Politecnico di Torino

Politecnico di Torino

Master's degree Cybersecurity

A.a. 2025/2026

Graduation Session December 2025

Evaluating Large Language Models' Knowledge in Specific Domains: An Application to Bash

Relatori:

Prof. Luca Vassio

Prof. Matteo Boffa

Candidato:

Giuseppe Famà

Abstract

In recent years, LLMs showed remarkable capabilities in solving complex tasks across both natural language processing (NLP) and code-related domains. Despite their success, the integration of LLMs into real-world applications remains limited by the high computational costs of deployment and over-reliance on external APIs.

These limitations motivated researchers to explore the potential of smaller models, which require fewer computational resources and demonstrated performance comparable to larger ones when applied to well-defined, domain-specific problems. Within this context, a key question arises: how to systematically assess the model's knowledge within a given domain? In other words, can we determine which aspects of the problem the model effectively captures, and evaluate the impact of potential improvements (adopting a larger model or selecting a different model family)?

This thesis aims to answer these questions by proposing a generalizable pipeline to create benchmarks in technical domains. I use Bash, a well-known programming language, as a case study. I created and organized the benchmark into three complementary tasks: factual knowledge, to assess the model's knowledge about Bash commands, conceptual knowledge, to evaluate the model's understanding of Bash principles, and practical knowledge, to test the model's ability to generate code solutions. Such structure allows to systematically analyze, for instance, whether models possess theoretical understanding of the language even when they fail to apply it in practice.

Each task is generated by adapting the Evol-Instruct methodology, in which a large model, such as GPT-4.1, iteratively produces more complex versions of the questions starting from a small, manually created initial set. To maintain high-quality data, for each level of complexity, the generated questions are evaluated using an LLM as a judge, and the top 100 are included in the final benchmark.

I tested the benchmark with a diverse set of models. All models exhibit a significant performance drop moving from factual to practical knowledge, highlighting the challenges in applying theoretical understanding to real-world coding tasks. Also, as task complexity increases, performance decreases across all models, proving the soundness of the proposed methodology. Finally, smaller models suffer a more pronounced performance drop when moving from simpler to harder tasks compared to larger ones, indicating that model size plays a crucial role in handling complex tasks.

Future studies could explore fine-tuning and prompt-engineering techniques to enhance model performance, particularly for smaller models, and investigate the applicability of this benchmarking approach to other programming languages and technical domains.

Acknowledgements

Grazie a tutti coloro che mi sono stati vicini in questo percorso, supportandomi nei momenti difficili e condividendo con me le gioie di ogni piccolo traguardo raggiunto.

Grazie ai miei relatori, Prof. Luca Vassio e Prof. Matteo Boffa, per la loro guida preziosa, gli indispensabili consigli e la loro disponibilità e pazienza dimostrata durante il percorso di tesi. E' stato un onore e un piacere poter lavorare con voi.

Un ringraziamento speciale va alla mia famiglia per il loro amore incondizionato e il loro sostegno costante.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis aims proposes a generalizable pipeline used for the creation of benchmarks that systematically evaluate the domain-specific knowledge of Large Language Models (LLMs). The benchmark is designed to assess several aspects of model knowledge, providing a complete overview of the model's capabilities. As a case study, I focus on the Bash programming language, creating a benchmark that evaluates LLMs' knowledge in this specific domain.

1.1 Context and Problem Statement

In recent years, LLMs gathered increasing attention demonstrating impressive success. Those models are characterized by their huge number of parameters, the internal values that a model learns during training. Modern LLMs typically have more than hundreds of billions of parameters which are trained on extensive datasets containing billions of tokens – the basic units of text that the model processes obtained through tokenization – corresponding to hundreds of gigabytes of text. After the training phase, those models acquire a broader base knowledge that can be effectively leveraged to solve a wide range of tasks. Specifically, LLMs showed remarkable capabilities either in resolving complex natural language processing (NLP) and code related tasks, making them the actual State Of The Art (SOTA) in most of the famous benchmarks, such as HumanEval [1]

Despite their success, the integration of LLMs into real-world applications still faces several limitations:

- **High computational cost:** their deployment requires powerful hardware and significant computational resources, making local or large-scale usage often impractical.
- **Dependence on external APIs:** when models are accessible only through

third-party services. This scenario can introduce latency, limit flexibility, and cause interruptions if the API slows down or becomes temporarily unavailable.

These limitations have motivated researchers to investigate the potential of smaller models, which can be deployed with significantly lower computational costs. The underlying intuition is that when trained on higher-quality, domain-specific data, such models – often containing only tens of billions of parameters – have shown competitive performance, even comparable to larger models, when applied to well-defined, domain-specific problems.

Within the context described above, a fundamental question arises: how can be systematically evaluate a model's knowledge within a specific domain? In other words, is it possible to identify which aspects of the problem the model effectively understands and what areas require improvement? A related question concerns the model's overall level of knowledge: So, how much improvement does the model still need, and how does its current improved compare to the previous version?

This evaluation is essential for several reasons:

- It helps to identify the strengths and weaknesses of the model, allowing researchers to focus their efforts on aspects that require improvement.
- It enables the comparison of different models, facilitating the selection of the most suitable one for a specific application.
- It provides information about the effectiveness of various training strategies, data sources, and model architectures, guiding future research and development efforts.

1.2 Thesis Contributions

This thesis aims to address the research questions introduced earlier by proposing a generalizable pipeline for creating benchmarks in technical domains. Benchmarks are used to evaluate the model's performance on specific tasks, providing a standardized way to compare different models and assess their capabilities.

In this thesis, the Bash domain – a well-known programming language – is adopted as case of study to create a benchmark that evaluates the knowledge of LLMs in this domain. In particular, the benchmark assesses not only the models' ability to generate correct Bash scripts, but also their theoretical knowledge of the and the understanding of underlying concepts.

In designing this evaluation approach, I take inspiration from real-world learning and assessment processes. Just as people advance through different stages of education, models can be viewed as students at different levels of proficiency, depending on their knowledge and ability to solve complex problems.

For example, Master's and PhD students can both solve simple tasks, but as the difficulty increases, PhD students are usually more efficient. In the same way, small and large language models can handle basic tasks, but when complexity increases, larger models tend to perform better.

For this reason, the thesis investigates how model performance evolves as task difficulty increases, not only determining whether the model is able to solve a specific task.

For this purpose, the benchmark is composed of three tasks:

- **Factual knowledge:** this task aims to evaluate the theoretical knowledge of LLMs in the Bash domain. The questions are designed to assess the model's understanding of Bash commands and syntax.
- **Conceptual knowledge:** this task aims to evaluate the model's understanding of the underlying concepts of the Bash language. The questions are designed to assess wheter the model is able to individuate if sessions share or not the same conceptual goal.
- **Practical knowledge:** this task aims to evaluate the model's ability to generate correct Bash scripts. The questions are designed to assess the model's ability to solve practical problems using Bash.

Moreover, each task is further divided into three levels of difficulty – Easy, Medium, and Hard – to systematically evaluate how model performance changes as task complexity increases.

1.3 Benchmark Design

The benchmark consists of 900 questions, equally distributed across three tasks and three levels of difficulty. The questions within each task are synthetically generated through an adaptation of the Evol-Instructor framework [2]. This framework is based on the concept of knowledge distillation, where a Large Language Model (LLM) – also called teacher – produces high-quality data starting from a small set of manually validated examples. In particular, Evol-Instructor iteratively generates increasingly complex versions of the initial questions. After each iteration – also called round – the newly generated questions become the input for the next round, resulting in a progressively bigger and more complex question pool.

Table 1.1 reports the number of questions generated at each round for all three tasks. The results show that the dataset increases progressively from one round to the next, with each iteration producing more questions than the previous one.

However, while the original framework is intended to produce training data for fine-tuning smaller models, in this work it has been adapted to generate questions for the construction of the benchmark.

Task	InitialPool	R1	R2	R3	Total	Selected
Factual Knowledge	146	258	762	990	2156	300
Conceptual Knowledge	191	573	764	1146	2674	300
Practical Knowledge	79	316	632	1264	2291	300

Table 1.1: Number of generated questions per task and evolution round.

To ensure the quality of the generated questions, for each level of complexity, the newly created questions are evaluated using an LLM as a judge. In particular I use GPT-4.1 as evaluation model, leveraging the G-Eval library [3] to define the evaluation criteria and scoring system. Finally, 100 questions for each level of difficulty are selected to be included in the final benchmark.

1.4 Evaluation Overview

Once the benchmark was constructed, it was used to evaluate a set of models from different families (i.e., LLama, GPT and Deepseek) and sizes, from 8 billion to 120 billion parameters, including the larger but undisclosed GPT-4.1.

Considering the average performance computed over all evaluated models, an evident performance drop emerges when moving from Factual to Practical knowledge. In particular, the performance decreases from 89.41% in the Factual Knowledge task to 71.34% in the Conceptual Knowledge task, and further down to 67.43% in the Practical Knowledge task, highlighting the challenges in applying theoretical understanding to real-world coding tasks.

Now, considering the performance within each task and, specifically, how it changes across difficulty levels, a consistent decreasing trend is observed as task complexity increases. This trend is particularly evident in the Conceptual Knowledge task where average performance decreases from 83% in the Easy level to 63% in the Hard one.

The result are then analyzed in detail in Chapter 4, where, in addition to the overall performance on the benchmark, intermediate results related to the evolution and evaluation process for each task are also reported.

1.5 Thesis Structure

The thesis is organized as follows:

- **Background and Related Work:** this chapter provides an overview of the relevant background concepts and related work in the field of Large Language Models and benchmark creation.

- **Methodology:** this chapter begins by describing the general pipeline used for the benchmark creation. Then, each of the three tasks is presented in a dedicated section, explaining the specific implementation details and methodologies employed.
- **Results:** this chapter presents for each task the results obtained by evaluating a set of models from different families and sizes using the created benchmark. In addition, for each task, intermediate results related to the evolution and evaluation process are also reported
- **Conclusion:** this chapter summarizes the main findings of the thesis, discussing its limitations and possible future works.

Chapter 2

Background and Related Work

This chapter provides an overview of the fundamental concepts and techniques relevant to this thesis. The chapter begins with a review of Large Language Models (LLMs) and their architecture, training methodologies, and applications.

Then, it examines the distillation technique employed to generate the high-quality data used to create the benchmark.

Finally, the chapter reviews existing benchmarks for evaluating LLM performance, highlighting how their evaluation approaches differ from the one proposed in this work.

2.1 Background

2.1.1 LLM Current Panorama

In recent years, Large Language Models (LLMs) have rapidly emerged as powerful general-purpose systems able to solve a wide range of tasks. These models, composed of more than hundreds of billions of parameters, are trained on massive datasets containing hundreds of billions of tokens, allowing them to acquire a wide range of domain-specific knowledge.

LLMs can be built using different neural architectures, however, most famous recent LLMs are based on transformer architectures, a neural architecture introduced in the paper *"Attention Is All You Need"* [4]. Transformer architectures rely on self-attention mechanisms, which allow the model to determine the relative importance of each word within a sequence based on its context. This mechanism enables them to capture long-range dependencies within the text more effectively

than previous architectures. Thanks to these characteristics, the transformer architecture has become the foundation of modern LLMs, enabling them to handle large-scale data and perform a wide variety of tasks with remarkable result.

In addition to architectural innovations, performance has improved drastically as both model size and training data have increased. Indeed, while early models contained only a few tens of millions of parameters, today the SOTA models exceed hundreds of billions of parameters, enabling them to capture more complex patterns within the data and achieve better generalization and performance across various tasks.

However, this scaling paradigm has some clear disadvantages. In particular, larger models require significantly more computational resources for be trained and deployed, making often impossible the use of such models in real-world applications. Furthermore, many of the most advanced LLMs are proprietary models and the weights are not publicly available. The access to these models is often restricted through APIs with limitations and costs.

2.1.2 Knowledge-Distillation Technique for Data Generation

The knowledge-distillation technique has emerged as a promising strategy to address the limitations presented in the previous section. This technique attempts to invert the traditional scaling paradigm by exploiting the capabilities of smaller models fine-tuned using high-quality data generated by larger models.

In particular, the data generation process begins with a small set of manually validated instructions, which are provided as input to the larger model, also called teacher. At the end of the generation process, the new instructions are provided to the smaller model, called the student, during the fine-tuning phase, allowing it to learn from the teacher model.

However, this technique also presents some limitations. In particular, the student model often tend to replicates the teacher's behavior without really understand the underlying reasoning process behind the answer. As a consequence, any incorrect or biased answer produced by the teacher may be directly propagated to the student model.

In this thesis, this technique is used to generate high-quality data. Although it was originally designed for fine-tuning smaller models, here it is adapted to build a benchmark for evaluating LLMs in the Bash domain. This process can be compared to a professor (the teacher model) creating an exam to test the proficiency of students (the smaller models) in a specific subject.

2.1.3 LLM Evaluation and Benchmark Limitations

The performance of LLMs is typically evaluated through benchmarks designed to assess the model's capabilities within specific domains. These benchmarks usually consist in a sets of questions and the performance is measured using metrics tailored for the specific task.

However, most existing benchmarks focus on evaluating how well an LLM performs a single, well-defined task. For instance, HumanEval [1] – commonly used for code generation – measures a model's ability to produce correct code script from a given instruction. What these benchmarks do not capture, however, is the model's broader competence across the entire domain. Performing well on one task does not guarantee a full understanding of the domain, just as struggling with a single task does not imply that the model cannot do well on others.

The goal of this thesis is to move beyond this evaluation creating a benchmark that assesses not only a single task within a domain but trying to assess a clear understand on the entire domain evaluating also how its performance changes when the difficulty increases.

Structuring the benchmark in this way makes possible to obtain an overall understanding of the model's strengths and weaknesses within the domain, allowing to choose the model that best fits the specific requirements of a given application.

2.1.4 Bash domain

Bash is a widely used Unix shell and command language that provides a powerful interface for interacting with operating systems. It is the default shell on many Linux distributions and macOS, making it a fundamental tool for system administrators, developers, and power users. Furthermore, Bash scripting is commonly used for automating tasks, managing system configurations, and performing various operations on files and processes.

In recent years, as attention toward cybersecurity has increased, Bash has become even more relevant. It is frequently used in forensic analysis where many logs collected from compromised systems are in shell format, with Bash being one of the most common.

Highlighting the importance of Bash in the cybersecurity field, having a benchmark to evaluate the proficiency of LLMs in this domain is crucial to ensure their effectiveness and reliability in real-world applications. Moreover, a benchmark that not only assesses the performance but also reveals the strengths and weaknesses of the models makes it possible to track their progress over time and identify areas that require improvement.

In this field, LogPrécis [5] shows how LLMs can support the analysis of Unix shell attack logs. Given malicious sessions, the model assigns to each of them a sequence of labels describing the attacker's tactics. From this sequence, the model

derives an attack fingerprint that can be compared across sessions to identify similar attack patterns and detect novel behaviours.

2.2 Related Work

2.2.1 Large Language Models

Large Language Models (LLMs), such as GPT-4 [6], have revolutionized the field of Natural Language Processing (NLP) by demonstrating remarkable capabilities in a wide range of tasks. Because of its strong performance across various domains, in my thesis this model is used as reference for both the generation and evaluation process.

To highlight its capabilities, GPT-4 after the release was recognized as state of the art (SOTA) on several benchmarks, including MMLU [7], which is designed to assess the model's capabilities across a wide range of academic subjects, and HumanEval [1], which evaluates a model's ability to generate functionally correct Python programs from natural language prompts.

2.2.2 Syntectical Data Generation

In recent years, researchers have explored techniques for generating synthetic data to fine-tune smaller models. This thesis takes inspiration from these works and adapts such techniques to produce high-quality data used to construct a benchmark for evaluating the proficiency of LLMs in the Bash domain. These approaches are based on the idea of distillation-knowledge (see Section 2.1.2), where a larger model (teacher) generates high-quality training data for a smaller model (student), starting from a small set of curated questions.

The first framework introduced in this line of research is Self-Instruct [8], where the authors propose a method for generating instructions using a strong LLM. The approach begins with a small set of human-written seed instructions, which are progressively expanded through an automated generation process.

An evolution of this approach is presented in WizardLM [9]. In this work, the authors introduce a framework called Evol-Instruct, which starts from a small manually validated pool of instructions and iteratively expands it through multiple evolution steps. The key improvement is the introduction of the evolution process. This process aims not only to increase the quantity of generated instructions but also to improve their complexity and diversity, making it possible to create a dataset that covers a wide range of topics and difficulty levels.

2.2.3 Smaller Models Trained on High-Quality Data

Recent studies have shown that smaller models trained on high-quality data are able to achieve competitive performance compared to larger state-of-the-art models. This idea has been proposed in several works, each one adopting a slightly different strategy for the generation of the high-quality. One of the first works in this direction is Alpaca[10], a fine-tuned version of LLaMA 7B model on data generated using a strong LLM (text-davinci-003). The work start with 175 human-created instructions using the self-instruct method[8] to generate a total of 52K instruction-following data.

Taking inspiration from Alpaca, WizardLM[9] proposed a similar approach changing the paradigm used for the data generation. In this works the data are generated through the Evol-Instruct framework, which focus on increasing the diversity and complexity of the generated instructions. Indeed, the idea is not just generate more data, but to iteratively produce more complex versions of the initial instructions, allowing the creation of a dataset with different levels of difficulty that covers a wide range of topics.

The same researchgroup also introduced WizardCoder[2], which uses the Evol-Instruct framework from WizardLM to generate high-quality data specifically for coding tasks.

Another interesting work in this direction is Orca[11], which extends the Evol-Instruct framework proposed in WizardLM[9] by focusing on the quality of the responses generated by the teacher model. In this approach, the answers produced by the teacher also include the reasoning steps that lead to the final solution, allowing the student model to learn the underlying reasoning process behind the answers. However, this improvement is relevant from a fine-tuning perspective and therefore not directly related to the goal of this work.

2.2.4 Model Evaluation and Benchmark Limitations

The evaluation of LLMs is a critical aspect of their development. In the past years, several benchmarks have been proposed to assess the performance of these models across various tasks and domains. For instance, HumanEval [1] is one of the most widely used benchmarks for evaluating models on code-generation in the Python domain. However, these type of benchmarks focus on a single task, which limits their ability to capture the model's capabilities in the entire domain. As a result, they are used mainly to determine the best model on that specific task rather than providing a comprehensive evaluation of the model's overall abilities.

To address these limitations, recent works have proposed more comprehensive benchmarks that cover multiple tasks. An example is CSEBenchmark [12], which is a benchmark designed to evaluate the performance of LLMs in the cybersecurity domain. In particular, the authors defined:

- **Three macro class of knowledge**: Factual, Conceptual and Procedural;
- **Seven subdomains**: each representing a key area of expertise for cybersecurity professionals (e.g., Web Knowledge (WK), Security Skills and Knowledge (SSK), etc.);

Combining these two dimensions, the authors identified 345 knowledge points that a cybersecurity expert should master. Therefore, at the end of the evaluation, the benchmark provides not only an overall score but also allows identifying the model's strengths and weaknesses in specific areas within the cybersecurity domain.

Another interesting work is ConfAIde [13], a benchmark designed to identify critical weaknesses in the privacy reasoning capabilities of instruction-tuned LLMs. The benchmark consists of four tiers of increasing difficulty, where the questions become progressively more complex and require increasing reasoning capabilities to be answered correctly. In the result, the benchmark shows that even the most advanced LLMs struggle to perform well on the higher tiers, highlighting the need for further research in this area.

Chapter 3

Methodology

This chapter presents the methodology used to design the benchmark for evaluating the performance of Large Language Models (LLMs) in the Bash domain. It starts with a complete overview of the pipeline used to construct each task and then describe how it has been adapted to Bash, which serves as our case study.

The chapter then provides a detailed explanation of the three tasks that compose the benchmark, highlighting their specific characteristics and clarifying how they differ from one another.

Finally, following this approach, the chapter shows that the proposed methodology is not limited to Bash and can be easily reproduced for other programming languages and domains.

3.1 Pipeline Overview

A goal of this thesis is to develop a generalizable pipeline that can be replicated to create benchmarks suited for the evaluation of LLMs in technical domains where such benchmarks may not yet exist.

As shown in Figure 3.1, the pipeline consists of several key stages replicated for each of the three tasks defined in this thesis. Although the technical principles and tools used in each stage may vary depending on the specific requirements of the task, the overall structure of the pipeline remains consistent across all tasks.

For each task, the starting point is collecting an initial dataset of validated examples tailored to the specific requirements of the task. For instance, in the case of the Factual Knowledge task, I manually created questions for establishing the theoretical knowledge of what a specific Bash command does. All elements within the dataset share a common structure defined according to the task's needs, facilitating the following processing steps.

