A pipeline to validate agentic AI parser generation

Supervisor:
Prof. Fabio Pinelli

Candidate:
Mirko Verardo

Co-supervisor:
Dr. Letterio Galletta

Department of Statistics, Informatics, Applications «Giuseppe Parenti»
Master in Data Science and Statistical Learning (MD2SL)
Academic Year 2024-2025

Outline

- Objectives
- Application domain
  - Parsing functions in the C programming language
  - Agentic AI systems
- The validation pipeline
  - Frameworks (LangChain and LangGraph)
  - Architectures (single-agent vs multi-agent)
- Experiment setup and results
  - Overall results
  - Research Question results
- Conclusions and future developments

Objectives

- Introduce an **agentic pipeline** to create and validate parsers, with particular focus on **testing** and **cybersecurity analysis** of code.
- Advance the state of the art on autonomous code generation **benchmarking** with a **new approach** based on vulnerability assessment.
- Fix issues of the previous work *«LLM-based parser generation»*.
- Increase the dataset for *«ParserHunter»* project with **cybersafer** synthetic parsing functions.

Application domain: parsing functions

- A function is considered a **parsing function** (or simply **parser**) if it satisfies the IIDDO requirements or, alternatively, if it satisfies the Composition (C) requirement:
  - **Input handling (I)**: must use a pointer to a buffer of bytes or a file descriptor for reading unstructured data.
  - **Internal state (I)**: must keep in memory an internal parsing state in memory.
  - **Decision-making (D)**: must take decisions based on the input and the internal state.
  - **Data structure creation (D)**: must build a data structure representing the accepted input of the parsed file.
  - **Outcome (O)**: must return a data structure built from the parsed data indicating the outcome of the parsing.
  - **Composition (C)**: must be defined as a composition of other parsers.

Application domain: C programming language

- The **C programming language** is one of the most used one for parser implementation for its **high performance**. In fact, C has a minimal runtime overhead thanks to its **low-level abstraction** between the source and the executable code.

- However, C parsers are error-prone and high-risk targets of **cyber attacks** due to the lack of automatic memory management and built-in safety.

- The **GNU Compiler Collection (GCC)** is the main toolchain used for compiling and building C programs. It provides a lot of optimization options for:
  - **Static analysis**: performed at ***build-time***, without executing the code (e.g., **formal methods**).
  - **Dynamic analysis**: performed at ***run-time***, during code execution (e.g., memory error detection).

Application domain: agentic AI systems

- An agentic AI system can achieve complex tasks autonomously (i.e., **without recurrent human input**). It can be **single-agent** or **multi-agent**.

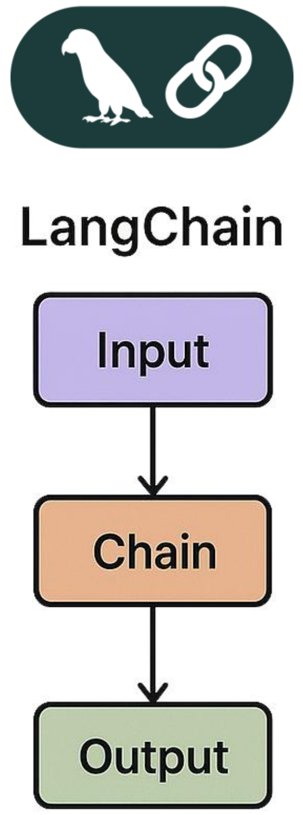
- An agent that uses a **LLM** to produce its output could improve performance through a **reasoning paradigm** (i.e., a step-by-step definition of how to solve the task).
  - Reasoning is usually activated through **prompting**.
  - e.g., **Reasoning and Acting (ReAct)** paradigm (will see an example soon).

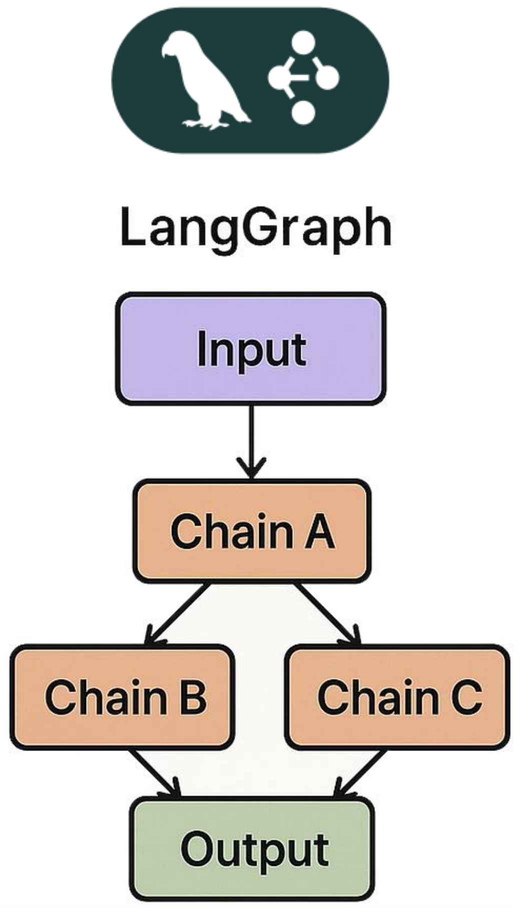
- A **tool** for an agent is an external software component invokable during reasoning. The tool output becomes **additional context** for the agent.
  - Tools are usually activated through **prompting** and integrated through **frameworks**.

The validation pipeline: frameworks

- **LangChain** is an open-source framework in Python that provides a **standard interface** for building LLM-centric workflows that combine prompt templates and tools.

- **LangGraph** is a LangChain extension that models workflows as **stateful directed graphs** where each node represents a computational step (e.g., agent) and edges define transitions between them where a common state object (fully customizable) is updated and passed.





The validation pipeline: main steps

1. Parser code **generation** in C language
   - Based on **IIDDOC requirements**.
   - More formats (user choice): **CSV, HTML, HTTP, JSON, PDF, XML**.

