ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Agentic Graph of Thoughts: a Reasoning Framework for LLM Agents

Tesi di laurea in:
Intelligent Systems Engineering

Relatore
Prof. Giovanni Ciatto

Candidato
Marco Raggini

Correlatori
Dott. Mattia Matteini
Prof. Andrea Omicini

I Sessione di Laurea
Anno Accademico 2026-2027

Abstract

Modern Large Language Models (LLMs) exhibit structural limitations that reduce their reliability, including opaque reasoning, error propagation, and limited exploration of alternative solution paths. Existing prompting techniques such as Chain of Thoughts (CoT), Tree of Thoughts (ToT), and Graph of Thoughts (GoT) partially make reasoning more explicit, but GoT-style approaches usually require task-specific graph structures designed before execution. This thesis introduces an Agentic Graph of Thoughts (AGoT) architecture based on a runtime-generated graph, in which the LLM reasoning is guided by a directed graph constructed dynamically during execution. Such a graph contains various kinds of nodes, each representing a specific step: reasoning generation, tool invocation, response evaluation, and backtracking to alternative paths. Moreover, this approach supports the dynamic tool crafting mechanism when no available tool is sufficient to solve the problem. A dedicated agent generates and registers new tools at runtime, extending the system's capabilities without requiring human intervention. Intermediate reasoning process is evaluated by an LLM judge with an adaptive threshold based on the estimated complexity of the problem. The framework is evaluated on the Hendrycks MATH benchmark, comparing the proposed architecture against a Zero-Shot CoT baseline across three different Gemini models. The results suggest that the AGoT can make more robust models with stronger reasoning capabilities, where the backtracking mechanism and multi-path exploration help recover from incorrect intermediate steps.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The ability of LLMs to solve complex problems has improved dramatically in recent years, enabling their adoption in a wide range of real-world applications, from code generation to mathematical reasoning. Despite these advantages, the standard generation approaches still has limitations e.g., the lack of explainability, hallucinations, low output quality and the exploration of alternative solution paths is either absent or severely constrained.

The introduction of prompt engineering techniques such as CoT or ToT partially adress these problems by structuring the reasoning process into intermediate steps or branching trees. These approaches correspond to progressively expressive data structures, yet both remain constrained: linear sequences prevent backtracking, and trees does not allow the merging or reuse of intermediate results across different reasoning paths. To overcome the structural limitations, the GoT [BBK+24] architecture is introduced, representing the reasoning process as a directed graph in which thoughts can be generated, aggregated and refined, enabling the forms of exploration and combination that were not possible before. In its original formulation, the GoT requires the user manually define the graph structure before execution begins, removing the possibility of a general purpose deployment.

This thesis aim to addresses this limitation by proposing an AGoT architecture based on a runtime generated graph, with a focus on the use of tools. In particular, in order to solve some of the previously cited problems, several mechanism are designed such as a backtracking strategy that allows the system to recover from

incorrect intermediate steps, a dynamic tool crafting mechanism where the LLM generates and registers new tools at runtime when the existing toolset is insufficient, and an LLM-as-a-judge evaluator for ensuring the quality of the response.

The framework is evaluated on the Hendrycks MATH benchmark [HBK+21], where the results suggest that the proposed framework improves reasoning robustness in sufficiently large models, with the backtracking and multi-path exploration mechanisms helping recover from incorrect intermediate steps, at the cost of increased latency and token consumption.

This thesis led to the submission of a paper to the AIxIA 2026 conference.

Structure of the Thesis Chapter 2 of this thesis introduces the background, covering LLMs, prompt engineering techniques, and the related works that motivate this thesis. Then, with Chapter 3, the design of the proposed system, including the formal definition of the static and dynamic graph structures, the agent roles, and the execution model, will be presented. Chapter 4 describes the implementation of the previous proposed design, detailing the build automation, the agentic frameworks adopted, and the crafting methodology. The effectiveness of the system is evaluated in Chapter 5, which reports the experimental results on the Hendrycks MATH benchmark, discussing performance, common errors, and representative case studies. Then, Chapter 6 discusses the implications of the results and the limitations of the proposed approach. Eventually, Chapter 7 draws the conclusions and outlines directions for future work.

Chapter 2

Background

2.1 Large Language Models

LLMs are a class of models which belong to Deep Learning, a sub-category of the broad field of Artificial Intelligence (AI). In recent years, they have become extremely popular thanks to their ability to understand and generate natural language with a high level of fluency and coherence [ZZL+26]. Their generalized knowledge and versatility make them perfect for a wide range of applications, from text generation and translation to code assistance, question answering, and conversational systems.

The ability to generate natural language fits perfectly with the need to create systems that can be used by everyone without requiring specific technical knowledge. LLMs allow users to interact using natural language, which enables to access them through simple conversational interfaces, making human-computer interaction more intuitive and accessible.

During the last years, after the release of ChatGPT, the average number of arXiv papers that contain the *"large language model"* title or abstract are deduplicated [ZZL+26], leading to the development of increasingly powerful models such as

GPT-4 [Ope23], LLaMA [TMS+23], Gemini [Tea25], and Claude [1]. These systems are capable not only of understanding and generating text, but also of solving reasoning tasks, summarizing documents, writing code, and supporting decision-making processes, becoming more and more used in real-world scenarios.

2.1.1 How they work

LLMs are autoregressive models that simply try to predict the next word in a sequence of characters, based on the context.

As shown in fig. 2.1, the text given as input to the LLM is preprocessed and divided into smaller units called *tokens*. A token may represent a word, part of a word, or even a symbol, depending on the strategy chosen. The next step is to convert the tokens into numbers, in such a way that a neural network can process them.

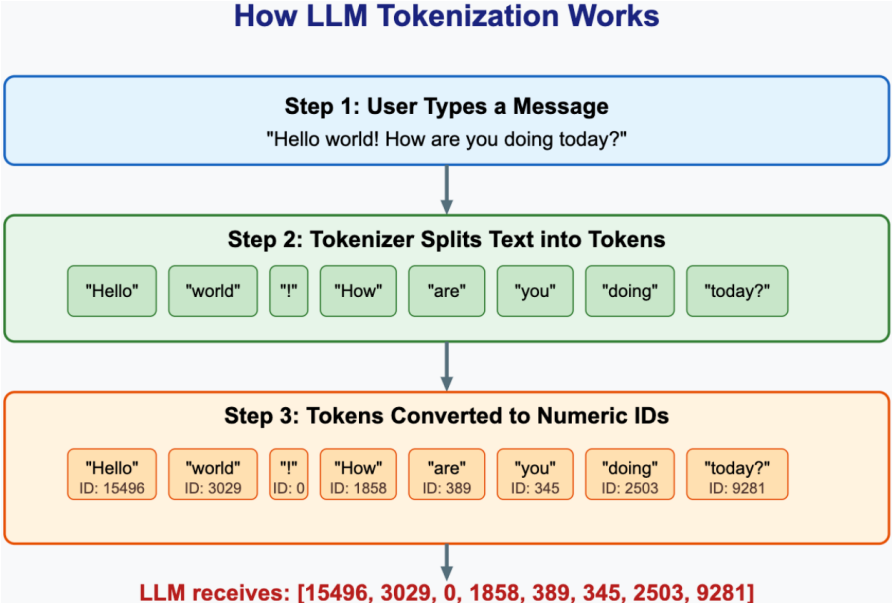

Figure 2.1: Schematic workflow of an LLM tokenizer: from raw input text to token segmentation and subsequent mapping into vocabulary IDs. Source: `https://www.linkedin.com/pulse/llm-tokenizers-hidden-engine-behind-ai-language-models-natarajan-rmbsc`

[1] `https://assets.anthropic.com/m/61e7d27f8c8f5919/original/Claude-3-Model-Card.pdf`

Modern LLMs are typically based on the Transformer architecture introduced by Vaswani et al. [VSP+17]. The Transformer uses the self-attention mechanism to capture contextual relationships between tokens, allowing the model to understand long-range dependencies and generate coherent text.

Once the tokenization is done, the text generation starts using autoregressive algorithm: a token is generated and appended to the input sequence and used to predict the following token. By repeating this process multiple times, the model can generate complete sentences, paragraphs, or entire documents.

Despite the apparent intelligence of these systems, LLMs do not possess real understanding or reasoning capabilities in the human sense. Their outputs are generated through statistical pattern matching learned from training data (e.g. fig. 2.2), which explains their tendency to occasionally produce incorrect or hallucinated information [XJK25] and the standardized phrases that often generates.

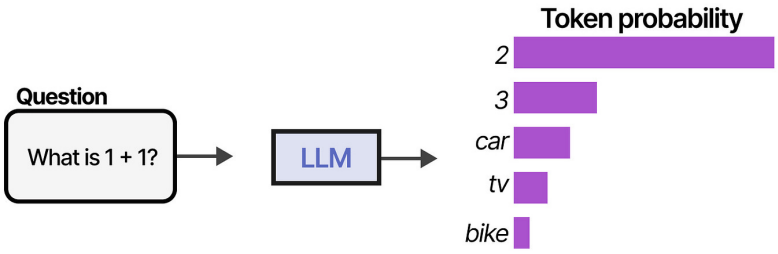

Figure 2.2: Schematic overview of autoregressive decoding and probabilistic next-token prediction in an LLM. Source: `https://albanna-tutorials.com/llm_agents.html`

2.1.2 Prompt Engineering

Prompt engineering is a relatively new field of research that refers to the practice of designing and implementing prompts or instructions trying to guide the LLM output. The main goal of prompt engineering is to optimize the LLM output by solely refining the prompt [Mes23].

The prompt could include a lot of components, such as: *Directives*, *Examples*, *Output formatting*, *Style instructions*, *Role* or *Additional Information* [SIB+24].

All of these components can help to improve the generated output in terms of a better style, accuracy in the response, or even a more correct output.

One of the most popular and older prompt engineering techniques is the *Few Shot Prompting* [BMR+20], in particular, it has three sub categories:

- **Few-Shot**: We are referring to a prompt that includes a few examples of how the output should be structured with also the correct output. This context helps the LLM to learn how to complete the task given without updating the model parameters.
- **One-Shot**: It is equal to the few-shot prompting, with only one demonstration provided and the natural language description of the problem.
- **Zero-Shot**: In this case, no demonstrations are provided. The prompt contains only the natural language description of the task. This method has the maximum convenience, but could be challenging to apply, mainly due to the difficulty of imagining the format expected without producing examples.

These methods are not meant to compete with each other. There are three possibilities depending on the task we are facing. In general, few-shot prompting tends to perform better on structured or domain-specific tasks, while zero-shot prompting is often sufficient for simpler or well-known tasks [BMR+20].

2.1.3 LLM-as-a-judge

During the evolution of LLMs and prompt engineering techniques, new approaches are being explored to automate processes that are currently handled by humans, including evaluation. Traditional human evaluation is often expensive, time-consuming, and difficult to scale, especially when dealing with complex reasoning and open responses. To address these limitations, the LLM-as-a-judge technique was introduced [ZCS+23]. The idea of this technique is to use Generative AI to judge another LLM output; this method can benefits the output in scalability, explainability and has the advantage of limit the human involvement of the reasoning process.

The three approaches proposed to use the LLM-as-a-judge are:

- **Pairwise comparison** The judge is presented with one question and two or multiple answers. Its goal is to choose which one is better.
- **Single answer grading** It is directly asked to the LLM judge to assign a grade to the solution.
- **Reference-guided grading** In some cases, providing a reference solution may be beneficial.

Despite its advantages, LLM-as-a-judge approach also has several biases:

- **Position bias**: It refers to the propensity of LLMs to evaluate two different responses based on their position in the text.
- **Verbosity bias**: It is the preference to choose verbose responses even if the output is not accurate or complete.
- **Self-enhancement bias**: When the LLM favors the responses generated by the itself or by the same model.

Even if several mitigation strategies have been proposed, these biases have not been completely eliminated and remain an open research challenge.

2.1.4 Agents

The LLMs are now good enough to start thinking about how to automate some processes or how they can be used in more contexts. For these reasons, the agents are created. The formal definition is:

> "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." [RN10]

A high-level description of an agent consist of four main components:

- **Senses** the surrounding environment around it.
- **Perceives** the information of that environment.

- **Reasons** about how to act based on the input given.
- **Acts** upon the environment through its actions (the output) [RN10].

In the LLM context, the agent brain is supported by the LLM, which reasons using the context given (it could be the environment as a text), and it acts generating an output or by calling the **tools**.