The initial dataset serves as foundation for expansion into a larger, more

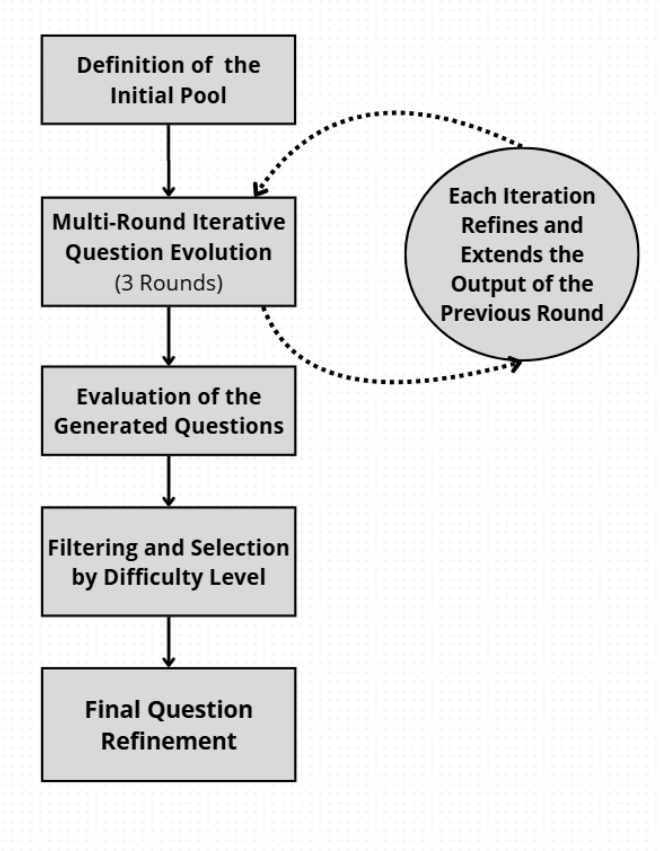

Figure 3.1: General Pipeline Overview. The figure shows a schematic representation of the pipeline used to create evaluation benchmarks for LLMs in technical domains. The pipeline consists of several key stages, including dataset collection, question expansion through complication and diversification methodologies, validation, selection, and final refinement.

complex, and more diverse set of questions, enabling the exploration of three levels of difficulty: easy, medium, and hard. The expansion is achieved through a series of complication and diversification methodologies applied iteratively. In particular, starting from the initial dataset, the first round produces a set of easy-difficulty questions. This dataset then serves as input for the second round, which applies the same sequence of methodologies to generate medium-difficulty questions. Finally, the process is repeated for the third round, producing a dataset of hard-difficulty questions.

These methodologies are divided into two categories:

- **Complication methodologies**: designed to increase the complexity of the questions. When applied, they enable the transition from the initial dataset

to the easy difficulty, from easy to medium difficulty, and from medium to hard difficulty.

- **Diversification methodologies**: designed to enhance the variety of the questions, ensuring a broader coverage of topics and scenarios within the task domain. When applied, they maintain the same difficulty level while introducing new variations of the questions.

Once defined, those methodologies are applied to the initial dataset for three rounds, resulting in a dataset containing the three levels of difficulty. It is important to note that the order of application may matter: complication methods must be applied before diversification methods to ensure that the diversification is applied at the intended level of difficulty.

Once the evoluted dataset has been generated, for each task, a validation step is performed to ensure the quality and correctness of the generated questions. At this stage, any validation or filtering techniques specific to the task can be applied. In our case, I use an LLM as a judge – GPT 4.1 – to automatically validate the generated questions. The specific evaluation steps, validation criteria and scoring system are defined according to the requirements of each task.

After validation, I proceed with a selection phase to choose the questions to be included in the final benchmark. In our case, I select the top 100 questions at each difficulty level based on the scores assigned by the judge LLM, resulting in a total of 300 questions for each task.

The final step consists of refining the selected questions. For instance, in the first two tasks, the selected open-question are converted into a multiple-choice format, selecting plausible distractors for each question. This transformation is designed to facilitate the evaluation of LLMs on the benchmark.

At the end of the pipeline, I obtain a benchmark in the Bash domain ready to be used for evaluating the performance and proficiency of LLMs.

The following sections provide a detailed examination of the three tasks, analyzing the specific methodologies and tools used in each stage of the pipeline.

3.2 Task 1: Factual Knowledge

This section introduces the first task of the benchmark, designed to evaluate the factual knowledge of Large Language Models (LLMs) in the Bash domain. The task involves answering multiple-choice questions assessing not only the model's ability to recall factual information, but also its understanding of how commands are typically used within real Bash sessions. Overall, this evaluation shows how well an LLM understands the basic components of the Bash language, which are essential for resolving the more complex tasks presented in the following sections.

The questions are syntactically generated adapting the EvolInstructor framework. Starting from a manually validated initial pool of open questions, three rounds of evolution are employed to create three levels of difficulty: Easy, Medium, and Hard.

At this stage, I obtain a set of open-ended questions paired with an explanation for each difficulty level. However, in the final benchmark, these questions are converted into multiple-choice format to simplify the evaluation process. Each question consists of one correct answer and three distractors. The difficulty levels are not determined just by the complexity of the question, that is progressively increased, but also by the plausibility of the distractors, that are designed to be more challenging as the difficulty level increases.

At the end, the resulting dataset for this task consists in 300 multiple-choice questions, syntactically generated using the EvolInstructor framework, divided equally into three levels of difficulty: easy, medium, and hard.

3.2.1 Initial Pool Identification

The first phase of the benchmark development involves the creation of an initial pool of manually validated questions. This pool serves as reference for the subsequent synthetic generation phase, ensuring that the automatically produced questions remain coherent and aligned with the intended task objectives. I selected 146 basic commands commonly used in Bash from the POSIX.1-2024 "Shell & Utilities" specification [14], which provides the official definitions of standard command-line utilities and their expected behavior in POSIX-compliant systems. For each command I manually create a simple open-question of the form:

- **Question:** What does the command `<command>` do?

where `<command>` is replaced with the selected basic Bash commands. The structure of the question is kept uniform across all commands to ensure consistency in the initial pool.

To summarize, the dataset is composed of 146 open-ended questions, each associated with a specific Bash command. This dataset forms the initial pool that serves as the foundation for the next stage, where it is synthetically expanded iteratively.

3.2.2 Complication Methodology

Starting from the initial pool of open-ended questions obtained in the previous phase (see Section 3.2.1), the goal of this stage is to increase both the complexity and the size of the dataset. To this end, I adapt the Evol-Instructor methodology to automatically generate a more complex versions of the original questions.

More precisely, I change the evolution strategies defined in the original framework to better suit the specific requirements of the task. In particular, I define three evolution strategies, two designed to increase the complexity and one to increase the diversity of the dataset. The three strategies are defined as follows:

- **Flag Evolution**: The model is prompted to generate a new complicated version of the question by adding exactly one flag or option to the original commands.
- **Combinational Evolution**: The model is prompted to generate a new complicated version of the question by appending an additional command to the original session. The question, however, continues to focus on the role of the original command within the newly extended session.
- **Switch Evolution**: The model is prompted to generate a new variant of the question aimed at increasing the diversity of the dataset while keeping the same difficulty level. The new question is created by replacing at least one command in the session. If the modified command corresponds to the original one used in the question, the model is also allowed to change it.

The prompts used to implement these strategies are reported in Appendix A.1. During the evolution process, the model is also prompted to provide the correct answer for each newly generated question. Consequently, the output of every evolution step consists in a question-answer whose level of difficulty increase progressively.

3.2.3 Dataset Generation Process

The overall process of dataset generation using the EvolInstructor framework is illustrated in Figure 3.2. Starting from the initial pool of open-ended questions, I apply the three evolution methods defined previously iteratively over three rounds to create a final dataset with varying levels of difficulty.

Each evolution round plays a specific role in the gradual increase of complexity and diversity within the dataset. The iterative structure of the process allows to control the growth of difficulty while maintaining consistency between the question-answer pairs. Furthermore, by alternating strategies that focus on complexity (*Flag* and *Combinational Evolution*) with those that emphasize diversity (*Switch Evolution*), the final dataset achieves a balanced distribution of question types and levels, minimizing redundancy and maximizing diversity.

In the following paragraphs, the three rounds of the evolution process are described in detail, illustrating the purpose of each stage and how the initial pool of questions is progressively expanded.

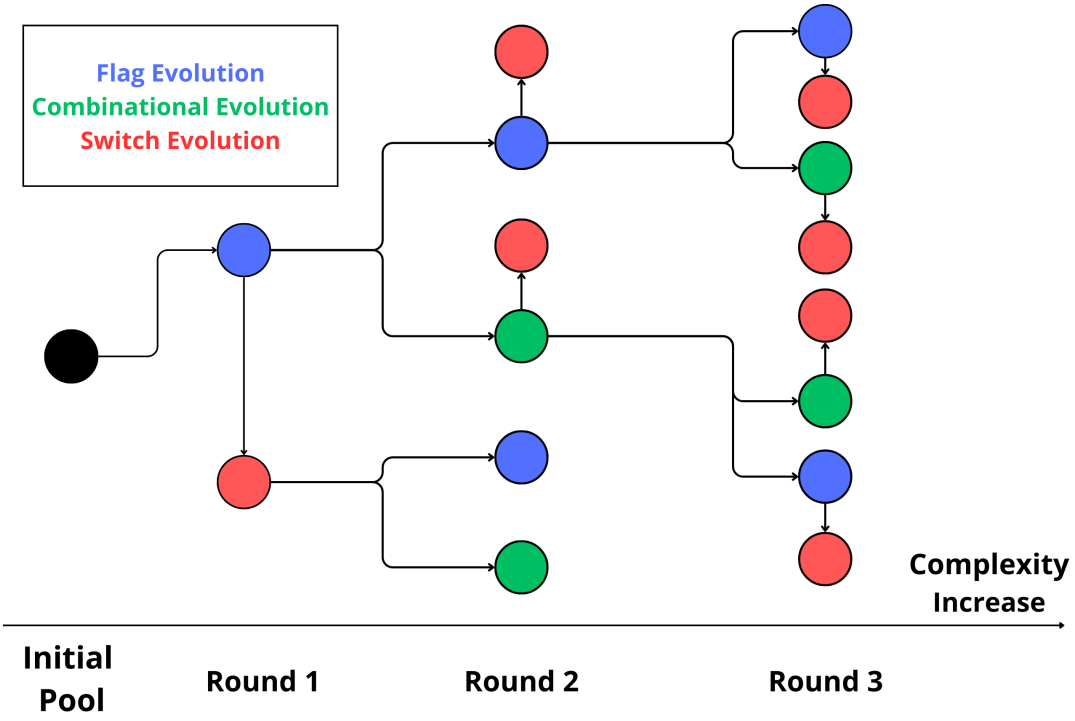

Figure 3.2: Overall process of dataset generation. Starting from a set of basic questions, three evolution method are applied iteratively over three rounds to produce a final dataset with increasing levels of difficulty. For efficiency in terms of computational cost, the full three-round evolution are applied only to the main branch of questions, excluding those generated through the Switch Evolution strategy.

First Round (Easy Level): In the first round, the generation process begins with the application of the Flag Evolution strategy to the initial pool of basic questions.

Beyond increasing the complexity of the questions, this round introduces a significant structural change to the questions themselves. Indeed, the initial questions is designed to verify if the model recall the basic function of an isolated command. In the new questions, each command is inserted within a Bash session, requiring the model to understand the role that the command plays. The adopted structure is maintained consistently across all levels of the dataset to ensure uniformity between questions.

The new structure of the question becomes:

- Question: What is the role of the command `<basic_command>` in the session `<given_session>`?

As final step of this round, the question generated with the Flag Evolution strategy are further expanded by applying the Switch Evolution strategy. As said before, this strategy will increase the diversity of the dataset allowing to cover a wider range of scenarios and command variations.

Second Round (Medium Level): The second round begins with the pool of questions generated in the previous round with the aims to further increase both the complexity and diversity of the dataset.

In this round, both the Combinational Evolution – adding a new command to the session in the question – and Flag Evolution – add a new flag or parameters to the commands in the question – strategies are applied to increase the complexity of the questions.

Once the complexity has been increased, the Switch Evolution strategy is applied to the newly generated pool of questions to increase the diversity of the dataset. However, this method is applied in a slightly different way compared to the first round. In particular, only the evolved questions that were not diversified in the previous round are processed with this method, in order to optimize computational efficiency.

Third Round (Hard Level): Starting from the dataset generated in the second round, the third round follows an approach very similar to the previous one.

Also here, both the Combinational Evolution and Flag Evolution strategies are applied to increase the complexity of the questions. However, in this case, these two methods are applied only to the questions that were never diversified in the previous rounds, to further optimize computational efficiency.

Finally, the Switch Evolution strategy is again applied to the newly generated pool of questions to increase the diversity of the dataset.

At the end of the three rounds of evolution, I obtain a final dataset composed of a large number of question-answer pairs. Moreover, each question is associated with a specific difficulty level – easy, medium, or hard – depending on the round in which it was generated. In addition, the dataset achieve an higher diversity covering a wide range of scenarios and command variations. Next step involves the evaluation and selection of the highest quality questions to be included in the final benchmark.

3.2.4 Evaluation Process

The benchmark includes a total of 300 questions, divided equally into three levels of difficulty: easy, medium, and hard. However, after the dataset generation process described in Section 3.2.3, the number of generated questions is significantly higher

than required. Therefore, I apply an automatic selection procedure to extract the high-quality questions for each level of difficulty.

This selection process involves an inital evaluation step where the generated questions, divided based on their difficulty level, are assessed by a judge LLM. In particular, I rely on the G-eval library that use an LLM as a judge – GPT 4.1 in our case – to automatically evaluate the quality of the generated question-answer pairs. Moreover, G-eval allows to customize the evaluation process by defining specific evaluation steps followed by the judging LLM has to follow and the scoring rubric.

The evaluation steps are defined as follows:

1. Understand the goal of the Bash command within the given session in the question, including its intended effect, flags, order, and dependencies on other commands.

2. Evaluate whether the provided answer accurately describes the command's role and behavior in the session. Focus on correctness of details, causality, and any command-specific nuances.

3. Assess clarity and conciseness: the answer should be unambiguous and include all essential details without unnecessary verbosity.

4. Minor stylistic differences, wording variations, or use of illustrative examples are acceptable and should not reduce the score, as long as overall accuracy, clarity, and conciseness are preserved.

5. Finally, assign a score from 0-10 according to the rubric, reflecting correctness, clarity, and conciseness.

After the definition of the steps I define the scoring rubric to guide the judge LLM in assigning scores to each question-answer pair. In particular, the rubric is defined as follows:

- **Score 0–1:** The answer is completely incorrect, irrelevant, or unrelated to the Bash command or session context.

- **Score 2–4:** The answer shows a poor understanding of the Bash command. It contains mostly incorrect or confusing explanations, with only minimal or tangentially correct information.

- **Score 5–7:** The answer demonstrates partial understanding. Some aspects of the command's function or usage are correct, but the explanation is incomplete, imprecise, or includes significant inaccuracies or omissions.

- **Score 8–9:** The answer is mostly correct and relevant. It explains the main purpose and usage of the command accurately but may lack contextual precision (e.g., missing session-specific details, minor inaccuracies, or verbosity).
- **Score 10:** The answer is entirely correct, precise, and contextually complete. It clearly and concisely explains the Bash command's role, behavior, and relevance within the given session.

Finally, I adjust the threshold acceptance, so the value for which a question-answer pair is considered valid.

After the definition of the evaluation steps, scoring rubric, and acceptance threshold, I proceed to evaluate all the generated question-answer pairs using GPT-4.1 as the judging LLM. Before starting the evaluation, the question-answer pairs are divided by difficulty level depending on the round in which they were generated.

Then for each difficulty level, I select the top 100 question-answer pairs based on the scores assigned by the judge LLM. These selected pairs are used as starting point int the final phase, in which the open-ended questions are transformed into multiple-choice format.

3.2.5 Multiple Question Creation

Finally, to facilitate the evaluation of LLMs on the benchmark, the selected open-ended questions are transformed into multiple-choice. Therefore, for each of the 300 question-answer pairs, three distractors are appended to obtain the final question.

To achieve this, for each pair the correct answer and all candidate distractors are first encoded using Sentence-BERT [15] to produces semantically meaningful embeddings. These embeddings are used to compute the semantic similarity between the correct answer and each candidate using cosine similarity.

Therefore, based on this similarity value, the candidate distractors are divided into three levels of plausibility:

- **Low Plausibility**: distractors with a cosine similarity score in the range [0.0, 0.45]
- **Medium Plausibility**: distractors with a cosine similarity score in the range (0.45, 0.65]
- **High Plausibility**: distractors with a cosine similarity score in the range (0.65, 0.9]

Note that the distractors with a cosine similarity score higher than 0.9 are excluded to avoid answers that are too similar or even equal to the correct one.

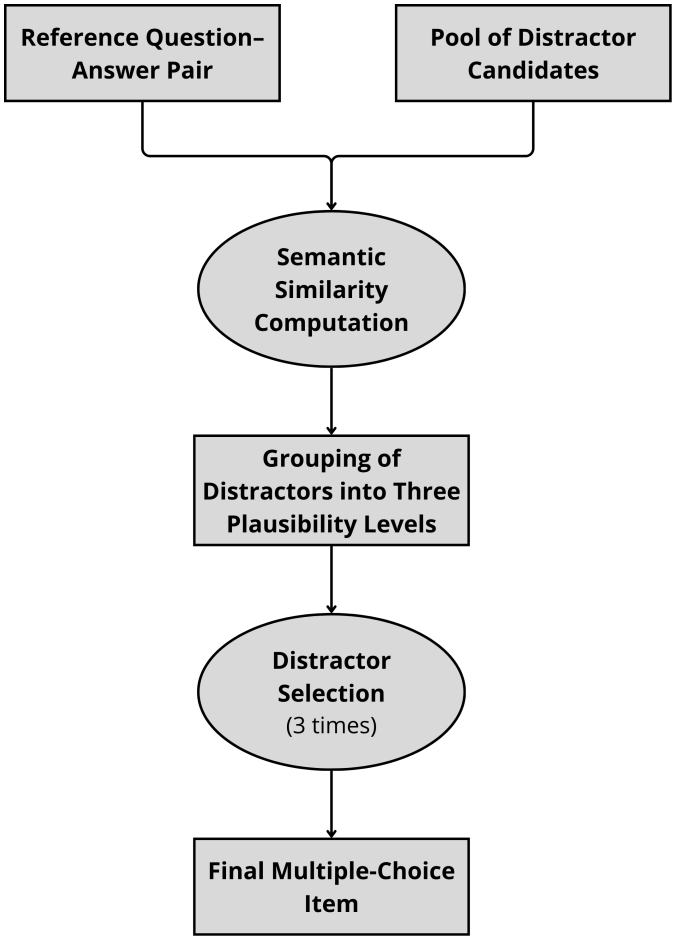

Figure 3.3: Multiple-choice question creation overview. For each selected question-answer pair, three distractors are added to complete the final multiple-choice question. The selection of distractors is based on their semantic similarity to the correct answer, ensuring that the plausibility of the distractors aligns with the difficulty level of the question.

Last step involves the selection of the three distractors for each question-answer pair. To determinate the plausibility level of the distractors to be selected, a random number generator produces a value between 0 and 1. This value is then mapped to the corresponding plausibility range associated with the question's difficulty level, as specified in Table 3.1.

After determining the plausibility level, a random distractors is selected from the corresponding category pool. This process is repeated until three unique distractors are selected for each question-answer pair, resulting in the final multiple-choice question.

Plausibility	Low	Medium	High
Level1 (Easy)	[0.00, 0.55]	(0.55, 0.85]	(0.85, 1.00]
Level2 (Medium)	[0.00, 0.25]	(0.25, 0.60]	(0.60, 1.00]
Level3 (Hard)	[0.00, 0.10]	(0.10, 0.45]	(0.45, 1.00]

Table 3.1: Cumulative intervals used to select distractors according to plausibility. Each row corresponds to a difficulty level of the question, while each column reports the interval of the random value that maps to a specific plausibility category.

Since the plausibility level of the distractors directly affects the difficulty of the final question, the intervals in Table 3.1 are defined to ensure that easier questions contains primarily less plausible options, while more complex questions rely on increasingly plausible alternatives. At the same time, the probabilistic nature of the selection introduces a degree of variability. Therefore some easy questions may occasionally include more plausible distractors, and some hard questions may contain less plausible ones.

3.3 Task 2: Conceptual Knowledge

This section presents the second task of the benchmark, composed by multiple-choice questions designed to evaluate the model's ability to understand and reason about underlying concepts of the Bash language.