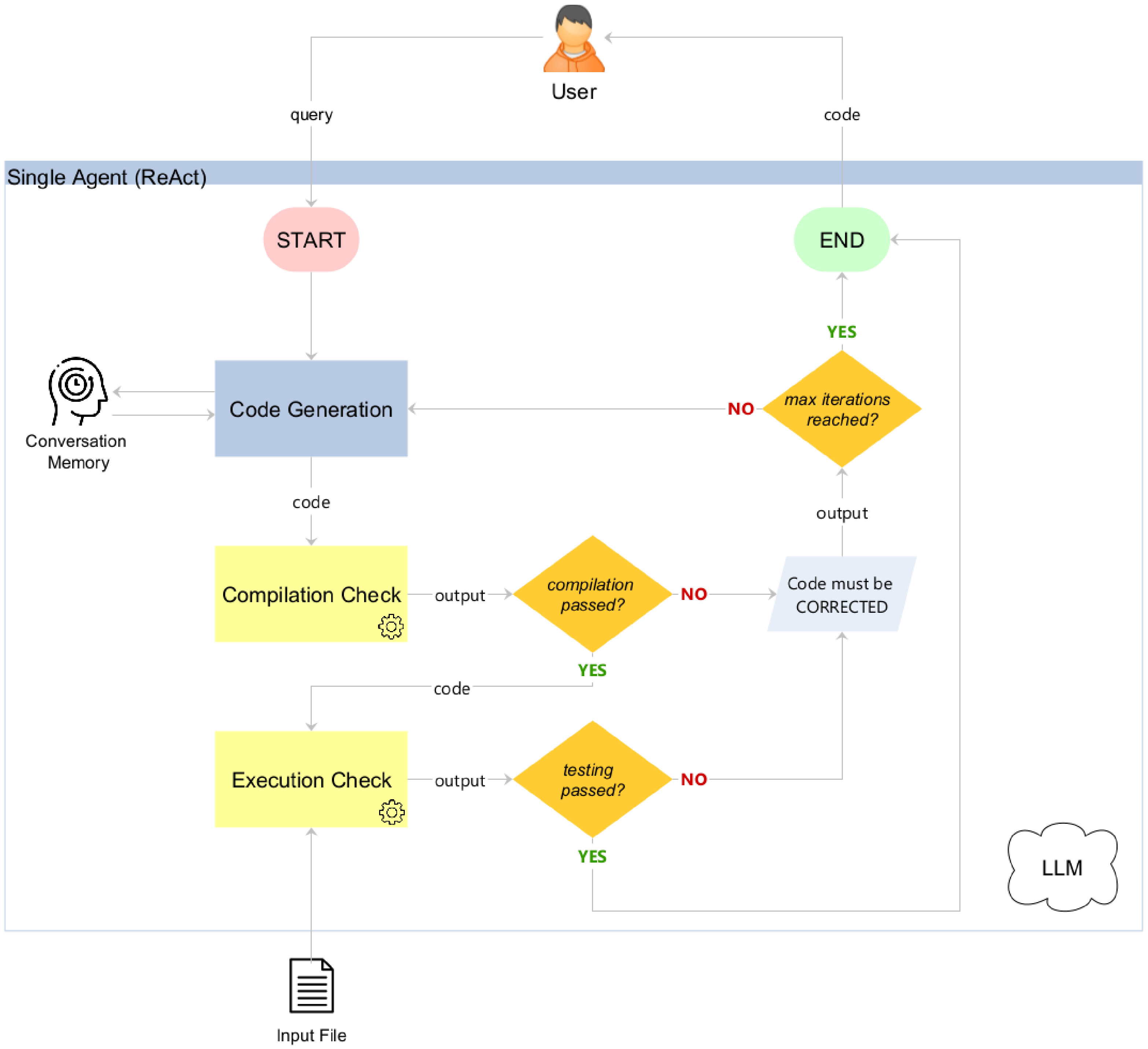
2. Code **compilation**
   - **Static and dynamic analysis** profiles through GCC.
   - If errors, they are given as input to the generation step to which the system goes back.

3. Code **testing**
   - **Parse the entire content of a file** in the chosen format (given as input) and print as output a text normalized summary of the parsed structure.
   - If errors, they are given as input to the generation step to which the system goes back.

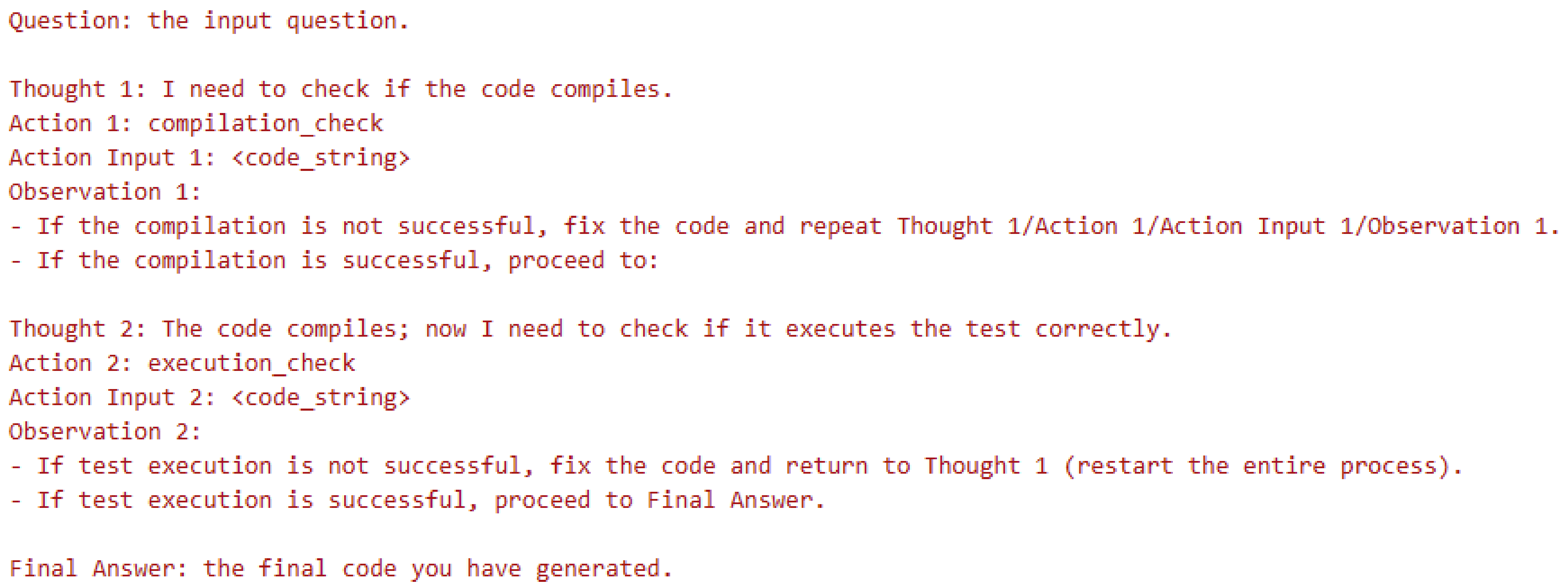
4. Loop until **satisfaction** or **max number of attempts** reached.

The validation pipeline: single-agent architecture

- The validation is made through the **ReAct** paradigm.
- Compilation and Execution Check are **tools**.
- **Conversation Memory** is useful if the conversation goes on (but limited to the context window).



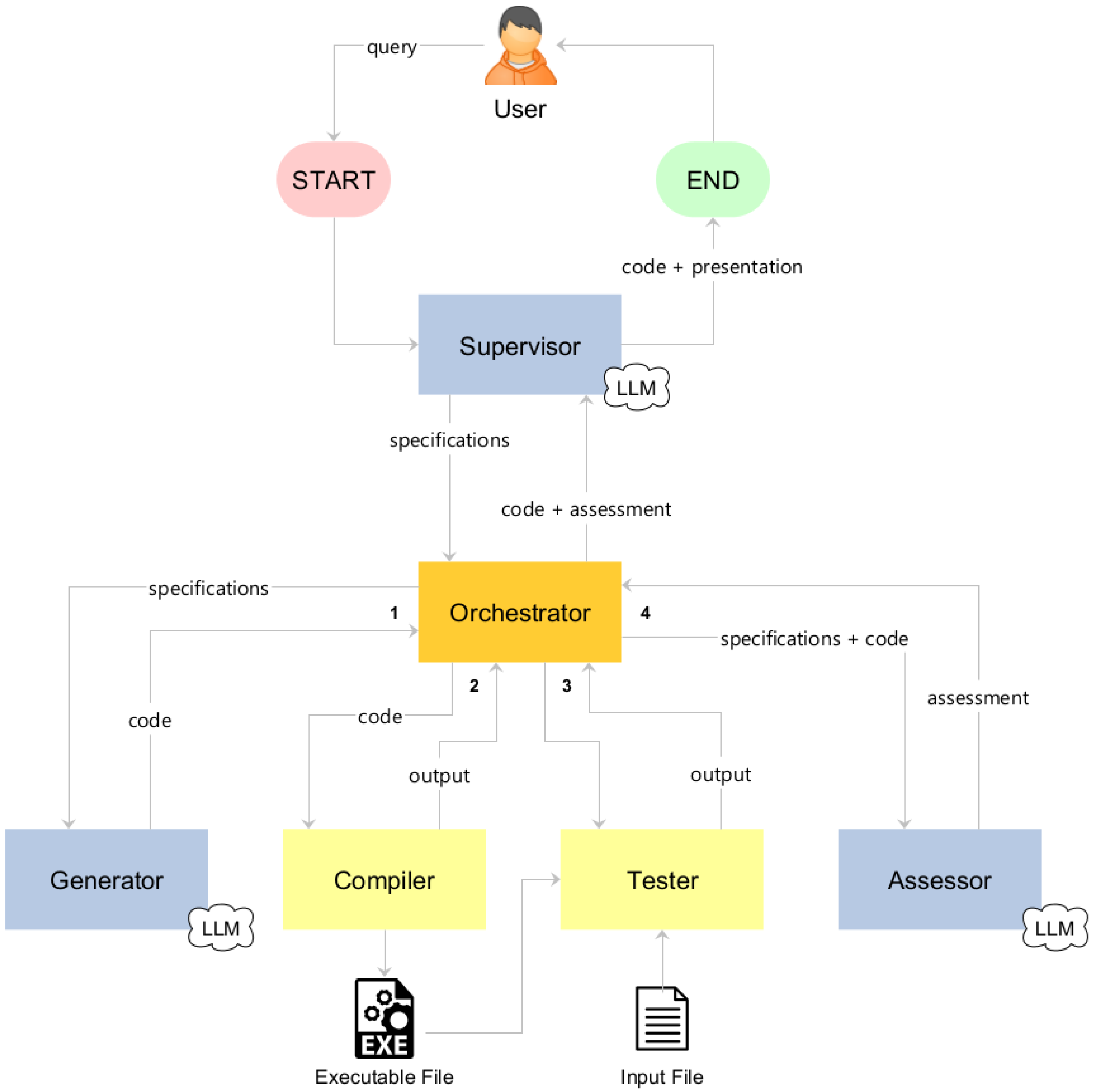
The validation pipeline: single-agent architecture (ReAct prompt)



```
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.
```