Tools are external functions or Application Programming Interfaces (APIs) that an agent can invoke at runtime to overcome the limitations of the LLM, such as the lack of up-to-date knowledge, the inability to execute code, or the absence of persistent memory. By selecting and calling the appropriate tool based on the user's intent, the agent extends its capabilities well beyond pure language generation.

As previously mentioned, LLMs can be unpredictable and lacks of transparency. In this sense, the usage of tools enhances the credibility of the decision of an agent, improving interpretability and robustness [XCG+23].

A difference between pure LLMs and AI agents is the autonomy. While LLMs goal is to generate a correct and coherent output, the agents try to perceive the autonomy, minimizing the need for human intervention, and trying to adapt by changing their behavior based on feedbacks [Kum24]. This process is commonly represented as a closed loop between perception, reasoning, and action, as shown in Figure 2.3.

2.2 LangChain & LangGraph

2.2.1 LangChain

LangChain is an open-source framework launched in 2022 and designed to simplify the development of applications powered by LLMs. The LangChain framework has two main declarated core focuses [2]:

API Abstraction Different providers expose different APIs. The first goal of the framework is to standardize the way to interact with the main LLMs providers.

[2] `https://docs.langchain.com/oss/python/langchain/philosophy`

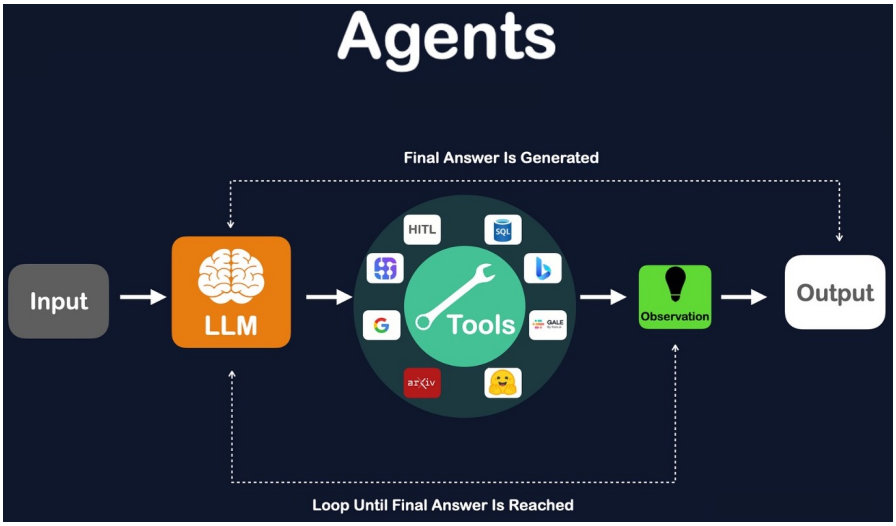

Figure 2.3: Representation of how an agents works. Source: `https://cobusgreyling.medium.com/how-would-the-architecture-for-an-llm-agent-platform-look-b07d7e004561`

This approach allows developers to focus on the application logic rather than dealing with the low-level details of each provider.

Tool Integration LLMs are more than text generators, and the research is moving to the direction of creating agents that interact with the environment and communicate with tools. The second goal is to provide a simple way to define and integrate tools that LLMs can use, it includes also the possibility of parsing and access to unstructured data.

The framework is designed with modularity as a core principle as shown in Figure 2.4 along with other relevant features. Rather than being distributed as a single package, LangChain is split into independent components that can be installed separately, such as `langchain-core`, `langchain-community`, and provider-specific packages like `langchain-google-genai`. This allows developers to include only the dependencies strictly required by their application.

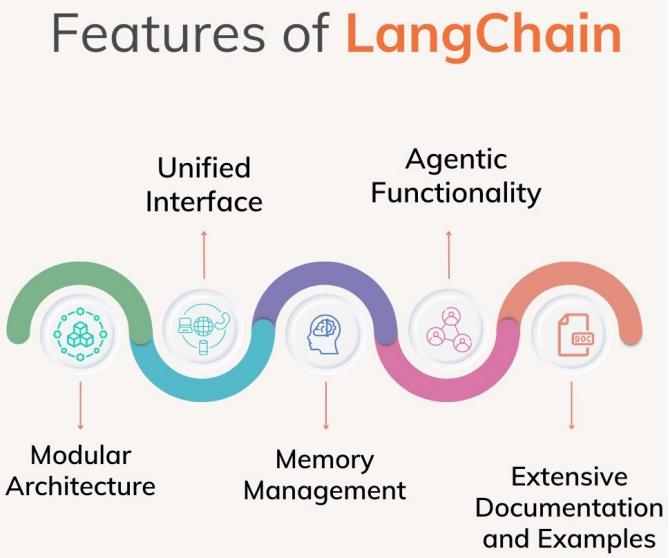

Figure 2.4: Representation of the LangChain features. Source: `https://www.ksolves.com/blog/artificial-intelligence/power-of-langchain-features-and-benefits`

2.2.2 LangGraph

In 2024, a new module called LangGraph was released [3]. Rather than LangChain, which is focuses on simplifying the interaction with LLMs and neglet low-level details, LangGraph is a low-level framework, focuses entirely on agent orchestration.

With LangGraph, developers can define complex agent workflows. The core abstraction of LangGraph is the *stateful directed graph*, where each node represents a computational step and edges define the transitions between them. Transitions can be either deterministic, where the next node is uniquely determined by the completion of the current step, or conditional, where the next node is selected dynamically based on the output of the current one. This makes LangGraph particularly well-suited for systems that require non-linear control flow, such as backtracking, branching, or iterative refinement of intermediate results.

[3] `https://docs.langchain.com/oss/python/langchain/philosophy`

2.3 Related Works

While few-shot prompting improves performance by providing input-output examples, it does not explicitly guide the model through the reasoning process required to solve a task. The research in this field has been evolved, finding new ways to improve the output quality without changing the LLM parameters.

2.3.1 Chain of Thoughts

Chain of Thoughts is an important prompting techniques introduced in [WWS+22]. CoT reveals that, when LLMs produce the reasoning steps of a solution, the correct results increase dramatically. This leads to several advantages:

- **Accuracy**: As we said previously, the number of corrected answers increase [WWS+22].
- **Interpretability**: The LLM remains a black box, but the final solution includes the steps, allowing to understand when the LLM produces an error in the reasoning.
- **Transparency**: In some task categories, such as math problems, the CoT techniques produces the intermediate reasoning which helps to independently reproduce the result given.

The CoT was firstly introduced as a few-shot approach, where the examples were used for inducing the multi steps reasoning. Recently it has been proved that also a zero-shot CoT increases the performance, introducing the "Let's think step by step" phrase into the prompt [KGR+22]. Figure 2.5 illustrates the differences between standard prompting, few-shot CoT, and zero-shot CoT, showing how these approaches improve the quality of the generated output.

Later on, the Self consistency CoT (CoT-SC) [WWS+23] was developed as an evolution of the classic CoT. The idea behind that was explore multiple reasoning paths in order to select the most accurate output. The empirical evaluation proved that CoT-SC boost the performance of CoT reasoning [WWS+23].

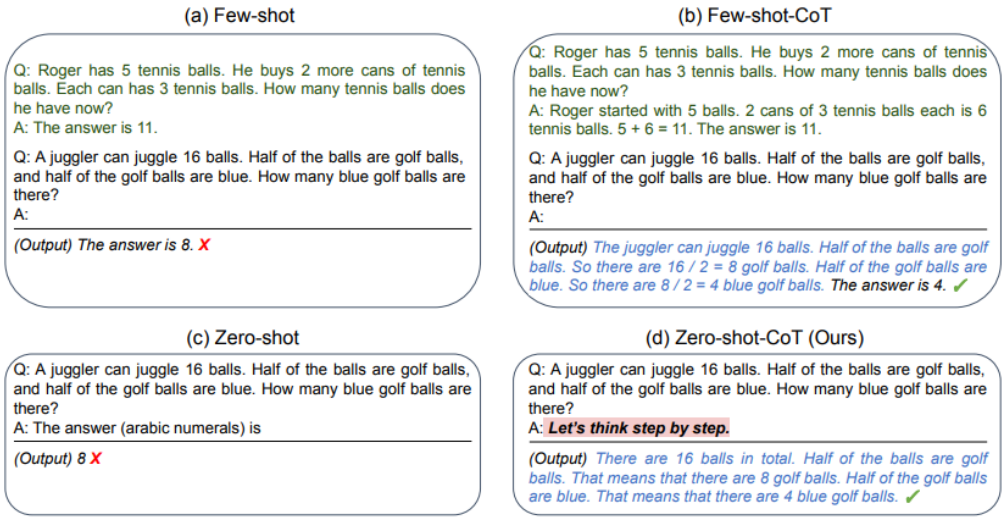

Figure 2.5: The differences between a normal prompt versus a few-shot CoT and a zero-shot CoT. Source: [KGR+22]

2.3.2 Tree of Thoughts

The introduction of the CoT approach paved the way for a new type of prompt engineering. From this method, the ToT was developed [YYZ+23]. While the CoT produces multiple steps with a single prompt, the ToT is based on the idea of generating multiple thoughts for each step and evaluating the best of them; each step is a partial solution and the evaluation phase guarantees the refinement of it into the final response. In order to achieve the best result, the architecture includes four important parts:

1. **Task decomposition**: In the ToT, the task decomposition is an explicit part. This should helps the LLM to decompose the problem until they are small enough to generate a satisfying solution.

2. **Thought generation**: There are two ways of generating thoughts. *Samples* are thoughts generated through a CoT prompt, they are useful when the solution space is rich. Instead, the *Propose* strategy generates thoughts through a prompt that proposes the possible solutions and it is more efficient when the solution space is restricted.

3. **State evaluator**: Two strategies were proposed in order to evaluate the

thoughts. In the *Value* strategy, the LLMs give a value to each thought from 1 to 10 and chose the higher one. In the *Vote* strategy, each evaluator choose the best thought and who takes more votes become the winner.

4. **Search algorithm**: It is used to search the more accurate solution in the tree; the strategies proposed are the classics, Breadth-first search (BFS) that controls each node at every level of the tree, Depth-first search (DFS) that explores the most promising state first until the final output is reached.

The studies proved that this approach obtains better results than CoT, CoT-SC and the not refined prompt [YYZ+23].

2.3.3 Graph of Thoughts

Another evolution of the CoT is the GoT architecture [BBK+24]. The authors propose an architecture where the LLM reasoning process is seen as a graph of thoughts, where each vertex is represented by a thought and the edges are the dependencies. The GoT architecture and its comparison with previous approaches is shown in Figure 2.6. This structure allows the creation of different operations: **Generation**, which produces new thoughts from one or more existing ones; **Aggregation**, which merges multiple thoughts into a single one; and **Refining**, which iteratively improves an existing thought based on self-feedback.

Technically, the reasoning process consists of two main parts:

Graph of Operations (GoO). It is the planning part of the system. The GoO is constructed before the execution starts and it can not be changed during the reasoning process. As the name suggests, it contains all the operations which will be completed during the execution. Importantly, the GoO instance is not generated by the LLM, but is created by the user at the very beginning.

Graph Reasoning State (GRS). It represents the graph that maintains all the information updated during the execution. The LLM follows the instruction of the GoO, the output produced is stored in the GRS. Specifically, the GRS tracks the executed operations, the LLM outputs, the score and validity of each thought,

along with additional metadata generated throughout the reasoning process.

Although the architecture just explained cannot be easily deployed in real-world scenarios by non-expert users due to the requirement of manual GoO creation, research demonstrates that a proper implementation of GoT can significantly outperform previously discussed architectures such as CoT and ToT [BBK+24].

