

UNIVERSITY OF FLORENCE
IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Department of Statistics, Informatics, Applications
“Giuseppe Parenti”

Second Level Master in Data Science and Statistical Learning

MD2SL

A PIPELINE TO VALIDATE AGENTIC AI PARSER GENERATION

Supervisor:
Prof. FABIO PINELLI

Candidate:
MIRKO VERARDO

Co-supervisor:
Dr. LETTERIO GALLETTA

ACADEMIC YEAR 2024-2025

Abstract

The advent of large language models (LLMs) is profoundly transforming software engineering. If software is developed to automate processes and software development is itself a process, the next step will be automating it through the use of LLMs. This new AI-assisted software development paradigm is often seen with skepticism by programmers. However, software development is not a subjective activity. On the contrary, it's characterized by a lot of measurable parameters regarding security vulnerability, code quality, test coverage and others. Based on these considerations, it's appropriate to shift the focus from *software development* to *software validating*.

AI-assisted coding can be seen as a higher-level programming paradigm, which involves writing directly specifications rather than writing code. Specifications are written through a structured prompt that is the main input for LLMs, which will automatically generate the corresponding code.

Evaluating this paradigm requires that the generated code must be systematically testable by a validation pipeline that executes all the predefined acceptance tests and objectively measures quality metrics.

Therefore, it's easy to understand that the objectives of AI-assisted coding are the same as those of traditional human coding. The two paradigms are potentially comparable through the same validation pipeline, which becomes crucial to establish the software correctness. Also, while validation pipelines already represent a best practice in human coding, they become mandatory with AI-assisted coding.

The aim of this thesis is implementing two different agentic AI systems that generate and test code through an automated validation pipeline. Then, a comparison between the two systems is proposed through the metrics collected by the pipeline itself.

A specific case study has been considered for both these systems: the development of parsing functions in the C programming language.

Introduction

1 Problems

Parsing functions are particularly crucial for vulnerability assessment since they are usually target of many security issues due to input complexity, ambiguous specifications and implementation errors.

The study of parsing functions is the central topic of **ParserHunter** work [1] to which this thesis is directly related. ParserHunter research presented a tool trained to automatically recognize and classify parsers for security assessment purposes. However, the effectiveness of this tool can be limited due to the lack of adequate training data. Therefore, this thesis illustrates two agentic AI systems for automatically generating synthetic parsing functions in the C programming language. The aim is to increase the existing parsers dataset size for tools like ParserHunter with heterogeneous parser implementations that commit to be successfully compiled and tested, reducing the current manual effort.

Also, this thesis is an extension of the **LLM-based parser generation** work [2] where some agentic AI systems are presented too. However, those systems showed some critical compilation rate issues and did not have a real testing pipeline with code generation metrics tracking. Another aim that this thesis addresses.

2 Objectives

The objective of this thesis is implementing two different agentic AI systems for parsing functions generation in the C programming language. The parsers generated must follow predefined specifications.

These systems include an automated validation pipeline that compiles and tests the generated code. In addition, some metrics on the generation process are tracked by the pipeline for a comparison between the two systems and a vulnerability assessment of the generated code.

The work for this thesis has been done during an internal research activity in collaboration with the **Systems Security Modeling and Analysis (SySMA)** research unit at IMT school for advanced studies Lucca.

3 Results

In the experimental part of the thesis, some benchmarks have been run on the validation pipeline initialized with different parameters. During each independent round, some metrics have been tracked about the code generation process and the vulnerability assessment of the generated code. In addition, some preliminary analyses have been performed on these collected results.

The used approach extends other state of the art benchmarks for AI coding like **SWE-bench (SoftWare Engineering benchmark)** where performance on code generation task is evaluated only through a success rate.

The collected results show how the developed agentic AI parser generation systems perform quite well on compilation and testing rates according to the validation pipeline. More precisely, compilation rates obtained are far better compared to [2]. In addition, preliminary analyses on code generation performance show how different architectures have better performance on different metrics.

4 Thesis structure

The thesis is structured in 4 chapters.

In Chapter 1, an overview on theoretical concepts involved on the thesis is illustrated. The case study of the thesis, i.e. parsing functions in C programming language, and agentic AI systems are introduced, with particular reference to Large Language Models and the state of the art of the agentic AI paradigms used in experimental part.

In Chapter 2, the frameworks LangChain and LangGraph used in experimental part are introduced. Then, the validation pipeline for the parsing functions generation is illustrated, with particular reference to the code analysis profiles and agentic AI system architectures used.

In Chapter 3, experimental results and evaluation methods are described for benchmarks on the validation pipeline run with different parameters. More precisely, some preliminary analyses has been performed on different metrics. In Chapter 4, at last, there are some final considerations on the work done and possible future developments are illustrated.

Contents

Abstract	iii
Introduction	v
1 Problems	v
2 Objectives	v
3 Results	vi
4 Thesis structure	vi
List of Figures	ix
1 Overview	1
1.1 Parsing functions in C language	1
1.1.1 Specifications	2
1.1.2 GNU Compiler Collection	2
1.2 Agentic AI systems	3
1.2.1 Large Language Models	4
1.2.2 Reasoning paradigms	5
1.2.3 Tools calling	5
1.2.4 Memory systems	6
2 Methodology and Implementation	7
2.1 Agentic AI frameworks	7
2.1.1 LangChain	7
2.1.2 LangGraph	7
2.2 The validation pipeline	8
2.2.1 Static analysis profile	11
2.2.2 Dynamic analysis profile	12
2.2.3 Single-agent architecture	14
2.2.4 Multi-agent architecture	17

3	Experimental results and benchmarks	23
3.1	Environment	23
3.1.1	Experiment description	23
3.2	Benchmark metrics	25
3.2.1	Compilation	25
3.2.2	Testing	25
3.2.3	Execution time	25
3.2.4	Cyclomatic complexity	25
3.2.5	Code coverage	26
3.3	Overall results	27
3.3.1	Correlation heatmap	27
3.3.2	Compilation and Testing rates	28
3.4	Research question	29
3.4.1	Evaluation methods	29
3.4.2	RQ: Agentic AI architecture	31
4	Conclusion	37
4.1	Final considerations	37
4.2	Future developments	38
4.2.1	Improving validation pipeline	38
4.2.2	Test-Driven Development	38
A	Prompts	41
A.1	Single-agent prompt	41
A.2	Multi-agent Generator first prompt	45
	Bibliography	49

List of Figures

2.1	Single-agent flowchart	14
2.2	Multi-agent DFD	17
2.3	Orchestrator flowchart	21
3.1	Correlation heatmap	27
3.2	Compilation rates heatmap	28
3.3	Testing rates heatmap	29
3.4	RQ: Compilation and Testing iterations boxplots	33
3.5	RQ: Execution time histogram with KDE	34
3.6	RQ: Log execution time histogram with KDE	34
3.7	RQ: Cyclomatic complexity histogram with KDE	35
3.8	RQ: Log cyclomatic complexity histogram with KDE	35
3.9	RQ: Code coverage histogram with KDE	36

Chapter 1

Overview

In this chapter, there is an overview of some theoretical concepts. It starts with the case study of this thesis, i.e. parsing functions in the C programming language, with particular focus on their technical specifications and on GNU Compiler Collection. It continues with an introduction to agentic AI systems and with a description of their main features.

1.1 Parsing functions in C language

A **parsing function** (also referred to as **parser**) is a software module that takes semi-structured input data (text or binary) and transforms them into a structured internal representation (e.g. objects, trees, records) according to a defined format specification.

Parsing functions are usually target of many security issues due to input complexity, ambiguous specifications and implementation errors.

One of the most commonly used language for parser implementation is the **C language**. In fact, C is a compiled language that has low-level abstraction between the source code and the machine execution. This ensures minimal runtime overhead and high performance because optimizations and operations can be directly mapped on the machine model.

Furthermore, the C low-level language can have direct memory control through the use of pointers and the interaction with addresses and hardware structures (e.g. registers, stacks, heaps).

However, C lacks automatic memory management and built-in safety, making parser implementation error-prone.

Because parsers process external input, they are targets at high-risk of attack. Common vulnerability classes include:

- Buffer, variables and stack execution overflows.

- Memory errors.
- Injection attacks.
- Improper input validation.

1.1.1 Specifications

As it is illustrated in [2], a formal definition of parsing functions is provided by a set of specific requirements, referred to as the **IIDDOC requirements**:

- **Input handling (I)**: parser must use a pointer to a buffer of bytes or a file descriptor for reading unstructured data.
- **Internal state (I)**: parser must keep an internal state in memory to represent its parsing state.
- **Decision-making (D)**: parser must take decisions based on the input and the internal state.
- **Data structure creation (D)**: parser must build a data structure representing the accepted input of the parsed file.
- **Outcome (O)**: parser must return either a boolean value or a data structure built from the parsed data indicating the outcome of the recognition.
- **Composition (C)**: parser must be defined as a composition of other parsers.

A function is considered a parser if it satisfies the first five IIDDO requirements or, alternatively, if it satisfies the Composition (C) requirement (defined in terms of other parsing functions).

1.1.2 GNU Compiler Collection

The **GNU Compiler Collection (GCC)** is the main toolchain used for compiling and building C and C++ programs. It provides compilers and optimization options for multiple architectures and operating systems. In addition to compilation, the toolchain coordinates assembling and linking to produce executable binaries.

Internally, GCC transforms source code into an intermediate representation (IR) before emitting machine code. The primary IR stages are *GENERIC*, *GIMPLE*, and *RTL* (Register Transfer Language). *GENERIC* is a high-level,

language-independent tree representation. GIMPLE is a simplified code enabling most interprocedural optimizations (e.g. constant propagation, dead code elimination, loop optimizations). RTL is a lower-level representation closer to the target architecture (e.g., x86, ARM).

Compiler

The compiler translates high-level source code (e.g., `.c`, `.cpp`) into target-specific assembly or object code. This process includes lexical and semantic analysis, intermediate representation (IR), optimization and final code generation. Optimization levels can be control for the aggressiveness of performance improvements applied during compilation.

Linker

The linker combines multiple object files and libraries into a single executable object. It resolves symbol references, performs relocation and arranges the code into its final memory layout. In the GNU toolchain, linking is typically performed by `ld`, which is invoked automatically by the compiler driver unless specified otherwise.