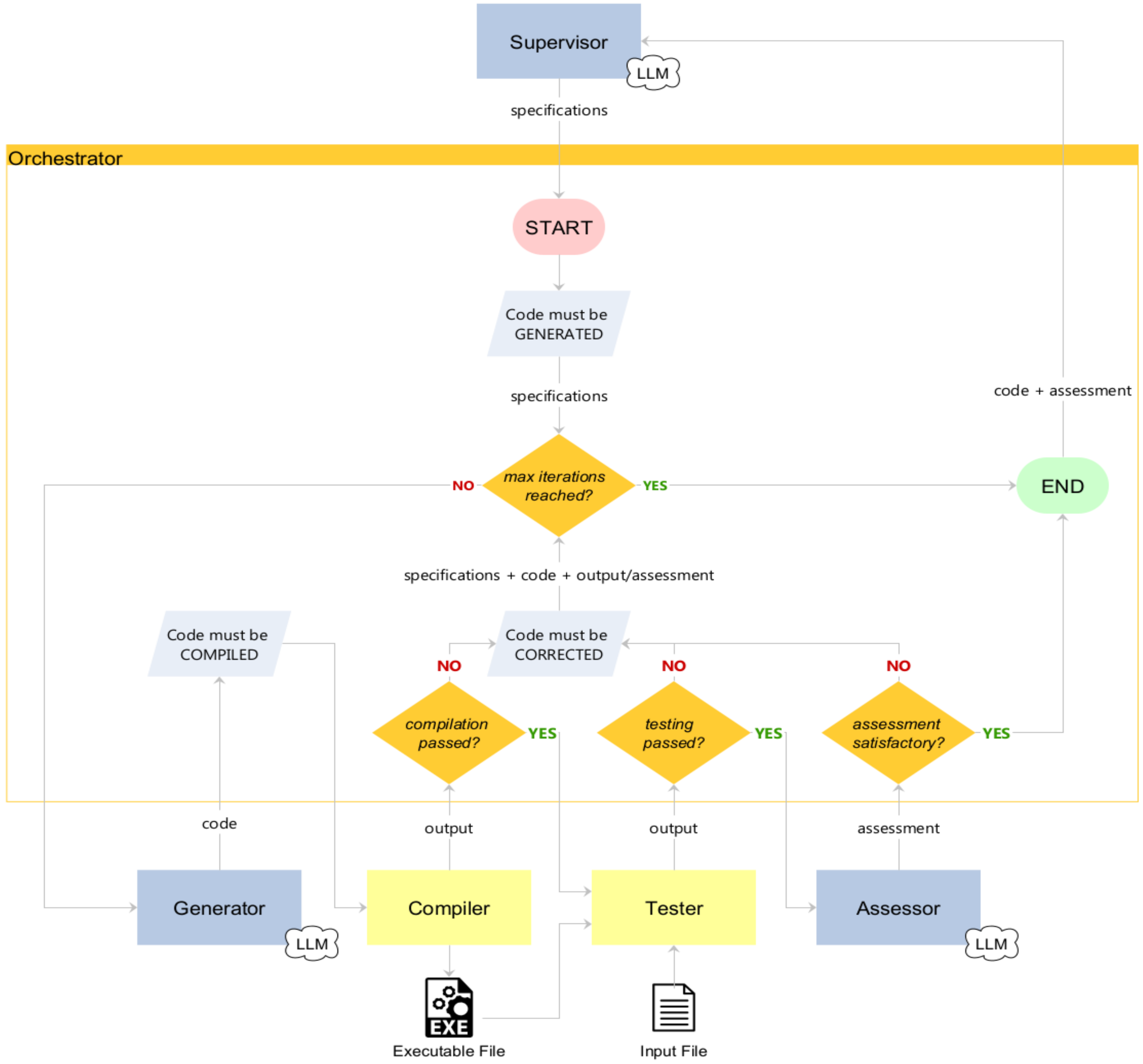
The validation pipeline: multi-agent architecture

- **Supervisor**: interacts with the user to create formal specifications for the parser and to present the final parser.
- **Assessor**: qualitatively verifies that code generated respects the initial specifications.
- **Orchestrator**: centralizes the communication between the agents.



The validation pipeline: multi-agent Orchestrator

- Addresses the workflow based on common state updates by other agents.
- Tracks the iterations count to decide if the limit has reached (pipeline end).
- Decide if the code is satisfactory (pipeline end).



Experiment setup

- The validation pipeline has been run **many times** for both **single-agent** and **multi-agent** architectures. During each round, some **benchmark metrics** have been tracked.

- Each round is **independent** from each other and it is initialized with a different set of parameters: **LLM**, **file format** to generate parser for and a random starting **seed**.

- Different LLMs are initialized with the **same fixed parameters** for each round: *max_iterations*, *max_output*, *temperature*, *timeout* and *max_retries*.

- The **user query** is injected to be equal for each round: *«Generate a parsing function for {file_format} files»*

- In the end, a dataset with **828 parsers** is created.

Experiment setup: LLMs comparison

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%

SWE-bench Verified: LLM evaluated with a minimal agent architecture on real-word GitHub issues. Evaluation is expressed only as **success rate** determined by **unit test verification** and human-validated for quality.