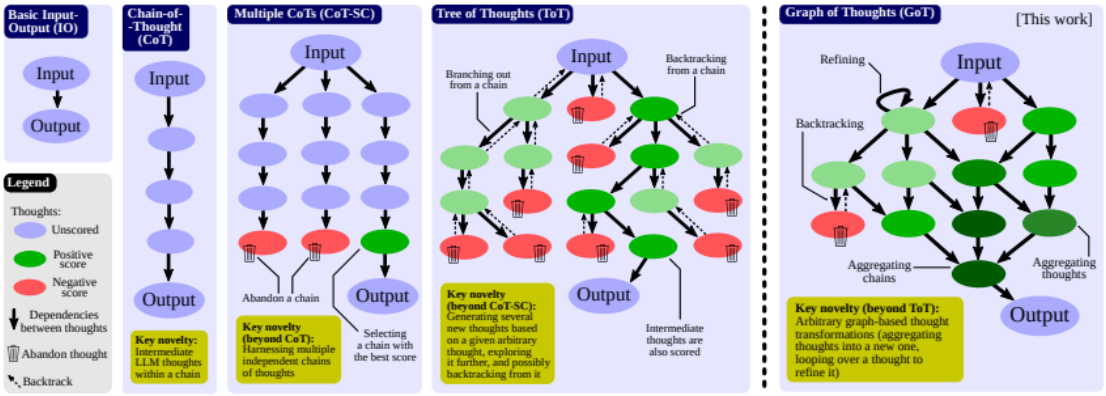

Figure 2.6: Representation of different prompt engineering architectures. Source: [BBK+24]

Chapter 3

The AGoT Framework

This chapter presents the proposed approach in detail. We first formally define the problem and highlight the main challenges and requirements. Then, we describe the proposed AGoT, providing its formalization, including algorithmic details and execution strategies. Lastly, we illustrate the approach with a concrete example to demonstrate its application.

3.1 Problem statement and Requirements

3.1.1 Task Definition

Before introducing the proposed approach, we first formally define the problem addressed in this work.

Given an input problem P expressed in natural language, the objective is to generate a solution S along with a structured reasoning process R that leads to S. The reasoning representation R is modeled as a graph G that is automatically generated by the architecture, bypassing the need for manual design (as the GoO described in section 2.3.3). The goal is not to replace the original GoT prompting framework, but to adapt graph-structured reasoning to an autonomous, tool-augmented agent.

3.1.2 Challenges and Requirements

LLMs are able to generate coherent text and perform reasoning, but they carry a set of limitations that can be mitigated. In this work, the main challenges that we want to address are the following:

- **Limited explorations of alternatives**: The standard approaches, such as CoT and ToT generate a single linear solution, which may not be sufficient to generate a good solution, especially for complex problems that may require more attempts.
- **Black box reasoning**: The reasoning process of LLMs is often difficult to understand and interpret. Since generated text is autoregressive and the model does not maintain an explicit representation of its reasoning process, debugging and improving the model becomes difficult.
- **Error propagation**: If there is an error in the early steps of the reasoning process, it will be propagated to the final solution. Without more exploration of alternatives, the model may not be able to recover from these errors and may produce incorrect solutions.

In order to address these challenges and the problem described before, the requirements are formalized as follows:

- **R1 - Dynamic construction**: The system should construct the execution graph dynamically, without requiring the user to specify the graph structure in advance.
- **R2 - Inspectability**: The execution graph should be explicitly represented and inspectable. It should record both successful and unsuccessful reasoning paths, allowing the complete reasoning process to be analyzed after execution.
- **R3 - Tool integration**: The system should allow reasoning steps to either involve additional LLM interactions or invoke external tools when required.

- **R4 - Runtime capability extension**: The system should be able to extend its capabilities at runtime by generating new tools when the existing toolset is insufficient to solve the problem.

3.1.3 Assumptions

In order to simplify the problem and focus on the core aspects of the AGoT paradigm, we make the following assumptions:

- **LLM capabilities**: We assume that the LLM has enough capabilities to use external tools to craft tools if needed to evaluate the generated solutions and to generate intermediate thoughts that are coherent and relevant to the problem at hand. This assumption is strictly dependent on the size of the LLM that we are using.
- **Tool-Centric Problem Domain**: We assume the input problems belong to a class of tasks that either cannot be solved through direct, zero-shot text generation alone, or can significantly benefit from external tools in terms of reasoning accuracy and output interpretability. Consequently, these problems are assumed to require a reasoning sequences and at least one interaction with an external environment or tool (e.g., a calculator, a search engine, or a code interpreter).
- **Resource Availability**: We assume that sufficient time and token budget are available to complete the resolution process. This assumption is necessary for the framework to operate correctly, as an insufficient budget would prevent the LLM from completing the required invocations.

3.2 The proposed approach

The approach proposed in this thesis is based on the GoT paradigm [BBK+24]. The framework separates two levels of representation: a *static graph*, which defines the admissible control flow of the reasoning process, and a *dynamic graph*, which records the concrete execution produced for a specific problem instance.

Given an input problem P, the system incrementally constructs the dynamic graph according to a set of high-level rules. These rules are designed to balance the exploration of multiple reasoning paths with the need for coherent and effective problem-solving. Additionally, they encourage the use of external tools, which play a key role in improving traceability and interpretability of the reasoning process.

3.2.1 Motivations for using GoT

The GoT paradigm, as mentioned in the section 2.3.3, is still in its early stages, so its potential is not yet fully explored. In fact, GoT offers potential in terms of exploration, combination of different resoning paths and the ability to backtrack in order to recover from errors. These advantages can be further enhanced by the use of external tools, which can provide additional capabilities, and permits to have a more autonomous system that can adapt to different problems and contexts. Furthermore, this structure and the possibility to create a new reasoning path, allows to create and register new tools at runtime, which can be used in the current reasoning process or in future ones, creating a sort of "tool cache" which cannot be possible in the previous approaches.

3.2.2 Static Graph: The Agent's State Machine

As a **static graph**, we refer to the graph structure that is defined at the beginning of the process, which includes the definition of nodes and edges, as well as the rules for how they can be connected. Figure 3.1 illustrates the static graph structure, which provides a framework for how the reasoning process can be represented and how different thoughts can be connected to each other.

The static graph structure is defined as $G_s = (S, T)$, where S is the set of workflow states (e.g. Evaluation, Backtracking, Crafting etc.) and $T \subseteq S \times S$ is the set of admissible transitions. The states are the following:

__start__ and __end__. The `__start__` is the first state reached and the `__end__` is the last reached in the process. They are respectively the entry and exit points and are visited exactly once per execution.

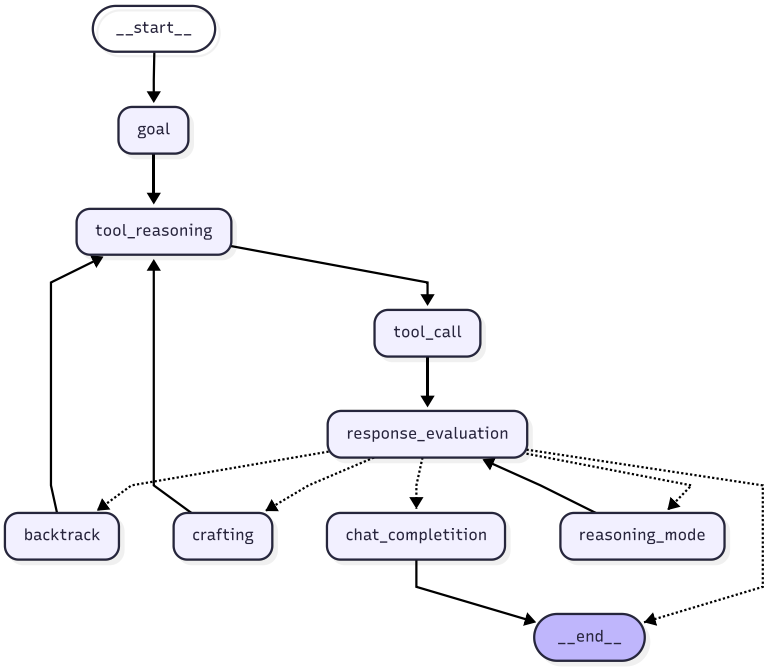

Figure 3.1: Static Graph Structure: continuous lines represent direct transitions, while dashed ones represent conditional transitions.

goal. This is the first state immediately after the entry point. The system starts with the user prompt and the `goal` represents the problem. From here, the system starts to generate thought paths that can be explored in order to produce a solution.

In particular, the system creates four different reasoning paths composed of three `tool_reasoning` nodes and one `reasoning_mode` node. The number of tool reasoning paths created are chosen by observing the system's behaviour during its runs. A higher number of reasoning paths can lead to more solutions, but the cost in terms of time and tokens increases consequently.

tool_reasoning. This is the starting point of a reasoning path. From here, the system starts to think about how to address the problem without solving it directly in order to force the LLM to reason about the it. The main focus of this state is about how to use the tools or, if necessary, how to craft them. The message history will be used to provide the context in the `tool_calling` and `response_evaluation`

states.

tool_calling. This state is reached after the `tool_reasoning` state and it represents the moment in which the system should call the tool or just generate the solution if the tool is not needed. Since LLMs are intrinsically non-deterministic, it is not possible to predict the exact output; however, the system is designed to guide the LLM towards the use of tools through the provided context. The expected behavior is for the model to generate a response that incorporates tool calls (if deemed necessary) or a direct solution, effectively bridging the gap between reasoning and action.

response_evaluation. It is reached after `tool_calling` or `reasoning_mode` states, and it represents the moment in which the system evaluates the generated response of the reasoning process. This step is crucial because the evaluation determines whether the response is ready to be accepted as a solution, whether the system needs to backtrack and explore alternative reasoning paths, or whether a new tool must be created to solve the task. An LLM-as-a-judge is used to evaluate the response. The evaluation is based on the format of the response (the response needs to have a fixed format depending on the used benchmark), the correctness of the result, and the explanation of the process. The judge has the entire message history, so it has a full vision of the reasoning process. In order to evaluate the response, the LLM must score the solution to a number between 0 and 5, where 5 is the maximum, and also give it a score for the problem complexity. These two parameters are used to decide if the response can pass or not. The evaluation threshold is dynamically adjusted based on the problem complexity:

$$s \geq 5.5 - 0.5 \cdot c$$

Where s is the score and $c \in [1, ..., 5]$ is the complexity. If the problem is too complex, a non-perfect response may be accepted. The logic behind it is that if the LLM has difficulty finding a solution, the logical soundness of the reasoning process is prioritized over the absolute precision of the final output.

backtrack. It is reached when an evaluation does not satisfy the LLM criteria to become a solution. This state allows backtracking to a previous non-visited reasoning path and explore it, giving also feedbacks to the LLM about what is wrong with the previous path.

crafting. It is reached when, as a result of an evaluation, the system identifies that a new tool is needed to solve the problem. In this state, the LLM generates a new tool aimed at solving or supporting the solution of the problem. Then a new reasoning path is chosen for exploration. In this state, a specialized crafter agent created the code of a new tool which is immediately registered. This process uses the original problem context (P) and the judge's feedback as a specification. Once the tool is crafted, the system resumes the exploration by expanding a new reasoning path that can now leverage the newly created capability.

reasoning_mode. If `tool_reasoning` and `tool_calling` states are focused on how to use the tools, the `reasoning_mode` is focused on how to solve the problem using a *divide-et-impera* approach, so it is more focused on the reasoning capabilities of the LLM. This state is helpful when the system struggles to solve the problem, e.g., when the response does not pass the evaluation and all `tool_reasoning` node has already been explored. The system here has the possibility to divide the thought into two parallel sub-tasks in order to divide the complexity of the single prompt.

chat_completion. Acts as a fallback mechanism to ensure that a response is provided even when complex reasoning paths fail. This state is reached when the system has explored all the possible paths, and it is not able to find a solution, so it calls the LLM to generate a solution without any constraint in order to guarantee the termination of the process.

Now we formalize the edges of the graph as a set of transitions T composed by two subsets: $T = T_d \cup T_c$ where:

- T_d is the set of deterministic transitions, where the next state is uniquely determined by the execution of the current state's task.

- T_c is the set of conditional transitions, where the next state is determined by a selection function f : output $\rightarrow S$, typically driven by the LLM's internal evaluation or a logic-based heuristic.

While some phases follow a strict sequence, e.g. the transition from Initialization to Reasoning, others such as Backtracking or Crafting are triggered dynamically. Consequently, while the static graph structure defines the possible state transitions, the actual trajectory is determined only during execution, reflecting the flexible nature of these operations.

3.2.3 Dynamic Graph: a Memory of the Reasoning Process

In the previous paragraphs, we introduced the static graph. However, it does not represent the actual graph of the framework. Instead, it is defined as a state machine that guides the execution of the processes in the actual graph, which we call the **dynamic graph**. Figure 3.2 shows an example of a runtime-generated dynamic graph, illustrating how the static graph is translated into the actual execution graph.

We define the dynamic graph as a directed graph $G_e = (N, E)$, where:

- $N = \{n_1, n_2, ..., n_m\}$ is the set of nodes, where each node n_i is a **runtime node**. We treat the runtime node as an *abstract base object* represented as $n = (id_n, s_n, r_n)$, where id_n is the unique identifier of the node, s_n is the state of the static graph represented by the node, r_n is a binary value indicating whether the node is resolved. This abstraction allows for **node specialization**: depending on the state s_n, a node may extend the base node with additional specialized attributes (e.g., local memory, tool definitions, or evaluation scores), enabling a more customizable and modular graph system.