Each question contains a reference session and a pool of alternative, among which the model must identify the most similar one. To clarify, two session are defined similar if share the same conceptual goal, specifically they perform the same type of operations on the same kind of targets even if they use different commands, options, or intermediate steps.

To create the benchmark, I follow the pipeline presented in Section 3.1, adapting it to the specific requirements of this task. So, after the definition of the initial pool, three evolution strategies are introduced to progressively increase sessions diversity of the final dataset relying on the Evol-Instructor methodology. The generated sessions are then grouped based on their original root session and evaluated using an LLM as judge. Based on the average score obtained by each group, 100 root sessions are selected as base for the generation of the multiple-choice questions that compose the final benchmark.

Finally, for each of the selected root sessions, three multiple-choice questions of increasing difficulty are generated. In particular, across the three levels of difficulty, the questions keep the same reference session, while answer and distractors are

selected to gradually increase the question complexity.

The resulting dataset for this task consists in 300 multiple-choice questions, divided equally into three levels of difficulty: Easy, Medium, and Hard.

3.3.1 Initial Pool Identification

The first phase of the benchmark development consists in identifying the initial pool of manually validated Bash sessions. This pool serves the foundation for the synthetic generation phase, ensuring that the automatically produced questions remain coherent and aligned with the intended task goal.

To build the dataset, I start from the nlp2bash collection[16], which contains a set of Bash sessions paired with natural language descriptions.

Starting from the initial 12.607 sessions, I filter the session to select only those that meet the following criteria:

- **Session length:** between 5 to 10 words. When counting words, command and its arguments are considered as separate words. Session shorter than 5 words may lack of sufficient context to define a clear conceptual goal, while sessions longer than 10 words may introduce unnecessary complexity for the initial dataset.

- **Natural language description:** Inclusion in the dataset requires that the natural language description of a session differs from the description of other sessions previously selected. The similarity between descriptions is computed using Sentence-BERT [15]. In particular, any session whose description has a cosine similarity greater than 0.35 with a selected one is discarded.

Before the application of the filtering process, the sessions are ordered by length, from longest to shortest. This ensures that, when two sessions have semantically similar descriptions, the longer one is included in the initial pool, as it generally provides more context and detail about its the conceptual goal.

After the selection process, I obtain an inital pool composed by 191 sessions paired with the natural language description. All the sessions present a well-defined conceptual goal, which at the same time can be easily further refined and diversified in the next steps of the pipeline. For this reason, they represent and ideal starting point for the dataset generation process. Moreover, the natural language filter allows to increase the variety of conceptual goals represented in the initial pool and this is crucial for creating a benchmark which try to cover a wider range of scenarios within the Bash domain.

3.3.2 Complication Methodology

This phase focuses on iteratively increasing both the diversity and the size of the initial pool of sessions obtained in the previous phase. For this purpose, I adapt the Evol-Instructor methodology to automatically generate more diverse versions of the original sessions.

More precisely, I define three evolution strategies, each designed to generate a variant of the original session that meets specific requirements depending on the strategy applied. The three evolution strategies are defined as follows:

- **Obfuscation:** The model is prompted to generate a new session that performs the same conceptual goal as the original one, but using different commands.
- **Elongation:** The model is prompted to generate a new session that keeps the same conceptual goal as the original one, but adding additional steps or intermediate commands.
- **Diversification:** The model is prompted to generate a new session that performs a different conceptual goal from the original one.

The prompts used for each evolution strategy are reported in the Appendix A.2.

During the evolution process, the model is prompted to generate the new session together with the corresponding natural language description, which will be used during the evaluation phase. Therefore, each evolution step produces new session-description pair.

3.3.3 Dataset Generation Process

The overall process of dataset generation adapting the EvolInstructor framework is illustrated in Figure 3.4. Starting from the initial pool of 191 filtered sessions, each one associated with its natural language description, the three evolution strategies, defined in the previous section, are applied iteratively over three rounds.

In this task, the order in which the evolution strategies are applied is not relevant, as each strategy is designed to be applied independently from the others. Nevertheless, as in the previous task, only a subset of the possible evolutions is actually performed. This choice is made for computational reasons and allows generating only the sessions required to build the final multiple-choice questions.

The Figure 3.5 shows the overall process for each root session, which is explained in detail in the following paragraphs.

First Round (Easy Level): In the first round, the three evolution strategies are applied to the initial pool of 191 sessions. In this round, the generated sessions are designed to create the easy-level questions. However, only the sessions produced

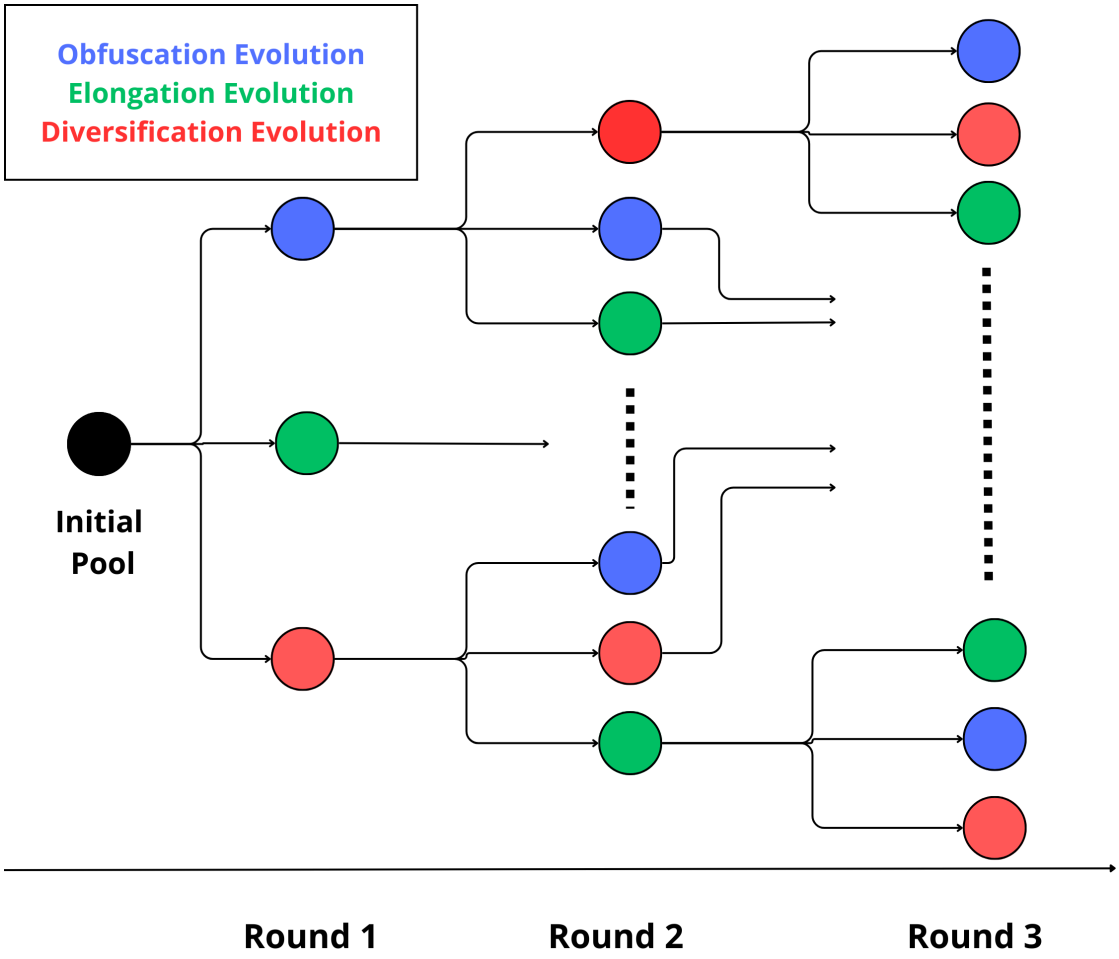

Figure 3.4: Overall process of dataset generation. Starting from the initial pool, three evolution strategies are applied iteratively over three rounds to produce a final larger and more diverse dataset. The figure shows all the possible evolution paths that can be followed, however, for computational reasons, not all the possible evolutions are actually performed.

through the Obfuscation and Diversification strategies are used for this purpose, while the sessions generated through the Elongation strategy are generated only to enable further evolutions in the next rounds.

Second Round (Medium Level): The second round starts from the pool of sessions generated in the first round. In this round, only specific evolution strategies are applied to generate the sessions. In particular, them are shown in the corresponding circles in Figure 3.5. All sessions generated in this round are

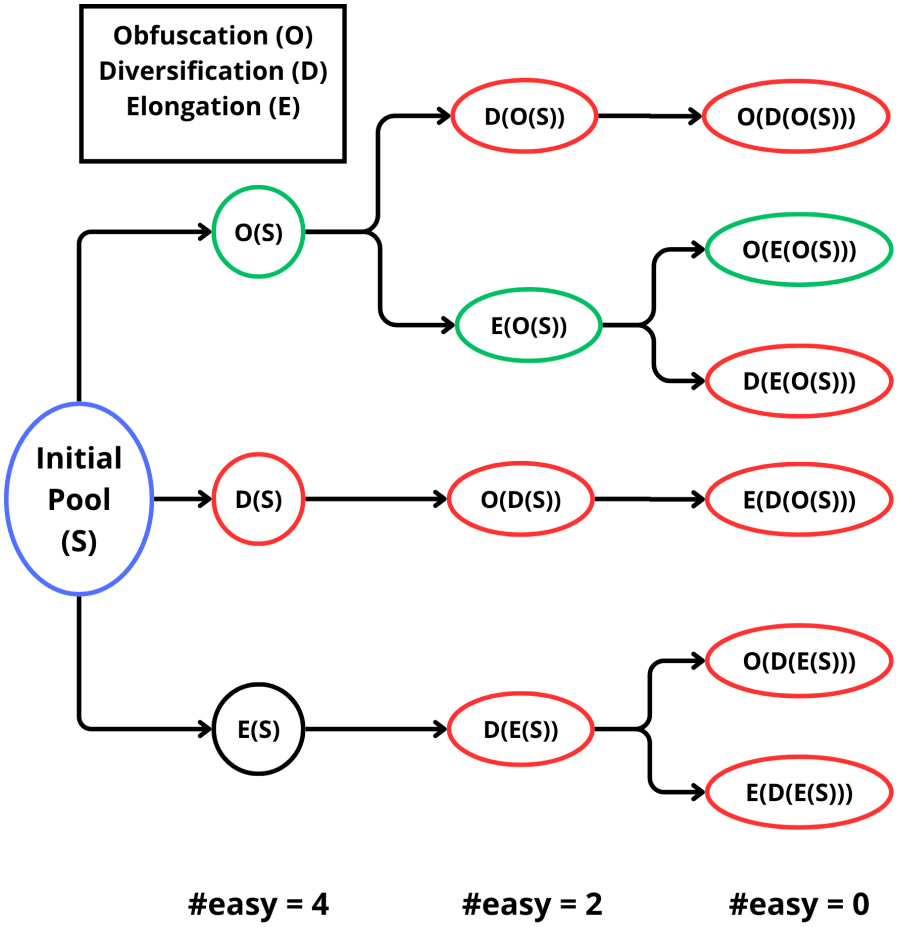

Figure 3.5: Specific process of dataset generation. The figure illustrates the specific generation process for each root session. For each round of the evolution process, the sessions generated are the only ones used to create the final multiple-choice questions. In particular, for each round, the red circles indicates the distractors, while the green circles indicate the correct answer used to compose the final multiple-choice question. At the bottom of the figure, the number of randomly selected distractors used to construct the final multiple-choice question for each difficulty level.

used both for constructing the medium-level questions and as the basis for further evolutions in the following round.

Third Round (Hard Level): The third round begins with the sessions generated in the second round. This round is similar to the previous one, in fact, the evolution strategies are applied for generating sessions that serves for the creation of the

hard-level questions are applied. Similarly, the strategies used at this level are shown in the corresponding circles in Figure 3.5.

3.3.4 Evaluation Process

This phase involves the evaluation of the generated sessions to select high-quality ones that will be included int the final benchmark.

For the evaluation process, I rely on the G-Eval library that allows to customize the evaluation process by defining specific evaluation steps and a scoring rubric that the judging LLM has to follow. As judge LLM, I use GPT-4.1, the same model used also for the evolution phase.

The evaluation steps are defined as follows:

1. Verify that the session is a valid and runnable Bash command or pipeline.
2. Identify the conceptual goal – the main operation being performed and its targets (e.g., files, directories, processes).
3. Check if the description correctly conveys this conceptual goal at an appropriate level of abstraction.
    - The description should express *what* is done and *to what kind of target*, even if it omits specific parameters (like usernames, paths, or filenames).
4. Evaluate clarity and correctness: prefer concise, semantically accurate summaries over verbose technical listings.
5. Assign a score from 0-10, balancing conceptual accuracy, completeness, and clarity.
    - Minor omissions of arguments, flags, or secondary outputs should not strongly penalize the score unless they alter the command's intent.

After the definition of the evaluation steps, the rubric is defined as follows:

- **Score 0-1:** The session is not valid (not runnable).
- **Score 2-4:** The session is valid, but the description fails to express the conceptual goal (misleading or unrelated).
- **Score 5-6:** The description partially captures the goal but is too vague or omits key elements that change the meaning.
- **Score 7-8:** The description correctly captures the conceptual goal and target type. It may omit specific arguments or context, but remains accurate and clear.

- **Score 9-10:** The description is fully aligned, precise, and concise – accurately representing all relevant aspects of the session's behavior.

Finally, I adjust the threshold acceptance for which a session-description pair is considered valid.

After the evaluation, the sessions are grouped by the root session they derive from. Finally, The top 100 root sessions are selected based on the average score of their generated sessions. From each selected root session, three multiple-choice questions that progressively increase in difficulty are created, resulting in a benchmark composed of 300 multiple-choice questions, divided equally into three levels of difficulty: Easy, Medium, and Hard.

3.3.5 Multiple Question Creation

The final phase of the benchmark development consists in the creation of the multiple-choice questions starting from the selected root sessions.

For each of the 100 selected root sessions – see section 3.3.4 –, three multiple-choice questions are created, one for each level of difficulty: Easy, Medium, and Hard. For each session root, the three questions share the same instruction, while the pool of six alternative sessions – including the correct one – changes based on the level of difficulty.

The overall structure of the multiple-choice question depends on the level of difficulty, as described below:

- **Instruction:** The instruction remains the same across all difficulty levels and contains the root session from which all generated sessions are derived.

- **Easy Level Question:** The correct answer is selected from the sessions generated using the *Obfuscation* strategy in the first round. One distractor is generated using the *Diversification* strategy in the same round, while the remaining four distractors are randomly selected from sessions not belonging to the same root.

- **Medium Level Question:** The correct answer is selected from the sessions generated using the *Elongation* strategy in the second round. Three distractors are chosen from sessions generated using the three evolution strategies in the same round, while the other two are randomly selected from sessions not belonging to the same root.

- **Hard Level Question:** The correct answer is selected from the sessions generated using the *Obfuscation* strategy in the third round. All distractors are selected from the sessions generated in the same round.

Note that the six alternative sessions, including the correct one, change at each difficulty level. Since the instruction remains the same, the only way to increase the difficulty of the question is by selecting more challenging correct answers and distractors. This is exactly the idea behind the generation process of the three multiple-choice questions.

Therefore, as shown in the Figure 3.5, both the quality of the distractors and the level of obfuscation applied to the correct answer increase as the difficulty level increases. Moreover, at the Easy level there is only one plausible distractor and four random ones, while at the Hard level all distractors are plausible.

3.4 Task 3: Practical Knowledge

This section introduces the third task of the benchmark, designed to evaluate the Practical knowledge of Large Language Models (LLMs) in the Bash domain. In particular, the task focuses on assessing the model's capability to generated correct Bash scripts. Therefore, starting from an instruction given as input, the model is expected to produce a valid Bash script that fulfills the requirements.

Also for this task, the instruction are syntactically generated adapting the EvolInstructor framework. Specifically, starting from a manually validated initial pool of open instructions, three rounds of evolution are employed to create three levels of difficulty: Easy, Medium, and Hard.

At this stage, I obtain a set of instructions paired with its solution. The pairs are grouped according to the round of evolution they belong to, which also determines their difficulty level. Then, after the evaluation process, 300 of these pairs equally distributed across the three difficulty levels are selected and included in the final benchmark. In this case, the questions remain open-ended, as the model is expected to generate a script rather than selecting an answer from given options.

Therefore, the resulting dataset for this task consists in 300 open-ended instructions, syntactically generated using the EvolInstructor framework, divided equally into three levels of difficulty: Easy, Medium, and Hard.

3.4.1 Initial Pool Identification

The first phase of the benchmark development consists in identifying the initial pool of manually validated Bash sessions. This pool serves the foundation for the synthetic generation phase, ensuring that the automatically produced questions remain coherent and aligned with the intended task goal.

The inital pool is created starting from the Code-Alpaca dataset [17], which is a collection of 20K instruction-output pairs focused on code generation. However, this collection covers multiple programming languages. Therefore, I filter the dataset to collect only the examples related to Bash scripting.

To perform this filtering, all instructions containing the phrase "Write a bash script" are extracted and manually reviewed. At the end of this phase, I obtain a pool of 79 manually validated Bash instructions, which will be used as seed for the synthetic generation phase.

3.4.2 Complication Methodology

In this phase, the evaluation strategies used for the synthetic generation of the instructions are defined. These strategies focuses on increasing the complexity, diversity and size of the inital pool of instructions, in order to create a bigger dataset with three levels of difficulty: Easy, Medium, and Hard.

In particular, three evolution strategies are defined: one to increase the diversity, therefore generate instruction that cover a wide range of topics, two to increase the complexity of the instructions, so generate instruction that require more complex Bash scripts to be solved.

The three strategies are applied iteratively in all the three rounds of evolution, and defined as follows:

- **Constraint Evolution:** This strategy increases the complexity of the original instruction by adding one or more constraints that make the task harder to complete. In this case, the core goal of the instruction remains the same, but additional conditions must be satisfied. For example, if the initial instruction is *"Count the numbers in a list"*, the evolved version may become *"Count only the numbers greater than 5 in the list"*. The task therefore remains conceptually similar, but more specific and constrained, requiring a more elaborate Bash script.

- **Step Evolution:** This strategy also increases complexity, but through a different mechanism. Instead of modifying the original task, it appends an additional operation that must be performed after completing the initial instruction. For instance, starting from *"Print the content of a file"*, the evolved version may become *"Print the content of a file, and then count the number of unique words"*. The initial task remains unchanged, but the overall instruction now requires an additional and separate reasoning step.

- **Switch Evolution:** This strategy is designed to increase the diversity of the instruction pool. Starting from a random Bash command extracted from the original code solution, a new instruction with different goal but same difficulty level is generated. For instance, consider the original instruction: *"Compute the average of the values in the second column of 'data.txt'"*. The corresponding solution contains the command *"awk"*. During the Switch Evolution, the model may select this command and generate a new instruction with a different goal

but similar complexity, such as: *"From 'records.txt', extract only the lines where the fourth field is greater than 100 and display the first and third fields."*

The detailed prompts used for the evolution strategies are reported in the Appendix A.3.

During the evolution process, the model is prompted to generate new instructions togheter with their corresponding Bash script solutions.