Experiment setup: benchmark metrics

- **Compilation** and **Testing** rate and iterations.
  - **Rate**: successfully compiled or tested parsers (global)
  - **Iterations**: min number of attempts to compile or test successfully the parser (1 to 15)

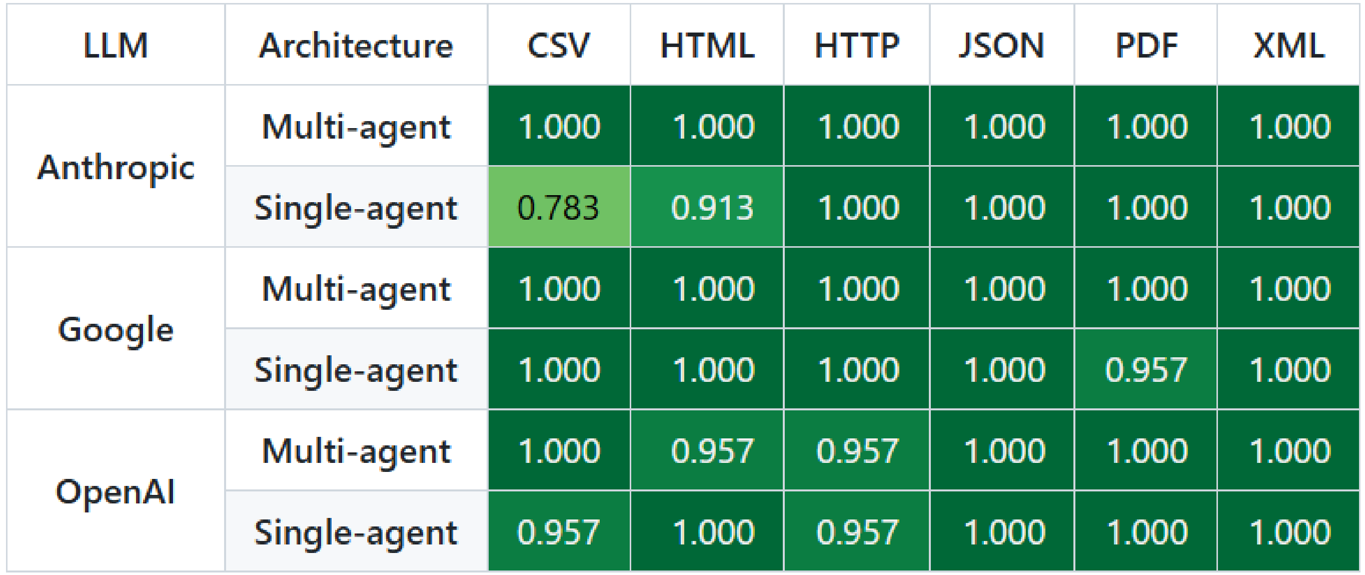
- **Execution time**: total pipeline duration in seconds.

- **Cyclomatic complexity**: the number of independent paths in the control-flow graph of the code.
  - Based on decision points (i.e., if, loops, etc.)
  - Natural number (higher the metrics, higher the risk)

- **Code coverage**: the rate of the code being executed when test is run.
  - Float number from 0 to 1 (higher the metrics, lower the risk)

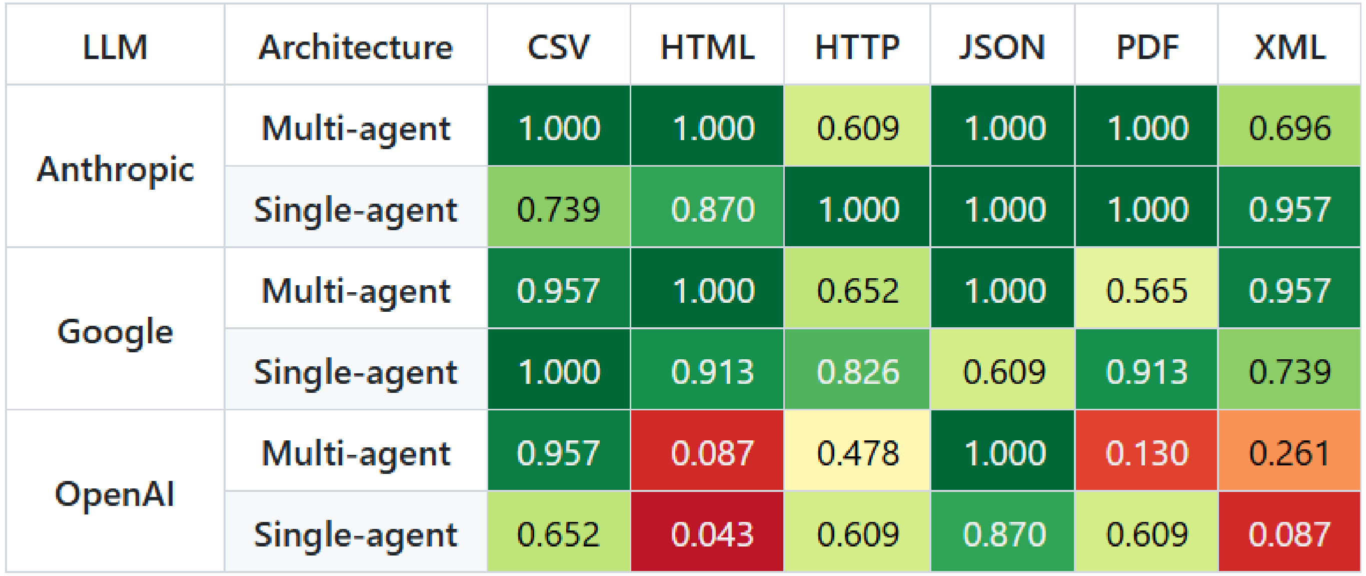
Overall results: compilation and testing rate

Compilation rate



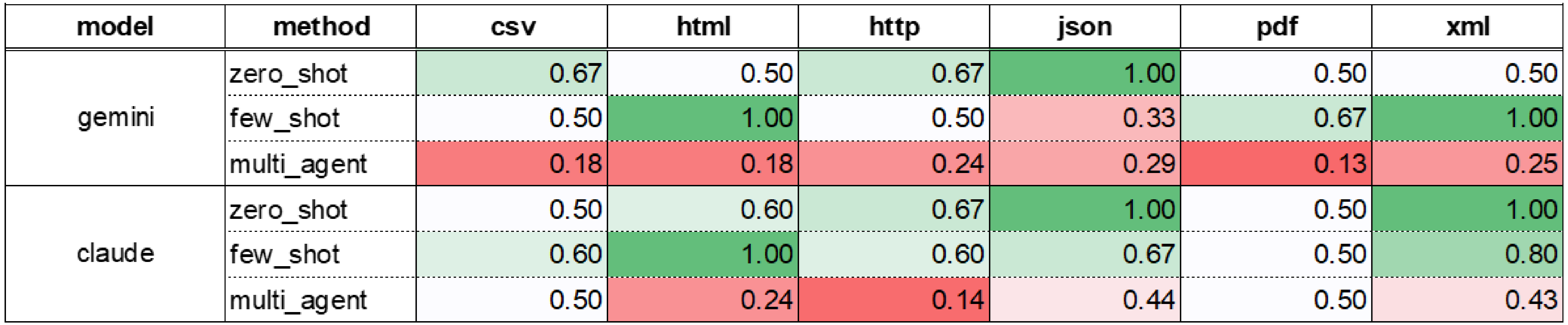
LLM	Architecture	CSV	HTML	HTTP	JSON	PDF	XML
Anthropic	Multi-agent	1.000	1.000	1.000	1.000	1.000	1.000
	Single-agent	0.783	0.913	1.000	1.000	1.000	1.000
Google	Multi-agent	1.000	1.000	1.000	1.000	1.000	1.000
	Single-agent	1.000	1.000	1.000	1.000	0.957	1.000
OpenAI	Multi-agent	1.000	0.957	0.957	1.000	1.000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

Testing rate



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

Overall results: compilation rate on previous work



model	method	csv	html	http	json	pdf	xml
gemini	zero_shot	0.67	0.50	0.67	1.00	0.50	0.50
	few_shot	0.50	1.00	0.50	0.33	0.67	1.00
	multi_agent	0.18	0.18	0.24	0.29	0.13	0.25
claude	zero_shot	0.50	0.60	0.67	1.00	0.50	1.00
	few_shot	0.60	1.00	0.60	0.67	0.50	0.80
	multi_agent	0.50	0.24	0.14	0.44	0.50	0.43

- Single agent systems compilation rates was not enough to satisfy the expectations.
- Multi-agent system was particularly bad.
- Current work has **fixed compilation for both** architectures.