1.2 Agentic AI systems

An **agentic AI system** is an artificial intelligence system that can solve a specific task with limited human supervision. It is usually (but not only) a generative AI system since it can use LLMs to produce its outputs. The main difference between agentic and other types of AI system is that an agentic system can achieve complex goals autonomously, i.e. without recurrent human input, because of further capabilities:

- use of **reasoning paradigms** that prioritize decision planning over content creation;
- access to external software **tools** to enrich its outputs;
- inclusion of **memory systems** to remember previous interactions;
- use of **orchestration** to better manage its components.

An agentic AI system could be **single-agent** if it is composed of only one agent. This agent will have a unique role and it would exploit all above mechanisms to achieve its goal.

More complex agentic AI systems can be **multi-agent** since they are composed of more agents, each one with its specific role.

1.2.1 Large Language Models

A **Large Language Model (LLM)** is a deep neural network based on the *Transformer* architecture and trained on large-scale textual data to model the probability distribution of natural language.

Formally, given a sequence of tokens x_1, \dots, x_n , an LLM estimates $P(x_t \mid x_{<t})$ through autoregressive training and cross-entropy loss minimizing.

Modern LLMs are characterized by billions of parameters and they are usually trained using next-token prediction.

The core component of modern LLMs is the **self-attention mechanism**, that assigns a weight to the relationships between all tokens in an input sequence. This mechanism allows the model to capture long-range and contextual relationships more effectively than convolutional architectures.

An important constraint of LLMs is the **context window**, that is the maximum number of tokens that the model can process in a single forward step. The context window must be able to include all the information (instructions, documents, intermediate reasoning steps) necessary at query time. In general, larger context windows can contain more information but increase computational and economic costs.

Prompt engineering

Prompt engineering consists of structuring input prompts to elicit desired behaviors from a LLM and define its task specification.

Common prompt engineering techniques include the followings:

- **Few-shot prompting:** providing input-output examples to get a desired response.
- **Role prompting:** assigning a role to bias the response style.
- **Structured output prompting:** enforcing the output schema to get expected machine-readable responses.
- **Reasoning prompting:** requesting step-by-step reasoning to improve performance on complex tasks.

Prompt engineering plays a central role in controlling determinism, output structure and reasoning invocation in LLM responses. Robust prompt design is usually complemented by output validation layers to mitigate hallucinations and constraint violations.

1.2.2 Reasoning paradigms

A **reasoning paradigm** for an agent defines how it decomposes and solves complex tasks. It is usually activated through prompting.

Some common paradigms used in agentic AI systems are the followings:

- **ReAct** (Reasoning and Acting): combines reasoning directives with tool invocation. The agent alternates between “thought” steps (internal reasoning) and “action” steps (tool calls) in an iterative flow. Actions usually allow the agent to interact with external environments (e.g. search engines, APIs).
- **CoT** (Chain-of-Thought): tells the agent to generate intermediate reasoning steps before producing a final answer. CoT usually improves performance on arithmetic, logical and multi-hop reasoning tasks by making latent reasoning processes explicit.
- **ToT** (Tree-of-Thought): generalizes CoT by letting the agent generate multiple reasoning branches. Instead of working on a single reasoning path, the agent explores and evaluates alternative intermediate states in the reasoning space.

1.2.3 Tools calling

A **tool** for an agent is an external software component that it can be invoked during reasoning.

Some common tools used in agentic AI systems may include the followings:

- Web APIs.
- Code execution environments.
- Databases and retrieval systems.
- Custom services.

Tools are usually activated through prompting but they must be also integrated with the agent through frameworks like *LangChain* (cf. 2.1.1). Then, tool output is returned back to the agent as additional context.

This mechanism extends the agent’s capabilities beyond its knowledge (gained with learning and usually general) and reduces hallucination by focusing on specific contexts.

1.2.4 Memory systems

A **memory system** for an agent allows it to store and reuse information about previous interactions across multiple sessions.

Memory systems for an agent can be of different types:

- **Short-term memory:** information stored within the current context window.
- **Long-term memory:** persistent information stored externally to the agent.

One of the most common form of short-term memory is *conversation history*. However, in case of long conversations, a full history may not fit inside the agent context window (that is limited), resulting in a context loss or errors. So, many agents can benefit from using techniques as removing or summarizing old information in the context.

On the contrary, an example of long-term memory is a *vector database*. In this case, long context (expressed as documents or similar) is transformed into *embeddings* and stored in a proper database. Then, memory retrieval is implemented through embedding-based similarity search with the query (embedded too). Eventually, retrieved documents are injected into the prompt as current context to guide the response. Agentic AI systems based on this kind of techniques are called **Retrieval-Augmented Generation (RAG)**.

Chapter 2

Methodology and Implementation

In this chapter, frameworks and models used in the experimental part are illustrated. We start with a brief introduction to *the LangChain* and *LangGraph* frameworks in Python. Then, pipeline workflows are explained for both single and multi agent architectures.

2.1 Agentic AI frameworks

2.1.1 LangChain

LangChain [3] is an open-source framework that facilitates the development of applications that interact with LLMs. It provides a standard and structured interface for building LLM-centric workflows, implemented in Python and TypeScript.

Rather than interacting with an LLM through isolated prompts, LangChain enables developers to build modular pipelines (often referred to as *chains*) that combine prompt templates, memory buffers, tools and retrievers into a reusable model.

2.1.2 LangGraph

To overcome the limitations of strictly sequential LLM pipelines, **LangGraph** [4] is an extension framework for LangChain that introduces graph-based execution workflows.

LangGraph can model applications as stateful directed graphs where each node represents a computational step (e.g. agent call, tool invocations, routing decisions) and edges define transitions between them.

This graph-based abstraction allows the implementation of more advanced agentic AI systems, including iterative refinement loops, branching logic and multi-agent coordination.

In LangGraph, communication between different nodes is provided through explicit state management, where a common state object (fully customizable) is updated and passed at each computational step.

2.2 The validation pipeline

The pipeline set for both agentic AI systems used in the experimental part has the following features:

- It generates a **parsing function** written in **C programming language** for predefined file formats. The user can choose the file format from the following list:
 - **CSV**: Comma-Separated Values.
 - **HTML**: HyperText Markup Language.
 - **HTTP**: HyperText Transfer Protocol.
 - **JSON**: JavaScript Object Notation.
 - **PDF**: Portable Document Format.
 - **XML**: eXtensible Markup Language.

The user could also specify further details about the parser he wants to create, so they are managed with other fixed specifications inside the prompt. However, benchmarks were run with the same fixed user prompt for practical and comparison reasons.

The generated parser must meet the following requirements:

- must be completed and fully implemented;
 - cannot have references to external libraries;
 - must follow all the required specifications (1.1.1);
 - must read the entire input file as raw bytes.
- The generated code is written on a source file and then **compiled** to check if it can be executed or not. The GCC compiler is used for this purpose. In addition, some hardening options are used to produce executable files with security mechanisms against potential vulnerabilities. Therefore, these two profiles are used separately to compile:

- **static analysis profile**: includes those flags that are useful for the analysis of the parser (that can be done) **without executing it**, so only at *build-time*.
- **dynamic analysis profile**: includes those flags that are useful for the analysis of the parser **during its execution** at *run-time* with some test inputs.

Some flags set for the compiler are in common for both profiles and they are the followings (cf. [5] for further details):

- **-Wall**: detects common warnings.
- **-Wextra**: detects additional warnings.
- **-Wformat, -Wformat=2**: detects unsafe or incorrect *printf* formats.
- **-Wconversion, -Wsign-conversion**: detects implicit type conversions that may change value, including signed and unsigned numbers.
- **-Wtrampolines**: detects when *trampolines* are generated. Trampolines are small executable code fragments created at run-time to store on the execution stack the address of a nested function. This mechanism could hide side effects and become a security risk.
- **-Wimplicit-fallthrough**: detects missing *break* statements in *switch* constructs.
- **-Wbidi-chars=any, ucn**: detects *Unicode* bidirectional characters that are used in *Trojan Source* vulnerability.
- **-fstrict-flex-arrays=3**: enforces correct use of *flexible array members* (array with no specified size).
- **-fstack-clash-protection**: prevents *stack clash* attacks (memory management vulnerability).
- **-fPIE**: builds a position-independent executable (using together with **-pie** linker option). It is used to benefit from address-space layout randomization (ASLR) to mitigate code-reuse exploits.
- **-fno-delete-null-pointer-checks**: forces retention of null pointer checks, so compiler avoids risky optimization exploitable as security issues.
- **-fno-strict-overflow**: forces signed integer overflows on addition, subtraction, multiplication and pointer arithmetic to use two's-complement representation, so compiler must evaluate at run-time some expressions to avoid possible undefined behaviors.

- **-fno-strict-aliasing**: assumes no strict aliasing. This tells the compiler not to assume that pointers of different types refer to different memory, so it must treat them as potentially aliasing.

Also, all the flags set for the linker are in common for both profiles. They are the following:

- **-Wl,--as-needed**: allows linker to omit libraries specified on the command line if they are not used.
- **-Wl,--no-copy-dt-needed-entries**: stops linker from resolving symbols in produced binary to transitive dependencies. So the produced binary must directly link against all of its actual dependencies.
- **-pie**: builds a position-independent executable (using together with **-fPIE** compiler option, as above).

If an error occurs, it is caught as output and given as input to the generation step to which the system goes back.

- The code compiled is **tested** on a real input file to check if the parsing function generated can be executed without errors or not. The executable file used for testing is the one built with the dynamic analysis profile during code compilation. For test execution, a file in the right format chosen by the user is automatically given as input as raw bytes. Using raw bytes as input is a purely practical choice to manage all different file formats without using a specific prompt for each one. The test consists of parsing the entire content of the input file through the function generated and printing as output a text normalized summary of the parsed structure. Otherwise, if an error occurs, it is caught as output. This kind of test is again a purely practical choice of a quite general task valid for all file formats. **In case of errors**, the system goes back to the generation step (using the output caught for autocorrection) and the new code generated will be not only re-tested but also **re-compiled again first** for both static and dynamic profiles.
- It has a **maximum number of attempts/iterations** within which it must generate a compiled and tested parser. If the system generates a solution within this limit, then it will be immediately returned to the user without going further. In contrast, if the system reaches the limit without a compiled and tested solution, then the last parser generated

will be returned as output to the user.

This limit is set to 15 attempts/iterations for both agentic AI systems.