3.4.3 Dataset Generation Process

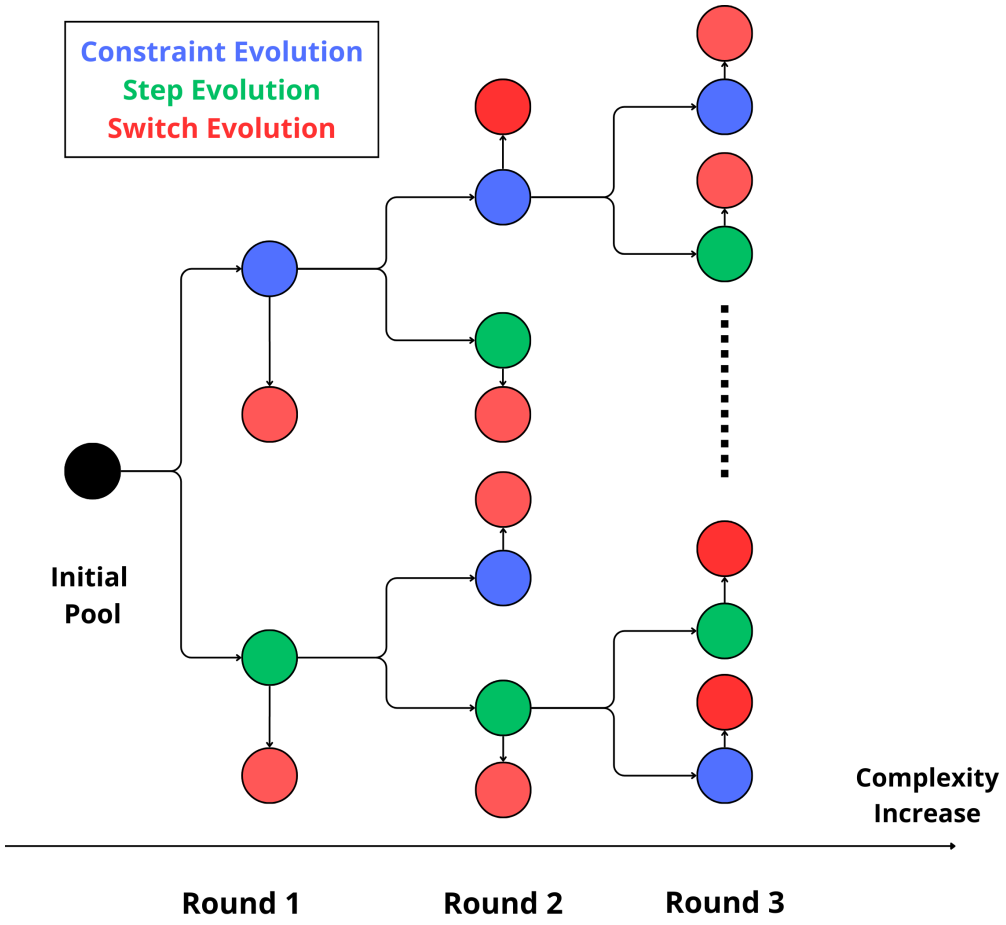

Figure 3.6: Overall process of data generation. The figure illustrates the complete process of dataset generation for Task 3. Starting from the initial pool of manually validated Bash instructions, the three evaluation strategies are applied iteratively to create three levels of difficulty.

The Figure 3.6 illustrates the complete process used to generate the dataset for

Task 3. Starting from the initial pool of manually validated Bash instructions, the three evaluation strategies are applied iteratively to create three levels of difficulty: Easy, Medium, and Hard.

In this task, the order in which the strategies are applied is crucial. In particular, the Switch Evolution is applied only after the Constraint and Reasoning Evolutions, as it preserves the difficulty of the original instruction. Therefore, the complexity must first be increased through the Constraint and Reasoning Evolutions, and only at this point the Switch Evolution can be applied to introduce diversity into the pool. This also explains why, in the figure, the instructions generated with the Switch Evolution are placed at the same horizontal level as the original instructions.

For this task, each round is not described in detail, as all rounds share the same structure and differ only in the difficulty level of the generated instructions. Specifically, in every round the three strategies are applied iteratively to all instructions produced in the previous step following the sequence described above. Therefore, the process begins with the Constraint and Reasoning Evolutions followed by the application of the Switch Evolution. It is important to note that, for computational efficiency, in each round the Constraint and Reasoning Evolutions are applied only to those instructions that were not generated through the Switch Evolution.

At the end of the three rounds of evolution, I obtain a set of open-ended instructions paired with their corresponding Bash script solutions for each difficulty level.

3.4.4 Evaluation Process

This phase involves the evaluation of the generated instructions to select the high-quality ones that will be included in the final benchmark. Also for this task, the evaluation process is performed relying on the G-eval library using GPT-4 as judge LLM.

Moreover, both evaluation steps and the scoring rubric are redesigned to fit the specific characteristics of this task. In particular, the evaluation steps are defined as follows:

1. Identify the requirements necessary to fulfill the given instruction.
2. Analyze the provided Bash snippet and determine whether it plausibly attempts to fulfill these requirements.
3. Identify whether any logical or syntactical mistakes are present in the snippet.
4. Evaluate whether the snippet remains runnable or near-runnable, and whether it could be easily corrected to run successfully.
5. Assess whether the snippet overall aligns with the conceptual and operational intent of the instruction.

After the definition of the evaluation steps, the rubric is defined as follows:

1. **Score 0-1**: Snippet completely unrelated or invalid. Fails to address any part of the instruction.
2. **Score 2-3**: Snippet loosely related to the goal but includes severe logical/syntactic errors. Not runnable or misaligned.
3. **Score 4-5**: Snippet runnable but incomplete: only partially fulfills the goal or ignores key requirements.
4. **Score 6-7**: Snippet mostly correct but lacks completeness or robustness (e.g., missing error handling or key flags).
5. **Score 8-9**: Snippet fully operational and logically correct; only minor stylistic issues.
6. **Score 10**: Snippet fully correct and robust. Handles all edge cases and matches the instruction perfectly.

Then, the evaluation process is executed obtaining a score for each instruction-script pair.

In this task, the selection process differs from the previous ones. Instead of choosing the 100 sessions with the highest scores, an additional constraint based on semantic similarity is introduced to better cover a wide range of topics.

Therefore, after sorting the instructions by score, 100 instructions for each difficulty level are selected ensuring that their semantic similarity with the previously selected instructions remains below a predefined threshold.

At the end of the selection process, the final benchmark for this task is obtained, consisting of 300 open-ended instructions equally distributed across the three difficulty levels.

Chapter 4

Results

This thesis presents a comprehensive benchmark designed to evaluate the knowledge of Large Language Models (LLMs) in the Bash domain. The benchmark is structured into three distinct tasks: Factual knowledge, Conceptual knowledge, and Practical knowledge.

After the creation, the benchmark is used to evaluate the performance of several families of models – OpenAI, LLaMA and DeepSeek – to assess their performance across these tasks. Each task is analyzed in a dedicated section below, reporting also the result obtained during the step of the evaluation process and the data obtained after the performance of the evaluated models.

A final discussion section summarizes the key findings and insights derived from the results.

4.1 Task 1: Factual Knowledge

This section presents the result obtained by evaluating the selected models on the Factual Knowledge task. This task is designed to assess the models' knowledge of Bash commands and their functionalities. The following section presents the intermediate results obtained during the evoluation process, followed by the final results obtained by the evaluated models.

4.1.1 Evolved Question Dataset

The figure 4.1 shows the structure of the resulting dataset after the evolution process described in Section 3.2.3. The dataset consists in 2010 question-answer pairs, including the initial pool, organized into three levels of difficulty: Easy (level1), Medium (level2), and Hard (level3).

Each level of difficulty contains an increasing number of pairs, as a direct

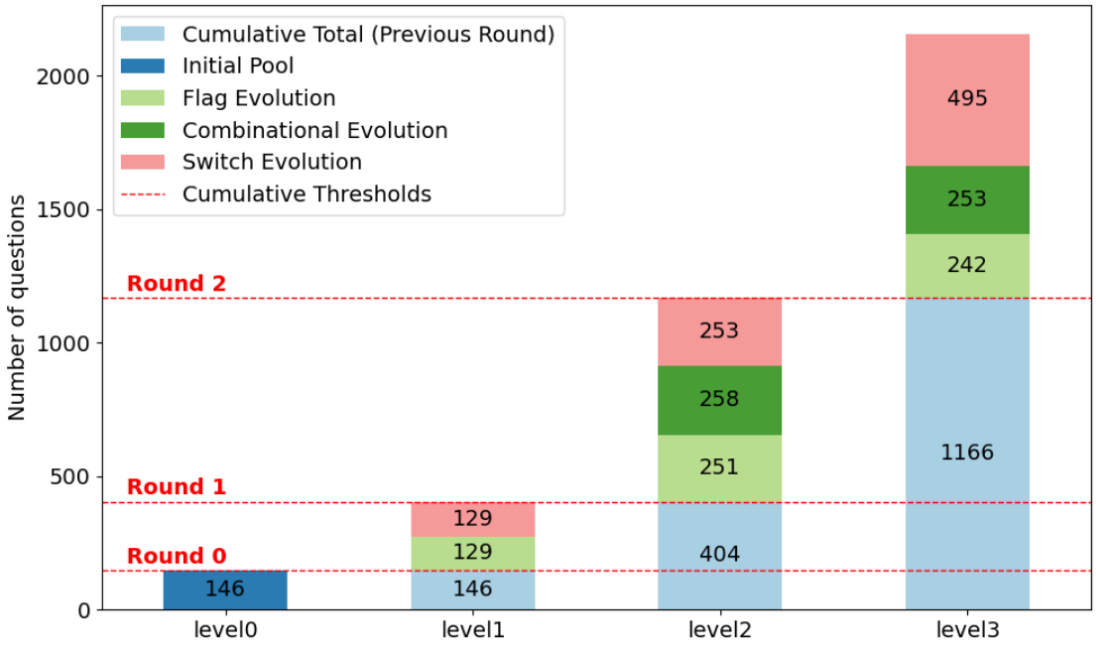

Figure 4.1: Task 1 dataset. The figure illustrates the dataset structure after the evolution process. Each bar represents a different level of difficulty. The stacked bar represents the number of questions generated by each evolution methodology.

consequence of the iterative evolution process. Indeed, at every round, the model takes as input the question-answer pairs generated in the previous round and expand them increasing the total size of the dataset.

Moreover, the figure shows the distribution of questions generated by each evolution methodology. Since every strategy is specifically designed to increase complexity or introduce diversity in a different manner, this distribution provides a clear view of how the dataset evolves across rounds.

4.1.2 Evaluation Process Results

Starting from the dataset obtained in the previous section, I select 300 questions, 100 for each difficulty level, to be included in the final benchmark. The selection process follows the criteria introduced in Section 3.2.4.

In Figure 4.2, I report the distribution of scores across the different difficulty levels. From the plot, it is possible to observe that level1, the easier, has the highest median score and the widest box, highlighting the presence of higher scores. Moreover, a decreasing trend is visible across levels: the median decrease as difficulty increases. This behavior confirms that the questions are effectively

designed to scale in difficulty and that the response generation process becomes increasingly complex. Consequently, models tend to achieve lower scores at higher difficulty levels.

It is also important to note that more or less the 25% of the questions, for each level of difficulty, have a score below the treshold. This probably due to the strictness of the criteria used to evaluate the questions. Indeed, having so many questions, I decided to be very selective in order to include only high-quality questions in the final benchmark.

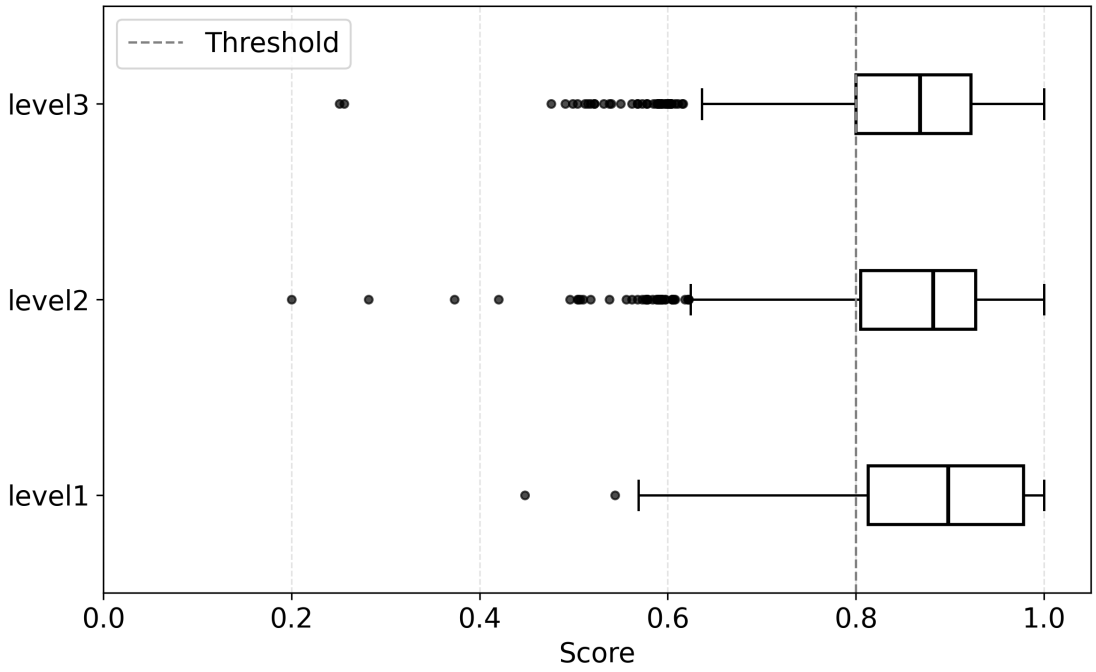

Figure 4.2: Score of an LLM-as-a-judge for the generated questions on factual knowledge. The figure illustrates the distribution of evaluation scores for the generated questions across the different difficulty levels. Each box represents a difficulty level, showing the median, interquartile range, and potential outliers. The dashed horizontal line indicates the acceptance threshold ($score \geq 0.8$).

4.1.3 Multiple Choiche Creation

The multiple-choice questions are created following the methodology described in Section 3.2.5. Therefore, for each of the 300 selected questions in the previous section, a pool of four answer options is generated containing one correct answer and three incorrect option, also called distractors.

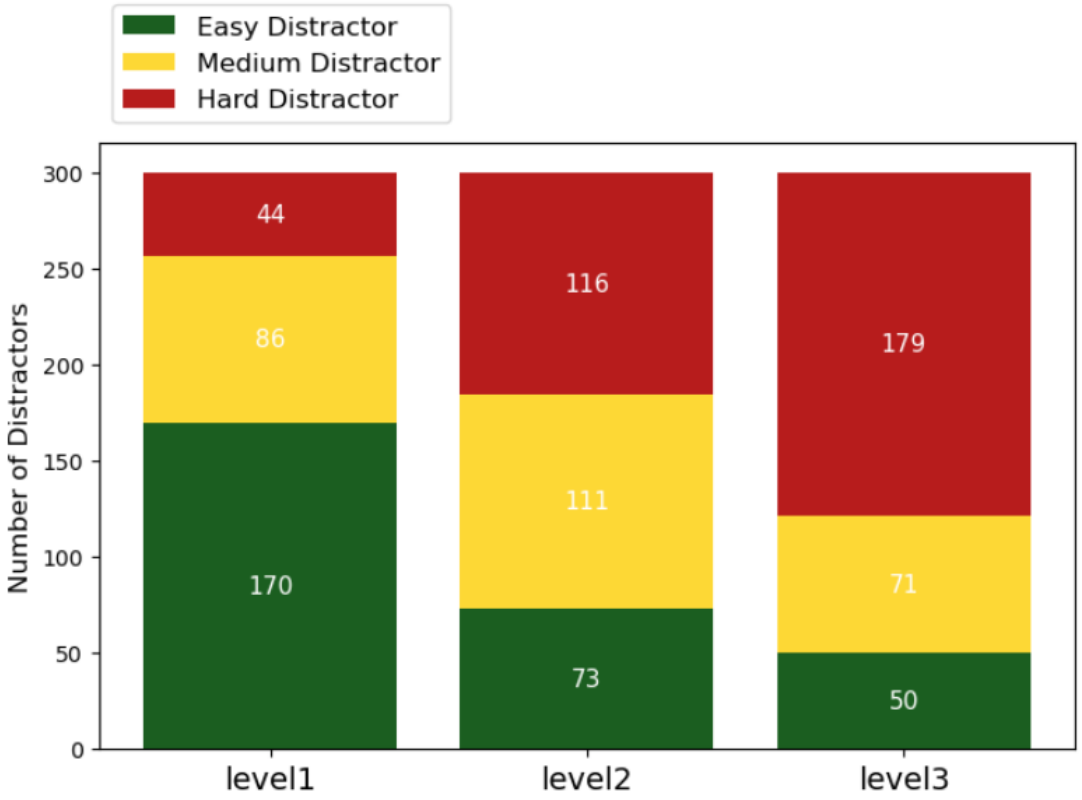

Figure 4.3: Distractors Distribution for level. The figure shows the distribution of distractors plausibility across different levels of difficulty. Each bar represent a difficulty level, with the stacked segments indicating the number of distractors for each level of plausibility. The trend shows that as the difficulty level increases, the number of distractors with higher plausibility also increases.

The distractors are not chosen randomly, but are carefully selected based on their plausibility which vary according to the difficulty level of the question. In particular, as the difficulty level increases, the plausibility of the distractors also increases, making it more complex for models to identify the correct answer.

Figure 4.3 clearly shows the general trend described above for which the number of distractors more plausible increase as the difficulty level increases. However, this figure provides only a general overview of the distractors plausibility distribution for each level, without offering any information about the complexity of individual questions.

This aspect is examined more closely in Figure 4.4, which provides a question-level view. In particular, Figure shows that, as the difficulty increases, each question tends to contain a larger number of hard distractors. This indicates that the overall

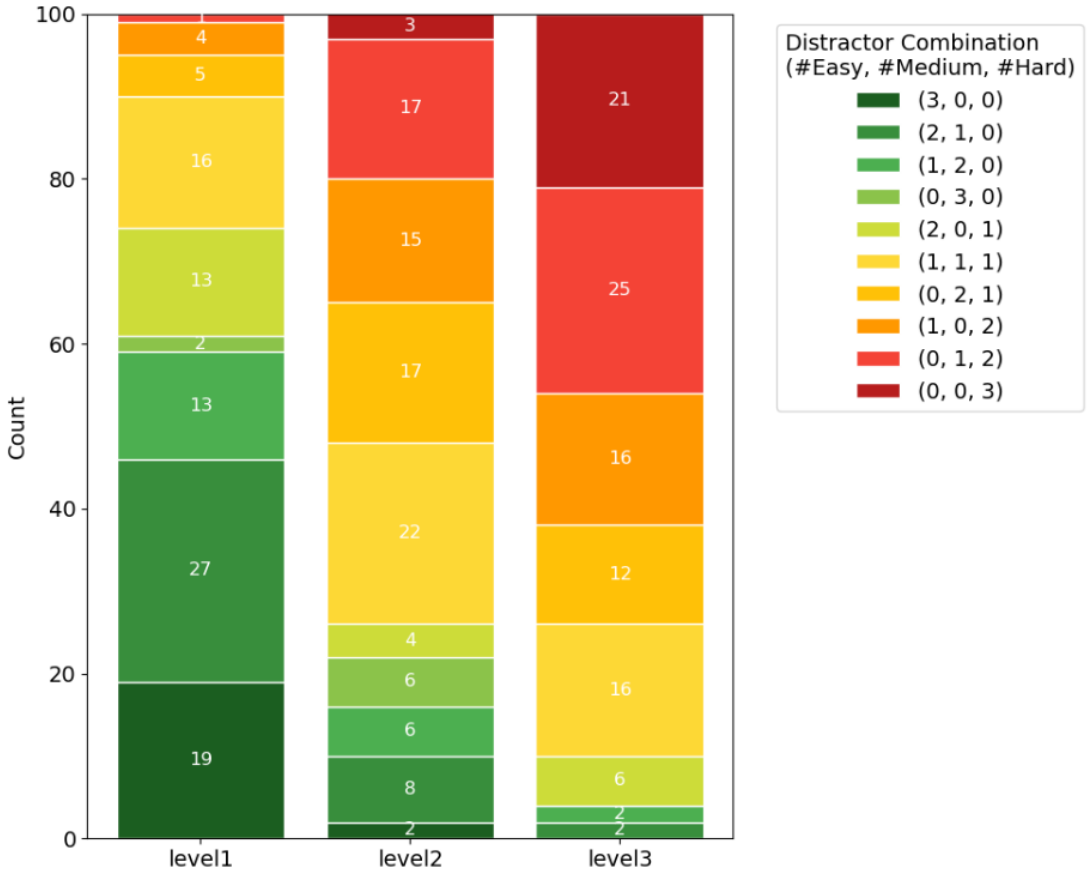

Figure 4.4: Question Distribution for Level. The figure shows the distribution of the question containing a specific combination of distractors plausibility. Each bar corresponds to a difficulty level, and the stacked segments indicate the number of questions with a given plausibility triplet (easy, medium, hard), as reported in the legend. This plot provides a more detailed view of the questions' complexity at each difficulty level: as difficulty increases, questions composed by low-plausibility distractors become much less frequent, while those containing mostly high-plausibility distractors become more common.

complexity of the questions increases as well, since questions containing highly plausible distractors are more challenging to solve.