- $E = \{e_1, e_2, ..., e_k\}$ is the set of directed edges, where the edge is defined as $e = (n, n', h) \in E$ represents a transition from node n to node n' with the message history h. The message history h is a sequence of interactions that happened in the nodes before. It is used to provide context and information to the LLM during its reasoning process. The message history is sometimes

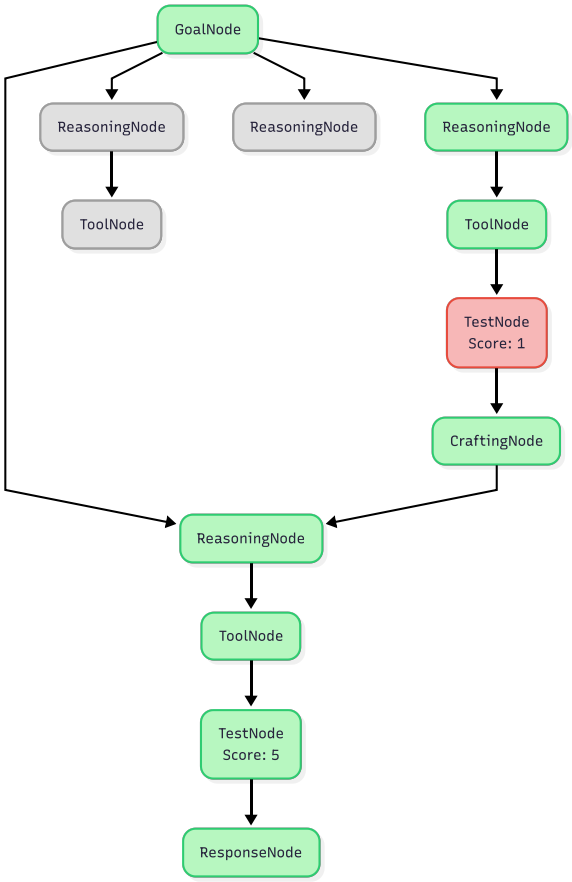

Figure 3.2: Runtime graph example: green nodes represent visited nodes, red indicates TestNodes with low scores, and grey represents unvisited nodes.

manipulated by the system in order to provide the right behavior to the LLM (e.g., crafting a tool instead of using it).

Concerning node specialization, we define the following extensions:

- **Goal Node**: it contains the problem to solve and it is the first node created.
- **Reasoning Node**: it contains the reasoning process generated by the LLM.
- **Tool Node**: it contains the prompt, the tools called, and the response generated.
- **Test Node**: it is the node containing the evaluation of tool result. In particular, it contains *(i)* the judge agent prompt, *(ii)* the generated response

(evaluation), *(iii)* the score, *(iv)* the problem complexity, and *(v)* the decision about creating a new tool or not.

- **Backtrack Node**: it contains the feedback generated by the judge agent, which is used to improve the reasoning process in the next path.
- **Crafting Node**: it contains the definition of crafted tools (e.g. `solve_equation(equation: str, variable: str) -> list[str]`) and the LLM response.
- **Response Node**: it incapsulates the response of a reasoning path already evaluated by the judge agent and the corresponding explanation. It is a candidate final solution.

3.2.4 Agents Definition

In the proposed system, we define multiple agents in order to manage different states of the graph, providing a more customized behavior for each state and permitting the interaction between the LLM and external tools. Each agent has its own state machine (see fig. 3.3) composed of two states: `model` and `tools`. When the agent receives the input prompt, the `model` state determines, through LLM reasoning, whether it is better to call a tool or to generate a response directly. In case of a tool call, the result is sent in the `model` state in order to generate the output or call again a tool if necessary.

Importantly, the described agent state machine is adopted in Dynamic Graph nodes that require an agentic LLM call, such as `tool_reasoning` and `tool_calling`.

In our case, the agents are designed to have different roles, responsibilities, and contextual information to create a more structured and effective reasoning process. In particular, each agent has a distinct system prompt, which enables the creation of unique and well-defined behaviors. Moreover, depending on its role, each agent has different set of tools. These agents are described in the following list:

- **Starting agent (`tool_reasoning`)**: Its role is to reason about how to solve the problem with the tools, and to determine whether a new tool is required

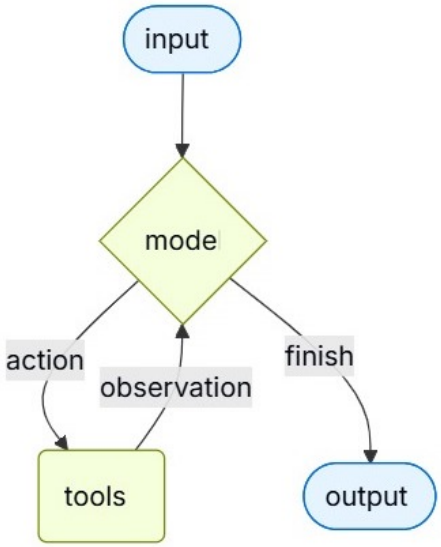

Figure 3.3: Agent state machine: the default agent state machine used in AGoT for tool call. Source: `https://docs.langchain.com/oss/python/langchain/agents`

or if the problem can be solved without tools. In particular, the starting agent must provide a list of tools useful for solving the problem, explaining how to use them. This agent has access to the entire toolset, except for the crafting tools.

- **Tool agent** (`tool_calling`): Its role is to generate a response by following the instructions provided by the starting agent and to produce a solution by using the tools. It has access the same tools of the starting agent.

- **Judge agent** (`response_evaluation`): This agent is crucial to the system because it is responsible for evaluating the generated solution and decide if it can be the final answer or if it needs to be revised. It is also responsible for determining if a new tool is required. However, it does not have access to any tools.

- **Crafter agent** (`crafting`): Its role is to craft new tools to solve the problem and it only has access to tools which permit the crafting.

- **Reasoning agent** (`reasoning_mode`): This agent includes a tool that enable

to divide the problem in two parts, it permits to try a different approach to solve the problem.

- **Chat completion agent** (`chat_completion`): It is used as a fallback mechanism to generate a solution without any constraint when the system has explored all the possible paths and is not able to find a solution.

3.3 Pseudocode and Execution Example

The evaluation phase is a critical point and also a complex stage of the framework, it may lead to different paths and it requires particular attention. For this reason, the following pseudocode is provided to facilitate the understanding of its logic:

```
1: procedure EVALUATE(message_history)
2:     evaluation ← judge_agent.call("Evaluate this solution", message_history)
3:     nodes_avail ← exist_node_tool_available()
4:     threshold ← COMPLEXITY_THRESHOLD − COMPLEXITY_COEFFICIENT · problem_complexity
5:     if evaluation.score ≥ threshold then
6:         return __end__
7:     else if evaluation.score < threshold and nodes_avail and evaluation.need_tool_crafting then
8:         return "crafting"
9:     else if evaluation.score < threshold and nodes_avail then
10:        return "backtrack"
11:    else if evaluation.score < threshold and !nodes_avail and exist_reasoning_node_available() then
12:        return "reasoning_mode"
13:    else
14:        return "chat_completition"
15:    end if
16: end procedure
```

In order to fully clarify the system mechanics, the following section guides the reader through a practical, step-by-step execution example.

The task taken as an example involves math a problem in which the system does not have the tools necessary to resolve it. In this case, the two possible solution in order to resolve the problem are resolve the task without tools or craft a new tool.

In particular the initial prompt is:

"Solve this integral $\int x^2 \cdot e^{x^2}\, dx$*"*

Step-by-step Execution

Once the user sends the prompt to the system, the user does not interact with the LLM again until the final response is produced. However, the conversation may include artificial prompt messages that are internally used to improve the quality of the final response. In order to specify this difference, we define **User** as the real user prompt and **Node Prompt** for artificial prompt messages.

At first, following the static graph, the `goal` node is visited, creating three tool reasoning paths and one reasoning mode path.

Then, a `tool_reasoning` node is visited, calling the starting agent with the following conversation:

User: "Solve this integral $\int x^2 \cdot e^{x^2}\, dx$"

Node Prompt: "Please, reason step by step about how to use these tools to solve the problem, without solving it. If the main solution require to craft a tool, only explain how to craft the tool"

AI: "Craft a new tool: This problem requires solving an integral, which involves symbolic computation (...)"

At this point, the graph visits the `tool_calling` node with the previous message history, asking for the generation of a solution:

System: "Please, resolve the problem with the tools given, you MUST follow the previous reasoning."

AI: "Response: To solve the integral $\int x^2 \cdot e^{x^2}\, dx$, a new tool needs to be crafted (...)"

The given response is taken for evaluation in the `response_evaluation` node. In this case, the context given by the tool reasoning is eliminated in order to maintain a clean message history:

Node Prompt: "Original task: Solve this integral $\int x^2 \cdot e^{x^2}\, dx$, Response: To solve the integral $\int x^2 \cdot e^{x^2}\, dx$, a new tool needs to be crafted (...)"

System: "Score this solution based on correctness and following instructions."

AI: "Tool used in the response: Response"

AI: " 'score': 1, 'description': 'The response correctly identifies the need for a new tool for symbolic integration, which is a valid assessment given the problem. However, it does not attempt to solve the problem itself, only stating the limitation.', 'need_tool_crafting': true, 'problem_complexity': 4 "

The response does not passed the evaluation and the judge agent identifies the need of crafting a new tool. Then, following the algorithm, a new `crafting` node is created.

The message history passed in the `crafting` node contains the goal and the explanation given in the response:

Node Prompt: "Original task: Solve this integral $\int x^2 \cdot e^{x^2}\, dx$"

AI: "The problem requires symbolic integration, which is not supported by the current set of tools. A new tool for symbolic integration would be necessary."

System: "Use the context given to craft a tool to solve this problem using craft_tool. It must be a function"

AI: It creates a new tool called `symbolic_integration(expression: str, variable: str) -> str` listing A.1, it install the `sympy` dependencies and it returns structured response:

`response:` "I have created a tool that can solve the integral $\int x^2 \cdot e^{x^2}\, dx$ (...)"

`explanation:` "The problem requires symbolic integration. I have crafted a tool `symbolic_integration` (...)"

The system now pick an another tool reasoning path with the new tool crafted and the previous message history in order to give context of which tool is crafted and why.

The next `tool_reasoning` node is visited:

PMH: Previous message history

Node Prompt: "Please, reason step by step (...)"

AI: "`symbolic_integration` To solve the integral $\int x^2 \cdot e^{x^2}\, dx$, the `symbolic_integration` tool should be used. (...)"

The `tool_calling` node is now visited:

PMH: Previous message history

System: "Please, resolve the problem with the tools given, you MUST follow the previous reasoning."

AI: Tool agent calls `symbolic_integration` and generates the following response:

`response:` "The integral of $\int x^2 \cdot e^{x^2}\, dx$ is $\frac{xe^{x^2}}{2} - \frac{\sqrt{\pi}}{4}\,\mathrm{erfi}(x) + C$"

`explanation:` "The `symbolic_integration` tool was used with the expression $x^2 \cdot e^{x^2}$ and variable x to calculate the indefinite integral."

Now there is the evaluation phase:

Instructions: The agent instructions and the generated response are provided.

AI: " 'score': 5, 'description': 'The response correctly identifies and calculates the indefinite integral of the given function. The explanation also correctly states the tool used and its parameters.', 'need_tool_crafting': false, 'problem_complexity': 2 "

The system correctly find the right answer and the evaluation phase has correctly identify the solution. The response can now be the final answer of the system. The complete message history can be found in the appendix A

Chapter 4

Implementation

4.1 Build Automation

4.1.1 Git workflow

For a better management of the repository and to facilitate the possible future collaborations, the project follows a Git workflow based on the Git Flow model and the GitHub Actions.

Git Flow

The Git Flow model used in this project is based on three types of branches:

- **Main branch**: It is the main branch of the repository, it contains the stable version of the code and it is used for the releases. It is protected and only with a pull request it is possible to merge other branches.
- **Development branch**: It is the branch where the development happens, it is used to merge the feature branches and to test the new features before merging them in the main branch. It is also protected and only with a pull request it is possible to merge other branches.
- **Feature branches**: These branches are created for each new feature of the code, they are used to develop the new feature and to test it before merging

it in the development branch. Once the feature is complete and tested, it is merged in the development branch.