Research Question

- **RQ**: *«How does agentic AI architecture affect the metrics considered on parser generation?»*

- Choose for each metric the **best architecture** and estimate its **effect** through:
  - **Classical inference** methods
  - **Generalized Linear Model (GLM)** to control for the other predictors (LLM and file format)

RQ results: Classical inference for each metric (1)

- Plots, sample median, mean and standard deviation calculated (see on next slide).

- **95% Confidence Intervals (CI)** calculated for each sample mean using the quantile function of **t-distribution** (large sample size). Since samples are right skewed, bootstrap CI have been calculated too.

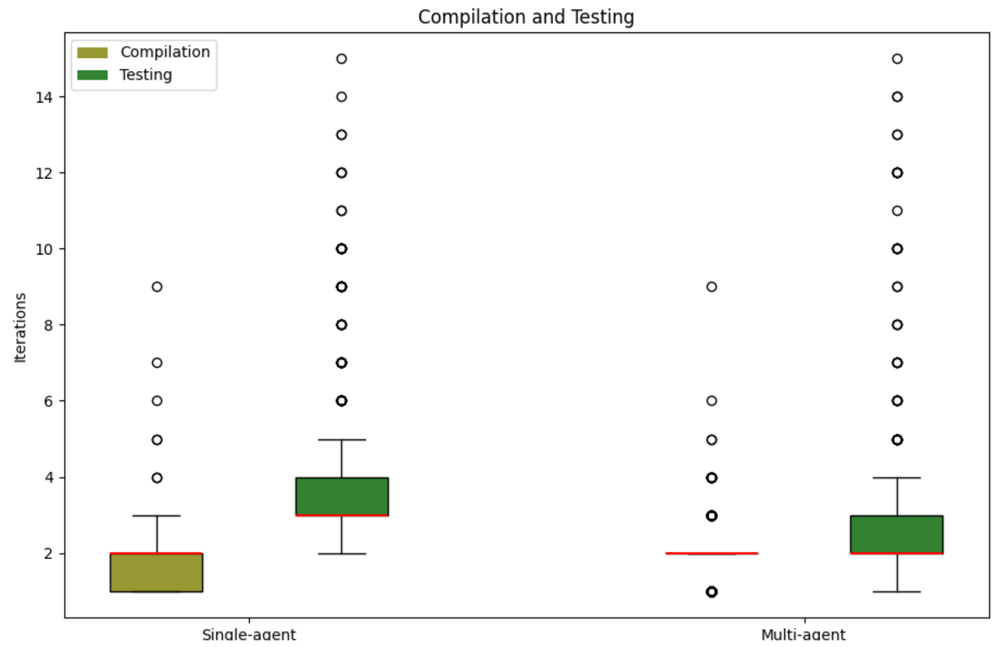
- **Parametric hypothesis tests (Welch’s ANOVA)** to check whether the difference between the architectures is statistically significant or not.

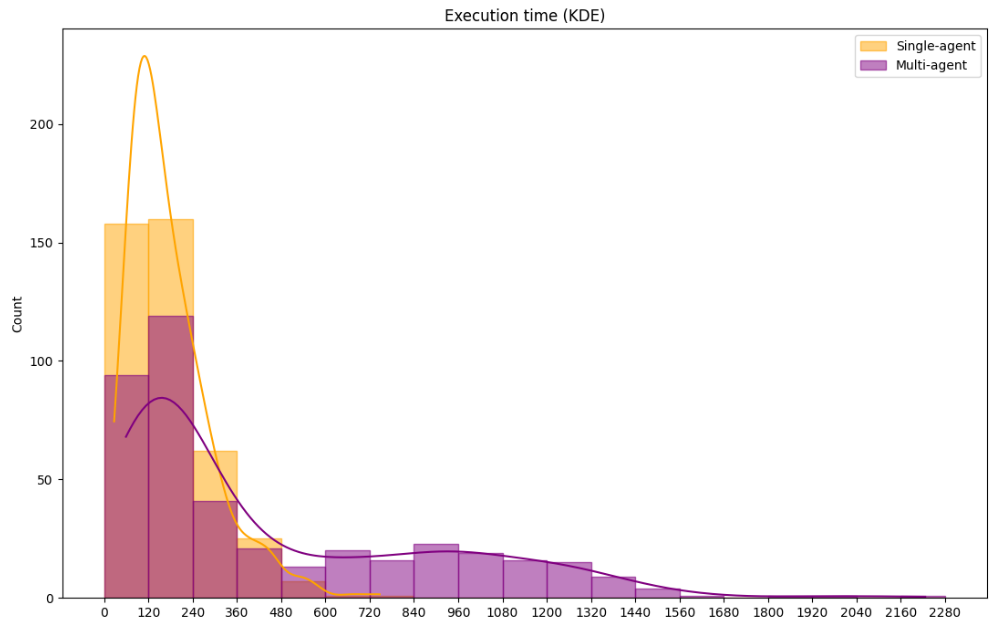
$$H_0\colon \mu_{ma} = \mu_{sa}$$

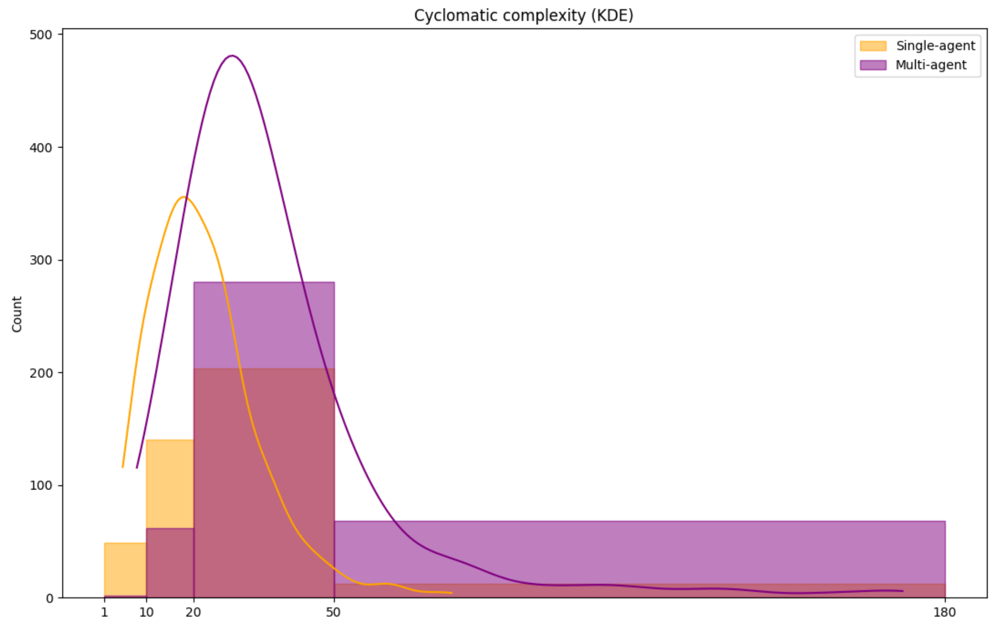
- **Cohen’s *d* effect** is calculated for each pairwise sample mean difference to see the effect magnitude.

