pre> 1     (a2 != 2)){ </pre>	<p><b>Basic Block 1</b></p> <pre> 0x001011ab:24: u0x00004780(0x001011ab:24) = RSP(i) +                 #0xffffffffffffff0 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 </pre>

<sup>1</sup>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
<pre> 1 putchar(L'2'); 2 if (a1 != a2) { </pre>	<p><b>Basic Block 2</b></p> <pre> 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) +     #0xfffffffffffff4 0x001011c7:4a: u0x00011e80:4(0x001011c7:4a) =     *(ram,u0x00004780(0x001011c7:49)) 0x001011ca:4d: u0x00004780(0x001011ca:4d) = RSP(i) +     #0xfffffffffffff0 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 </pre>
<pre> 1 putchar(L'4'); 2 goto LAB_001011e5; </pre>	<p><b>Basic Block 3</b></p> <pre> 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 </pre>
<pre> 1 } else { 2     putchar(L'3'); 3 } </pre>	<p><b>Basic Block 4</b></p> <pre> 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 </pre>
<pre> 1 putchar(L'5'); 2 } </pre>	<p><b>Basic Block 5</b></p> <pre> 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)) </pre>

Source Code (C)	P-Code / Basic Blocks
<pre> 1 LAB_001011e5: 2 putchar(L'6'); 3 return; 4 } </pre>	<p>Basic Block 6</p> <pre> 0x001011ea:3c: RSP(0x001011ea:3c) = RSP(i) + #0xffffffffffffffe0 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) </pre>

Basicblocks are created in `flow.cc` by the function `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

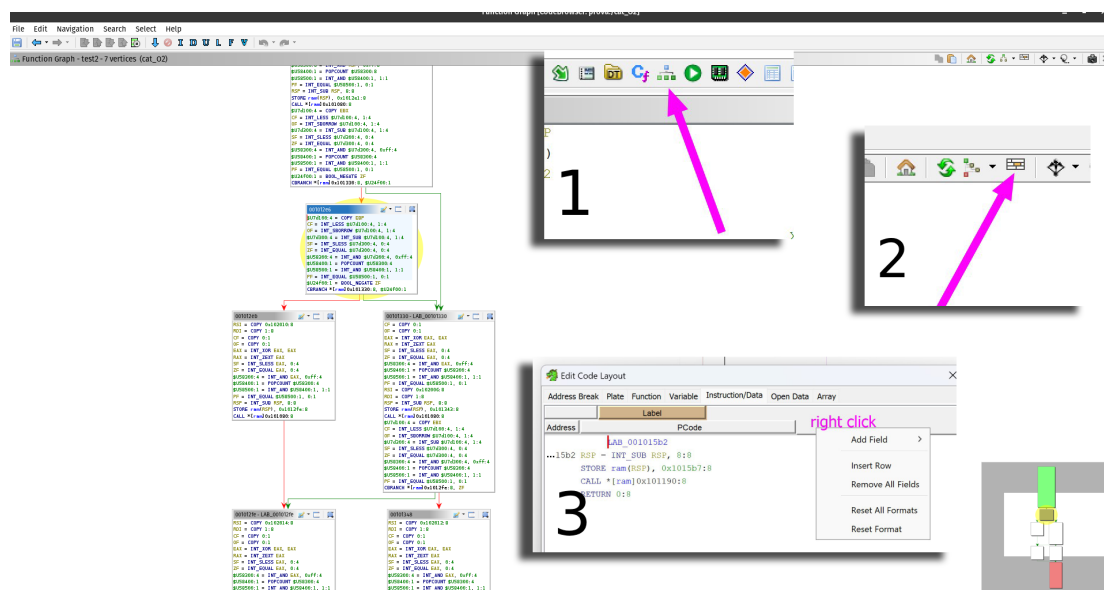


Figure 3.2: Control Flow Graph with P-code in Ghidra

### 3.4.2 The Structuring Algorithm

To transform the CFG into C statements, Ghidra employs a structuring algorithm implemented in the `ActionBlockStructure` class. 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`

This "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 (or `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()`, the matched `BasicBlocks` are replaced with a single `FlowBlock` representing the high-level construct (e.g., a `BlockWhileDo` for a while loop). This new block encapsulates the original blocks, 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 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<sup>2</sup>: 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.

---

<sup>2</sup>The transformation is triggered only if the architecture option `analyze_for_loops` is enabled.

3. **Opcode 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 Flow-Block remains and it will be represented as a **goto**<sup>3</sup>. 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.