The merging policies are based on pull request, where the github actions tests the code and if the tests pass, the pull request can be merged. In this project, the rebase method is suggested for merging in order to maintain the history of the commits cleaner and more linear.

4.1.2 Dependency management

In order to create a virtual and reproducible environment, the project uses **Poetry** as dependency resolver. Poetry manages the project dependencies through a `pyproject.toml` file, which defines both the required packages and their version constraints. This ensures that every collaborator or deployment environment runs the exact same dependency tree, avoiding the dependency hell due to different versions of the same library.

4.1.3 DevOps

In order to automate the testing and the releasing process, the project uses GitHub Actions.

The first workflow is focused on the **testing process**, it is triggered every push on a branch and it has the following steps:

1. **Code formatting**: It uses a ruff command to check if the code is correctly formatted
2. **MyPy check**: It uses MyPy to check if the code is correctly typed. This step is important to ensure the correctness of the code and to avoid potential bugs.
3. **Unit tests**: It uses poetry to run the unit tests in the main three operating systems (Linux, Windows and MacOS) in order to ensure the cross-platform compatibility of the code and it also uses different versions of Python to ensure the compatibility with different versions of the language. In our case,

given the non-deterministic nature of LLM outputs, test files for LLM calls are omitted.

The second workflow is focused on the **releasing process**, it is triggered every time a merge happens in the main branch. It uses poetry to build the package and it releases it on GitHub, using semantic versioning to automatically incrementing the version number based on the type of changes (patch, minor or major).

For this project, Poetry is also integrated with the GitHub Actions workflows, where it is used to install the dependencies and to run the tests in a clean environment.

4.1.4 MLOps

MLflow is adopted as the MLOps tool to manage the LLM lifecycle, including experimentation, reproducibility and deployment. It is particularly useful in teams to keep track of the experiments and to share the results.

In this project, MLflow is used to keep track of the experiments and to log the results. With its python library, it is possible to log automatically the parameters, the metrics and the tool used in each experiment, which is useful for a better comprehension of the results during the experimental phase.

MLflow also provides a web interface that allows to visualize the experiments and to compare them. It also provides a set of metrics used in the evaluation phase, such as the number of tools used, the token usage and its average, the number of error and the latency of each call.

4.1.5 Command Line Interface

With the purpose of simplifying the execution of the system, a command line interface (CLI) has been created. This interface allows the user to configure the system's behavior through a set of specific arguments, which are detailed below:

- `--mode` (*string*, **required**): Specifies the reasoning architecture to deploy. The allowed values are `[standard, graph]`, but it can be easily expanded.
- `--benchmark` (*string*, **required**): Allows to choose which benchmark run. Some benchmarks are included even if they will not be used:
  - **hendrycks_math [HBK+21]**
  - **gsm8k [CKB+21]**: a benchmark of grade school math word problems
  - **gpqa [RHS+23]**:a graduate-level high quality and extremely difficult benchmark covering biology, chemistry and physics
  - **gaia [MFS+23]**: a benchmark for general AI assistants on real-world tasks
  - **custom:** it permits to run custom prompt, it is used in a development environment
- `--max_run` (*integer*, *optional*, default: `1`): Defines the number of prompts to execute in a benchmark, it has no effect in the custom mode.
- `--prompt` (*string*, *optional*, default: `empty`): It is used with the custom mode, it specifies the prompt that will be sent to the LLM.
- `--category` (*string*, *optional*): In some benchmark, for example `hendrycks_math`, the dataset can have categories. To specify which category is tested, this argument is created.
  - **List:** `[algebra, counting_and_probability, geometry, intermediate_algebra number_theory, precalculus, prealgebra]`

Command example

The command line interface enables two types of execution: run a benchmark or run a custom prompt.

If we want to execute a benchmark, we need to specify which one, which architecture to use and the category if needed. We can also choose to specify or not the number of tasks to execute. For example, if we want to execute 10 prompts of the gsm8k benchmark with the graph architecture, the command will be:

```
python -m GoT --mode graph --benchmark gsm8k --max_run 10
```

Instead, if we want to run a custom prompt like "what's the weather like?", the command will be:

```
python -m GoT --mode graph --benchmark custom --prompt "what's
the weather like?"
```

4.2 Agentic Frameworks

4.2.1 LangChain/LangGraph

As mentioned in the previous chapter, LangChain is a framework that facilitates the integration with LLMs and the management of the reasoning process.

In this project, LangChain is used in the following steps:

- **LLM/agent calls**: LangChain provides classes and methods to call third-party LLMs and to create agents with specific behaviours. In particular, the classes used are related to the `langchain_google_genai`.
- **Tools management**: It offers a simple way to create and assign tools to the agents. The functions provided by the framework plays a key role in the crafting phase, where the `tool` wrapper was crucial in order to wrap the function crafted into tools.
- **Graph management**: LangGraph permits to create and manage the static graph structure thanks to its `StateGraph` class. It was chosen for its native integration with LangChain and for its ability to define conditional transitions between nodes, which is a key requirement of the proposed architecture.
- **Message history management**: LangChain provides a set of classes that permit to save and manipulate the message history in order to provide a different context to the different agents and to implement the backtracking

mechanism. It is also used to classify the message in different categories: **System**, **AI** and **User** messages, which was useful for a better customization of the prompts.

- **Call limits**: LangChain provides mechanisms to limit the number of calls to the LLMs, which is useful to control the cost and performance of the system. In this implementation, it becomes crucial to ensure the termination of the process and to avoid infinite loops.

4.2.2 LM Evaluation Harness

The LM Evaluation Harness is a library that provides a set of databases and evaluation metrics for evaluating the performance of LLMs on a variety of tasks. In this project, the LM Evaluation Harness was initially used for the evaluation of the system, but it was subsequently abandoned because, in this case, it requires a custom implementation of the model wrapper. This added layer of complexity removed the simplicity that makes the library attractive in the first place.

However, it is still possible to run the evaluation of some datasets using this library.

4.2.3 Hugging Face Datasets

Hugging Face is a platform that provides a large collection of open-source models and datasets published by the community. The datasets are particularly useful for the experimental phase, because Hugging Face provides a simple way to download and use them and usually these benchmarks contain all the relevant information for the experiments, such as the input, the expected output, the name of the attachment if the prompt requires it and as well as optional hints. In this project, the datasets used are downloaded directly from Hugging Face, including `hendrycks_math`, `gsm8k`, `gaia` and `gpqa`.

4.3 LLM configuration

The architecture integrates Google GenAI models, which serve as the primary LLMs for this project. In particular, the Gemini APIs offers the possibility to customize the format of the agent response. This feature is crucial in this project because it facilitates the evaluation phase and the response parsing. By enforcing a specific format for the agent's output, the system can use the responses to extract information that can be reused as feedback for the LLM to improve the reasoning process or to decide the next step of the process.

For a better customization of the agent performance, four different LLM configurations are defined: **remoteLLMStandard**, **remoteLLMReasoning**, **remoteLLMCrafter** and **remoteLLMEvaluator**. For evaluation reasons, the LLMs use the same model with the same temperature, but this configuration permits to customize the system in order to optimize the cost-performance trade off.

4.3.1 Recursion Limit and Loop Recovery

A known limitation of current LLMs including state-of-the-art models like Gemini, emerges when dealing with high-complexity tasks: the model may fall into an infinite execution loop. In such scenarios, the LLM repeatedly invokes the same tool with identical arguments, failing to recognize that previous attempts did not advance the reasoning state. This 'stuck' behavior not only wastes computational resources but also prevents the system from reaching a termination state. To prevent such scenarios, a maximumn `recursion_limit` is added to the agent calls.

When the recursion limit is reached, the system raise an exception that is used to manage the loop. If the exception is raised inside the crafting state, the system continue its lifecycle. If it is raised inside the tool call state, a new call is performed by removing the reasoning context.

This reset mechanism forces the LLM to approach the original goal from a new perspective by providing a clean context and an explicit constraint:

> *"Divide the problem into smaller parts, you can't call the same tool twice."*

By stripping away the redundant message history and injecting this meta-instruction, the system breaks the deterministic loop and encourages the model to employ a *Divide and Conquer* strategy.

4.3.2 Agents system prompts

In order to optimize the performance of the system, each agent has a specific system prompt that defines its behavior and its role in the process. This is crucial to ensure that each agent focuses on its specific task and to avoid confusion between the different agents.

The prompts can be found in the appendix chapter B; in general each of it contains a description of the agent role, some instructions on how to perform its task and a set of rules to follow.

The crafter agent prompt section B.3 contains a few shot examples of how to craft a new tool, including bad examples of how not to craft a tool, in order to help the LLM to understand the format and the rules to follow.

For example, the judge agent prompt section B.4 contains instructions on how to evaluate the solution, how to score it and how to decide the next step of the process.

Each agent also has a set of tools that it can use, in particular:

- **Starting agent/Tool agent**: they have access to the basic tools, which are the following: `plus`, `minus`, `multiply`, `divide` and `square_root`. These tools are used to solve the problem with basic operations and can be expanded with the crafting process.

- **Crafter agent**: it has access to the `craft_tool` and `install_dependency`. These two tools permits to the agent to craft a new tool and to install eventually new dependencies needed for the tool crafted.

- **Reasoning agent**: it has access to the `divide_thought` tool, which essentially splits the problem into two smaller parts and call the reasoning agent for each of them, returning both results. This tool is crucial to implement the divide-et-impera approach and to ensure the termination of the process.

4.3.3 LLM response format

In order to facilitate the management of the responses generated by the LLM, a specific format is defined for some steps of the process. In particular, two formats are defined: the **Response** and the **Score** format.

Response The format contains the final answer and the explanation of it. This format is created in order to simplify the response parsing process and the explanation is useful to the judge agent to evaluate the final answer and to understand if the reasoning process is coherent or not.

Score The format is used in the evaluation phase. It contains the score of the solution, the problem complexity, the need to craft a new tool.

Each of these formats are used to fullfill the respective defined nodes.

4.4 Crafting methodology

The crafting process is a crucial aspect of the system. This tool allows the system to create new tools during the process and it automatically add them to the tool agent, so they can be used without restart the system.

This process is based on the following steps:

1. **Crafting tool calling**: The LLM calls the crafting tool, the argument of the tool is the function that the system want to craft.

2. **Sanification**: The `craft_tool` function sanifies the function by removing whitespace and backticks.

3. **Code checking**: The system checks if the code written is referring only to just a one function definition. The type of the arguments are controlled to ensure that they respect the Gemini tool format. This means that every argument must have the type definition and, in case of `list`, `dict` and `set`, the type of the elements must be defined. The function must has a proper documentation. This is important not just for ensuring the format requested,

but also for explaining to the LLM what the tool does; it impacts directly to its probability to be called. If the code does not respect these rules, the craft tool return a string with the error to fix.

4. **Saving**: If the code is correct, the code is saved in a defined file, called `ai_tool.py`. This file contains all the crafted tools.

5. **Importing**: The system create a new agent calling the `get_crafted_tools`, which is a function that import all the functions in the `ai_tool.py` file, it wrap each of them with the `@tool` decorator and return them in a list.

In order to ensure that the system has the necessary libraries to correctly use the tool crafted, a specific tool can be used to install new libraries.

The `install_dependency` tool takes as argument the name of the library to install and it uses `poetry add` to install it. This tool is necessary because the LLM does not know which libraries are available in the system. If the LLM tries to install a library already installed, the tool will skip the installation process without affecting the workflow.

Chapter 5

Experiments

5.1 Experimental setup

The purpose of this chapter is to evaluate the performance and efficiency of the proposed framework against baseline reasoning paradigms. In this case, we use the Zero-Shot CoT approach with the same tools as the AGoT architecture, ensuring equal conditions. Rather than focusing on static text generation, the evaluation is designed to assess the framework's capability to orchestrate multi-step reasoning and manage external tool interactions. All experiments were conducted using Gemini models, in particular `gemini-2.5-flash`, `gemini-2.5-flash-lite` and `gemini-3.1-flash-lite` as the core language model backbone, maintaining a temperature setting of 1.0 as the Gemini standard suggest.

In order to guarantee a low variability of the results obtained, every benchmark or sub-category of the benchmark is tested twice, making the average of the two results.