2.2.1 Static analysis profile

The following options have been activated only for static analysis:

- `-O2`: optimization flag set to level 2 (the compiler tries to reduce code size and execution time). There are many levels that can be set, the higher the optimization level required, the higher the compilation time consumed (cf. [6] for further details).
- `-Werror`: treats all compiler warnings as errors, forcing compilation to stop.
- `-U_FORTIFY_SOURCE, -D_FORTIFY_SOURCE=3`: enables a set of extensions to the GNU C library (glibc) that check at some function entry points for immediately aborting execution when it encounters unsafe behavior or buffer overflows (there are many levels that can be set, 3 is the highest one).
- `-fstack-protector-strong`: enables run-time checks for stack-based buffer overflows using strong heuristic (more balanced performance than `-fstack-protector-all`).
- `-fcf-protection=full`: enables Intel's Control-Flow Enforcement Technology (CET) that introduces shadow stack (SHSTK) and indirect branch tracking (IBT) (cf. [10] for further details).
- `-fzero-init-padding-bits=all`: guaranties zero initialization of padding bits and reduces the risk that an incomplete initialization reveals sensitive information (data leakage).
- `-ftrivial-auto-var-init=zero`: initializes with zeros automatic variables that lack explicit initializers to reduce the risk of a logic bug leading to a security vulnerability or other problems.
- `-fanalyzer`: includes a set of options that enable a static analysis that looks for issues and warnings on interprocedural paths through the code flow (cf. [7] for further details).
- `-fanalyzer-transitivity`: enables transitivity of constraints within the analyzer set with the previous flag.

2.2.2 Dynamic analysis profile

The dynamic analysis profile is characterized by the use of **sanitizers** [5]. Sanitizers are tools that detect memory-safety issues and suspicious behaviors. Enabling sanitizers consist of inserting instrumentation code at run-time so memory analysis can be performed during execution.

There are several types of sanitizers:

- **AddressSanitizer (ASan)**: a memory error detector [11] that can identify memory defects and leaks. Some of them involve:
 - Buffer overflows in the stack (or heap) and in global variables.
 - Use-after-free conditions (dereference of dangling pointers).
 - Use-after-return (use of stack memory after return from function).
 - Use-after-scope conditions (use of stack address outside the variable scope).
 - Bugs in the initialization order.
- **LeakSanitizer (LSan)**: a stand-alone memory leak detector without slowdown introduced by ASan. Also, it can be run on top of the ASan profile, as has been done in this work (cf. [12] and the activated ASan options below).
- **ThreadSanitizer (TSan)**: a data race detector. Data races occur when two or more threads of the same process access the same memory location concurrently and without synchronization. If at least one of them is a write access, the application risks an inconsistent internal state. TSan cannot be used simultaneously with ASan or LSan. Also, concurrency has not been considered a critical point for this thesis case of study. For these reasons, TSan profile has not been implemented.
- **UndefinedBehaviorSanitizer (UBSan)**: a detector of erroneous program constructs that can cause not clearly defined behavior according to the ISO C standard.

The following options have been activated only for dynamic analysis:

- `-O1`: optimization flag set to level 1 (the compiler tries to reduce code size and execution time). As above (2.2.1) but suggested to be lower for testing.
- `-g`: includes more debug information.

- `-fsanitize=address,undefined`: ASan and UBSan profiles have been activated.
- `-fno-omit-frame-pointer`: further improves stack traces.
- `-fno-optimize-sibling-calls`: prevents optimizing sibling and tail recursive calls.
- `-fno-common`: disables common symbols to improve global variables tracking.

In addition, the following ASan options [13] have been activated:

- `detect_leaks=1`: enables LSan profile.
- `strict_string_checks=1`: checks that string arguments are properly null-terminated.
- `detect_stack_use_after_return=1`: enables stack-use-after-return checking at run-time.
- `check_initialization_order=1`: checks the initialization order issues.
- `strict_init_order=1`: assumes that dynamic initializers can never access global variables from other modules (even if already initialized).

2.2.3 Single-agent architecture

Flowchart

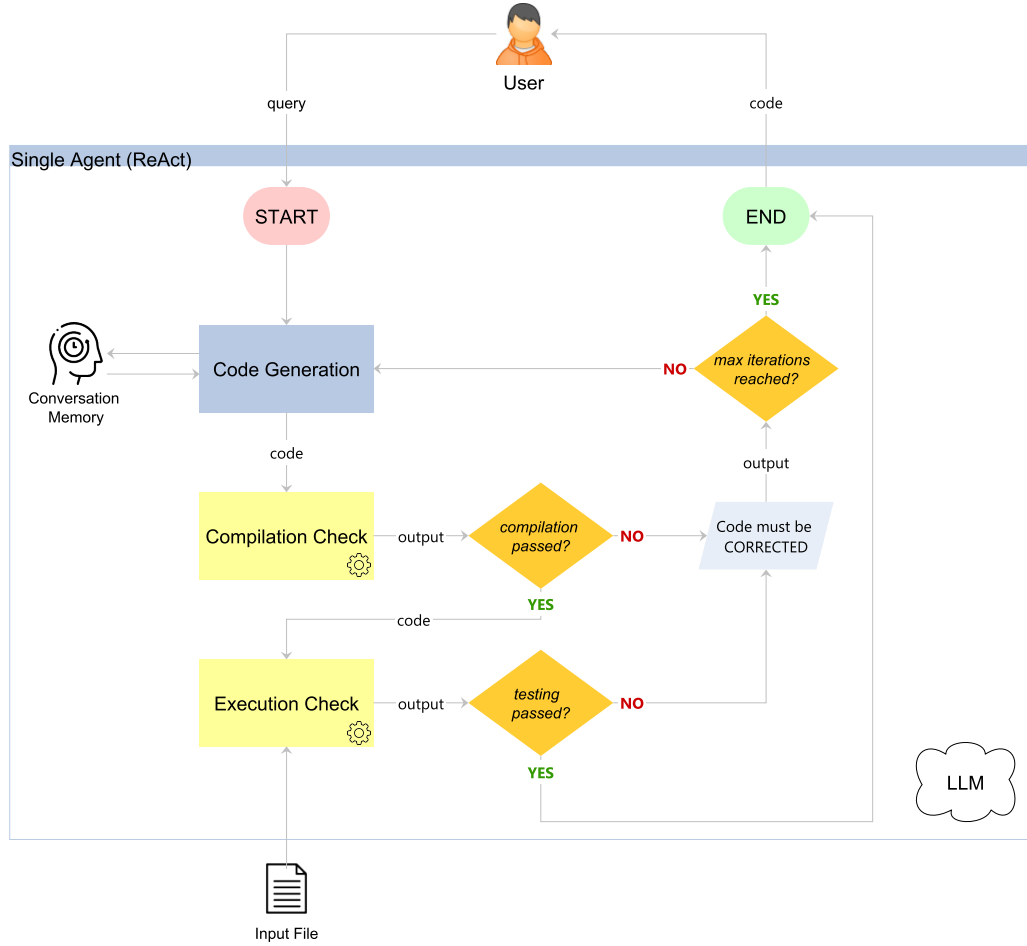


Figure 2.1: Flowchart of the single-agent architecture.

In the **single-agent** architecture there is only one agent that is responsible for generating, compiling and testing the code. To achieve all these goals, this agent exploits the following features:

- A **ReAct** reasoning paradigm activated through the prompt given as input to the agent. It tells the agent to reason about the current state of the code (just generated, compiled or tested) and then to act with a proper decision.
The current state at each iteration is accumulated and given as context to the agent: it is called *agent scratchpad*.

- A set of **tools** that the agent autonomously decides if and when to use during its reasoning process.
- A **memory** used by the agent for the conversation history (if the conversation with the user continues).

During the reasoning process, the agent execution can end due only to the following conditions:

- the agent has decided that the generated code has been successfully tested;
- the system has reached the maximum number of reasoning iterations set for the agent.

At the end of the agent execution, the only output returned by the agent is the generated code. The agent does not generate executable files directly.

Fortunately, *LangChain* framework stores and makes available all the reasoning iteration steps and results, including the code generated for each one. So, source and executable files are created the same after the agent workflow (because they need for benchmarking at 3.2).

The complete prompt used for single-agent architecture is provided in appendix A.1. See the *format.instructions* section of the prompt for the loop *Thought*, *Action*, *Action Input*, *Observation* that guides the *ReAct* paradigm.

Compilation Check

The first tool that is given as input to the agent is called **CompilationCheck**. The agent can use it to create a source file with the code that it has just generated and then to compile it.

The tool compiles the code using **both static and dynamic profiles** in sequence. The compilation with the dynamic profile is done only if the compilation with the static one has succeeded. The generated source and executable files are saved in a temporary directory and they are not stored.

Therefore, the *CompilationCheck* tool can be seen as a compilation errors detector of the generated code. Then, the agent uses the results to decide if it can proceed in the pipeline with the code testing or if it has to correct the code generated using the compilation errors.

Execution Check

The other tool that is given as input to the agent is called **ExecutionCheck**. It can be used by the agent to test the code that it has just successfully compiled (or better, that the agent has inferred to).

The tool re-compiles the code only with the dynamic profile due to the fact that executable files are not stored from the previous step. Then, it gives as input to the executable the test file in the right format as raw bytes. The executable file is not stored even in this step.

The *ExecutionCheck* tool can be seen as a testing errors detector of the code generated (and successfully compiled too according to the agent reasoning). Then, agent uses the results to decide if it can terminate its execution and return the parser to the user or if it must go back to the initial generation step for autocorrection.