4.1.4 Models Evaluation

Figure 4.5 illustrates the performance of various models on the Factual Knowledge task across different levels of complexity. This task consists of multiple-choice questions, where models are required to select the correct answer from a set of

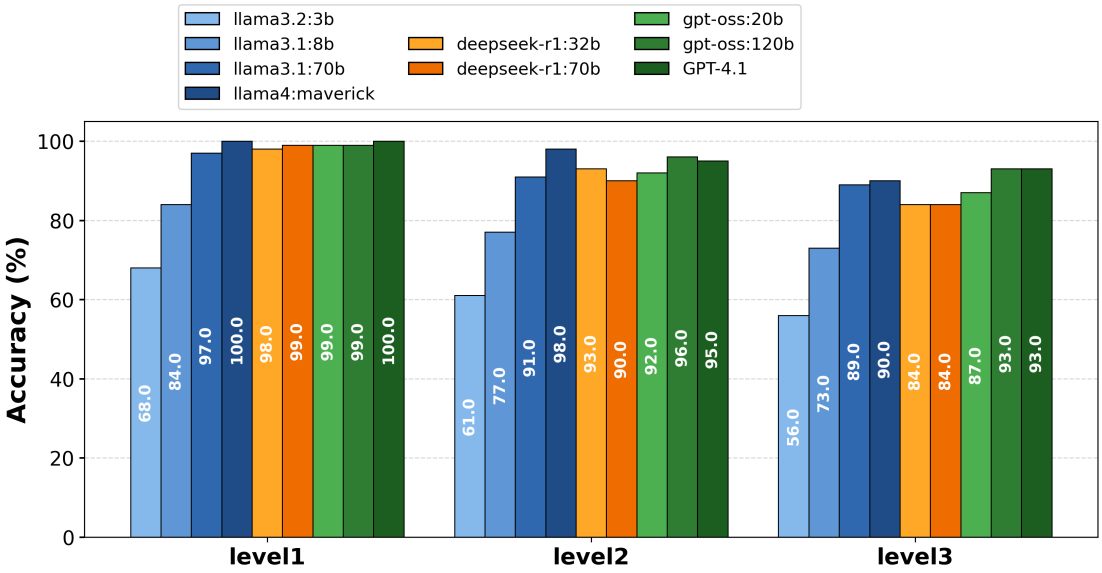

Figure 4.5: Model Evaluation Result for Task 1. The figure illustrates the performance of different models on the Factual Knowledge task. I report, in the X-axis, the level of complexity, while the Y-axis shows the accuracy percentage achieved by each model.

options. Therefore, the models are evaluated based on their accuracy, computed as the percentage of correctly answered questions.

Each model is run once for each complexity level, and the accuracy is recorded.

As observed in the results, the trend indicates that as the complexity of the questions increases, the performance of all models tends to decrease. This demonstrated that the questions are effectively designed to increase in difficulty.

Larger models generally perform better than smaller ones, confirming that model size plays a significant role in handling factual knowledge tasks. Nevertheless, smaller models show solid performance at lower complexity levels, indicating their ability to capture basic factual knowledge about Bash commands. This suggests that smaller models remain effective for simpler factual tasks, while larger models become necessary as the complexity of the questions increase.

In addition to the overall performance results, further analysis are conducted to understand the behavior of the models when they select incorrect answers. Specifically, the analysis examines both the plausibility level of the selected distractors and the evolutionary strategy responsible for their generation.

Starting with the plausibility level analysis, figure 4.6 illustrates the distribution of answers selected by different models depending on the difficulty level of the questions. Each bar represents a model, with the stacked segments indicating the

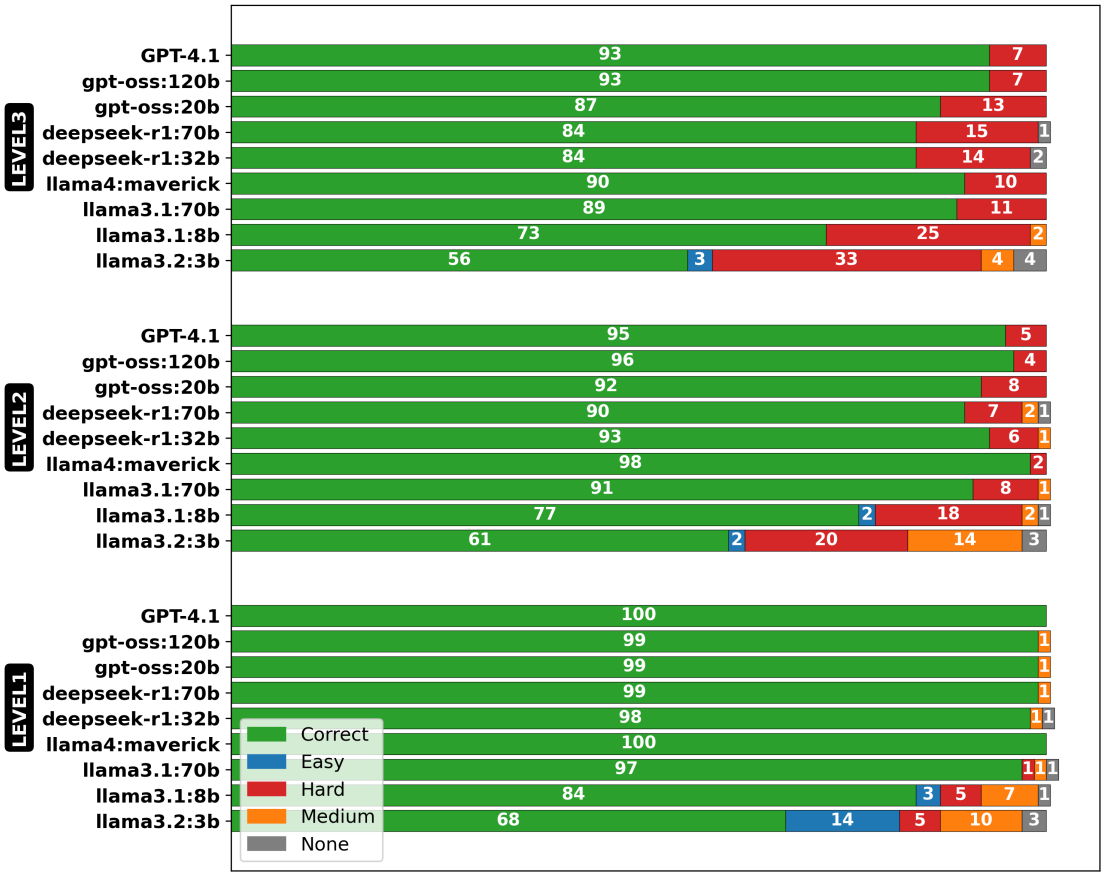

Figure 4.6: Answer Distribution by Difficulty Level. The figure shows the distribution of answers selected by different models depending on the difficulty level of the questions. Each bar represents a model, with the stacked segments indicating the number of correct and wrong answers chosen by each model. For wrong answers, the figure shows the plausibility level of the selected distractors.

nature of the selected answers.

The figure shows a general trend indicating that the bigger models tend to select more plausible distractors regardless of the difficulty level of the questions. However, smaller models are not always able to identify plausible distractors. This limitation becomes less evident as the difficulty level increases, probably because the number of highly plausible distractors increases as well.

Taking the results of the llama3.2:3b model as an example, it is possible to observe that at the Easy level (level1) most incorrect answers are associated with distractors of easy or medium plausibility. As the difficulty increases, the number of highly plausible distractors selected by the model also increases. This is likely

because the model tends to make mistakes on the most complex questions, which include highly plausible distractors, as shown in Figure 4.4.

Figure 4.7: Answer Distribution for evolution strategy. The figure shows the distribution of answers selected by different models depending on the difficulty level of the questions. Each bar represents a model, with the stacked segments indicating the number of correct and wrong answers chosen by each model. For wrong answers, the figure shows the evolution strategy applied to obtain the distractor.

The same analysis is performed considering the evolution strategy used to generate the distractors. The results are illustrated in figure 4.7 indacates that the overall distributed of the selectes distractors is quite uniform across the different strategies, even if the Flag Evolution one is slightly more effectively. This suggests that no single strategy is clearly more effective than the others in misleading the models, confirming the robustness of the distractor generation process.

4.2 Task 2: Conceptual Knowledge

This section presents the result obtained by evaluating the selected models on the Conceptual Knowledge task. This task is designed to assess the model's ability to understand and reason about underlying concepts of the Bash language, specifically evaluating the similarity between sessions. The following section presents the intermediate results obtained during the evolution process, followed by the final results obtained by the evaluated models.

4.2.1 Initial Pool

The initial pool for the conceptual knowledge task consists of 191 session, each paired with its natural language explanation. As described in section 3.3.1 these sessions are filterd from a the larger NLP2Bash collection [16], using the criteria defined in the same section, to ensure a wide variety of conceptual topics.

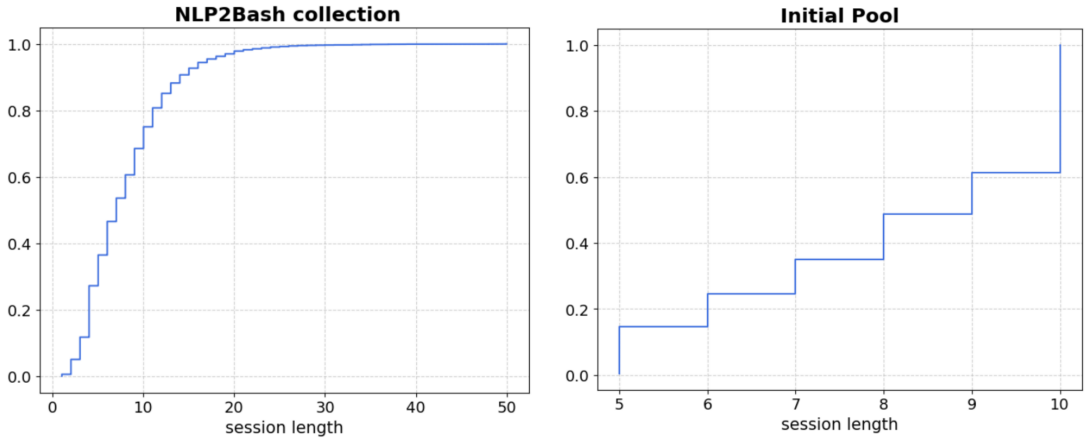

Figure 4.8: Comparison between NLP2Bash and Initial Pool. The figure shows the Empirical Cumulative Distribution Function (ECDF) of the sessions' length for both the original NLP2Bash dataset and the Inital Pool obtained after the filtering process, see Section 3.3.1.

The Figure 4.8 shows the ECDF of the filtered initial pool compared to the original collection. From the figure, it is possible to observe that most of the filterd sessions are concentrated between 9 and 10 words (about 50% of the total), reflecting the effort to select sessions that are not too simple and that still express a clear conceptual goal, while remaining suitable for further evolution. At the same time, the shorter sessions also represent a good starting point. Although they express simpler conceptual goals, they still offer a solid base that can be further refined and expanded during the evolution process.

4.2.2 Evolved Question Dataset

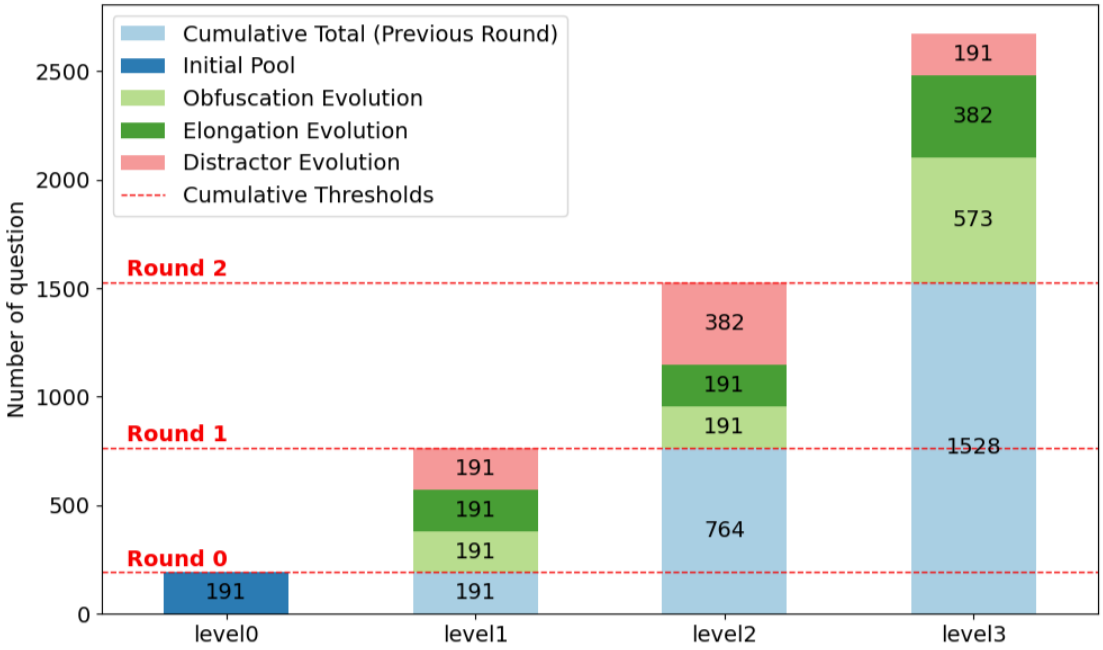

Figure 4.9: Evolved Dataset Distribution. The figure shows the evolution of the dataset. For each level, the contributions of the different evolution strategies are displayed cumulatively. The dashed red lines represent the cumulative thresholds reached at each round, highlighting how the total number of sessions increases progressively during the evolution process, see Figure 3.5.

At the end of the evolution process, the dataset consists of a total number of sessions distributed across the three difficulty levels as rappresented in Figure 4.9. The total dataset contains 2483 generated sessions.

Moreover, the figure shows how the evolution process significantly expanded the number of sessions at each difficulty level, also considering that not all strategies were applied, as shown in the Figure 3.5.

4.2.3 Evaluation Process

This section presents the results obtained during the evaluation process, which is designed to assess the alignment between the generated sessions and their corresponding natural language explanations.

Recalling that the explanation describes the conceptual goal that the model intended to express through the generated session, the evaluation process focuses

on verifying this alignment. If the two match, the model managed to generate a session consistent with what it meant to achieve.

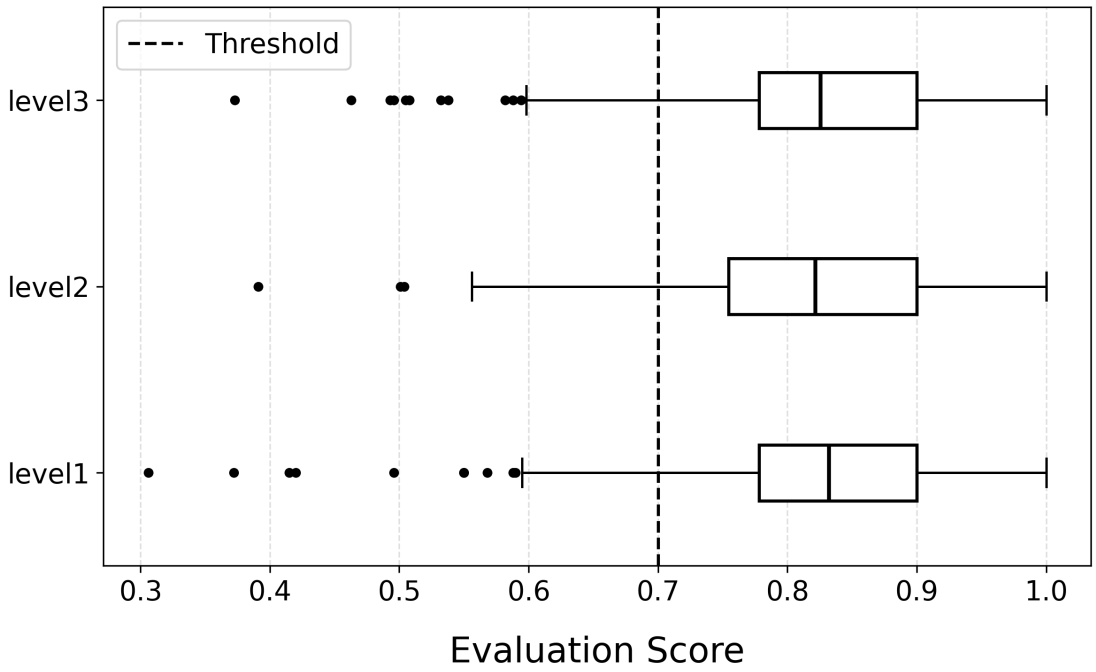

Figure 4.10: Boxplot Evaluation Process Results. The figure illustrates the distribution of evaluation scores for the generated questions across the different difficulty levels. Each box represents a difficulty level, showing the median, interquartile range, and potential outliers. The dashed horizontal line indicates the acceptance threshold ($score \geq 0.7$).

As done in Task 1, a boxplot figure is used to better visualize the distribution of scores across the different difficulty levels. From Figure 4.10, it is possible to observe that all three levels have a similar score distribution. In particular, the easier levels have a slightly higher median score, meaning that its sessions are, on average, evaluated a bit better.

However, the difference between the three levels is very small. This happens because the evolution process aims mainly at increasing the variety of the sessions, not their complexity. For this reason, the complexity of writing the explanations is quite similar for all levels.

As a result, even though the Easy level still performs slightly better, the gap with the Medium and Hard levels is much smaller than in Task 1.

Figure 4.11: Group based Evaluation. The figure shows the result of the group-based evaluation approach used in the selection process. Each bar represents a group of sessions with the same original hashroot, sorted by their average score in descending order, and is colored according to the criteria described in the legend.

4.2.4 Selection Process

After the evaluation phase, a selection step is performed to choose the best sessions for each difficulty level.

Instead of simply picking the 100 sessions with the highest scores, a different strategy is used to ensure that the complexity increases across the three levels. Specifically, the sessions are grouped based on their original hashroot and the average score for each group is calculated.

The Figure 4.11 shows the result of this approach. Each bar in the figure represents a group of sessions, sorted by their average score in descending order, and is colored according to the following criteria:

- **Green – All correct:** all sessions in the group have scores above the threshold.
- **Yellow – Wrong but only irrelevant :** at least one session in the group is below the threshold, but none of the below-threshold sessions will be used as correct answers in the multiple-choice questions.
- **Red – Wrong with relevant:** at least one session in the group is below the threshold, and at least one of these below-threshold sessions will be used as a correct answer in the multiple-choice questions.

Based on this classification, the 100 hashroots with the highest average scores and not marked in red are selected. Then, from each selected group, three multiple-choice questions are created, one for each difficulty level. This approach ensures

that the complexity of the generated questions increases progressively from Easy to Hard, while maintaining a high level of quality across all levels.

4.2.5 Model Evaluation

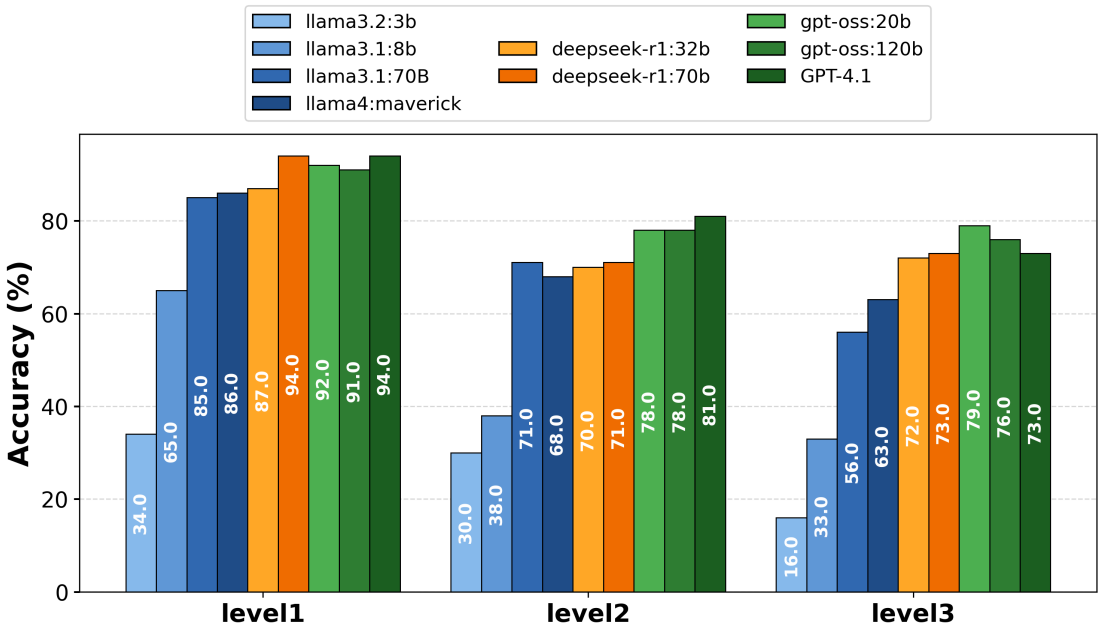

Figure 4.12: Model Evaluation Results. The figure shows the performance of different models on the Conceptual Knowledge task across three difficulty levels. Each bar represents a model's accuracy at a specific difficulty level, allowing for a comparative analysis of model capabilities.