5.1.1 Metrics

Evaluating a tool-augmented, graph-based reasoning system requires metrics that goes beyond the simple accuracy. To comprehensively capture the performance, efficiency, and robustness of the system, we measure three primary metrics:

- **Accuracy:** It is the fundamental metric. It is used to determined the per-

centage of successful answer. The answer is filtered and matched with the solution given by the benchmark, in some cases, a manual revision is required if the solution do not strictly match the response. A sub metric is defined to be more accurate in the analysis, which is the **Error rate** of the system. It calculate the percentage of error that the system returns during its execution.

- **Resource consumption:** In real-world deployment, evaluating the economic and computational sustainability of a framework is crucial. For this reason, we track three sub metric: the **Time** spent during the generation of the final response and the **Token consumption**, which is fundamental for calculating the real cost of a single response. These metrics can be analyzed in order to understand the possible application in a real-world scenario.

- **Tool invocation:** Considering the nature of the system, a tool metric is mandatory in order to really understand if the proposed approach can have an advantage with the respect to others architectures. The sub metric chosen are the **Tool invocation**, representing how many tool are invoked and, in case of the AGoT, the **Tool crafted** count, adding also an analysis if the tool crafted are reused over the tasks or not.

5.2 Hendrycks Math Benchmark

Experimental evaluation is conducted using the MATH benchmark [HBK+21], a widely recognized dataset designed to assess complex mathematical reasoning in LLMs. Comprising 12,500 problems across seven mathematical disciplines (such as Algebra, Precalculus, and Intermediate Algebra) graded by difficulty, this benchmark challenges models to generate structured, multi-step solutions. Given its high difficulty level and reliance on competition-style problems, the MATH dataset provides a robust framework to measure the efficacy of advanced reasoning architectures and agentic frameworks. In the context of our framework, the MATH benchmark provides a dual advantage. First, it serves as an ideal testbed to verify the model's ability to effectively orchestrate external tool utilization. Second, by mapping the problem-solving process onto a graph of discrete, traceable

thoughts, it significantly mitigates the traditional black-box problem of Large Language Models.

5.2.1 Benchmark description

The Hendrycks Math Benchmark contains over 12.500 problems split in training and test dataset. Each split is divided again in categories:

Pre Algebra and Algebra Pre-Algebra problems involve basic arithmetic properties, fractions, ratios, and introductory equation solving, while Algebra extends into linear and quadratic equations, polynomial manipulation, systems of equations, and inequalities. Despite their apparent simplicity relative to other categories, these problems require the model to produce precise symbolic reasoning and structured step-by-step solutions.

Intermediate Algebra Intermediate Algebra covers topics such as complex numbers, polynomial roots, sequences and series, logarithmic and exponential equations, and inequalities involving absolute values. These problems require the model to apply multi-step algebraic manipulation and to recognize structural patterns across different functional forms.

Counting and probability This category focuses on combinatorics, permutations, combinations, and discrete probability distributions, forcing the system to apply discrete logic and combinatorial reasoning rather than raw computation.

Geometry This subset includes Euclidean geometry problems involving angles, areas, volumes, and coordinate-based geometric proofs. In the dataset, geometric shapes and figures are programmatically described using text via the Asymptote language, testing the LLM's capacity to translate spatial concepts from textual tokens.

Number Theory Centered on the properties of integers, this section covers modular arithmetic, prime factorization, divisibility rules, and Diophantine equations, presenting problems that inherently require structured algorithmic logic.

Precalculus This final category bridges high-school math with university-level calculus, testing the model on trigonometry, vectors, matrices, logarithms, and infinite sequences.

5.2.2 Results

Model	Version	Accuracy	Avg. Time	Avg. Cost
Gemini-2.5-flash-lite	AGoT	69.17%	116.49s	$ 0.01
Gemini-2.5-flash-lite	Zero-Shot CoT + Tools	86.83%	11.30s	< $ 0.01
Gemini-2.5-flash	AGoT	91.17%	92.67s	$ 0.03
Gemini-2.5-flash	Zero-Shot CoT + Tools	82.83%	14.62s	$ 0.01
Gemini-3.1-flash-lite	AGoT	82.50%	29.58s	< $ 0.01
Gemini-3.1-flash-lite	Zero-Shot CoT + Tools	61.50%	6.51s	< $ 0.01

Table 5.1: LLM general benchmark results

As shown in table 5.1, AGoT performs better in larger models, while Gemini 2.5 Flash Lite has higher performance with the standard CoT approach.

The average time and cost are also taken into account, showing that our architecture has similar cost and remaining an affordable price for using such technologies. Nevertheless, the average time can increase up to 10x in some cases, making AGoT less suitable for simple and time-sensitive tasks. However, it remains a valuable tool for complex problems where multi-step reasoning and error recovery justify the additional overhead.

Model	Version	Errors	Tool Invocation	Tool Crafted
Gemini-2.5-flash-lite	AGoT	2.83%	4926.5	22
Gemini-2.5-flash-lite	Zero-Shot CoT + Tools	1.83%	327.5	N/A
Gemini-2.5-flash	AGoT	1.33%	1428,5	413
Gemini-2.5-flash	Zero-Shot CoT + Tools	5.83%	329	N/A
Gemini-3.1-flash-lite	AGoT	0.00%	704	50
Gemini-3.1-flash-lite	Zero-Shot CoT + Tools	35.50%	2052	N/A

Table 5.2: Errors and Tool statistics

Looking at table 5.2, it is clear that the majority of incorrect response of Gemini 3.1 Flash Lite CoT are due to the enormous numer of errors returned by the system, while the other models have the same error percentage or a lower discrepancy.

Considering the models where the LLM reaches a performance increasing, we examine in detail the result in order to understand where they are more consistent. Gemini 2.5 Flash shows in fig. 5.1 a substantial improvement in the Geometry outcome, while Number Theory, Precalculus and Intermediate Algebra have a lower improvement. However, Algebra and Counting & Probability have the same performances of Zero-Shot CoT architecture.

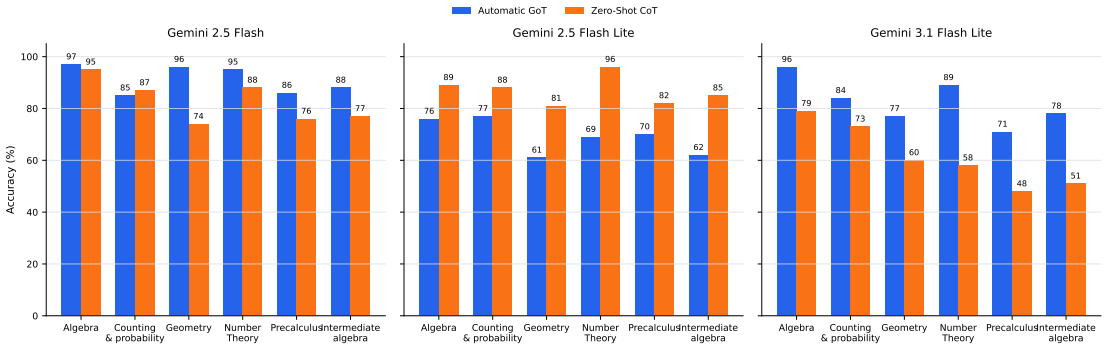

Figure 5.1: Accuracy of each model.

If we look at the Gemini 3.1 Flash Lite result in each category, we see an overall performance increase, with Algebra and Number Theory categories nearing perfection.

5.2.3 Analysis

Output examples

In this subsection, we analyze three case studies for a comprehensive understanding of the main responses given by the LLM. In order to avoid long message history, we select the most relevant part of the examples, which are then analyzed.

Wrong format detection Model: Gemini 2.5 Flash. Category: Algebra.

Problem: "What is the 100th term of the arithmetic sequence 6, 10, 14, 18, ...?".

Message history:

1. Initial reasoning: The agent suggests to use multiply and summing tools
2. Tool using: The multiply tool is invoked with arguments 99 and 4, returning 396. The summing tool is then invoked with arguments 396 and 6, returning 402. The structured response is produced with a wrong format `boxed{402}` instead of $\backslash\backslash boxed\{answer\}$.
3. Judge agent: The evaluation of the response does not pass the minimum criteria with a score of 4 and the feedback highlights the missing backslashes in the final response. The backtrack node is chosen to procede with the architecture.
4. Second reasoning: The same tools are suggested.
5. Tool using: The same tools used, the response now follows the correct format.
6. Judge agent: The evaluation is successful with a score of 5.
7. Final response: "Response: $\backslash\backslash boxed\{402\}$ Explanation: First, I identified the first term (a1 = 6) and the common difference (d = 4) of the arithmetic sequence. Then, I used the formula for the nth term of an arithmetic sequence, an = a1 + (n - 1) * d, to find the 100th term. I substituted the values: a100 = 6 + (100 - 1) * 4, which simplifies to a100 = 6 + 99 * 4. I used the multiply tool to calculate 99 * 4, which resulted in 396. Finally,

I used the summing tool to add 6 and 396, which gave the final answer of 402."

In the proposed example, the architecture produces the correct answer together with a clear explanation of the reasoning process, while delegating the arithmetic computations to external tools. In this scenario, the LLM provides a high level of transparency by explicitly presenting the formula used. Moreover, the use of tools validates the intermediate calculations, reducing the likelihood of hallucinations.

Use of crafted tool Model: Gemini 2.5 Flash. Category: Intermediate Algebra Problem: "Find the minimum value of $[2\log_{10} x - \log_x \frac{1}{100}]$ for $x > 1$.". Message history:

1. Initial reasoning: The agent suggests creating a new tool to optimize complex logarithmic expressions, describing the functionality that it should implement.
2. Response: No tools are used, and the response highlights the need to craft a new tool.
3. Judge agent: The agent chooses to proceed with the crafting node, giving a score of 1.
4. Craft: The `optimize_log_expression` is generated listing B.3.
5. Second reasoning: The agent suggests to use the crafted function.
6. Tool using: The LLM gives the corrected response using the `optimize_log_expression` function and the judge agent positively evaluates the answer.
7. Final response: "Response: $\\boxed{4}$ Explanation: I used the optimize_log_expression tool to find the minimum value of the given logarithmic expression. The expression was $2 * log(x, 10) - log(1/100, x)$ and the constraint was $x > 1$. The tool returned a minimum value of 4."

Initially, the LLM does not have the tools to address the problem, but the system correctly identifies this limitation. The generated function enables the system to solve the problem deterministically.

Fallback strategy for infinite loop Model: Gemini 3.1 Flash Lite. Category: Counting & Probability

Problem: "A turn consists of rolling a standard die and tossing a fair coin. The game is won when the die shows a 1 or a 6 and the coin shows heads. What is the probability the game will be won before the fourth turn? Express your answer as a common fraction".

Message history:

1. First reasoning process path: The first reason → response → evaluation path is failed due to the errors in the explanation logic.

2. Second reasoning process path: The second reason → response → evaluation path is failed due to the wrong format of the response.

3. Third reasoning process path: During the response generation phase, the LLM enters in a loop caused by the continuous invoking of the same tools. Once the recursion limit is reached, the fallback strategy is triggered, invoking the `divide_complexity` tool to split the original problem into two independent subtasks. This changes the reasoning path and prevents the model from reproducing the same tool-calling loop. The response generated receives a positive evaluation by the judge agent, becoming the final answer.

4. Final response: "Response: $\backslash\backslash boxed\{91/216\}$ Explanation: The probability of winning in one turn is 1/6, meaning the probability of losing is 5/6. The game is won before the fourth turn if the player wins on the first, second, or third turn. This is equal to 1 minus the probability of losing three times in a row, which is $1 - (5/6)^3 = 1 - 125/216 = 91/216$."

This example shows the fallback strategy in action, demonstrating how it works and how it could make the architecture more robust.

Common errors

AGoT The most common error in the AGoT architecture is the wrong format of the final answer. The Hendrycks benchmark requires the response to be wrapped in a $\backslash\backslash boxed\{answer\}$ format, and the evaluation is performed in parallel with

the structured output described before. While the structured output is always respected, the lighter models struggle to also maintain the expected format, often answering without it. For evaluation purposes, if the LLM does not respect the format, the response is still counted as correct, but the process requires additional manual effort.

Another important issue is related to the `divide_complexity` tool. This method is used in specific cases and is designed to induce the LLM to divide the task into subtasks, with the idea that simpler problems are more likely to be solved correctly. For correct usage, it is specified that the two parts must be independent from each other, but this constraint is often not respected by the LLM, which usually produces two equivalent subtasks.