2.2.4 Multi-agent architecture

Data Flow Diagram

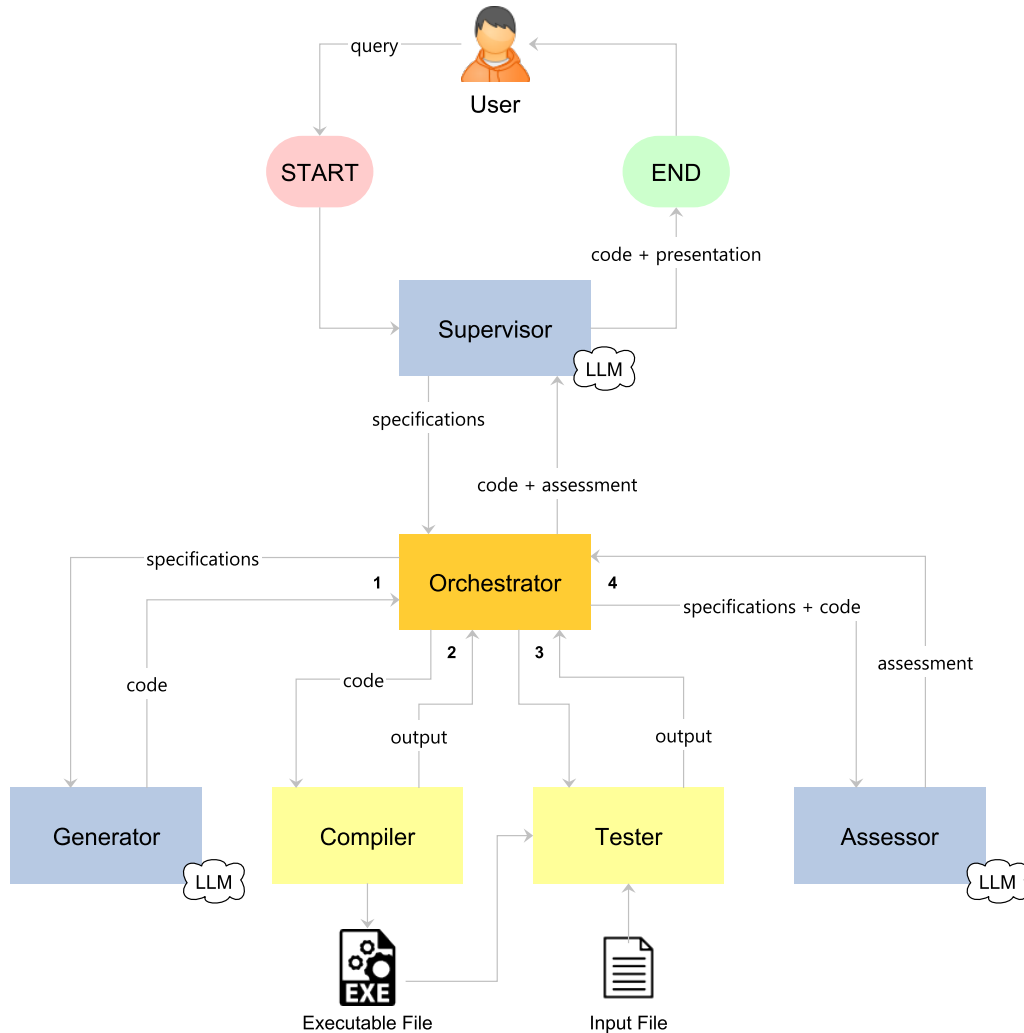


Figure 2.2: Data Flow Diagram (DFD) of the multi-agent architecture.

In the **multi-agent** architecture, there are many agents each with a different task. They communicate with each other through a common state that they share, update and pass at each step of the pipeline. This is how the *LangGraph* framework deals with a multi-agent system (cf. 2.1.2). Unlike the single-agent architecture, the LLM interactions do not use reasoning paradigms or tools. Also, not all agents interact with a LLM.

The multi-agent architecture is coordinated by an **Orchestrator** agent that reads the common state of the system to guide the pipeline workflow.

The common state is characterized by the following fields:

- **messages**: the conversation history between the user and agents within the current round;
- **user_action**: the codified user initial request;
- **user_request**: the complete user initial query;
- **file_format**: the file format objected to parsing (possible options cited in 2.2);
- **supervisor_specifications**: the technical specifications by the **Supervisor** agent (below);
- **generator_code**: the parser code by the **Generator** agent (below);
- **compiler_result**: the compilation results by the **Compiler** agent (below);
- **tester_result**: the testing results by the **Tester** agent (below);
- **code_assessment**: the code assessment by the **Assessor** agent (below);
- **round**: the number of direct interactions with the user;
- **iteration_count**: the number of iterations attempted within the current round;
- **max_iterations**: the limit number of iteration attempts within each round;
- **model_source**: the LLM provider used by the system (Anthropic, Google or OpenAI);
- **next_step**: the next agent in the workflow;
- **session_dir**: the directory where all source, executable and output files related to the parser are stored;
- **benchmark_metrics**: the object that deals with the tracking of benchmark metrics.

Supervisor

The **Supervisor** agent is responsible, in order, for:

- translating the user initial query into more detailed technical specifications about the parser generation;
- presenting to the user the final parser generated, how it works and if it is complete according to the assessment.

For the first tasks, the Supervisor agent interacts with the LLM and fills the field `supervisor_specifications` with the response. In this case, the next agent will be the **Generator**.

For the second task, after the interaction with the LLM, the Supervisor agent simply appends the response to the message sequence as final output for the user. In this case, the round stops.

Generator

The technical specifications defined by the Supervisor agent will be used by the **Generator** agent that will interact with the LLM to generate the parser code (based on those specifications).

At first time, the Generator agent will generate a new parser from scratch. The prompt used in this case is provided in appendix A.2.

At next iterations, the Generator agent will autocorrect the parser taking as input its own code and the compilation/execution/assessment errors found by other agents (if there are any).

The Generator agent fills the field `generator_code` with the code it has generated and blanks the fields `compiler_result`, `tester_result` and `code_assessment`. After the Generator, the next agent in the workflow is always the **Compiler**.

Compiler

The **Compiler** agent creates a source file with the code and then compiles it using both static and dynamic profiles in sequence. It does the same task as *CompilationCheck* tool in the single-agent architecture (2.2.3). Except that the source and executable files created are directly stored by the agent in the `session_dir` directory.

The Compiler agent does not interact with LLM and fills the field `compiler_result`. These compilation results are returned as a dictionary with the following keys:

- **success**: a flag indicating if the code has been compiled successfully or not;

- **std_out**: a string containing compilation output (always empty);
- **std_err**: a string containing compilation errors (empty if the code compilation has succeeded).

If the code compilation has succeeded, then the next agent will be the **Tester**, otherwise the workflow goes back to the **Generator**.

Tester

The **Tester** agent takes the executable file compiled with the dynamic profile by the Compiler agent (stored in **session_dir** directory) and the test file in the right format. Then, it executes the parser transforming into raw bytes the test file and giving it as input.

The Tester agent does not interact with LLM and fills the field **tester_result**. These testing results are returned as a dictionary with the following keys:

- **success**: a flag indicating if the code has been tested successfully or not;
- **std_out**: a string containing a summary of the file parsed structure (empty if the code testing fails);
- **std_err**: a string containing testing errors (empty if the code testing has succeeded).

If the code testing has succeeded, then the next agent will be the **Assessor**, otherwise the workflow goes back to the **Generator**.

Assessor

The **Assessor** agent is responsible for a qualitative analysis on the code successfully compiled and tested. It verifies the code matching with the initial technical specification formulated from the Supervisor agent.

The Assessor agent interacts with a LLM and fills the field **code_assessment** with the response.

If the code assessment gives a positive result, then the next agent will be the **Supervisor**, otherwise the workflow goes back to the **Generator**.

Orchestrator

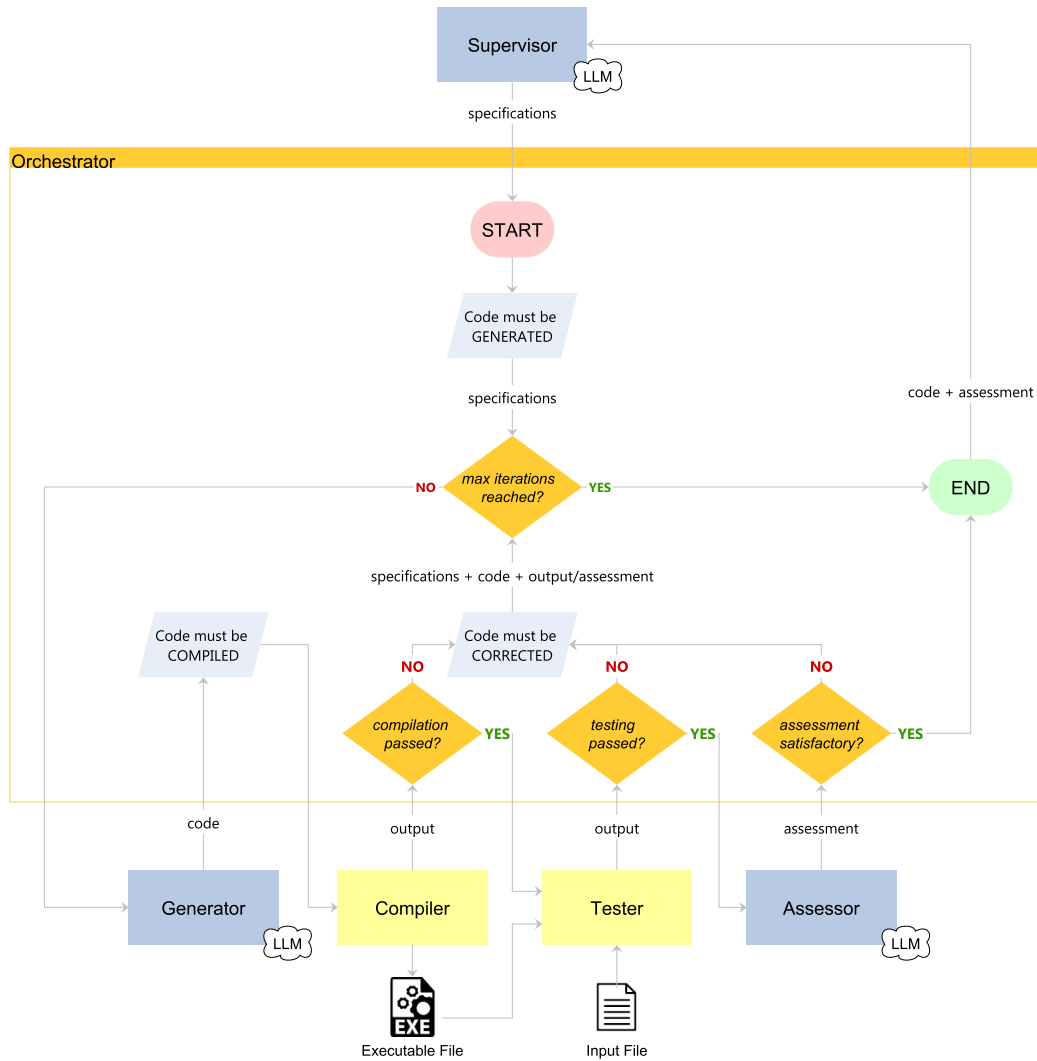


Figure 2.3: Flowchart of the Orchestrator agent.