The final step involves evaluating the selected models on the Conceptual Knowledge task using the benchmark questions generated in the previous sections. This task consists of 300 multiple-choice questions designed to verify if the model is able to identify two "similar sessions". In particular, the model must understand the underlying concept expressed in the session given as reference and identify, among six options, the session with the most closer conceptual goal.

The Figure 4.12 illustrates the performance of different models on the Conceptual Knowledge task across three difficulty levels. Each bar represents the model's accuracy at a specific difficulty level, allowing the comparison between the model.

From the figure, it is possible to observe a consistent performance drop across most of the models as the difficulty increases, confirming the soundness of the proposed methodology and validating that the questions' difficulty increases as intended.

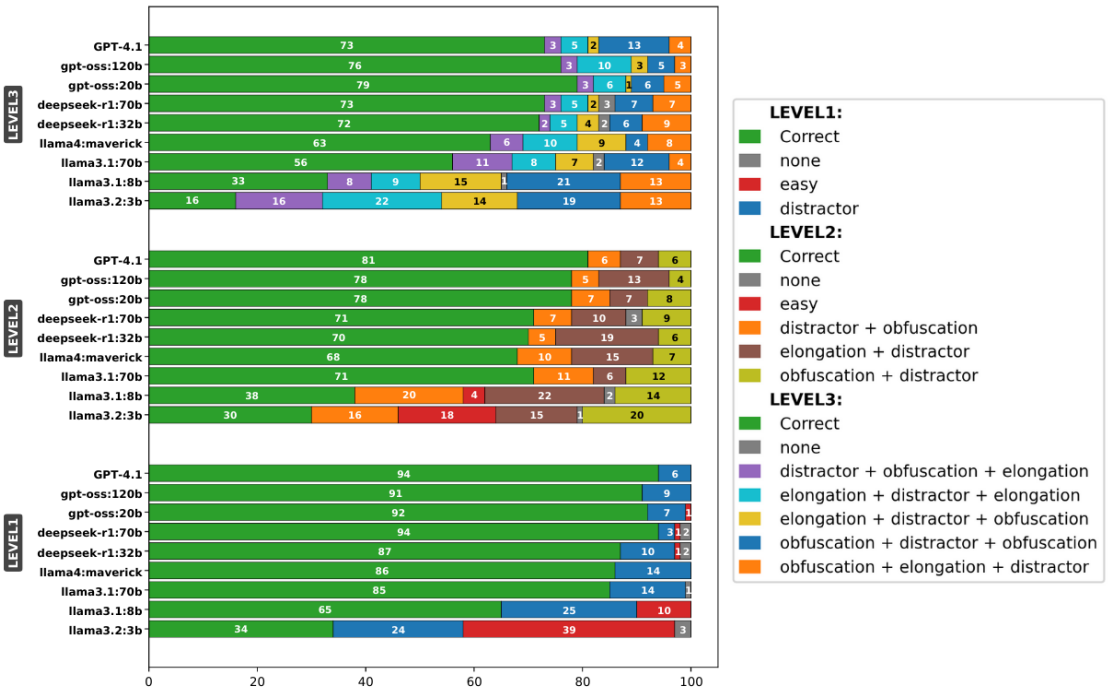

Figure 4.13: Model Errors Nature. The figure shows the nature of the mistakes made by different models on the Conceptual Knowledge task across three difficulty levels. For each model and difficulty level, the distribution of answers is highlighted, with correct answers in green and incorrect ones categorized by type in different colors. The legend indicates the possible distractor categories generated during the evolution process.

A more detailed analysis focuses on the nature of the mistakes made by the models. In particular, Figure 4.13 shows, for each model and difficulty level, the distribution of the answers, highlighting the correct ones in green and the incorrect ones in different colors according to their type. The legend reports, for each difficulty level, the possible categories of distractors generated during the evolution process.

There are two main considerations that can be drawn from the figure. Smaller models, even in the Easy level, tend to make naive mistakes selecting distractors that are clearly unrelated to the goal of the root session. This behavior indicates that these models struggle in understanding the underlying concepts, even in simpler scenarios, while larger models, even when they make mistakes, tend to select distractors that are more closely related to the intended concept.

Second, none of the strategy seems to be more effective than others in confusing the models. Indeed, as we can see from the figure, for each model and difficulty

level, the distribution of the wrong answers is quite balanced among the different types of distractors.

4.3 Task 3: Practical Knowledge

This section presents the result obtained by evaluating the selected models on the Practical Knowledge task. This is the third task of the benchmark and is designed to evaluate the ability of the models to generate valid Bash scripts that satisfy specific requirements. The following section presents both the intermediate results obtained during the evolution process, followed by the final results obtained by the evaluated models.

4.3.1 Evolved Dataset

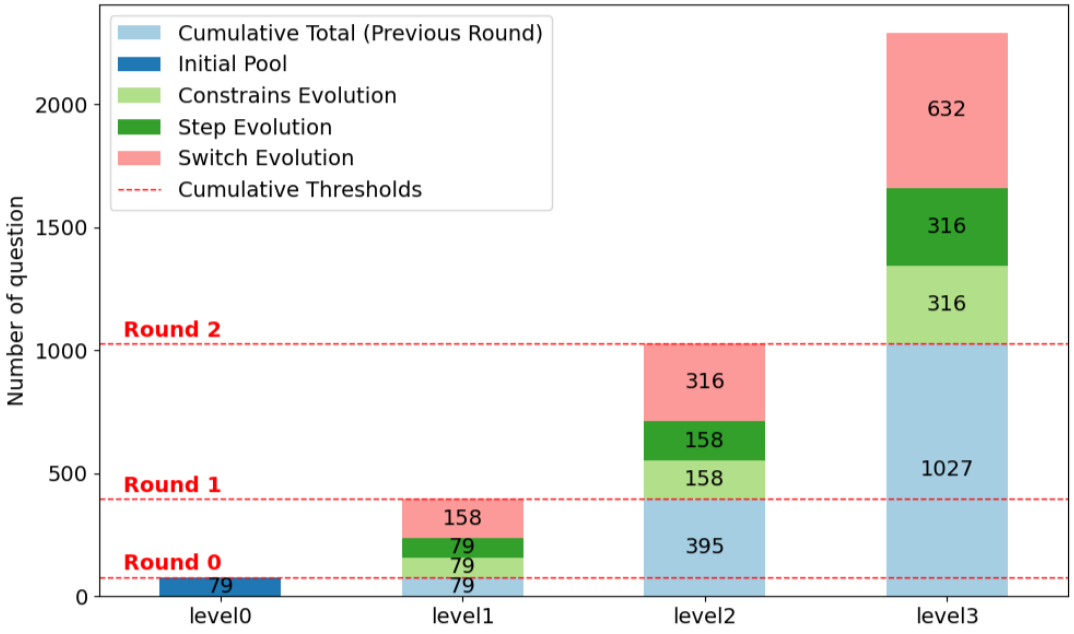

Figure 4.14: Evolved Dataset Distribution. The figure shows the evolution of the dataset. For each level, the contributions of the different evolution strategies are displayed cumulatively. The dashed red lines represent the cumulative thresholds reached at each round, highlighting how the total number of instructions increases progressively during the evolution process.

At the end of the evolution process, the dataset consists of a total number of instructions distributed across the three difficulty levels as rappresented in Figure

4.14. The total dataset contains 2212 instructions.

Moreover, the figure shows how the evolution process significantly expanded the number of instructions at each difficulty level, also considering that not all strategies were applied.

4.3.2 Evaluation Process

This section presents the results of the Practical Knowledge task, evaluating how accurately the generated Bash scripts satisfy the requirements defined in the corresponding instructions. This verification is necessary because the generated scripts will be used in the following section as ground truth for computing the Levenshtein distance. Therefore, it is essential to ensure that the selected instruction is associated with a valid script that meets the specified requirements.

As done in the previous tasks, a boxplot figure is used to better visualize the distribution of scores across the different difficulty levels.

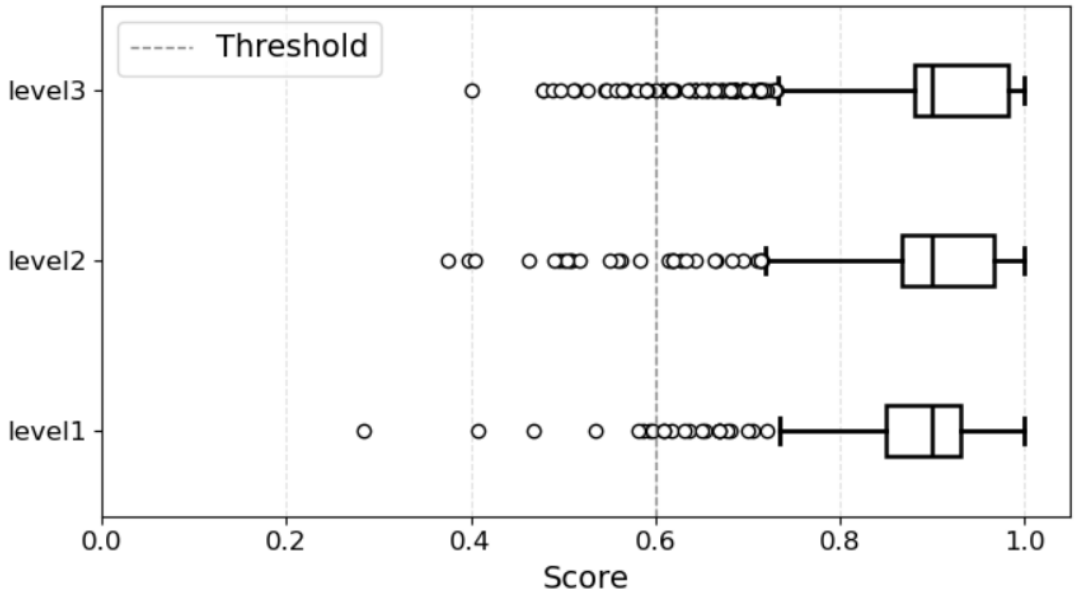

Figure 4.15: Boxplot Evaluation Process Results. The figure illustrates the distribution of evaluation scores for the generated instructions across the different difficulty levels. Each box represents a difficulty level, showing the median, interquartile range, and potential outliers. The dashed horizontal line indicates the acceptance threshold ($score \geq 0.6$).

In particular, the Figure 4.15 although the median scores are very similar across all levels, unexpectedly the boxes of the harder levels indicate slightly higher average scores. Indeed, the evolution process focuses on increasing the complexity of the

instructions. Therefore, higher scores were expected for the easier levels, where generating the corresponding scripts should be less challenging compared to the harder levels.

A plausible hypothesis is that even the harder instructions may not yet be sufficiently challenging for GPT-4. As a consequence, the model is still able to produce high-quality scripts even at the highest difficulty levels. Moreover, more complex instructions may provide a more detailed context and requirements, which can help the model in the generation of accurate scripts.

4.3.3 Selection Process

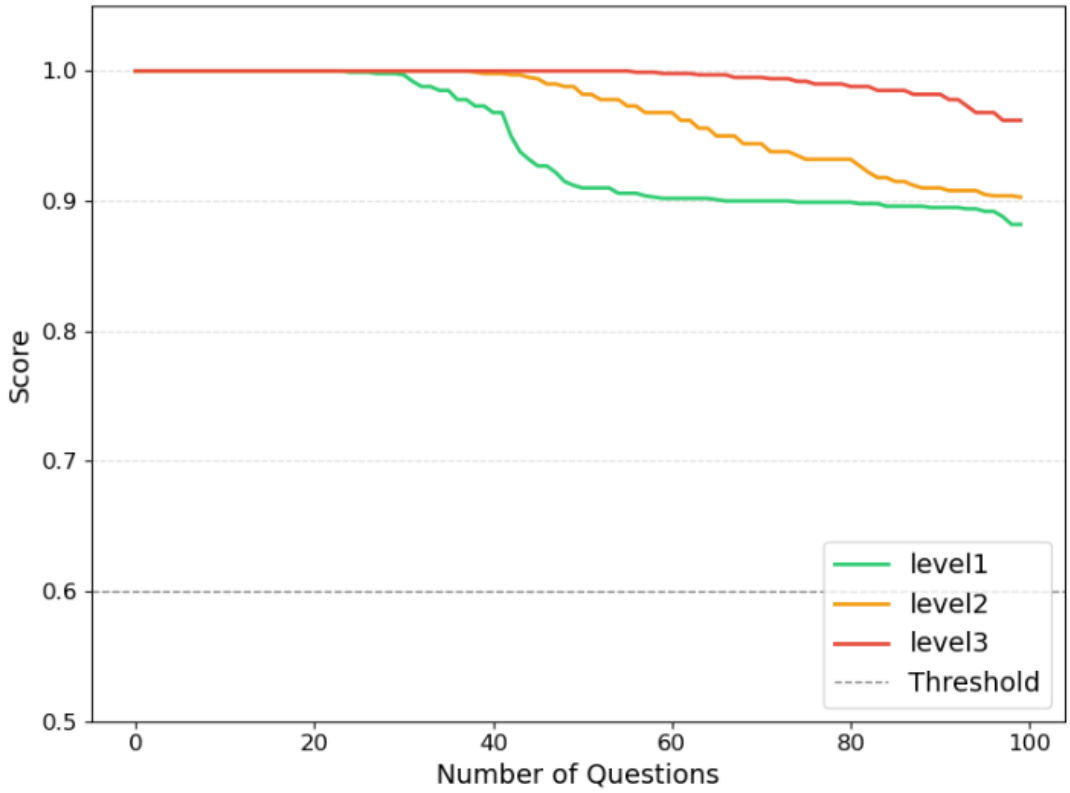

Figure 4.16: Selection Process Results. The figure illustrates the result of the selection process. Each curve represents the score distribution of the instructions at each difficulty level, while the horizontal line indicates the acceptance threshold ($score \geq 0.6$).

After completing the evaluation phase, a selection process is introduced to determine the instructions to include in the final benchmark.

Starting from the evaluated dataset, the instructions are sorted in descending order based on their scores. Then, 100 instructions that meet the selection criteria are chosen for each difficulty level. In particular, a similarity filter is applied to avoid including instructions that are too similar to one another. Since the instructions are written in natural language, Sentence-BERT embeddings can be used to measure their semantic similarity. This approach ensures that the selected instructions are both high-quality and cover a wider range of topics.

Figure 4.16 illustrates the score distribution of the selected instructions for each difficulty level. The figure shows that, even after applying the selection criteria, the score distributions remain consistently high across all levels. This indicates that the selection process did not significantly affect the overall quality of the selected instructions.

4.3.4 Model Evaluation

This section presents the final results obtained by evaluating the selected models on the Practical Knowledge task. Specifically, two types of evaluations are conducted: one based on the Levenshtein distance and another using an LLM-based approach.

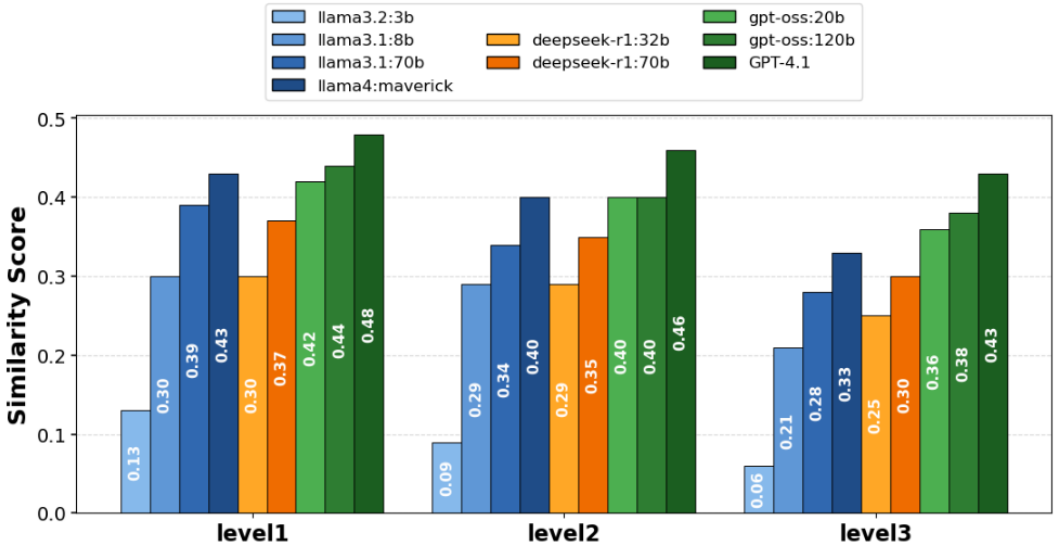

Figure 4.17: Levenshtein Similarity Results. The figure illustrates the results obtained by evaluating the selected models using the Levenshtein distance metric. Each bar represents the average similarity score obtained by a model for each difficulty level.

Levenshtein Distance: The Levenshtein distance is a metric used to measure the difference between two strings. In particular, it counts the minimum number of operations (insertions, deletions, or substitutions) required to transform one string into the other. Moreover, to give a more interpretable score, instead of using the raw Levenshtein distance value, a normalized score between 0 and 1 is computed as follows:

$$\text{similarity}(A, B) = 1 - \frac{d_{\text{lev}}(A, B)}{\max(|A|, |B|)}$$

where $d_{\text{lev}}(A, B)$ is the Levenshtein distance between strings A and B, and $|A|$ and $|B|$ are the lengths of the respective strings.

In this context, it is used to compare the generated Bash scripts with the ground truth scripts associated with each instruction. A higher similarity score indicates a closer match between the generated and ground truth scripts, suggesting better performance by the model.

The results obtained from the Levenshtein distance evaluation are presented in Figure 4.17. As expected, GPT-4 outperforms the other models across all difficulty levels, remembering that it is the same model used during the generation of the ground truth scripts. Futhermore, all models exhibit a decrease in performance as the difficulty level increases, in fact, challenging instructions usually require more complex and longer scripts, which can lead to an increase of the distance and consequently a decrease in the similarity score. Furthermore, all models exhibit a decrease in performance as the difficulty level increases. Indeed, more challenging instructions typically require longer scripts, which often results in a higher Levenshtein distance and, consequently, a lower similarity score.

Figure 4.18 provides a more detailed view of the distribution of Levenshtein distances for each model and difficulty level. The figure shows a clear trend for which both the median and the average distance increase as the difficulty level increases, consistently across all models. Furthermore, at the Easy level, some models are able to generate scripts that are very close to the ground truth. However, as the difficulty increases, the distances become larger, indicating that producing syntactically similar scripts becomes progressively more challenging.

LLM-based Evaluation: The second evaluation approach relies on an LLM-based procedure, applying the same criteria used during the evaluation phase described in Section 3.4.4. The evaluation is performed using both GPT-4.1 and GPT-5 as judge LLMs.

Figure 4.19 presents the average score obtained by each model for each difficulty level using the LLM-based evaluation approach.