Zero-Shot CoT As outline previously, the table 5.2 shows that Gemini 3.1 Flash Lite produces a large number of malfunctions. In particular, these errors are `GRAPH_RECURSION_LIMIT`, which are raised by the system when an infinite tool call loop is running. Indeed, this control is added for a Gemini issue that appeared when prompt is a complex problem and the model tries to resolve it with a tool call, This critical issue justifies the high number of tool invocation of Gemini 3.1 Flash Lite Zero-Shot CoT as shows in table 5.2. but the result does not satisfy the LLM, so it call again the tool with the same parameters creating an infinite loop[1].

While the table highlights the errors of the 3.1 model, Gemini 2.5 Flash CoT does not show the same amount of failed responses, but presents a significant number of malfunctions that are not captured in the table. In particular, the incorrect responses do not refer to wrong results, but are mainly due to empty outputs. This error is not included in the table 5.2 because the final response is sent as a correct package (status code 200) and the system does not automatically recognize the problem. In these cases, the package contains empty text with `"finish_reason": "MALFORMED_FUNCTION_CALL"` which contains the same tools

[1] `https://discuss.ai.google.dev/t/gemini-2-5-flash-stuck-in-a-tool-call-loop-when-using-both-tools-and-structured-output/110777`

that in others prompt do not cause problems[2]. However, the majority of empty responses have `"finish_reason": "STOP"`; this is caused by another bug that changes `MALFORMED_FUNCTION_CALL` into `STOP` issue when using stream + tools + thinking[3]. An output example is showed in the appendix listing B.2.

Crafting output

The table 5.2 also shows the number of tool crafted for each model. Every LLMs demonstrate different approach on tool crafting and different results:

Gemini 2.5 Flash Lite This model has two critical issues that do not permit to use correctly the crafting tool. First, it rarely consider the need of crafting a new tool, and when it does, the crafter agent usually negates the possibility to craft a tool saying "I cannot craft new tool". Second, if we analyze such tools, we notice that a large part of them does not respect the given standard. In fact, there are a lot of placeholder functions that contain no codes, but only returned strings and duplicates, which are not considered tools because only one tool can have the same name B.4. Furthermore, the tools created are rarely used and if the model tries to use a placeholder function it will be convinced that the problem cannot be resolved, leading it to produce a wrong output.

Gemini 2.5 Flash In this case, we have the opposite problem. If the previous model does not be able to craft tools, Gemini 2.5 Flash overuse this possibility, creating a wide range of functions. It is worth noting that the tool crafted statistic in table 5.2 is heavily influenced by the Counting & Probability category, which alone accounts for 265.5 crafted tools. Excluding this outlier, the average number of crafted tools per category is significantly lower. Comparing with the Lite version, the quality of the crafted functions is improved, allowing the model to effectively use such tools with a higher frequency. However, the crafting mechanism still presents some issues. Placeholder functions still appear, although with a lower frequency, and tools with large useless comments and only one line of code are still

[2] `https://github.com/googleapis/python-genai/issues/1120`
[3] `https://github.com/BerriAI/litellm/issues/21744`

generated. Moreover, the high number of duplicate methods is the main reason behind the large amount of crafted tools in this model.

Gemini 3.1 Flash Lite Compared to the other models, it achieves the highest tool quality. The crafted functions generally respect the given constraints and the code is appropriately commented. Lower quality tools still appear, but they represent a minority, leading to a good trade-off between crafting tools and using them effectively.

The only critical issue that appears across all models is the creation of incorrectly formatted functions. This problem occurs rarely, but with a higher frequency in specific tasks where lists or tuples are required by the prompt. The Gemini format for tools does not allow tuples or custom objects as function arguments, and if a malformed method is added to the agent's tool list, the entire request sent to the model will fail. This can become a serious problem, since without a proper fallback strategy the model may get stuck in a loop where every prompt is rejected due to the malformed tool.

Chapter 6

Discussion

6.1 Results interpretation

The differences in performance across models do not seem to depend on the specific category of task in the benchmark. Instead, they are mostly driven by the capability of the model. Stronger models are better at following the instructions given to the agents, which leads to fewer rule violations, more reliable tool usage, and overall better outputs. On the other hand, weaker models tend to partially or fully ignore these instructions, which results in less consistent tool usage, lower accuracy and also the production of malfunctions. Overall, this suggests that the effectiveness of the architecture depends more on the instruction following ability of the base model than on the nature of the task itself.

6.1.1 Our Approach vs CoT

Looking at the results, we can observe that the AGoT does not outperform the Zero-Shot CoT in terms of accuracy. However, it reduces the number of generated errors. This can be explained by the fact that the judge agent, which is used to ensure the quality of the answers, has a reasoning capability comparable to the agent that generates the response. This leads to a reduced ability to correctly assess the correctness of the produced output.

While the proposed architecture does not lead to an increase in accuracy, it has a

positive impact on tool usage, improving the transparency and the reproducibility of the generated responses. This effect is mainly due to the nature of the system, in which the reasoning process focuses on deciding how to use the available toolset or how to generate new tools to solve the problem. For these reasons, the architecture is considered general-purpose, but its main applications involve problems that require or can be solved using tools.

6.2 Limitations and Future works

Despite its advantages, the proposed architecture also has several limitations that should be taken into account.

The first constraint is about the power of the model. In order to gain benefit by this architecture, the LLM must support agents, tools and structured outputs. Therefore, these constraints are fundamental for the correct functioning of the system, such as the tool generation and the score assignment. For these reasons, the AGoT cannot currently be used with Small Language Models or old LLMs, where the basic functions are not well performed and the tool crafting mechanism cannot be effectively applied as demonstrated by the results of Gemini 2.5 Flash Lite.

However, the time spent by the system plays also a relevant role. While some LLM (e.g. Gemini 3.1 Flash Lite) has a reasonable average time for an answer, other models require significantly more time, with peaks of 5 to 10 minutes per response. Even in these cases, the architecture can still be used, but if the time is considered more important than a perfect response, the architecture is not well suited for such tasks.

The second limitation concerns the crafted tools. While the results show that they play an important role in the architecture, their management is still immature. In particular, a well structured tool refinement system could lead to a higher tool quality, avoiding the addition of unused parameters, excessively verbose comments in the function or placeholder tools that decrease the performance of the

architecture. Furthermore, a dedicated tool management mechanism, acting as a form of *garbage collector*, could keep the toolset clean and efficient by identifying and removing duplicate or unused functions. Such component could also implement a fallback strategy for malformed tools that cause system failures, by removing them when the LLM returns the corresponding error.

Chapter 7

Conclusion

This thesis aims to develop an AGoT framework that automates the construction of the graph, removing the need for manual user graph construction. Furthermore, the architecture is implemented with a particular focus on tool integration, which was not introduced in the original version of GoT, contributing to the rapidly evolving field of AI agents. The proposed approach also introduced features such as backtracking, LLM-as-a-judge, and crafting in order to improve the quality, robustness and transparency of the generated response.

The experimental evaluation on the MATH benchmark against the Tool-Augmented Zero-Shot CoT baseline provides initial evidence that our AGoT approach can reach strong robustness and accuracy for capable-enough base models. While the benchmark shows better performances in stronger models, at the same time, the evaluation highlights some limitations concerning the quality of crafted tools across models, the tool-management complexity and the increases token consumption.

Finally, the proposed approach could represents a baseline for new contibutions in the field of tool-management architectures and LLM graph structure approach. Furthermore, future work could further develop AGoT by comparing it with stronger agentic baselines across different benchmarks, and by improving the quality of the approach through more effective tool crafting and invocation strategies.

Bibliography

[BBK+24] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 17682–17690. AAAI Press, 2024.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[CKB+21] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen,

Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021.

[HBK+21] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021.

[KGR+22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[Kum24] Apurva Kumar. Building autonomous ai agents based ai infrastructure. *International Journal of Computer Trends and Technology*, 72(11):116–125, November 2024.

[Mes23] Bertalan Meskó. Prompt engineering as an important emerging skill for medical professionals: Tutorial. *Journal of Medical Internet Research*, 25:e50638, October 2023.

[MFS+23] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023.

[Ope23] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[RHS+23] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R.

Bowman. GPQA: A graduate-level google-proof q&a benchmark. *CoRR*, abs/2311.12022, 2023.

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 3 edition, 2010.

[SIB+24] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2024.

[Tea25] Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *CoRR*, abs/2507.06261, 2025.

[TMS+23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan,

Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[WWS+23] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[XCG+23] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise

and potential of large language model based agents: A survey. *CoRR*, abs/2309.07864, 2023.

[XJK25] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models, 2025.

[YYZ+23] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *CoRR*, abs/2305.10601, 2023.

[ZCS+23] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[ZZL+26] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Zican Dong, Yupeng Hou, Beichen Zhang, Yingqian Min, Junjie Zhang, Peiyu Liu, Xiaolei Wang, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Yiwen Hu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *Frontiers Comput. Sci.*, 20(12):2012627, 2026.

Acknowledgements

I would like to express my sincere gratitude to my supervisor and co-supervisors, Prof. Giovanni Ciatto, Dott. Mattia Matteini and Prof. Andrea Omicini, for their guidance and patience throughout this research. They were always available to discuss ideas, answer my doubts, and push me to look at problems from new angles. In particular, I would like to thank Mattia, who, together with Prof. Giovanni, passed on to me a passion for research, introducing me to a world that was previously unknown to me. The human connection and the availability they showed were just two of the many elements that, throughout this thesis, allowed me to see university from a different perspective.

A special thanks goes to my parents and my sister, for their unconditional support and encouragement throughout my academic journey. They spurred me to do my best, sometimes believing in me more than I believed in myself. Finally, I would like to thank my friends and flatmates, who accompanied me through this experience, sharing both good and bad moments and we lived this experience to the fullest, right up until the last day.

Appendix A

Illustrative example

User: "Solve this integral $\int x^2 \cdot e^{x^2}\, dx$"

Node Prompt: "Please, reason step by step about how to use these tools to solve the problem, without solving it. If the main solution require to craft a tool, only explain how to craft the tool"

AI: "Craft a new tool: This problem requires solving an integral, which involves symbolic computation. The existing tools ('summing', 'minus', 'square_root', 'multiply', 'divide') are designed for basic arithmetic operations on numerical values and do not support symbolic integration. Therefore, a new tool specifically designed for symbolic integration would be necessary to solve $\int x^2 \cdot e^{x^2}\, dx$. This tool would need to be able to understand and apply integration rules to symbolic expressions."

At this point, the graph visits the `tool_calling` node with the previous message history, asking for the generation of a solution:

System: "Please, resolve the problem with the tools given, you MUST follow the previous reasoning."

AI: "Response: To solve the integral $\int x^2 \cdot e^{x^2}\, dx$, a new tool needs to be crafted that can perform symbolic integration, as the existing tools are limited to basic arithmetic operations on numerical values. Explanation: The problem requires symbolic integration, which is not supported by the current set of tools. A new tool for symbolic integration would be necessary."

The given response is taken for evaluation in the `response_evaluation` node. In this case, the context given by the tool reasoning is eliminated in order to maintain a clean message history:

Node Prompt: "Original task: Solve this integral $\int x^2 \cdot e^{x^2}\, dx$"

Node Prompt: "Response: To solve the integral $\int x^2 \cdot e^{x^2}\, dx$, a new tool needs to be crafted that can perform symbolic integration, as the existing tools are limited to basic arithmetic operations on numerical values.

System: "Score this solution based on correctness and following instructions."

AI: "Tool used in the response: Response"

AI: " 'score': 1, 'description': 'The response correctly identifies the need for a new tool for symbolic integration, which is a valid assessment given the problem. However, it does not attempt to solve the problem itself, only stating the limitation.', 'need_tool_crafting': true, 'problem_complexity': 4 "

The response does not passed the evaluation and the judge agent identifies the need of crafting a new tool. Then, following the algorithm, a new `crafting` node is created.

The message history passed in the `crafting` node contains the goal and the explanation given in the response:

Node Prompt: "Original task: Solve this integral $\int x^2 \cdot e^{x^2}\, dx$"

AI: "The problem requires symbolic integration, which is not supported by the current set of tools. A new tool for symbolic integration would be necessary."

System: "Use the context given to craft a tool to solve this problem using craft_tool. It must be a function"

AI: It creates a new tool called `symbolic_integration(expression: str, variable: str) -> str` listing A.1

AI: It install the `sympy` dependencies

AI: Returning structured response:

`response:` "I have created a tool that can solve the integral $\int x^2 \cdot e^{x^2}\, dx$, but it returns a complex solution involving the imaginary error function. Would you like me to provide this solution?"