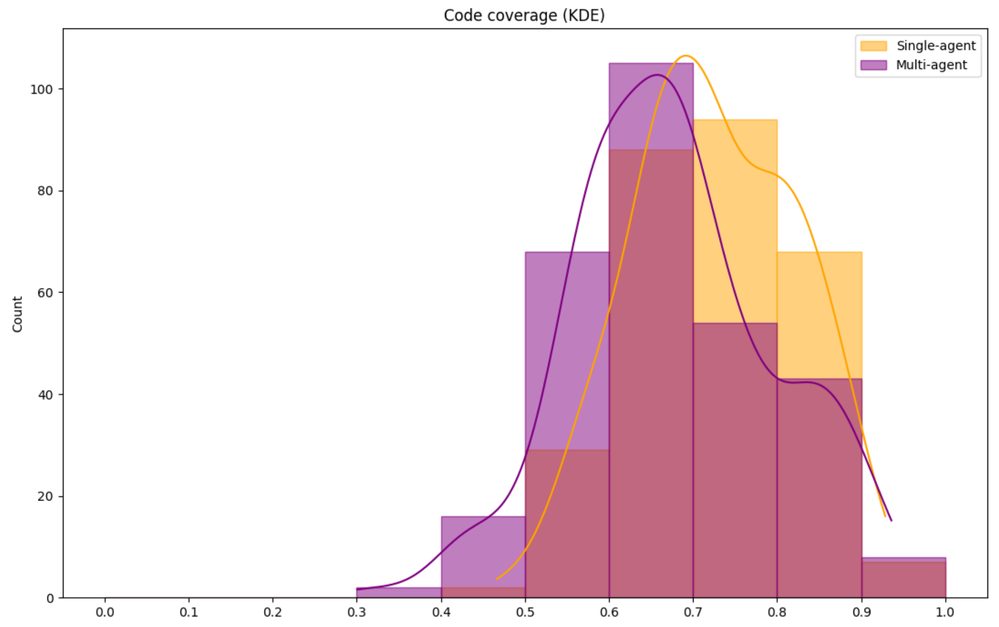
$$d = \frac{\overline{x_1} - \overline{x_2}}{s}$$

RQ results: Classical inference for each metric (2)



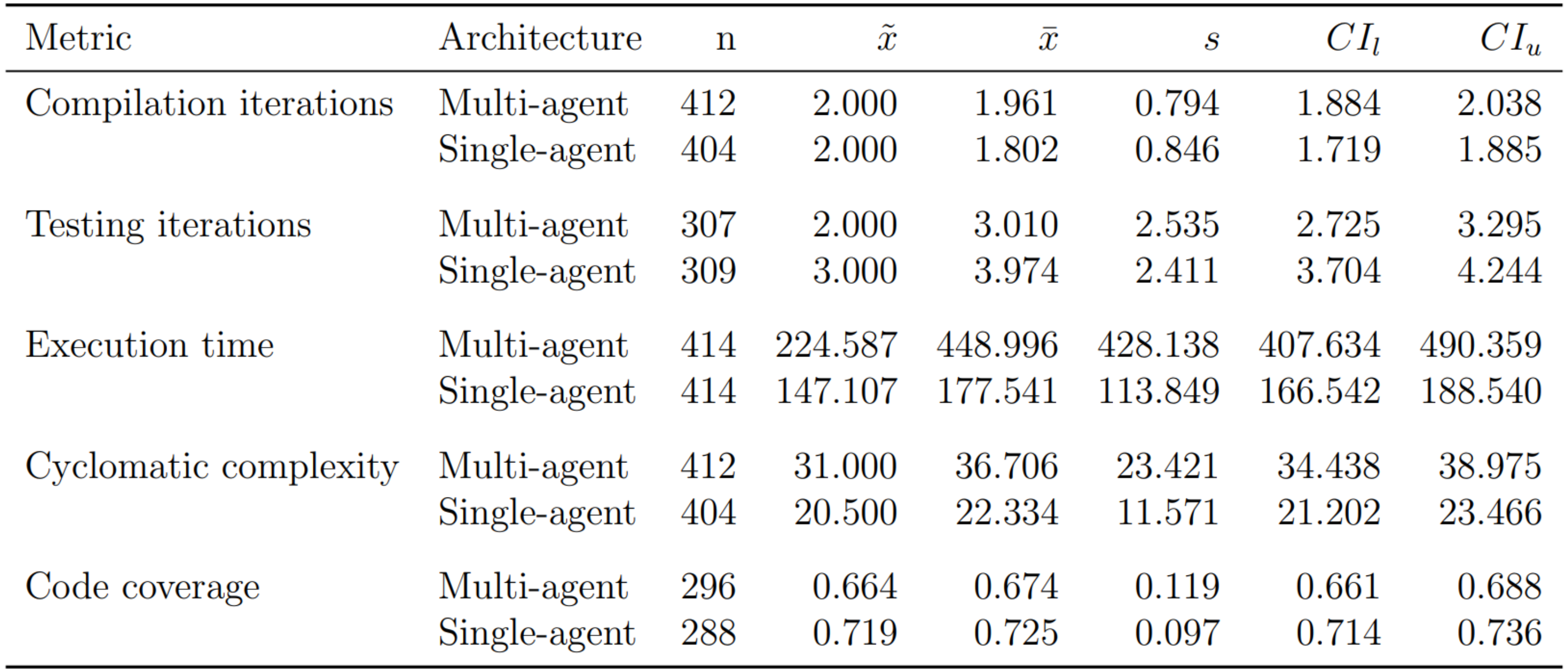




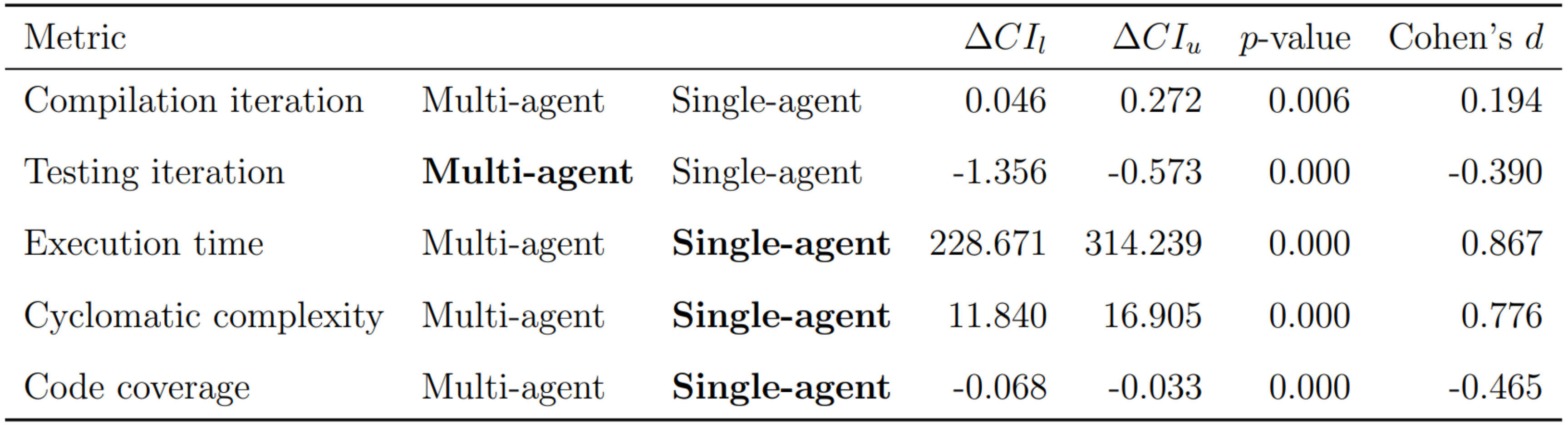


RQ results: Classical inference for each metric (3)

- *p*-values against H_0 are all statistically significant according to α = 0.05.
- *ΔCI* indicate the **absolute magnitude** (not crossing zero if significative).
- Cohen's *d* effect indicates the **effect magnitude** (standardized mean difference)
  - < 0.2 : low
  - > 0.7 : high



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



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

RQ results: Generalized Linear Model (1)

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

where:

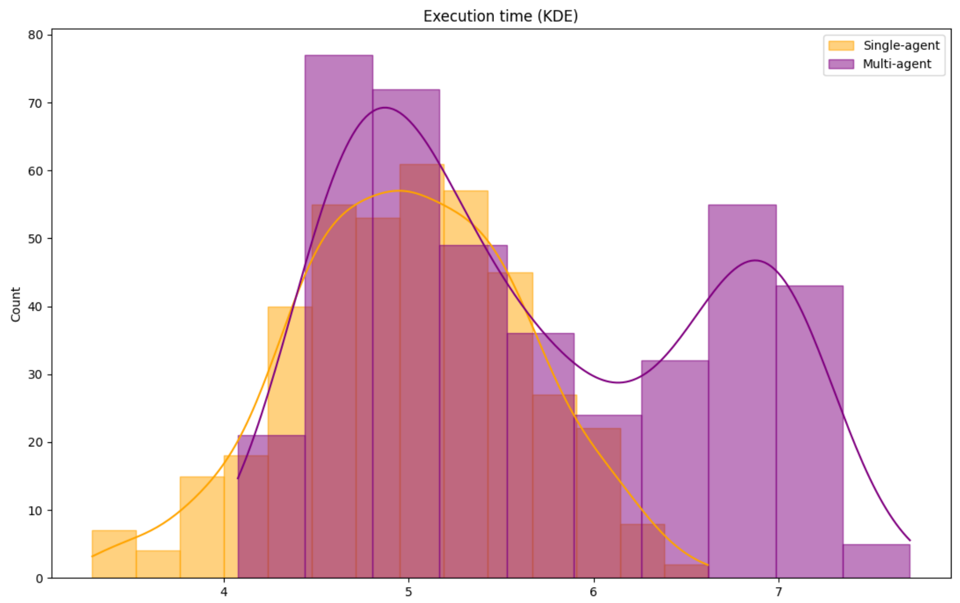
- Y: the considered metric outcome.
- $log(\cdot)$: the link function chosen looking at the nature of Y and at its visual distribution.
- X_1, X_2, X_3: the architecture, LLM and file format predictor variables respectively (all categorical).

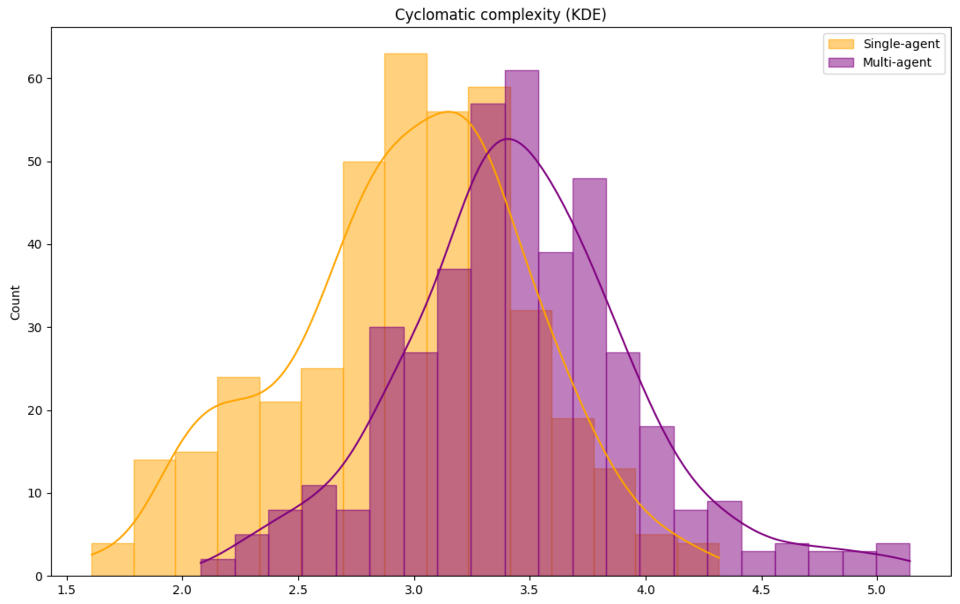
and with the following **ratio effect** estimate:

$$\frac{\mathbb{E}(Y \mid X_1 = 1)}{\mathbb{E}(Y \mid X_1 = 0)} = \frac{e^{\beta_0 + \beta_1 \cdot 1 + \beta_2 X_2 + \beta_3 X_3}}{e^{\beta_0 + \beta_1 \cdot 0 + \beta_2 X_2 + \beta_3 X_3}}$$

$$\frac{\mathbb{E}(Y \mid X_1 = 1)}{\mathbb{E}(Y \mid X_1 = 0)} = e^{\beta_1}$$

RQ results: Generalized Linear Model (2)



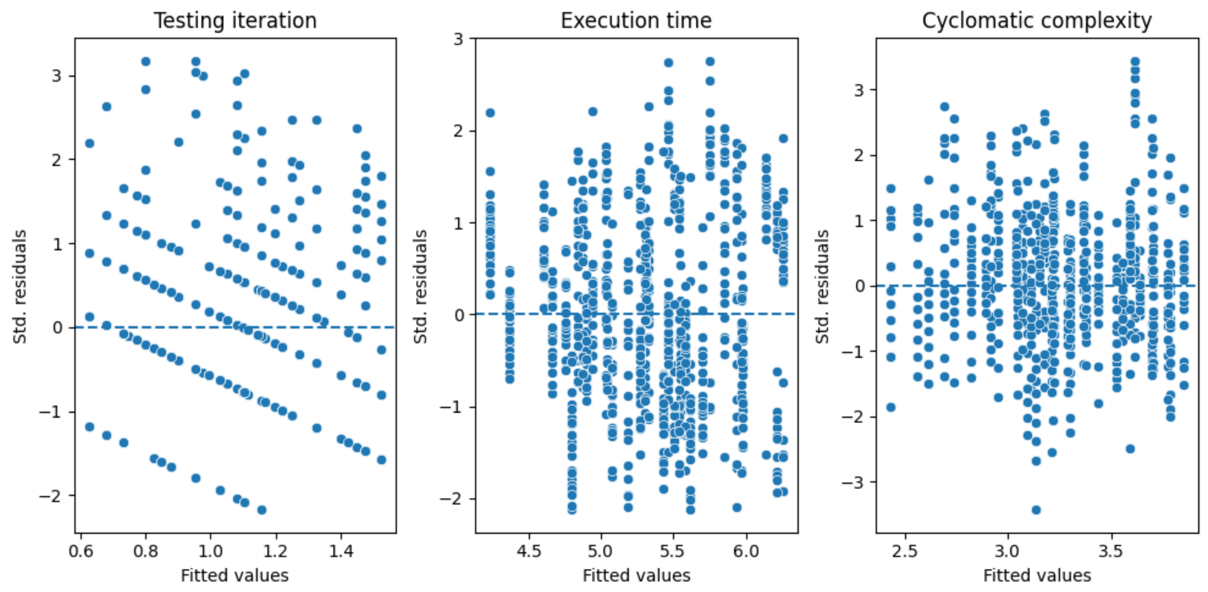


Metric	Model	Link	p-value	Effect	CI_l	CI_u
Testing iteration	Negative Binomial	log	0.000	1.324	1.183	1.481
Execution time	Gaussian	log	0.000	0.280	0.238	0.330
Cyclomatic complexity	Gaussian	log	0.000	0.581	0.535	0.631