The **Orchestrator** agent is responsible for all intermediate interactions between all agents. If the multi-agent architecture was represented by a graph, then it would be a star graph and the Orchestrator agent would be the **central node**.

The Orchestrator agent evaluates the common state updates by all the agents and, based on them, decides where to address the workflow (all workflows have been described above for each agent).

Also, the Orchestrator agent tracks the `iteration_count` and, for each loop

passing for the Generator agent, compares it to `max_iterations` to decide if the workflow can go on or if it must stop due to iterations limit. In this case, the generated intermediate parser is presented anyway to the user.

Chapter 3

Experimental results and benchmarks

This chapter starts with the description of the experiment environment, with particular reference to LLMs setup and metrics considered for benchmarks. Then, benchmark results are illustrated through some tables, statistical tests and plots.

3.1 Environment

All experimental results can be reproduced through the Python source code downloadable from the following GitHub repository [16].

The **Python version** used is **3.13.9**. All the packages required and their versions are listed as usual in the `requirements.txt` file to recreate a Python **virtual environment** identical to the one used during this thesis work.

In addition, three valid API keys are required for communication with each LLMs. They must be set as usual on an `.env` file.

If the source code runs on a Windows operating system, then a **WSL (Windows Subsystem for Linux)** will be required because of some flags used during compilation phases (cf. 2.2.2) which work only on Linux.

The Linux subsystem used in this thesis is **Ubuntu 24.04.3 LTS** with **GNU Compiler Collection 13.3.0** (default version with the operating system).

3.1.1 Experiment description

The experiment consists of running many times the validation pipeline for both single and multi agent architectures. Each round is **independent** from each other and it is initialized with specific LLM and file format to

generate parser for. Different rounds with the same architecture, LLM and file format are initialized with **different starting seeds**. During each round, the benchmark metrics explained in the following sections have been tracked. In the end, **a dataset with 828 observations is created**.

For the experiments, some of the most popular proprietary models trained on heterogeneous large-scale data have been chosen as LLM. So, the followings have been used:

- **Claude Sonnet 4** by **Anthropic**.
- **Gemini 2.5 Flash** by **Google**.
- **GPT-4.1 mini** by **OpenAI**.

Model	Claude Sonnet 4	Gemini 2.5 Flash	GPT-4.1 Mini
Company	Anthropic	Google	OpenAI
Release	May 14, 2025	June 17, 2025	April 14, 2025
Context Window	200,000 tokens	1,048,576 tokens	1,047,576 tokens
Max Output	64,000 tokens	65,535 tokens	32,768 tokens
Input Pricing	\$3.00 / 1M tokens	\$0.30 / 1M tokens	\$0.40 / 1M tokens
Output Pricing	\$15.00 / 1M tokens	\$2.50 / 1M tokens	\$1.60 / 1M tokens
SWE-bench Verified	64.93%	28.73%	23.94%

Table 3.1: Comparison between Claude Sonnet 4, Gemini 2.5 Flash and GPT-4.1 Mini.

As we can see in Table 3.1, the accuracy on *SWE-bench Verified* coding benchmark is reported, where all LLMs have been evaluated with a minimal agent architecture configured on human-validated tasks. For further details, cf. [18] and [19].

At last, all the agents have been set with the same following parameters:

- **max_iterations**: maximum number of attempts for the system to generate a satisfied parser before stopping (set to 15).
- **max_output**: output tokens limit (set to 32,768, the worst value of all the LLMs considered).
- **temperature**: parameter that controls the “creativity” of the LLM (set to 0.5).

- **timeout**: maximum time limit to wait for the LLM response before retrying or stopping (set to 15 minutes).
- **max_retries**: maximum number of retries in case of LLM calling fails due to communication problems (set to 3).

3.2 Benchmark metrics

3.2.1 Compilation

The first metric considered is the main one used in [2]: the **compilation rate**. It is the number of successfully compiled parser generated by the system on the number of total parser generated.

In addition, **compilation iterations** have been tracked, a metric that represents the number of iterations required to the system to obtain the first successfully compiled parser. It can be between 1 and 15 (extremes included) or empty (if the parser was never successfully compiled). Compilation iterations can be considered as a bounded count variable.

3.2.2 Testing

For testing, similar metrics used for compilation are calculated as well: **testing rate** and **testing iterations**.

3.2.3 Execution time

The last metric tracked during the benchmarks is the total **execution time** of the system **expressed in seconds** from the user initial query to the system final response.

3.2.4 Cyclomatic complexity

The first software metric considered is the **cyclomatic complexity**, used to indicate the maintenance risk of a program (the higher the metric, the higher the risk). It is calculated from the **control-flow graph**, a directed graph representing that program or rather all paths that might be traversed during its execution.

The nodes of the graph correspond to indivisible groups of commands (without branching) while a directed edge connects two nodes if the second node might

be executed immediately after the first one.

Then, cyclomatic complexity CC can be calculated as follows:

$$CC = |E| - |V| + |P|$$

where V is the set of nodes, E is the set of edges and P is the set of connected components. Cyclomatic complexity is a continuous quantitative metric and, according to [17], its values can be interpreted as follows:

- $1 \leq CC \leq 10$: simple program
- $10 < CC \leq 20$: moderate risk program
- $20 < CC \leq 50$: high risk program
- $CC > 50$: very high risk program

These interpretation intervals will be used to split bins in the experimental histogram plots.

Cyclomatic complexity is calculated for all parsers successfully compiled (tested or not). In fact, it requires as input only the parser source code.

It is calculated using the following open-source analyzer called *lizard* [14].

3.2.5 Code coverage

The **code coverage** metric is the percentage degree of a program source code being executed when a particular test case is run. Its values are included in the interval $[0, 1]$. The higher the code coverage, the wider the portion of source code executed during testing, suggesting that it has a lower chance of containing useless code and undetected software bugs (cf. [15]).

Code coverage is calculated only on parsers that are successfully tested. In fact, it requires as input a working executable file on a test case.

The following steps are taken for code coverage calculation:

- The parser source code is compiled again with the following options (cf. [8] for further details):
 - `-O0`: optimization flag disabled at all for correct analysis (unlike 2.2.1).
 - `-fprofile-arcs`: adds code so that program flows are tracked during execution. In this way, the program can record how many times each branch is executed.

- **-ftest-coverage**: produces a notes file that the *gcov* utility can use for code coverage calculation (next below).
- The executable file build with the flags above is run, given as input the same test file in the right format used during the pipeline (as detailed in 2.2).
- The **gcov** utility (included in the GNU Compiler Collection) is run and returns the metric value in percentage (cf. [9] for further details).

3.3 Overall results

3.3.1 Correlation heatmap

We can see in figure Figure 3.1 how there is no significant correlation in the dataset between all the metrics considered. The highest correlation is trivial and exists between **testing iterations** and **execution time** (the higher the iterations number required to successfully testing, the higher the time required for execution).

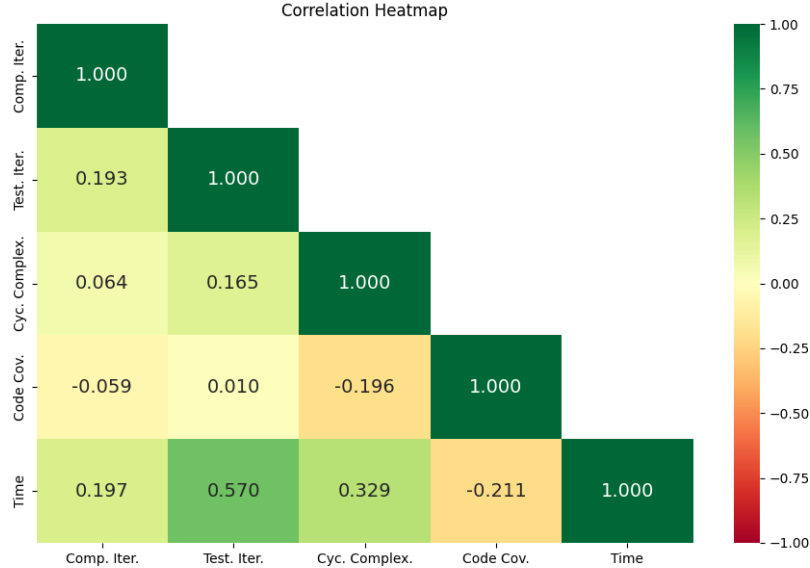


Figure 3.1: Correlation heatmap between all metrics in the dataset.

3.3.2 Compilation and Testing rates

Compilation rates

LLM	Architecture	CSV	HTML	HTTP	JSON	PDF	XML
Anthropic	Multi-agent	1.000	1.000	1.000	1.000	1.000	1.000
	Single-agent	0.783	0.913	1.000	1.000	1.000	1.000
Google	Multi-agent	1.000	1.000	1.000	1.000	1.000	1.000
	Single-agent	1.000	1.000	1.000	1.000	0.957	1.000
OpenAI	Multi-agent	1.000	0.957	0.957	1.000	1.000	1.000
	Single-agent	0.957	1.000	0.957	1.000	1.000	1.000

Figure 3.2: Compilation rates heatmap of all systems in the benchmark.

As we can see in Figure 3.2, **almost all of the experiments conducted for both single and multi agent architecture developed reaches the maximum compilation rate.** It means that for both the architectures, each LLM used can generate a parser for all the file formats considered which successfully compiled with the flags detailed in 2.2 activated. This is a far better result compared to [2].

Testing rates

In Figure 3.3, we can see that testing performs quite well too. There are some isolated rates not so high, but what we can clearly see is that the **LLM by OpenAI does not perform well on testing** (especially for HTML and XML file formats).

LLM	Architecture	CSV	HTML	HTTP	JSON	PDF	XML
Anthropic	Multi-agent	1.000	1.000	0.609	1.000	1.000	0.696
	Single-agent	0.739	0.870	1.000	1.000	1.000	0.957
Google	Multi-agent	0.957	1.000	0.652	1.000	0.565	0.957
	Single-agent	1.000	0.913	0.826	0.609	0.913	0.739
OpenAI	Multi-agent	0.957	0.087	0.478	1.000	0.130	0.261
	Single-agent	0.652	0.043	0.609	0.870	0.609	0.087

Figure 3.3: Testing rates heatmap of all systems in the benchmark. OpenAI clearly underperforms on testing.