---

<sup>3</sup>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  $(t_1, t_2, \dots, t_n)$ , accumulating error and dispersing the gradient over long sequences, the Transformer processes the entire sequence simultaneously, relying entirely on the attention mechanism.

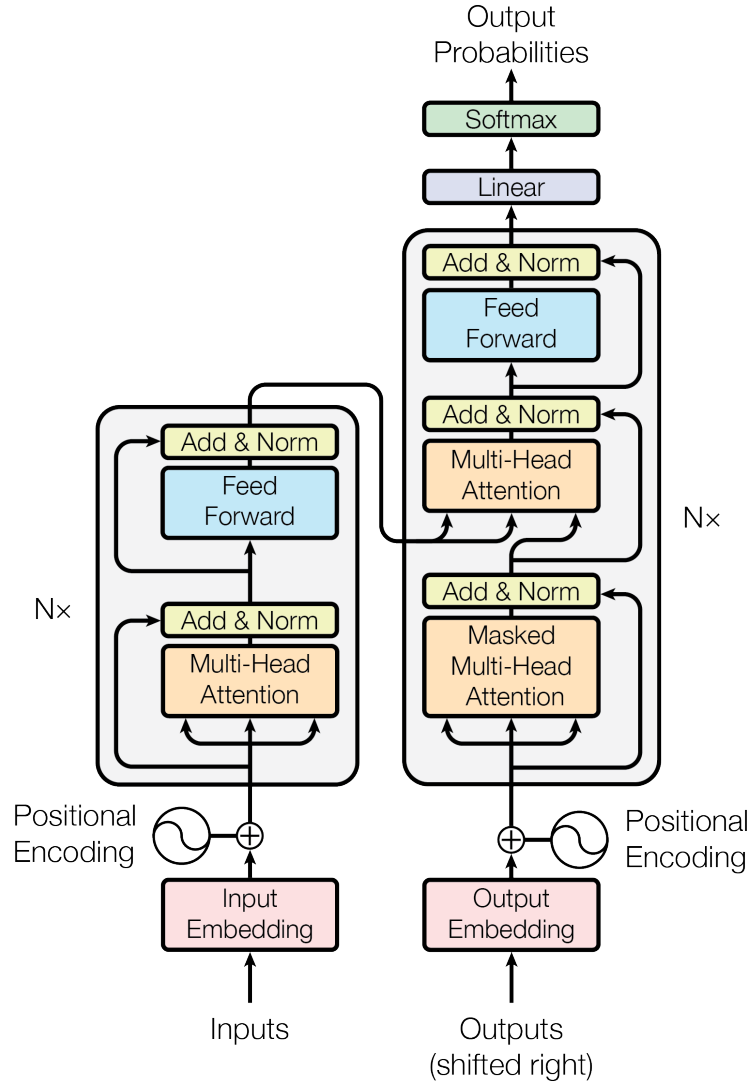


Figure 3.3: The Transformer - model architecture

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  $x = (x_1, \dots, x_n)$  and produces continuous representations  $z = (z_1, \dots, z_n)$ . The **decoder** (right) conditions on  $z$  and generates the output tokens  $y = (y_1, \dots, y_m)$  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<sup>4</sup>, a webtool for visualizing tokenization for different models, to tokenize a sentence:

---

<sup>4</sup><https://tiktokenizer.vercel.app/>

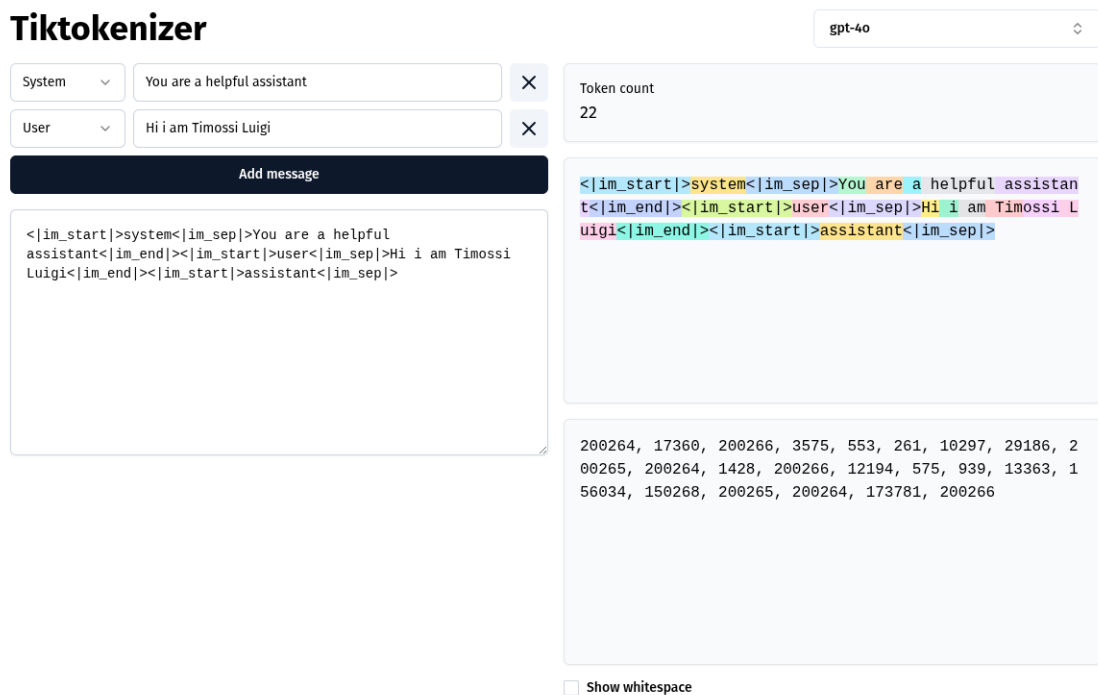


Figure 3.4: Tokenization example using tiktokenizer

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]

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

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

### 3.6.4 Quantization

## 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:

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



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).
- $T \rightarrow 0$ : 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  $p$ .[\[wikib\]](#)

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

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

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.[\[Mor\]](#)

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

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:  $\text{PPL} = 2^{H(P)}$ , 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 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

This chapter describes the core of your work: the methodology you used to solve the problem you described in the **Introduction** (Chapter 1). It is quite likely to have a different name (e.g., the name of the tool you wrote), and/or to be split into multiple chapters. *Methodology* suggests that you will spend space discussing your overall approach and justification for it: it is a good idea for a Master's Thesis, as opposed to just a list of things that you have done (which could be more correctly called *Method* or *Methods*, and would risk being boring and not very useful for the reader).

This chapter will likely be quite densely connected with things you have written in the **Background** (Chapter 3), and the results that you will show in the **Results** (Chapter 5). Put abundant references to help the reader navigate the document.

Try to keep a level of detail such that a skilled reader will be able to reproduce your result, but do not include unnecessary details that the reader can find out for themselves (e.g., the list of commands to perform a standard task). That is indeed valuable information that can help reproducibility, but if you do not think it is needed to understand your work, its place is likely to be in an appendix or in the documentation of the code you wrote, if you decide to release it.

This chapter should include work that has been done by you. If that is not the case, consider moving it to the **Background** (Chapter 3), or make it very clear that you are not the author. The same thing applies to images: if they aren't yours, you should cite the source.

The results of your work should generally not be included in this chapter. In some cases, preliminary results can be included to explain why you took a certain direction in your work (e.g., you took a choice rather than another because you measured that it was more efficient), but the overall evaluation of your work should be in the **Results** (Chapter 5).

You are likely to include algorithms in pseudocode. Check relevant L<sup>A</sup>T<sub>E</sub>X packages<sup>1</sup> for that.

---

<sup>1</sup><https://www.overleaf.com/learn/latex/Algorithms>

# Chapter 5

## Results

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

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

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

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

If you are using L<sup>A</sup>T<sub>E</sub>X, it will automatically place figures, tables, algorithms, etc. somewhere in the text. These parts of the documents are called *floats* because their position

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

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

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

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

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

## 5.1 Discussion

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

# Chapter 6

## Conclusion

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

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



# Bibliography

- [Bai] Yumo Bai. *Why are most LLMs decoder-only?* — yumo-bai. <https://medium.com/@yumo-bai/why-are-most-llms-decoder-only-590c903e4789>. [Accessed 25-01-2026].
- [Gao+25] Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Bolun Zhang, and Chao Zhang. *DecompBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios*. 2025. arXiv: [2505.11340](https://arxiv.org/abs/2505.11340) [cs.SE]. URL: <https://arxiv.org/abs/2505.11340>.
- [Git] GitHub. *GitHub Actions* — github.com. <https://github.com/features/actions>. [Accessed 26-01-2026].
- [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](https://arxiv.org/abs/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](https://doi.org/10.21105/joss.00655). URL: <https://doi.org/10.21105/joss.00655>.
- [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](https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html). [Accessed 19-01-2026].
- [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].
- [Tan+24] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. “LLM4Decompile: Decompile 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](https://doi.org/10.18653/v1/2024.emnlp-main.203). URL: <http://dx.doi.org/10.18653/v1/2024.emnlp-main.203>.
- [Vas+23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [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](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].