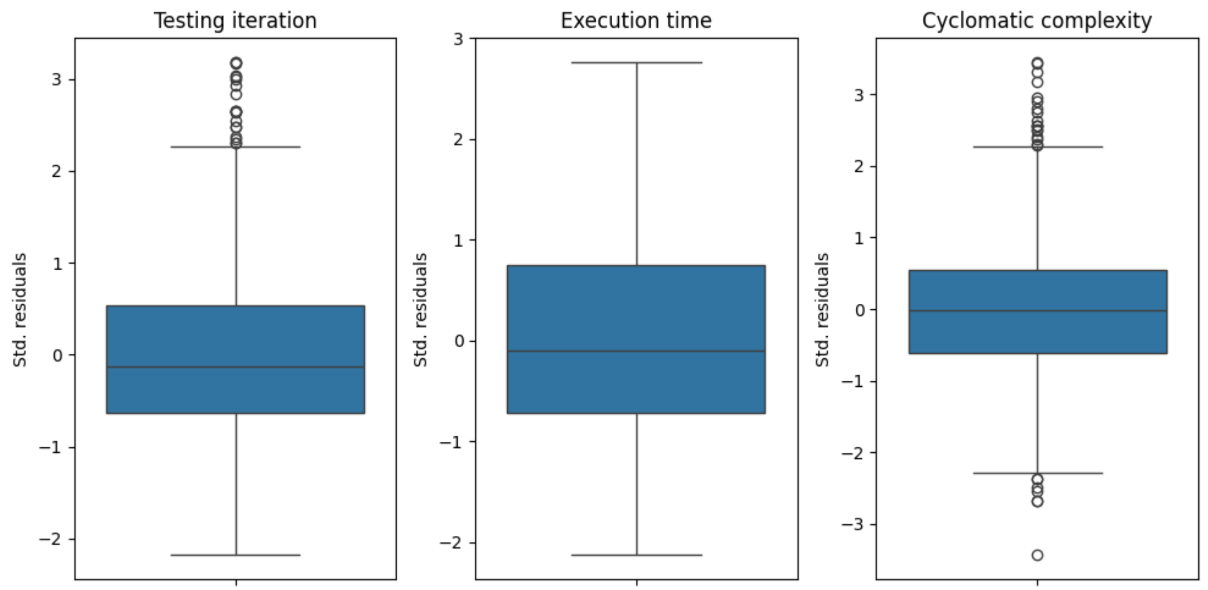
- *p*-values for architecture coefficients β_1 are all statistically significant according to α = 0.05.
- We can see the **multiplicative effect e^{β_1} on the mean** holding other predictors constant.
- e.g., Cyclomatic complexity: single-agent architecture expects a mean decrease of ≈ 42%.

Alternative: OLS model on log scale (residual analysis)

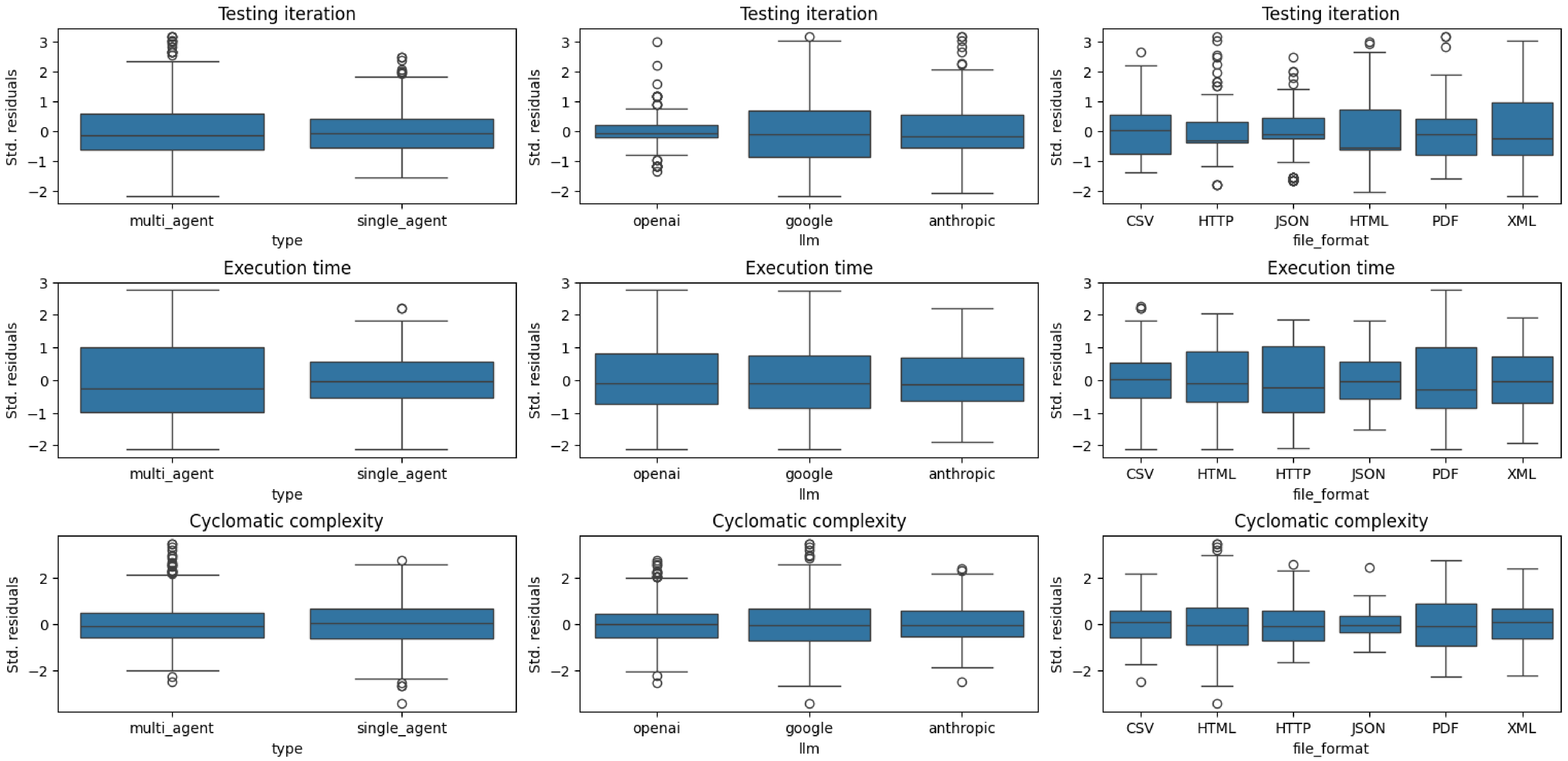
Check for **uncaptured component** of deterministic part.



Check for **Normal distribution** with **zero mean**.



Alternative: OLS model on log scale (check for heteroskedasticity)



Conclusions

- A validation pipeline that includes code **static and dynamic analysis**, **testing** and **vulnerability assessment metrics** tracking has been introduced:
  - Add **new heterogeneous parsers** to the dataset for «ParserHunter» project.
  - **Fix compilation** task from the previous work.
  - **Advance the state of the art** (e.g., SWE-bench) for autonomous code generation benchmarking (not only success rate).

- The **single-agent** architecture shows better benchmarks for code quality metrics while the **multi-agent** architecture has a more deterministic behavior and it is easier to manage and extend.

Future developments

- **Single-agent false negative** detection: since single-agent behavior is mostly delegated to reasoning, it could end sooner than expected and generate parsers that do not fully commit all the pipeline validation (but wrongly evaluated by the agent as so).

- **Test-Driven Development (TDD)**: is a software development paradigm that consists of iteratively writing a test case that fails, then writing just enough code to make the test pass, then repeating with another new test case.
  - If **test cases are pre-designed and given as skeleton** to the prompt, then the validation is shaped by design.
  - To adapt TDD to this thesis work, **specific test cases and prompts should be produced for each file format** considered. Even better if more test files for the same format with incremental parsing level complexity or also wrong (fuzzing).

Acknowledgments

I would like to thank:

- Prof. **Fabio Pinelli** and Dr. **Letterio Galletta** from the SySMA Research Unit, for the collaboration during this thesis work.
- The **commission**, for the kind attention.