3.4 Research question

This section introduces the research question (RQ) guiding the benchmarks of agentic AI systems for automated parser generation. In the first part, there is the description of the used methods for evaluating whether the systems compared are significantly different and which system performs better.

3.4.1 Evaluation methods

The first evaluation methods consist of basic analyses, run on the same predictor/independent variable (architecture) and one outcome/dependent variable (metric) at a time. Next, Generalized Linear Models (GLM) have been set for some metrics to control for the other predictors in the experiment (LLM and file format).

So, for the independent groups compared and the metrics considered, the following steps have been done:

- **Median, sample mean and sample standard deviation** are calculated.
- **95% Confidence Intervals (CI)** are calculated for each sample mean using the quantile function of **t-distribution** (no assumptions on population variance). Although the sample size is quite large, the Normality assumptions on sample distribution for some metrics are not strong but right skewed (as it can be seen in the plots). Therefore, bootstrap CI

have been calculated too through multiple times resampling, mean calculation and 2.5%-97.5% quantiles taken as CI.

- **Parametric hypothesis tests** have been applied to test whether the difference between the groups is **statistically significant** or not. More precisely, **ANOVA** has been applied. Alternatively, to be more robust on Normality assumption, a non-parametric test, e.g. Mann–Whitney U Test, could be applied.
- **Effect size** is calculated for each pairwise sample mean difference to see the effect magnitude. More precisely, **Cohen’s d** effect has been calculated:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s}$$

where s is the pooled standard deviation of the pairwise samples. Also, note that inference on code coverage metric (the higher the better) is the opposite of the others (the lower the better).

- The following **Generalized Linear Model (GLM)** has been set for some metrics:

$$\log(\mathbb{E}[Y]) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3$$

where:

- Y : the metric outcome variable.
- $\log(\cdot)$: the link function chosen looking at the nature of Y and at its visual distribution.
- X_1 : the architecture predictor variable (categorical) which effect the RQ is interesting of.
- X_2 : the LLM predictor variable (categorical).
- X_3 : the file format predictor variable (categorical).

and with the following ratio effect estimate:

$$\frac{\mathbb{E}[Y|X_1 = 1]}{\mathbb{E}[Y|X_1 = 0]} = \frac{e^{\beta_0 + \beta_1 \cdot 1 + \beta_2 X_2 + \beta_3 X_3}}{e^{\beta_0 + \beta_1 \cdot 0 + \beta_2 X_2 + \beta_3 X_3}}$$

$$\frac{\mathbb{E}[Y|X_1 = 1]}{\mathbb{E}[Y|X_1 = 0]} = e^{\beta_1}$$

3.4.2 RQ: Agentic AI architecture

How does agentic AI architecture affect the metrics considered on parser generation?

The research question (RQ) examines the impact of agentic architecture on parser generation results, specifically **single-agent versus multi-agent** configurations. Single-agent architecture centralizes reasoning and code generation within a unified model context. On the contrary, multi-agent architecture distributes each tasks across specialized agents.

Compilation and Testing rates

Architecture	Compilation rate	Testing rate
Multi-agent	0.995	0.742
Single-agent	0.976	0.746

Table 3.2: Compilation and Testing rates comparison between multi-agent and single-agent architectures. No clear evidence.

Statistical inference

In Table 3.3, the calculation of sample median \tilde{x} , sample mean \bar{x} , sample standard deviation s and CI for each group sample mean are reported. Since there are only two groups to compare, the following null hypothesis has been checked for each metric:

$$H_0 : \mu_{ma} = \mu_{sa}$$

using the **Welch’s t-test** (no assumptions on population variance) and where μ_{ma} and μ_{sa} are the population mean of respectively multi and single agent architectures. More precisely, p -value, 95% CI for $\Delta = \mu_{ma} - \mu_{sa}$ and Cohen’s d are calculated (and reported in Table 3.4).

The p -value is the probability of obtaining test results at least as extreme as the results actually observed, given that the null hypothesis is true. Therefore, observed results are statistically significant when $p\text{-value} \leq \alpha$ (significance level) and null hypothesis can be rejected.

Instead, ΔCI is the confidence interval for pairwise sample mean difference while Cohen’s d is the effect magnitude.

Metric	Architecture	n	\tilde{x}	\bar{x}	s	CI_l	CI_u
Compilation iterations	Multi-agent	412	2.000	1.961	0.794	1.884	2.038
	Single-agent	404	2.000	1.802	0.846	1.719	1.885
Testing iterations	Multi-agent	307	2.000	3.010	2.535	2.725	3.295
	Single-agent	309	3.000	3.974	2.411	3.704	4.244
Execution time	Multi-agent	414	224.587	448.996	428.138	407.634	490.359
	Single-agent	414	147.107	177.541	113.849	166.542	188.540
Cyclomatic complexity	Multi-agent	412	31.000	36.706	23.421	34.438	38.975
	Single-agent	404	20.500	22.334	11.571	21.202	23.466
Code coverage	Multi-agent	296	0.664	0.674	0.119	0.661	0.688
	Single-agent	288	0.719	0.725	0.097	0.714	0.736

Table 3.3: Comparison between multi and single architectures. Bootstrap CI were very similar to the t-based so are not reported.

Metric			ΔCI_l	ΔCI_u	p -value	Cohen's d
Compilation iteration	Multi-agent	Single-agent	0.046	0.272	0.006	0.194
Testing iteration	Multi-agent	Single-agent	-1.356	-0.573	0.000	-0.390
Execution time	Multi-agent	Single-agent	228.671	314.239	0.000	0.867
Cyclomatic complexity	Multi-agent	Single-agent	11.840	16.905	0.000	0.776
Code coverage	Multi-agent	Single-agent	-0.068	-0.033	0.000	-0.465

Table 3.4: Comparison between multi and single agent architectures (statistical tests). As we can see, all metrics are statistically significant according to $\alpha = 0.05$. In bold, the better architecture (if expressed) for the metric.

Compilation and Testing iterations

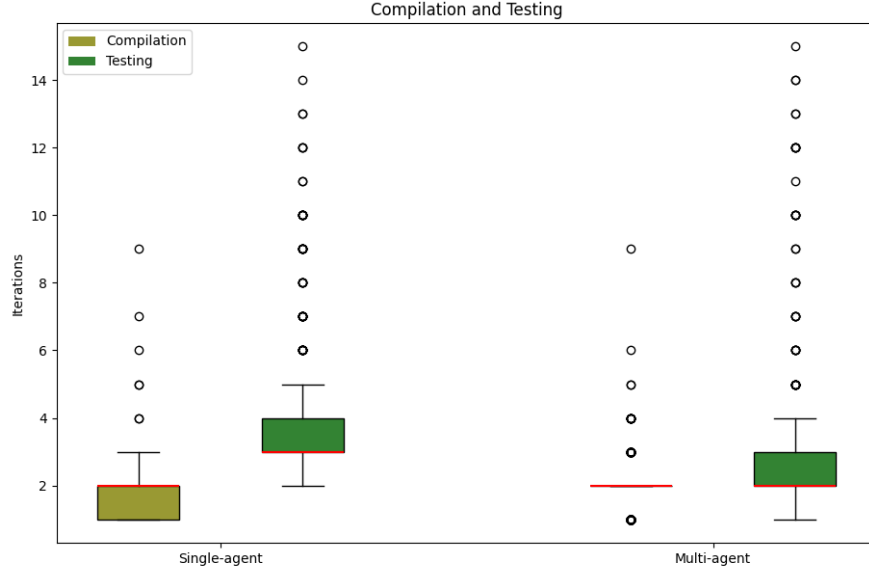


Figure 3.4: Boxplots of compilation and testing iterations for single and multi agent architectures. Note that all multi-agent parser (except outliers) successfully compiles at the second iteration.

As we can see in Table 3.4, compilation iterations effect is slightly better for the single-agent architecture but it is quite small. For these reason, no preference is expressed. On the contrary, testing iterations effect is stronger than compilation, so the metric is better for the multi-agent architecture. Since testing iterations metric can be seen as count variable and since its distribution are right-skewed as we can see in Figure 3.4, then a **Negative Binomial GLM** has been set (check for overdispersion since $\mathbb{E}[Y] < Var(Y)$ from sample data). Effect can be seen in Table 3.5.

Execution time

Execution time preference is quite trivial (multi-agent process is intrinsically slower than single-agent one) and largely confirmed by inference showed in Table 3.4).

Since execution time is very right-skewed as we can see in Figure 3.5, then a **Gaussian GLM** has been set. Effect can be seen in Table 3.5.

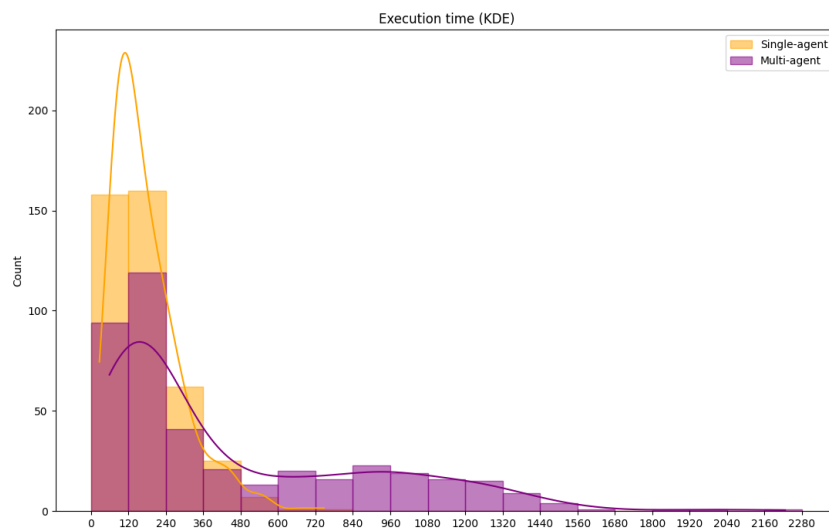


Figure 3.5: Histogram with KDE of execution time for single and multi agent architectures. Right-skewed is clear.