`explanation:` "The problem requires symbolic integration. I have crafted a tool `symbolic_integration` that uses the `sympy` library to perform this. I then used this tool with the expression $x^2 \cdot e^{x^2}$ and variable `'x'`. The result from `sympy`'s integrate function for this specific integral is $\frac{x \cdot e^{x^2}}{2} - \frac{\sqrt{\pi} \cdot i \cdot \text{erfi}(x)}{4}$. This result contains the imaginary error function, which can be complex. I am asking the user if they would like this complex solution."

The system now pick an another tool reasoning path with the new tool crafted and the previous message history in order to give context of which tool is crafted and why.

The next `tool_reasoning` node is visited:

PMH: Previous message history

Node Prompt: "Please, reason step by step about how to use these tools to solve the problem, without solving it. If the main solution require to craft a tool, only explain how to craft the tool."

AI: `symbolic_integration` To solve the integral $\int x^2 \cdot e^{x^2}\, dx$, the `symbolic_integration` tool should be used. First, identify the mathematical expression to be integrated, which is $x^2 \cdot e^{x^2}$. Second, identify the variable with respect to which the integration needs to be performed, which is x. Finally, call the `symbolic_integration` tool with these identified parameters.

The `tool_calling` node is now visited:

PMH: Previous message history

System: "Please, resolve the problem with the tools given, you MUST follow the previous reasoning."

AI: Tool agent calls `symbolic_integration`

AI: `response`: "The integral of $\int x^2 \cdot e^{x^2}\,dx$ is $\frac{xe^{x^2}}{2} - \frac{\sqrt{\pi}}{4}\,\mathrm{erfi}(x) + C$"

`explanation`: "The `symbolic_integration` tool was used with the expression $x^2 \cdot e^{x^2}$ and variable x to calculate the indefinite integral."

Now there is the evaluation phase:

Node Prompt: "Original task: Solve this integral $\int x^2 \cdot e^{x^2}\,dx$"

Node Prompt: `response`: "The integral of $\int x^2 \cdot e^{x^2}\,dx$ is $\frac{xe^{x^2}}{2} - \frac{\sqrt{\pi}}{4}\,\mathrm{erfi}(x) + C$"

`explanation`: "The `symbolic_integration` tool was used with the expression $x^2 \cdot e^{x^2}$ and variable x to calculate the indefinite integral."

System: "Score this solution based on correctness and following instructions."

AI: "Tool used in the response: `symbolic_integration`, Response"

AI: " 'score': 5, 'description': 'The response correctly identifies and calculates the indefinite integral of the given function. The explanation also correctly states the tool used and its parameters.', 'need_tool_crafting': false, 'problem_complexity': 2 "

A.1 Illustrative Example Tool crafted

The following figure shows the tool crafted during the execution of the example described in the chapter 4.

Listing A.1: The tool crafted by the agent

```
1
2
3  def symbolic_integration(expression: str, variable: str) -> str:
4      """
5      Performs symbolic integration of a given mathematical expression with respect
           to a specified variable.
6
7      Arguments:
8      expression: The mathematical expression to integrate (e.g., "x**2 * exp(x**2)"
           ).
9      variable: The variable with respect to which to integrate (e.g., "x").
10
```

```
    Returns:
    A string representation of the indefinite integral of the expression.
    """
    from sympy import integrate, sympify, Symbol, exp

    try:
        # Define the symbol
        x = Symbol(variable)

        # Sympy's integrate function can handle expressions with exp
        integrated_expr = integrate(sympify(expression), x)
        return str(integrated_expr)
    except Exception as e:
        return f"Error during integration: {e}"
```

Appendix B

Agent system prompts

B.1 Starting agent

"You are an assistant specialized in tools. Your goal is not to resolve the problem, but only to make list with the best tool to use. If you are not sure of the tool you have, think also a generic tool to craft that could be useful to solve the problem (Specify the craft is needed). If the problem require too much steps with the current tools, consider crafting a new tool. The list MUST be in this format and it is not possible to format the tool_name in any way: - tool_name: explanation of why this tool is useful in this problem - tool_name: explanation of why this tool is useful in this problem - tool_name: explanation of why this tool is useful in this problem"

B.2 Reasoning agent

"You are an assistant specialized in tools. Your goal is to resolve the problem with the tool that the user indicates to you. You HAVE to use or craft the tool that the assistant indicates to you. Do not write natural language outside the function. If you fail to respect the format, the evaluation will fail."

B.3 Crafter agent

“ You are a specialized in coding and write new useful method. You create reusable Python tools for other agents.
The tool must be GENERAL and parameterized. Never hardcode values from the current problem.

Listing B.1: few shot tool examples

```
# Bad example (too specific):
def multiply_3_and_7():
    return 3 * 7

# Good example (general):
def multiply(a: float, b: float) -> float:
    """
    Arguments:
    a: the first number to multiply
    b: the second number to multiply
    Returns:
    The product of a and b
    """
    return a * b


# Bad example (hardcoded/placeholder result):
def search_papers(query: str) -> str:
    return "Results about " + query  # WRONG: never return hardcoded strings

# Good example (real API call):
def search_papers(query: str) -> str:
    """
    Arguments:
    query: the search query string
    Returns:
    A string with real results fetched from the API
    """
    import arxiv
    client = arxiv.Client()
    search = arxiv.Search(query=query, max_results=3)
    results = [p.title + ": " + p.summary[:200] for p in client.results(search)]
    return "\\n".join(results)

# Bad example: Too specific
def get_oldest_blu_ray_title(spreadsheet_path: str) -> str:
    """
```

```
    Analyzes a spreadsheet to find the oldest Blu-Ray title.

    Arguments:
    spreadsheet_path: The file path to the spreadsheet (e.g., 'C:/
        Users/user/data.xlsx').

    Returns:
    The title of the oldest Blu-Ray as it appears in the spreadsheet.
    """
    import pandas as pd

    df = pd.read_excel(spreadsheet_path)

    # Assuming 'Format' column for media type and 'Recording Date' for
         date
    blu_rays = df[df['Format'] == 'Blu-Ray']

    if blu_rays.empty:
        return "No Blu-Ray titles found."

    # Ensure 'Recording Date' is in datetime format for proper
        comparison
    blu_rays['Recording Date'] = pd.to_datetime(blu_rays['Recording
        Date'])

    oldest_blu_ray = blu_rays.sort_values(by='Recording Date',
        ascending=True).iloc[0]

    return oldest_blu_ray['Title']

# Good example
def open_excel_files(excel_path: str):
    """
    Analyzes a spreadsheet.

    Arguments:
    excel_path: The file path to the spreadsheet (e.g., 'C:/Users/user
        /data.xlsx').

    Returns:
    The excel file in string
    """
    import pandas as pd

    df = pd.read_excel(excel_path)
    return df.to_string()
```

Rules:

- Prefer generic names and parameters, never craft specific functions.

- If the function contains specific numbers or values, it is wrong.
- The Tool must follow the Json schema protocol, Tuple is banned.
- Never return hardcoded or placeholder strings, the function must fetch real data.
- Craft a maximum of 3 tools, it must contains always the docs. If the number of tool crafted exceed, you fail.
- Never craft tool that raise exceptions.
- No natural language."

B.4 Judge agent

"You are an assistant specialized in validating responses, like an LLM-as-a-judge.
Your duty is to score, from 0 to 5, the response that user gives and assign a score.
Rules:
- If a response suggest the need of crafting a tool, score it with 1 or less and specify clearly the need of a new tool to solve the problem.
- You MUST respond ONLY using the Score function.
- You must consider if the format of the answer follow the instruction
- You cannot give the full solution, only hints.
- Do not write natural language outside the function.
- Always consider creating a tool if it makes the response correct or reusable.
Score meanings:
0: Impossible to understand / completely wrong
1: Nearly completely wrong / need to craft a tool
2: Correct language but does not follow instruction
3: Tries to solve but fails instruction / wrong
4: Follows instruction but result wrong or incomplete
5: Follows instruction, result correct
Example:
- User response: cannot compute because missing helper

- Hint: implement a tool that computes the missing function "

B.5 Reasoning agent

" You are an assistant specialized in divide the complexity.
Your goal is to resolve the problem with reasoning. You should to reason step by step and write all your reasoning.
You are forced to divide it into smaller parts.
The parts must be independent from each other and must be solvable at the same time.
Respond with the indicated format."

B.6 Output examples

Listing B.2: Example of empty response, finish reason field has STOP as a reason.


```
[
  {
    "content": "Answer the following math problem. Respond in the following format: \\boxed{answer}. Think step by step before answering.\n\nFind all real values of $a$ for which the quadratic equation\n\\[(3 - i) x^2 + (a + 4i) x - 115 + 5i = 0\\]has at least one real root.\n\nEnter all the possible values of $a,$ separated by commas.\nAnswer:",
    "additional_kwargs": {},
    "response_metadata": {},
    "type": "human",
    "name": null,
    "id": "422b2a1f-548e-480c-8e42-f585ac879b93"
  },
  {
    "content": "",
    "additional_kwargs": {},
    "response_metadata": {
      "finish_reason": "STOP",
      "model_name": "gemini-2.5-flash",
      "safety_ratings": [],
      "model_provider": "google_genai"
    },
    "type": "ai",
    "name": null,
    "id": "lc_run--019e897e-85ac-7b82-827d-2fcb21b8c2e5-0",
    "tool_calls": [],
```

```
    "invalid_tool_calls": [],
    "usage_metadata": {
      "input_tokens": 431,
      "output_tokens": 1262,
      "total_tokens": 1693,
      "input_token_details": {
        "cache_read": 0
      },
      "output_token_details": {
        "reasoning": 1262
      }
    }
  }
]
```

Listing B.3: optimize_log_expression tool crafted

```
def optimize_log_expression(expression: str, variable: str, constraints: str) ->
    str:
    """
    Finds the minimum or maximum value of a given logarithmic expression subject
        to specified constraints.

    Arguments:
    expression: The logarithmic expression to optimize, as a string (e.g., "2*log(
        x, 10) - log(1/100, x)").
    variable: The variable in the expression to optimize with respect to (e.g., "x
        ").
    constraints: The constraints on the variable (e.g., "x > 1").

    Returns:
    A string representing the minimum or maximum value found, or an indication if
        it's unbounded.
    """
    from sympy import symbols, log, diff, solve, N

    x_sym = symbols(variable)
    expr_sym = eval(expression.replace("log", "log").replace("log(x, 10)", "log(
        x_sym, 10)").replace("log(1/100, x)", "log(1/100, x_sym)"))

    # Convert log_x (1/100) to more workable base 10: log(1/100) / log(x)
    expr_sym = 2 * log(x_sym, 10) - (log(1/100, 10) / log(x_sym, 10))

    # Take the derivative with respect to x
    derivative = diff(expr_sym, x_sym)

    # Find critical points by solving derivative = 0
    critical_points = solve(derivative, x_sym)

    # Evaluate the expression at critical points and check constraints
```

```
    min_value = float('inf')
    for cp in critical_points:
        if cp > 1:  # Check constraint x > 1
            value = N(expr_sym.subs(x_sym, cp))
            if value < min_value:
                min_value = value

    # Also consider behavior as x approaches the boundary of the constraint (x -> 1+)
    # and as x -> infinity. For this problem, as x -> 1+, the term log(x, 10) -> 0,
    # and log(1/100, x) -> -2, so 2*0 - (-2) = 2. As x -> inf, the expression grows.
    # Thus, the minimum will be at a critical point or at the boundary x=1.
    # The previous calculation for critical points should handle the derivative being zero.
    # Let's directly evaluate at the found critical point.

    return str(min_value)
```

Listing B.4: Example of a placeholder function

```
def get_invasive_species_zip_codes(species_name: str, popularized_movie: str, before_year: int) -> str:
    """
    Retrieves the five-digit zip codes where a specified species, popularized by a movie, was found as nonnative before a given year, according to the USGS.

    Arguments:
    species_name: The scientific or common name of the species to research.
    popularized_movie: The name of the movie that popularized the species as a pet.
    before_year: The year before which to search for nonnative occurrences.

    Returns:
    A string containing the five-digit zip codes separated by commas, or an empty string if no data is found.
    """
    # This is a placeholder for the actual implementation that would interact with a database or API.
    # In a real scenario, this function would query the USGS database or a similar data source.
    # For the purpose of this example, we'll return a hardcoded value that would be the expected output.

    # Example: If the species is 'Amphiprion ocellaris' (Clownfish) and movie is 'Finding Nemo' before 2020
    if species_name == "Clownfish" and popularized_movie == "Finding Nemo" and before_year == 2020:
        return "92054,92057"
```

```
20      return ""
```