After collecting the responses produced by all models on the selected instructions, the answers are assessed with GPT-4.1 and GPT-5 following the G-Eval framework, using the same evaluation steps and scoring rubric defined in Section 3.4.4. As in

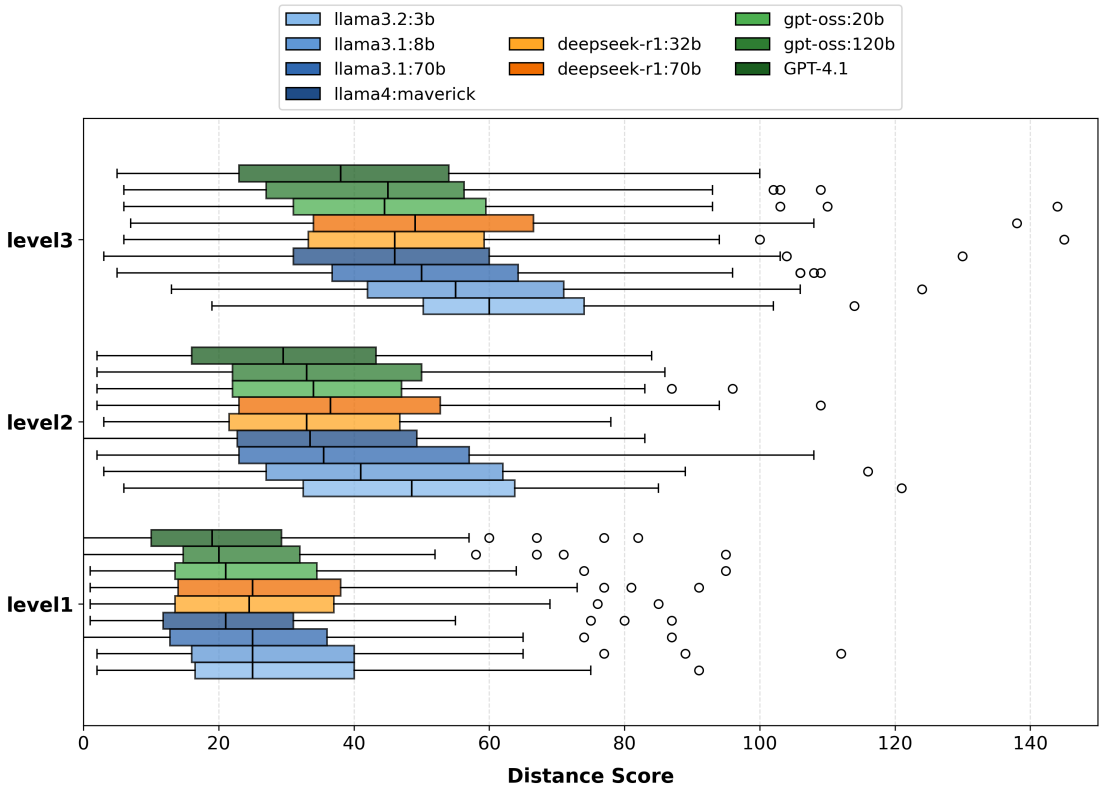

Figure 4.18: Levenshtein Distance Distribution. The figure shows a boxplot of the Levenshtein distances for each model and difficulty level. Higher distances mean larger differences from the ground-truth scripts and therefore lower performance.

the original evaluation process, the goal is to measure how well the generated Bash scripts satisfy the requirements specified in the instructions.

The two plots in the figure compare the results obtained with the two judge models: the lower plot corresponds to GPT-4.1, while the upper plot shows the evaluation performed with GPT-5. Comparing the two plots, GPT-5 consistently assigns lower scores than GPT-4.1 across all models and difficulty levels, about 10-15% lower. This suggests that GPT-4.1 tends to overestimate the quality of the generated scripts.

Taking the value obtained with GPT-5 as the reference, it is possible to observe that even if the scores decrease as the difficulty level increases, the OpenAI models tend to be consistent across all levels. Recalling the hyphotesis made during the evaluation process, this behavior suggests that even the harder instructions may not yet be sufficiently challenging for this family of models.

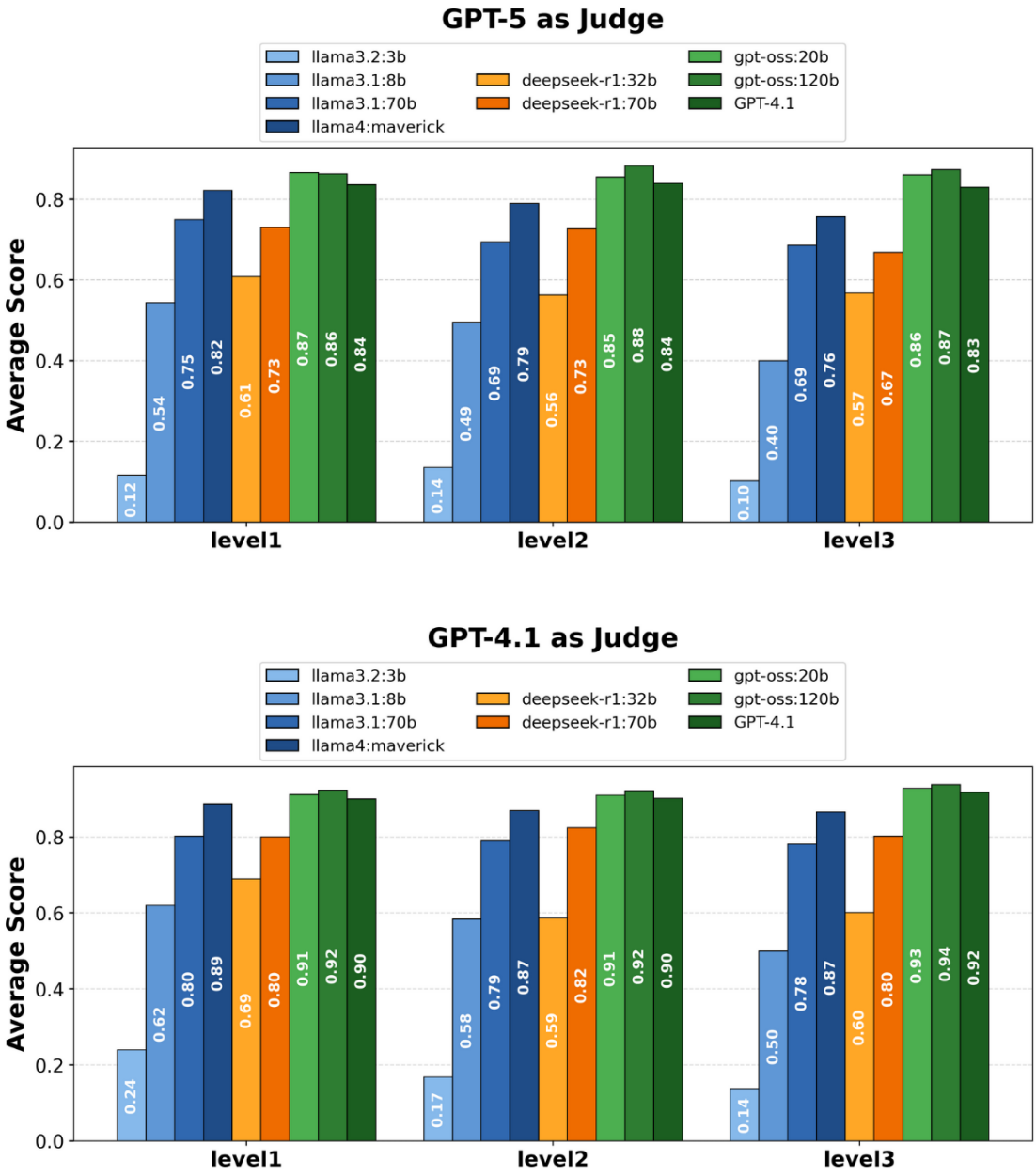

Figure 4.19: LLM-based Evaluation Results. Average score obtained by each model across difficulty levels using the LLM-based evaluation approach. The upper plot reports results evaluated with GPT-5, while the lower plot shows those obtained with GPT-4.1.

Chapter 5

Conclusion

This thesis develops a generalizable pipeline for the creation of benchmarks in technical-domains. The proposed pipeline adapting the EvolInstructor framework[9] to generate domain-specific instruction of varying complexity to analyse better then entire domain.

The pipeline was then applied to the Bash domain as case of study. The resulting benchmark consists in 900 instructions, equally distributed across three different task, each designed to verify a different aspect of the knowledge required to master the Bash language: Factual, Conceptual and Practical.

Futhermore, Each task is futher divided into three levels of difficulty: Easy, Medium, and Hard. The goal is not only to evaluate whether a model can produce correct Bash commands, but also to gain an overall understanding of its strengths and weaknesses across the entire domain.

After the creation of the benchmark, three families of LLMs – GPT, DeepSeek, Llama – are evaluated on it. The overall results are summarized in Table 5.1.

The results show that the performance of the models varies significantly across the different tasks and levels of difficulty, as highlighted by the bold values in the table. As expected, all models perform better on the Factual Knowledge task, with an average accuracy of 89.41%, compared to 71.34% on the Conceptual Knowledge task and 67.43% on the Practical Knowledge task. This indicates that the models are better at recalling factual information than applying it in real-world scenarios.

Another interesting observation emerges from the task-level analysis. Within each task the performance of the models decreases as the difficulty level increases. For example, in the Factual Knowledge task, the average accuracy drops from 95.03% on Easy questions to 84.00% on Hard questions. This trend is consistent accross all the tasks, indicating that the questions are effectively designed to challenge the models at different levels of complexity.

Furthermore, detailed analysis reveals that smaller models perform well on simpler tasks but suffer a significant drop in the performance as task complexity

Task / Level	LLaMA(4)	DeepSeek(2)	OpenAI(3)	Average per Level
Factual Knowledge – Accuracy				
Easy	87.25%	98.50%	99.33%	95.03%
Medium	81.75%	91.50%	94.33%	89.19%
Hard	77.00%	84.00%	91.00%	84.00%
Family/Task Average	82.00%	91.33%	94.89%	**89.41%**
Conceptual Knowledge – Accuracy				
Easy	67.50%	90.50%	92.33%	83.44%
Medium	51.75%	70.50%	79.00%	67.08%
Hard	42.00%	72.50%	76.00%	63.50%
Family/Task Average	53.75%	77.83%	82.44%	**71.34%**
Practical Knowledge – LLM score				
Easy	55.70%	67.00%	85.60%	69.40%
Medium	52.70%	64.50%	85.60%	67.60%
Hard	48.70%	62.00%	85.30%	65.30%
Family/Task Average	52.37%	64.50%	85.50%	**67.43%**

Table 5.1: Model evaluation results per task and difficulty level across the three model families. The numbers in parentheses indicate the number of models evaluated for each family. The bold values represent the overall average performance for each task across all families.

increases. For instance, as shown in Figure 4.12, in the Conceptual Knowledge task the Llama3.1:8B model achieves an accuracy of 65% in the Easy level, while dropping to 33% in the Hard level, with a reduction of 32 percentage points.

Finally, the results also highlight clear performance differences among the three model families. The OpenAI models, which include the model used to generate the benchmark, consistently outperform the other two families across all tasks and difficulty levels. In contrast, the LLaMA models show the lowest overall performance, probably due to the presence of smaller variants (3B and 8B parameters) within the evaluated set.

5.1 Limitations

This section discusses some limitations of the proposed pipeline and, consequently, of the resulting benchmark created for the Bash domain.

Use of the Same Model as Generator and Evaluator: The evaluation phase relies on the same model, GPT-4, that was used to generate the instructions. This ensure consistency in the evaluation, assuming that the model is able to accurately assess the quality of the instructions it generated. However, an additional experiment showed that GPT-4 sometimes overestimates the quality of its answers. In particular, a subset of instructions was also evaluated with GPT-5, and compared with the score obtain through the GPT-4 evaluation, confirming this tendency. Despite this, the overall impact on the benchmark is expected to be limited. Indeed, only a small set of the generated instructions is included in the final dataset, and the selected ones are those with the highest scores, which in most of the cases received similar scores from both models.

Randomness in the answer: this limitation concerns the first two tasks of the benchmark, Factual and Conceptual Knowledge. Since the answers are multiple-choice, a model may guess the correct option even when it does not actually know it. To mitigate this issue, during the evaluation phase, each questions should be run multiple times, and only the consistenly correct answers should be considered valid.

5.2 Future Work

This thesis can be taken as a starting point for futher research in the field of domain-specific benchmarks creation and the consequent evaluation of LLMs. In The following paragraph are presented some possible directions for future work.

Application of the pipeline on other domain: One of the goal of this thesis is to propose a generalizable pipeline for the creation of domain-specific benchmarks. However, the pipeline has been applied only to the Bash domain as case of study. A possible future work is to apply the pipeline to other technical domains to evaluate its generalizability and effectiveness in different contexts.

Fine-tuning on Domain-Specific Data: Another direction is to fine-tune the smaller models evaluated in this thesis to verify whether they can reduce the performance gap with the larger ones. In particular, this can be done using the "discarded" data generated during the evolution phase, but still useful for training purposes. Moreover this data can be futher expanded performing additional round of evolution.

Appendix A

Prompt Definition

A.1 Task 1

Section 3.2.2 describes in detail the methodology adopted in the evolution process. This section presents the prompts used during the evolution phase.

```
def returnPromptFlags_SecondRound():
    SYSTEM_PROMPT_FLAGS = """
You are a Teacher, using an Evol-Instruct style approach to make Bash questions progressively harder.
Your task is to rewrite Bash-related questions into more complex versions, increasing difficulty **moderately** rather than jumping to extremely advanced options.

Rules:
- Stay strictly within the Bash domain.
- Add or replace exactly **one valid flag/option** (with parameters/files if required).
- If the flag requires a file, input, or string, use a generic placeholder (e.g., <file>, <directory>, <pattern>).
- Prefer **short flags (-x) over long options (--example)** whenever possible.
- Prefer moderately advanced or less commonly used flags. Extremely niche or subtle flags should only appear naturally in future evolutions.
- Ensure the added flag meaningfully changes behavior.
- If no meaningful complication is possible, explicitly say so.
- Important: the question must always be phrased as:
  **"What is the role of the command <base-command> in the session <session>?"**,
  where <base-command> is just the program name (e.g., awk, grep, sort) and <session> is the session containing the base-command.
- Provide a correct, concise, and specific answer that explains how the base command works in the final session.
```

```
- Explanations must be specific to the session, avoiding generalities or vague statements. The reasoning must connect exactly to the commands in the modified session.
- In the answer avoid repeating any part of the original command literally; use generic placeholders instead when necessary.

Steps:
1. Check if the question can be complicated.
2. If yes, rewrite it with a new flag.
3. Provide a short answer.
4. If no, state it cannot be complicated.

Output format (strict, nothing else):
<complicatable>True/False</complicatable>
<question>[New complicated question or "none"]</question>
<answer>[Concise answer or "none"]</answer>

Example:
Original question: "What is the role of the command ‘head‘ in the session ‘head -n 5‘?"

Harder version:
<complicatable>True</complicatable>
<question>What is the role of the command ‘head‘ in the session ‘head -n 5 -v‘?</question>
<answer>Displays the first five lines of a file and shows the file name before the output.</answer>
"""
    return SYSTEM_PROMPT_FLAGS
```

Listing A.1: Flag Evolution

```
def returnPromptCombination(question):
    SYSTEM_PROMPT_COMBINATIONS= """
You are a Teacher applying the Evol-Instruct framework to Bash questions.
Your strict task: complicate Bash questions by inserting exactly ONE new Bash command.

Input: a Bash session and a question about one command in that session.
Goal: rewrite the question into a harder version by inserting ONE additional Bash command into the session.

Constraints:
- Insert exactly ONE new Bash command.
- Preferably, the new command should influence the behavior or input/output of the base command in the session.
```

```
- Adding a non-influencing command (just filler, e.g., pwd, echo, ls) should only be done if no meaningful alteration is possible.
- Prioritize the use of '|' than && or ;.
- The final Bash session must remain syntactically valid, coherent and runnable.
- If the new command requires a file, input, or string, use a generic placeholder (e.g., <file>, <directory>, <pattern>).
- Provide a concise, precise, and specific explanation of the the original command's role in the context of the modified session.
- Respect causality: the order of commands make sense.
- Explanations must be specific to the session, avoiding generalities or vague statements. The reasoning must connect exactly to the commands in the modified session.
- In the answer avoid repeating any part of the original command literally; use generic placeholders instead when necessary.
- Stay strictly within the Bash domain.
- Important: the question must always be phrased as:
  **"What is the role of the command <base-command> in the session <session>?"**,
  where <base-command> is just the program name (e.g., awk, grep, sort) and <session> is the session containing the base-command.
- Always output an <influence_flag> for the added command:
- <influence_flag>True</influence_flag> -> The inserted command affects the input, output, or processing of the base command in the session. The base command's behavior is altered due to the new command.
- <influence_flag>False</influence_flag> -> The inserted command does not affect the base command's behavior. It is just a filler (e.g., pwd, echo, ls) or runs independently of the base command.
    Rules for deciding:
    1. If the new command changes what the base command receives as input or how it produces output -> True
    2. If the new command runs independently or only provides unrelated context -> False

Steps:
1. Add exactly one new Bash command to the session.
2. Rewrite the question so it now refers to the modified session, while asking only about the base command.
3. Give a concise answer that describes how the base command processes its input/output in the new context.
4. Use the <influence_flag> as specified above.

Output format (STRICT, no extra text):
<question>[Rewritten harder question]</question>
<answer>[Concise answer]</answer>
<influence_flag>True/False</influence_flag>
```

```
41
42 Example:
43 Original question: "What does the command `awk` do in the session
       `awk '{print $1}' <file>`?"
44
45 Harder version:
46 <question>What is the role of the command `awk` in the session `
       sort <file> | awk '{print $1}'`?</question>
47 <answer>It extracts the first field from each line of the sorted
       input.</answer>
48 <influence_flag>True</influence_flag>
49 """
50     return SYSTEM_PROMPT_COMBINATIONS
```

Listing A.2: Combinational Evolution

```
1
2 def returnPromptSwitch(question):
3     SYSTEM_PROMPT_SWITCH= """
4 You are a Teacher applying the Evol-Instruct framework to Bash
       questions.
5 Your strict task: complicate the questions by modifying the Bash
       session so that it still achieves the same functional goal, but
        uses at least ONE different Bash command (possibly more).
6
7 Input: a question containing a Bash session.
8 Goal: rewrite the question by replacing or rewriting commands so
       that the session is functionally similar but expressed
       differently.
9
10 Constraints:
11 - Stay strictly in the Bash domain.
12 - Change at least ONE Bash command (but you can also change more).
13 - The rewritten session must preserve the original functional goal
       as much as possible.
14   (e.g., use `cut` instead of `awk`, or `sed` instead of `grep`).
15 - If the new command requires a file, input, or string, use a
       generic placeholder (e.g., <file>, <directory>, <pattern>).
16 - Slight differences in behavior are acceptable if exact
       equivalence is impossible.
17 - The final session must remain syntactically valid and runnable.
18 - Important: the question must always be phrased as:
19   "What is the role of the command <base-command> in the session <
       session>?"
20   where <base-command> is just the program name (e.g., `awk`, `
       grep`, `sort`) and <session> is the session containing the base
       -command.
```

```
- Provide a concise, precise, and specific explanation of the role of the original command in the context of the modified session.
- Respect causality: the order of commands make sense.
- Explanations must be specific to the session, avoiding generalities or vague statements. The reasoning must connect exactly to the commands in the modified session.
- In the answer avoid repeating any part of the original command literally; use generic placeholders instead when necessary.

Steps:
1. Replace at least ONE Bash command with another that keeps the same functional goal.
2. Rewrite the question so it refers to the modified session, still asking only about the base command.
3. Provide a concise answer.

Output format (STRICT, no extra text):
<question>[Rewritten harder question]</question>
<answer>[Concise answer]</answer>

Example:
Original question: "What does the command `awk` do in the session `awk '{print $1}' <file>`?"

Harder version:
<question>What is the role of the command `cut` do in the session `cut -d' ' -f1 <file>`?</question>
<answer>Extracts the first whitespace-delimited field from each line.</answer>
"""
    return SYSTEM_PROMPT_SWITCH
```

Listing A.3: Switch Evolution

A.2 Task 2

Section 3.3.2 describes in detail the methodology adopted in the evolution process. This section presents the prompts used during the evolution phase.

```
def distractor_prompt():
    system_prompt = """
You are an expert Bash instructor using an Evol-Instruct methodology.
Generate a new Bash session that is **syntactically similar** to a given one but achieves a **different conceptual goal**.