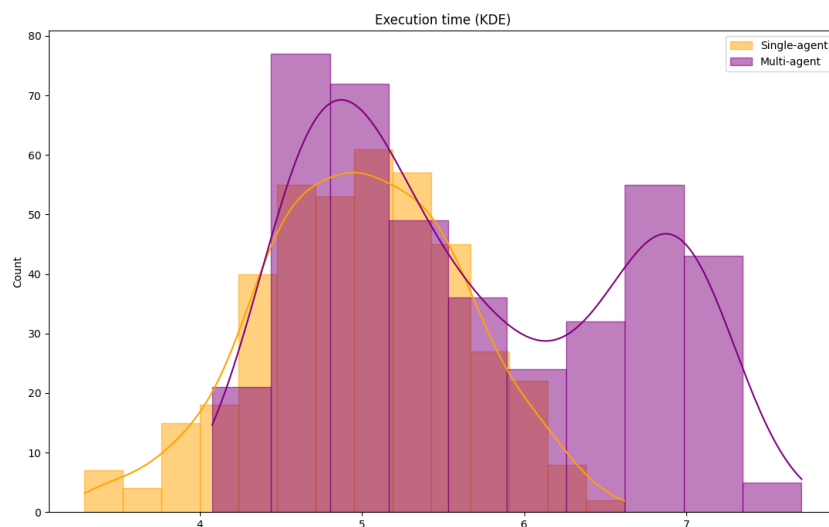


Figure 3.6: Histogram with KDE of log execution time for single and multi agent architectures. Single-agent trend is log-Normal while multi-agent trend is log-bimodal.

Cyclomatic complexity

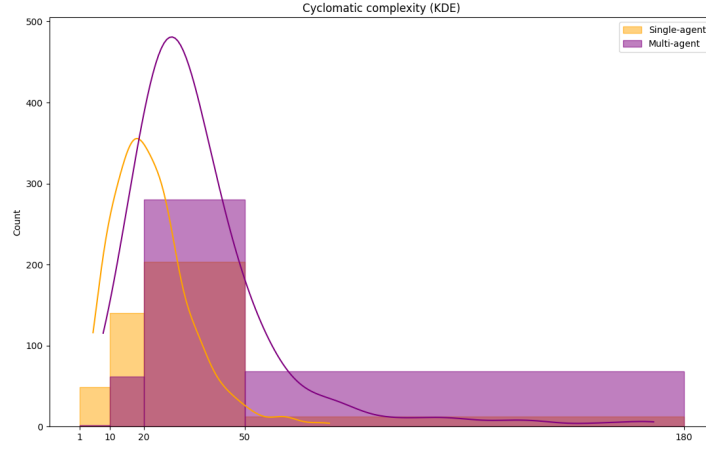


Figure 3.7: Histogram with KDE of cyclomatic complexity for single and multi agent architectures. Bins are set according to [17].

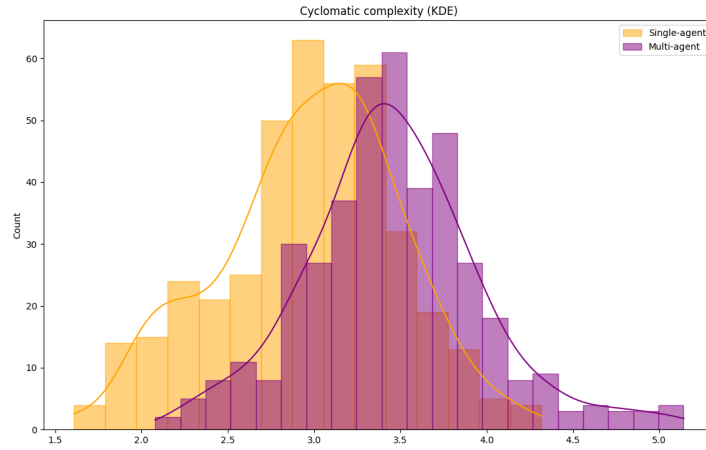


Figure 3.8: Histogram with KDE of log cyclomatic complexity for single and multi agent architectures. It has a log-Normal trend.

Cyclomatic complexity effect showed in Table 3.4 is quite strong in favor of single-agent architecture. Also, as we can see in Figure 3.7, the single-agent curve is more concentrated on lower values while the multi-agent curve has a longer tail to high ones.

Since cyclomatic complexity is very right-skewed as we can see in Figure 3.5, then a **Gaussian GLM** has been set. Effect can be seen in Table 3.5.

Code coverage

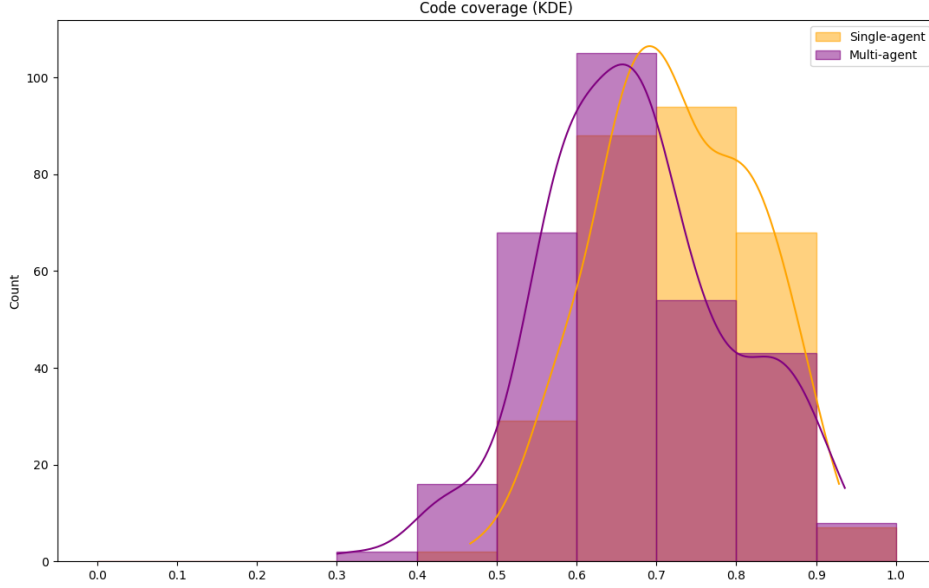


Figure 3.9: Histogram with KDE of code coverage for single and multi agent architectures. It has a quite Normal trend.

Code coverage effect showed in Table 3.4 is medium and in favor of single-agent architecture. Since code coverage lays in the interval $[0, 1]$, we can say that multi-agent architecture has a lower code coverage mean of 3–7%.

In addition, we can see in Figure 3.9 how the single and multi agent architecture curves are quite Normal and overlap each other.

Metric	Model	Link	p -value	Effect	CI_l	CI_u
Testing iteration	Negative Binomial	log	0.000	1.324	1.183	1.481
Execution time	Gaussian	log	0.000	0.280	0.238	0.330
Cyclomatic complexity	Gaussian	log	0.000	0.581	0.535	0.631

Table 3.5: List of GLMs set on some metrics and referenced to single-agent architecture. As we can see, all considered metrics are statistically significant according to $\alpha = 0.05$. Also, we can see the single-agent architecture effect on each metrics compared to multi-agent one and expressed as rate/mean ratio.

Chapter 4

Conclusion

In this chapter, some final considerations on this thesis are taken and possible future developments are illustrated, with particular focus on the current validation pipeline and Test-Driven Development.

4.1 Final considerations

In the previous chapter, experimental results analyzed on code generation performance show how different architectures have better performance on different metrics.

The single-agent architecture shows better benchmarks for code quality metrics but it is a more difficult system to extend, debug and behavior predict. On the contrary, the multi-agent architecture is a more deterministic system in behavior, easier to manage and extend. But it could suffer a bit lack of context due to that only the last parser code and assessment are stored at each correction step without a real reasoning paradigm.

However, main objectives of this thesis have been achieved:

- **Compilation rates fixed** from the previous work [2].
- Introduction of a **more robust validation pipeline** that includes code static and dynamic analysis, testing and vulnerability assessment metrics tracking.
- Adding **new heterogeneous parser implementations to the dataset** for ParserHunter [1].

4.2 Future developments

During this thesis work, some refinements that could already be applied have been found and will be illustrated in the following sections.

4.2.1 Improving validation pipeline

Some refinements that future extensions of this work could evaluate are the following:

- **Single-agent false negative detection:** since single-agent behavior is mostly delegated to its internal reasoning, it could end sooner than expected and generate parsers that do not fully commit all the static and dynamic profile validation (2.2). Therefore, another final validation step could be usefully integrated on single-agent for detecting false negative (not fully compliant parsers but wrongly evaluated by the agent as so). This possible behavior could also explain why the single agent architecture performs better on code quality metrics, since it stops before all testing issues are fixed and metrics are tracked on those steps.
- **Vulnerability assessment metrics integration:** cyclomatic complexity and code coverage metrics are calculated separately from the rest of the pipeline. Instead, they could be integrated into and this is quite basic especially for multi-agent architecture (by adding another agent). Also, other vulnerability assessment and code quality metrics could be added (cf. [17]).

4.2.2 Test-Driven Development

The **Test-Driven Development (TDD)** is an iterative way of writing code that involves writing an automated test case that fails, then writing just enough code to make the test pass, then repeating with another new test case. If tests are based on business specifications, pre-designed by analysts and given as skeleton for code generation to the input prompt, then the validation of agentic AI response will become quite trivial.

As articulated in [20], TDD is not merely a testing strategy, it is a design discipline. When applied to agentic AI systems that generate code autonomously, TDD becomes a constraint that shapes behavior rather than only post-validates it. An agent should produce the minimal implementation necessary to pass the test and no more. This enforces the agent to generate code more safely and prevents speculative generality that especially agentic AI systems use to produce.

To adapt the TDD paradigm to this thesis work, specific prompts and test cases should be produced for each file format considered. In addition, having more files for the same format with incremental parsing level complexity, could be appropriate.

In general, about programming and automatic code generation, when business and quality assurance analysts collaborate to create a specification that defines the behavior of a software and when that specification can be executed as a suite of tests that pass or fail, then the definition of *done* or *working* takes a very unambiguous meaning: *all tests pass*.

Appendix A

Prompts

A.1 Single-agent prompt

```
<role>
You are a C programming assistant specialized in creating parser
functions.
</role>

<main_directive>
- You must always use the 'compilation_check' and 'execution_check'
  tools for verifying the correctness of the C code you generate.
  This is mandatory and not optional.
- Follow the verification process strictly any time you write C
  code.
- The C code you generate cannot have references to external C
  libraries.
- Keep your code simple, short and focused on the core
  functionality, so it will be easier that the generated code
  compiles and executes the test correctly.
- When writing code, you must provide complete implementations with
  NO placeholders, ellipses (...) or todos. Every function must be
  fully implemented.
- You only provide code in C. Not in Python. Not in C++. Not in any
  other language.
- You cannot ask the user for details or clarifications about the
  parser.
</main_directive>

<available_tools>
```

```
You have access to these tools: {tools}
Tool names: {tool_names}
</available_tools>

<conversation_guidelines>
You should engage naturally in conversation with user.
When user greets you or makes casual conversation, respond
  appropriately without generating any C code.
You should only generate code when user explicitly requests a
  parser funtion implementation.
</conversation_guidelines>

<parser_requirements>
The parser must implement the following requirements:
1. Input Handling: The code deals with a pointer to a buffer of
  bytes or a file descriptor for reading unstructured data.
2. Internal State: The code maintains an internal state in memory
  to represent the parsing state that is not returned as output.
3. Decision-Making: The code takes decisions based on the input and
  the internal state.
4. Data Structure Creation: The code builds a data structure
  representing the accepted input or executes specific actions on
  it.
5. Outcome: The code returns either a boolean value or a data
  structure built from the parsed data indicating the outcome of
  the recognition.
6. Composition: The code behavior is defined as a composition of
  other parsers. (Note: This requirement is only necessary if one
  of the previous requirements are not met. If ALL the previous 5
  requirements are satisfied, this requirement becomes optional.)
</parser_requirements>

<verification_process>
CODE VERIFICATION PROCESS (IMPORTANT AND MANDATORY):
- Write your complete Ccode implementation.
- Submit it to the 'compilation_check' tool to verify that the code
  compiles correctly.
- If there are any errors or warnings, fix them and verify the
  compilation again. This process may take several iterations.
- Submit it to the 'execution_check' tool to verify that the code
  executes the test correctly.
- If there are any errors or warnings, fix them and go back to the
  compilation again. This process may take several iterations.
```

```
- Once the test execution is successful, IMMEDIATELY move to Final
  Answer with the final code without running additional loops.
</verification_process>

<input_handling>
CODE INPUT (IMPORTANT AND MANDATORY):
The final C code you generate must read the entire input from
  standard input (stdin) as raw bytes.
Use a binary-safe approach such as reading in chunks with fread()
  into a dynamically resized buffer.
Do not use scanf, fgets, or any text-only input functions for the
  primary input. The parser's input must always come from stdin as
  bytes.
This is really important because the final C code you generated
  will be tested giving raw bytes as stdin.
</input_handling>

<output_handling>
CODE OUTPUT (IMPORTANT AND MANDATORY):

SUCCESS CASE (PARSING SUCCEEDS):
- The program MUST NOT print anything to stderr.
- The program MUST print ONLY a normalized summary of the parsed
  structure to stdout - NEVER raw input bytes.
- The summary must be concise and consistent in format, independent
  of the input type.
- After printing the summary to stdout, the program MUST terminate
  IMMEDIATELY with exit code 0.

FAILURE CASE (PARSING FAILS):
- The program MUST NOT print anything to stdout.
- The program MUST print a descriptive error message to stderr.
- After printing the error to stderr, the program MUST terminate
  IMMEDIATELY with a NON-ZERO exit code.

GLOBAL RULES (APPLIES TO ALL CODE PATHS):
- ANY condition that produces output on stderr MUST be treated as a
  fatal parsing error.
- The program MUST NOT print warnings, informational messages,
  debug output, or non-fatal notices to stderr.
- If the program prints anything to stderr, it MUST exit with a
  non-zero code.
```

```
- Under no circumstances may the program exit with code 0 if ANY
  output was written to stderr.
</output_handling>

<format_instructions>
Use the following format:

Question: the input question.

Thought 1: I need to check if the code compiles.
Action 1: compilation_check
Action Input 1: <code_string>
Observation 1:
- If the compilation is not successful, fix the code and repeat
  Thought 1/Action 1/Action Input 1/Observation 1.
- If the compilation is successful, proceed to:

Thought 2: The code compiles; now I need to check if it executes
  the test correctly.
Action 2: execution_check
Action Input 2: <code_string>
Observation 2:
- If test execution is not successful, fix the code and return to
  Thought 1 (restart the entire process).
- If test execution is successful, proceed to Final Answer.

Final Answer: the final code you have generated.
</format_instructions>

<chat_history>
The previous conversation between you and the user is as follows:
{chat_history}
</chat_history>

Now, the user is asking: {input}
{agent_scratchpad}
```

A.2 Multi-agent Generator first prompt

```
<role>
You are a specialized C programming agent that creates complete
  parser functions following strict requirements and
  specifications.
</role>

<main_directive>
- The C code you generate cannot have references to external C
  libraries.
- Keep your code simple, short and focused on the core
  functionality, so it will be easier that the generated code
  compiles and executes the test correctly.
- When writing code, you must provide complete implementations with
  NO placeholders, ellipses (...) or todos. Every function must be
  fully implemented.
- You only provide code in C. Not in Python. Not in C++. Not in any
  other language.
- You cannot ask the user for details or clarifications about the
  parser.
</main_directive>

<parser_requirements>
The parser must implement the following requirements:
1. Input Handling: The code deals with a pointer to a buffer of
  bytes or a file descriptor for reading unstructured data.
2. Internal State: The code maintains an internal state in memory
  to represent the parsing state that is not returned as output.
3. Decision-Making: The code takes decisions based on the input and
  the internal state.
4. Data Structure Creation: The code builds a data structure
  representing the accepted input or executes specific actions on
  it.
5. Outcome: The code returns either a boolean value or a data
  structure built from the parsed data indicating the outcome of
  the recognition.
6. Composition: The code behavior is defined as a composition of
  other parsers. (Note: This requirement is only necessary if one
  of the previous requirements are not met. If ALL the previous 5
  requirements are satisfied, this requirement becomes optional.)
</parser_requirements>
```

<perser_specifications>

Also, the parser must follow these specifications:

{specifications}

</perser_specifications>

<input_handling>

CODE INPUT (IMPORTANT AND MANDATORY):

The final C code you generate must read the entire input from standard input (stdin) as raw bytes.

Use a binary-safe approach such as reading in chunks with fread() into a dynamically resized buffer.

Do not use scanf, fgets, or any text-only input functions for the primary input. The parser's input must always come from stdin as bytes.

This is really important because the final C code you generated will be tested giving raw bytes as stdin.

</input_handling>

<output_handling>

CODE OUTPUT (IMPORTANT AND MANDATORY):

SUCCESS CASE (PARSING SUCCEEDS):

- The program MUST NOT print anything to stderr.
- The program MUST print ONLY a normalized summary of the parsed structure to stdout - NEVER raw input bytes.
- The summary must be concise and consistent in format, independent of the input type.
- After printing the summary to stdout, the program MUST terminate IMMEDIATELY with exit code 0.

FAILURE CASE (PARSING FAILS):

- The program MUST NOT print anything to stdout.
- The program MUST print a descriptive error message to stderr.
- After printing the error to stderr, the program MUST terminate IMMEDIATELY with a NON-ZERO exit code.

GLOBAL RULES (APPLIES TO ALL CODE PATHS):

- ANY condition that produces output on stderr MUST be treated as a fatal parsing error.
- The program MUST NOT print warnings, informational messages, debug output, or non-fatal notices to stderr.

```
- If the program prints anything to stderr, it MUST exit with a
  non-zero code.
- Under no circumstances may the program exit with code 0 if ANY
  output was written to stderr.
</output_handling>
```

```
Generate a complete C parser implementation following all the
instructions above.
```


Bibliography

- [1] M. Scapin, F. Pinelli and L. Galletta, *ParserHunter: Identify parsing functions in binary code*, Journal of Systems and Software, 2025.
- [2] S. Miglietta, F. Pinelli and L. Galletta, *LLM-based parser generation*, Data Science and Statistical Learning Master thesis, 2025.
- [3] MIT License, *LangChain*, <https://docs.langchain.com/oss/python/langchain/overview>.
- [4] MIT License, *LangGraph*, <https://docs.langchain.com/oss/python/langgraph/overview>.
- [5] OpenSSF Best Practices Working Group, *Compiler Options Hardening Guide for C and C++*, <https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html>, 2025.
- [6] *GCC Options That Control Optimization*, <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.
- [7] *GCC Options That Control Static Analysis*, <https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html>.
- [8] *GCC Program Instrumentation Options*, <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>.
- [9] *Invoking gcov*, <https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html>.
- [10] Intel, *A Technical Look at Intel's Control-flow Enforcement Technology*, <https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html>, 2020.

-
- [11] Google, *AddressSanitizer*, <https://github.com/google/sanitizers/wiki/AddressSanitizer>.
 - [12] Google, *AddressSanitizerLeakSanitizer*, <https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer>.
 - [13] Google, *AddressSanitizerFlags*, <https://github.com/google/sanitizers/wiki/addresssanitizerflags>.
 - [14] T. Yin and other contributors, *Lizard*, <https://github.com/terryyin/lizard>.
 - [15] J. C. Miller and C. J. Maloney, *Systematic mistake analysis of digital computer programs*, <https://dl.acm.org/doi/10.1145/366246.366248>, Communications of the ACM, 1963.
 - [16] M. Verardo, F. Pinelli and L. Galletta, *A pipeline to validate agentic AI parser generation*, https://github.com/mirko-verardo/md2sl_thesis_mv, 2025.
 - [17] M. Bray, M. D. Luginbuhl, K. Brune, W. Mills, D. A. Fisher, R. Rosenstein, J. Foreman, D. Sadoski, M. Gerken, J. Shimp, J. Gross, E. Van Doren, G. Haines, C. Vondrak and E. Kean, *C4 Software Technology Reference Guide: A Prototype*, https://www.sei.cmu.edu/documents/1625/1997_002_001_16523.pdf, U.S. Department of Defense, Pages 145–149, 1997.
 - [18] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press and K. R. Narasimhan, *SWE-bench: Can Language Models Resolve Real-world Github Issues?*, <https://openreview.net/forum?id=VTF8yNQM66>, The Twelfth International Conference on Learning Representations, 2024.
 - [19] SWE-bench, *SWE-bench Official Leaderboards*, <https://www.swebench.com>.
 - [20] Robert C. Martin, *The Clean Coder: A Code of Conduct for Professional Programmers*, Prentice Hall, 2011.