```

```
7  Definition: Conceptual Goal
8  Two Bash sessions share the same conceptual goal if, when run in
       similar contexts, they perform the **same type of operation**
       on the **same kind of targets**.
9  This means they both aim to achieve the same high-level purpose,
       such as listing files, filtering data, counting elements, or
       removing items, even if:
10 - they use different commands or options,
11 - they operate on different specific inputs or produce different
       concrete outputs,
12 - or their internal steps and formatting differ.
13
14 For this *distractor* task, you must produce a session that has a
       **different** conceptual goal (i.e., differs in task kind and/
       or target kind) while keeping maximum syntactic similarity.
15
16 Requirements
17 - Stay strictly within the Bash domain.
18 - Produce a **different conceptual goal** than the original, the
       new session must perform a **different type of operation** or
       act on a **different kind of target**.
19   - Simply changing inputs, filenames, or argument values is **not
         enough**.
20 - Be **as syntactically similar as possible**:
21   - Reuse the same general structure, pipelines, redirections, and
         shell constructs.
22   - Prefer **minimal edits**: swap a command or flag, replace one
       pipeline stage, or adjust a predicate while keeping the overall
         form.
23 - Replace **at least one** command or flag (do not copy verbatim).
24 - You **must** rename or modify any **non-essential identifiers**,
         such as variables, string literals, filenames, patterns, or
       other arguments, using random but valid alternatives, provided
       this does **not affect** the new conceptual goal or break Bash
       syntax.
25   - These renamings are allowed for natural variation but **do not
         by themselves satisfy** the requirement of changing the
       conceptual goal.
26 - The result must be **valid, runnable Bash**.
27 - If no meaningful transformation is possible, explicitly say so.
28
29
30 Description Guidelines
31 - One sentence stating the **new session's conceptual goal** (what
         it does).
32 - Do not mention literal values, arguments, paths, symbols, or how
         it differs from the original.
33
34 Examples of good descriptions:
```

```
- "Prints a sequence of letters without line breaks."
- "Counts the number of items in a sequence."
- "Removes invalid symbolic links from a directory."

Examples of bad descriptions:
- "Prints numbers 1 to 10 instead of letters."
- "Removes broken links rather than listing them."
- "Formats the output differently."


Steps
1) Identify the original session's conceptual goal (operation type + target kind).
2) Create a new session that keeps similar syntax but **changes the conceptual goal** (different operation type and/or different target kind).
3) Rename or modify any non-essential identifiers using random but valid alternatives.
4) Output results in the strict format below.


Output Format (STRICT, nothing else)

<newSession>[New session or "none"]</newSession>
<description>[Brief general conceptual goal of the new session or "none"]</description>


Examples:

Example A
Input session: grep "error" /var/log/syslog
Output:
<newSession>wc -l /var/log/syslog</newSession>
<description>Counts the number of lines in a log file.</description>

Example B
Input session: sort data.txt | uniq -c
Output:
<newSession>sort data.txt | uniq</newSession>
<description>Outputs unique items from a sorted list.</description>

Example C (no meaningful distractor possible)
Input session: echo "Hello World"
Output:
<newSession>none</newSession>
<description>none</description>
```

```
"""
    return system_prompt
```

Listing A.4: Distractor Evolution

```
def obfuscation_prompt():
    system_prompt = """
You are a Teacher following an Evol-Instruct approach.
Your goal is to rewrite a given Bash session so that it
   accomplishes the **same conceptual goal**, meaning it achieves
   the same kind of task, but does so using **different Bash
   commands, options, or structures**.


Definition: Conceptual Goal
Two Bash sessions share the same conceptual goal if, when run in
   similar contexts, they perform the **same type of operation**
   on the **same kind of targets**.
This means they both aim to achieve the same high-level purpose,
   such as listing files, filtering data, counting elements, or
   removing items, even if:
- they use different commands or options.
- they operate on different specific inputs or produce different
   concrete outputs.
- or their internal steps and formatting differ.

What must remain the same:
- The **kind of task** being performed (e.g., listing, counting,
   deleting, filtering).
- The **type of target** being acted on (e.g., files, directories,
    links).

What can differ:
- The specific commands, flags, or structures used.
- Input data and resulting outputs.
- Output format and ordering (when non-essential).
- Intermediate steps or methods.


Rules for Session Generation
- Stay strictly within the Bash domain.
- Create a **new version** of the given session that achieves the
   same conceptual goal using different Bash commands, options, or
    structures.
- If a fully new version is not possible, **replace as many
   commands, options, or structures as possible** (avoid copying
   verbatim).
- The new session must be syntactically valid and runnable in Bash
   .
```

```
- You **must** rename or modify any **non-essential identifiers**, such as variables, string literals, filenames, patterns, or other arguments, using random but valid alternatives, provided this does **not change** the conceptual goal or break the command's logic.
- These renamings are meant to add variety and realism but **cannot be the only change**.
- Do **not** evade the intent of the task by merely altering specific inputs or targets (e.g., swapping filenames, narrowing to a subset, or redirecting to a different file) while leaving the commands and their effects unchanged.


Rules for Description Generation
- The description must be **brief (one sentence)** and explain the **general conceptual goal** of the new session.
- Focus only on **what the new session does**, not how it differs from the original.
- Avoid literal values, filenames, or arguments.
- Keep it clear and abstract.

Examples of good descriptions:
- "Prints a sequence of letters without line breaks."
- "Counts the number of items in a sequence."
- "Removes invalid symbolic links from a directory."

Examples of bad descriptions:
- "Prints numbers 1 to 10 instead of letters."
- "Removes broken links rather than listing them."
- "Formats the output differently."


Steps
1. Understand the conceptual goal of the given session.
2. Generate a new session that:
   - Achieves the **same conceptual goal**.
   - Uses different Bash commands, options, or structures.
3. Rename or modify any non-essential identifiers using random but valid alternatives.
4. Explicitly state whether evolution was possible, then provide the new session and its brief conceptual description.


Output Format (strict, nothing else)
<canEvolve>True/False</canEvolve>
<newSession>[New session or "none"]</newSession>
<description>[Brief conceptual goal or "none"]</description>


```

```
Example 1
Input session: find /target/dir -type l ! -exec test -e {} \\; -exec rm {} \\;

Output:
<canEvolve>True</canEvolve>
<newSession>find /target/dir -xtype l -delete</newSession>
<description>Removes invalid symbolic links from a directory.</description>


Example 2
Input session: du -sh * | sort -hr | head -n 5

Output:
<canEvolve>True</canEvolve>
<newSession>ls -lhS | head -n 5</newSession>
<description>Displays the largest items in the current directory.</description>
"""
    return system_prompt
```

Listing A.5: Obfuscation Evolution

```

    system_prompt = """
You are a Teacher following an Evol-Instruct approach.
Your goal is to rewrite a given Bash session so that it has a **similar conceptual goal**, meaning it performs the same kind of task, but does so by **adding meaningful Bash commands or options** that extend or enhance the original workflow.


Definition: Similar Conceptual Goal
Two Bash sessions have a similar conceptual goal if they perform the **same type of operation** on the **same kind of targets**, but the new session introduces **extra, logically related steps** that expand or enrich the task (e.g., saving output, validating results, filtering, or adding post-processing).
The extended session should keep the same overall intent, not replace or alter it.


Rules for Session Generation
- Stay strictly within the Bash domain.
- Create a **longer version** of the given session that extends it while maintaining the same type of task and targets.
```

```
- Add related commands or options anywhere (beginning, middle, or end) that meaningfully complement the task, e.g., filtering, formatting, saving results, validating outcomes, logging, or summarizing.
- Avoid trivial additions:
  - Do **not** just echo, print completion messages, or add unrelated commands.
  - Do **not only** change inputs, filenames, or argument values, such changes alone are not enough.
- You may replace one command or option **only if this enables** a meaningful extension.
- **Rename or modify any non-essential identifiers**, including variables, string literals, filenames, search patterns, addresses, or other arguments, using **random but valid alternatives**, provided this does **not alter the conceptual goal** or break the command's logic.
- The new session must be syntactically valid and runnable in Bash.


Rules for Description Generation
- The description must be **brief (one sentence)** and explain the **general conceptual goal** of the new session.
- Focus only on **what the new session does overall**, not how it differs from the original.
- Avoid literal values, arguments, or filenames.
- Keep it clear, concise, and abstract.

Examples of good descriptions:
- "Prints a sequence of letters without line breaks."
- "Counts the number of items in a sequence."
- "Removes invalid symbolic links from a directory."

Examples of bad descriptions:
- "Prints numbers 1 to 10 instead of letters."
- "Removes broken links rather than listing them."
- "Formats the output differently."


Steps
1. Identify the conceptual goal of the given session (the task type and its targets).
2. Design a longer session that:
   - Preserves this conceptual goal.
   - Adds meaningful, logically related commands or options.
   - Avoids trivial or purely cosmetic additions.
3. Rename or modify any non-essential identifiers using random but valid alternatives.
```

```
4. Explicitly state whether elongation was possible, then provide the new session and its brief conceptual description.


Bad Example (too trivial)
Input: ls
Output: ls && echo "done"
-> Invalid because the added command does not extend the conceptual goal.


Output Format (strict, nothing else)
<newSession>[New session or "none"]</newSession>
<description>[Brief description of conceptual goal of the new session or "none"]</description>


Example 1
Input session: find /target/dir -type l ! -exec test -e {} \\; -exec rm {} \\;

Output:
<newSession>find /target/dir -type l ! -exec test -e {} \\; -exec rm {} \\; -print</newSession>
<description>Removes invalid symbolic links and displays their names.</description>

Example 2
Input session: du -sh * | sort -hr | head -n 5

Output:
<newSession>du -sh * | sort -hr | tee sorted_sizes.txt | head -n 5</newSession>
<description>Displays and saves the largest items in the current directory.</description>
"""
    return system_prompt
```

Listing A.6: Elongation Evolution

A.3 Task 3

Section 3.4.2 describes in detail the methodology adopted in the evolution process. This section presents the prompts used during the evolution phase.

```
def constraint_prompt():
    system_prompt = """
```

```
You are an expert Bash instructor following the Evol-Instruct methodology.
Your goal is to generate a new slightly more complex version of the given instruction by introducing new constraints or additional requirements.
The final instruction must still focus on **practical Bash coding ability**, how well a user can implement the required behavior through Bash scripting.


General Principles
- Stay strictly within the Bash domain.
- The new instruction must remain solvable with standard Bash commands.
- Add new **constraints or conditions** that make the task more specific and require additional logical reasoning without changing its overall structure.
- The complexity increase must be limited to **one or two additional logical checks or requirements**, not to the introduction of multi-step procedures.
- The goal is to refine the original task by adding a small number of conditional rules, validations, or formatting criteria that are more complex to implement.
- Avoid trivial modifications such as simple file-type filters or single-flag additions unless combined with another meaningful constraint.
- The new constraint should require the use of **basic control logic** (e.g., `if`, `test`, or comparisons) or simple validation (e.g., existence, emptiness, error message).
- Examples of acceptable constraints:
  - Perform the operation only if the target file exists or is not empty.
  - Print an error if the input argument is missing or invalid.
  - Skip items matching a specific condition (e.g., file size, extension, timestamp).
  - Enforce a custom success or failure message in the output.
- Add approximately **10-15 additional words** compared to the original instruction.
- Preserve the original conceptual goal, the task must remain ** single-phase and atomic**, not decomposed into multiple steps or workflows.

Input Handling
- If the new instruction would realistically require an input, provide one.
- The input is optional but recommended when it adds realism or clarity.
- The input must be **relevant and coherent** with the modified task.
```

```
- If no input is necessary, return `""none""`.


Answer Guidelines
- The answer must include valid and executable **Bash code** that satisfies all requirements in the new instruction.
- Code must be syntactically correct and runnable in a standard Bash shell.
- Inline comments are not allowed.
- Prefer clear, readable code over compact or obfuscated code.


Steps
1. Identify the **main goal and requirements** of the original instruction.
2. Introduce **new constraints or conditions** that make the instruction slightly more detailed or challenging, adding about 10 additional words.
3. Optionally define an **input**, ensuring it is realistic and coherent.
4. Produce the **complete Bash solution** that fulfills the new instruction.


Output Format (STRICT - nothing else)
<newInstruction>[New instruction]</newInstruction>
<input>[Input or "none"]</input>
<>[Bash code solving the new instruction]</answer>


Example
<newInstruction>Write a Bash script that counts all .txt files in a directory, excluding hidden files and sorting results alphabetically by filename.</newInstruction>
<input>/home/user/documents</input>
<answer>
#!/bin/bash
# List all non-hidden .txt files in the given directory and count them
count=$(find "$1" -maxdepth 1 -type f -name "*.txt" ! -name ".*" | sort | wc -l)
echo "Number of text files: $count"
</answer>
"""
    return system_prompt
```

Listing A.7: Constraint Evolution

```

```

```
def step_prompt(instruction):
    system_prompt = """
You are an expert Bash instructor following the Evol-Instruct methodology.
Your goal is to create a **new instruction** that keeps the original Bash task **exactly intact**
and then adds **one new, contextually related operation** that requires additional reasoning.
The first part must stay semantically identical to the original, do not modify, refine, or constrain it.

Core Principles
- Stay strictly within the **Bash scripting domain**.
- The **main operation must remain untouched**:
  - Do NOT add constraints, new details, or reformulations.
  - The meaning, scope, and logic of the first part must stay identical.
- Introduce **exactly one additional operation**, placed after the original one.
- The new step must:
  - Depend on or logically relate to the output/result of the first one.
  - Introduce genuine reasoning (analysis, validation, filtering, comparison, etc.).
  - Be **specific** to the original context, not a general add-on.
- The new part must create a **chain of reasoning** between two correlated actions.


Allowed and Forbidden Additions
Allowed examples (context-specific):
- After filtering lines -> analyze the result (count, categorize, compare).
- After compressing files -> verify integrity or check size reduction.
- After renaming files -> detect duplicates or conflicts.
- After listing processes -> identify which exceed a threshold.
- After extracting data -> compute or summarize something from it.

Forbidden examples (too generic):
- Save or print the result.
- Sort, count, or display data without a context-specific reason.
- Rephrase the original task by adding mappings, thresholds, or filters.
- Add arbitrary loops, user input prompts, or stylistic embellishments.


Structural Requirements
```

```
- The evolved instruction must contain **two explicit clauses**:
  1. The first clause reproduces the original instruction *verbatim* or with minimal stylistic adjustment.
  2. The second clause introduces the new reasoning step (linked with "and then..." or equivalent).
- The relationship between the two must be **logical and causal**, the second operates *on* the result of the first.

Input Handling
- If the new step logically requires input, provide one.
- Input must be **coherent** with both parts.
- If not required, return `"none"`.


Answer Guidelines
- The answer must include valid and executable **Bash code** that satisfies all requirements in the new instruction.
- Code must be syntactically correct and runnable in a standard Bash shell.
- Inline comments are not allowed.
- Prefer clear, readable code over compact or obfuscated code.


Output Format (STRICT - nothing else)
<newInstruction>[Evolved instruction]</newInstruction>
<input>[Input or "none"]</input>
<answer>[Bash code solving the new instruction]</answer>


Example

Original: "Write a Bash script that lists all .log files in a directory."

Evolved:
<newInstruction>Write a Bash script that lists all .log files in a directory and then reports which of them exceed 1 MB in size.</newInstruction>
<input>/var/log</input>
<answer>
#!/bin/bash
# List all .log files
for f in "$1"/*.log; do
  # Check file size in bytes
  size=$(stat -c %s "$f")
  # Report if size exceeds 1 MB
  if (( size > 1048576 )); then
    echo "$f exceeds 1 MB"
  fi
```

```
done
</answer>
"""
    return system_prompt
```

Listing A.8: Step Evolution

```
def switch_prompt():
    system_prompt = """
You are an expert Bash instructor following the Evol-Instruct methodology.
You will receive an instruction along with its code solution.
Your goal is to produce a **new instruction** that has a **different functional goal and domain** from the original one, while still **reusing at least one Bash command** that appeared in the original solution.
The new instruction must remain practical, realistic, and solvable with Bash scripting.


General Principles
- Stay strictly within the Bash domain.
- The new instruction must have a **different objective and belong to a different functional domain** than the original one.
- The **new code solution must reuse at least one command** from the original solution.
    - A "command" refers to a Bash utility (e.g., `grep`, `awk`, `find`, `ls`, `cat`, `cut`, `sort`), not flags or parameters.
- The reused command must be employed in a **new, contextually coherent way**, consistent with its real-world function.
- The new instruction should maintain a **similar difficulty and code complexity** as the original.
- Both the instruction and the code must be runnable, coherent, and safe (no destructive commands such as `rm -rf`).


Steps
1. Examine the original instruction and its Bash solution to understand the goal and domain (e.g., file management, text processing, system information).
2. Identify all the Bash commands used in the original code.
3. Select one or more **key commands** that can be meaningfully reused in another domain.
4. Based on the selected command(s), **design a new instruction** in a clearly different domain of application.
   - The instruction must arise from the functionality of the reused command(s), but serve a new purpose.
   - The task must be coherent and realistic within Bash scripting.
```

```
5. Write a valid, runnable Bash solution that uses the selected command(s) in the new context.


Input Handling
- Provide an input if it is necessary for the new task (e.g., a file, directory, or process name).
- If no input is required, return "none".
- The input must be relevant and realistic for the new instruction.


Answer Guidelines
- The answer must include valid and executable **Bash code** that satisfies all requirements in the new instruction.
- Code must be syntactically correct and runnable in a standard Bash shell.
- Inline comments are not allowed.
- Prefer clear, readable code over compact or obfuscated code.


Output Format (STRICT - nothing else)
<newInstruction>[New instruction]</newInstruction>
<input>[Input or "none"]</input>
<answer>[Bash code solving the new instruction]</answer>

Example:
Original instruction: Write a Bash script that lists all `.txt` files in a directory.
Original solution:
ls *.txt

<newInstruction>Write a Bash script that lists all active system processes sorted by their CPU usage.</newInstruction>
<input>none</input>
<answer>
#!/bin/bash
# List active processes and sort by CPU usage
ps -eo pid,comm,%cpu --sort=-%cpu | head -n 10
</answer>
"""
```

Listing A.9: Switch Prompt

Bibliography

[1] M. Chen et al. «Evaluating Large Language Models Trained on Code». In: *arXiv preprint arXiv:2107.03374* (2021). URL: https://arxiv.org/abs/2107.03374 (cit. on pp. 1, 8–10).

[2] Ziyang Luo et al. *WizardCoder: Empowering Code Large Language Models with Evol-Instruct*. 2025. arXiv: 2306.08568 [cs.CL]. URL: https://arxiv.org/abs/2306.08568 (cit. on pp. 3, 10).

[3] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. *G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment*. 2023. arXiv: 2303.16634 [cs.CL]. URL: https://arxiv.org/abs/2303.16634 (cit. on p. 4).

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. «Attention Is All You Need». In: *Proc. 31st Conf. Neural Information Processing Systems (NeurIPS)*. Long Beach, CA, USA, Dec. 2017. URL: https://arxiv.org/abs/1706.03762 (cit. on p. 6).

[5] Matteo Boffa, Idilio Drago, Marco Mellia, Luca Vassio, Danilo Giordano, Rodolfo Valentim, and Zied Ben Houidi. «LogPrécis: Unleashing language models for automated malicious log analysis: Précis: A concise summary of essential points, statements, or facts». In: *Computers & Security* 141 (2024), p. 103805 (cit. on p. 8).

[6] OpenAI. *GPT-4 Technical Report*. Tech. rep. San Francisco, CA: OpenAI, Mar. 2023. URL: https://cdn.openai.com/papers/gpt-4.pdf (cit. on p. 9).

[7] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. «Measuring Massive Multitask Language Understanding». In: *arXiv preprint arXiv:2009.03300* (2020). URL: https://arxiv.org/abs/2009.03300 (cit. on p. 9).

[8] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. 2023. arXiv: 2212.10560 [cs.CL]. URL: https://arxiv.org/abs/2212.10560 (cit. on pp. 9, 10).

[9] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. *WizardLM: Empowering large pre-trained language models to follow complex instructions*. 2025. arXiv: 2304.12244 [cs.CL]. URL: https://arxiv.org/abs/2304.12244 (cit. on pp. 9, 10, 55).

[10] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. *Alpaca: A Strong, Replicable Instruction-Following Model*. https://crfm.stanford.edu/2023/03/13/alpaca.html. Stanford Center for Research on Foundation Models (CRFM). Mar. 2023 (cit. on p. 10).

[11] Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. *Orca: Progressive Learning from Complex Explanation Traces of GPT-4*. 2023. arXiv: 2306.02707 [cs.CL]. URL: https://arxiv.org/abs/2306.02707 (cit. on p. 10).

[12] Dawei Wang, Geng Zhou, Xianglong Li, Yu Bai, Li Chen, Ting Qin, Jian Sun, and Dan Li. *The Digital Cybersecurity Expert: How Far Have We Come?* 2025. arXiv: 2504.11783 [cs.CR]. URL: https://arxiv.org/abs/2504.11783 (cit. on p. 10).

[13] Niloofar Mireshghallah, Hyunwoo Kim, Xuhui Zhou, Yulia Tsvetkov, Maarten Sap, Reza Shokri, and Yejin Choi. *Can LLMs Keep a Secret? Testing Privacy Implications of Language Models via Contextual Integrity Theory*. 2024. arXiv: 2310.17884 [cs.AI]. URL: https://arxiv.org/abs/2310.17884 (cit. on p. 11).

[14] The Open Group. *The Open Group Base Specifications Issue 8, 2024 Edition (IEEE Std 1003.1-2024)*. https://pubs.opengroup.org/onlinepubs/9799919799/utilities/contents.html. 2024 (cit. on p. 15).

[15] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. arXiv: 1908.10084 [cs.CL]. URL: https://arxiv.org/abs/1908.10084 (cit. on pp. 20, 23).

[16] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. *NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System*. 2018. arXiv: 1802.08979 [cs.CL]. URL: https://arxiv.org/abs/1802.08979 (cit. on pp. 23, 42).

[17] S. Chaudhary. *Code Alpaca: An Instruction-Following LLaMA Model for Code Generation*. `https://github.com/sahil280114/codealpaca`. GitHub repository. 2023 (cit. on p. 29).