ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica — Scienza e Ingegneria

Master Degree in Computer Science

Large Language Models for Solver Selection: a Preliminary Study

Supervisor:
Professor
Roberto Amadini

Co-Supervisor:
PhD Student
Simone Gazza

Co-Supervisor:
Professor
Guido Tack

Presented by:
Vittorio Rossetto

March Session 2026

Academic Year 2025/2026

This is the DEDICATION:
you can write whatever you want here,
or nothing at all ...

Introduction

This thesis presents a systematic investigation of the use of Large Language Models (LLMs) [15] within the domain of Constraint Programming (CP) [2], more specifically on the task of algorithm selection [12], referred to as "solver selection" in the case of NP-complete problems. The objective is to assess whether general-purpose language models can effectively support or enhance decision processes that have traditionally relied on supervised learning.

Algorithm selection, aims to identify the most suitable algorithm for solving a specific problem prior to execution. this issue has been prevailing in many domains, as no single algorithm can perform best on all problem instances. Traditional algorithm selection and portfolio construction methods typically treat the problem as a classification or regression task [1].

LLMs are neural network architectures trained on large-scale text corpora to model statistical regularities in natural language. They represent the outcome of sustained advances in natural language processing and machine learning, and have recently demonstrated strong performance across a wide range of reasoning and generation tasks. Despite their broad adoption in general-purpose applications, their role in specialized technical domains such as CP, and specifically in automated solver selection, remains unexplored.

CP is a declarative paradigm for solving combinatorial problems, particularly those arising in planning, scheduling, and resource allocation [2]. Problems are modeled in terms of variables, domains, and constraints, and are processed by solvers, i.e., software systems that search for assignments satisfying the constraints and, in optimization settings, improving a given objective function. Different solvers rely on distinct underlying technologies and heuristics, leading to performance variability across problem classes. Selecting an appropriate solver for a given instance is therefore a central challenge [33].

The research presented in this thesis investigates the problem of algorithm selection, by using LLMs as decision-making components with a portfolio of solvers. This idea is inspired by [14], and takes inspiration from portfolio-based approaches, where multiple solvers are available and a central strategy determines which ones to execute for a given instance. Traditional portfolio methods rely on complex, supervised learning algorithms. In contrast, this work explores whether general-purpose LLMs, given suitable contextual information, can approximate or surpass these approaches without task-specific retraining.

After selecting a benchmark set of CP problems, specifically those used in the 2025 MiniZinc

Challenge [32, 31], we identified a collection of general-purpose LLMs for evaluation. We then conducted iterative experiments to assess whether these models could outperform existing single solvers, systematically testing different prompt configurations and progressively enriching the contextual information provided.

Initial results highlighted the limitations of LLMs for this task, yielding only marginal improvements over traditional solvers. To address these limitations, we incorporated structured problem representations using a feature extraction tool, `mzn2feat` [39], which produced the highest observed performance, even though by a minimal margin. Building on these insights, we also developed a tool to translate problem code into natural language descriptions, fzn2nl [52], on which we conducted further testing.

The structure of the thesis is as follows. Chapter 1 provides the technical background required for the remainder of the work. Chapter 2 describes the experimental methodology, evaluation metrics, and design choices underlying the study. Chapter 3 presents the core experimental results, beginning with baseline tests based on minimal problem representations, and progressively enriching the context with textual descriptions, structured features extracted from problem instances, and finally the implementation of temperature tuning. Chapter 4 introduces a complementary tool developed during this research, `fzn2nl`, which translates FlatZinc models into natural language descriptions, and finally we tested LLM performance when operating directly on such generated representations.

Contents

Chapter 1

Technical Background

In this chapter, we give the reader an initial knowledge base on all the theoretical concepts, and all the components that had been utilized in the development of this research.

In the first section we will provide a description of CP itself, starting from a general definition, delving deeper in the description of CP solvers, and then we are gonna provide a brief description of MiniZinc, a CP modelling language, in which all the tested problems are coded.

In the second section we will provide an high level description of what large language models (LLMs) are and how they work, starting from the implementation of inference and decoding, passing to prompting techniques and prompt engineering, finally explaining the use of Application Programming Interfaces (APIs) as a communication mean.

1.1 Constraint Programming

CP is a paradigm for declarative description and effective solving of large, particularly combinatorial, problems especially in areas of planning and scheduling [2].

A constraint can be thought of intuitively as a formalization of dependencies in physical worlds and their mathematical abstractions. A constraint is a logical relation among several unknowns (or variables), each taking a value in a given domain. The constraint thus restricts the possible values that variables can take; it represents partial information about the variables of interest. Constraints can also be heterogeneous, so they can bind unknowns from different domains, for example a length (number) with a word (string). The important feature of constraints is their declarative manner, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship [2].

Constraints arise naturally in most areas of human endeavor. They are the natural medium of expression for formalizing regularities that underlie the computational and (natural or designed) physical worlds and their mathematical abstractions.

CP is the study of computational systems based on constraints. The idea of CP is to solve problems by stating constraints (requirements) about the problem area and, consequently, finding a solution satisfying all the constraints.

1.1.1 Solvers

Throughout this thesis, the term solver denotes a system that, given a formal problem specification, computes assignments satisfying a set of constraints and, when required, optimizes an objective function.

A "Constraint Satisfaction Problem" (CSP) can be formalized as a tuple

$$\langle X, D, C \rangle,$$

where X is a finite set of decision variables, $D(x)$ denotes the domain associated with each variable $x \in X$, and C is a set of constraints defined over subsets of X. A solution is an assignment

$$s : X \to \bigcup_{x \in X} D(x)$$

such that $s(x) \in D(x)$ for all $x \in X$ and all constraints in C are satisfied [5].

The objective in a CSP is to determine whether at least one feasible assignment exists, or to enumerate all feasible assignments.

A "Constraint Optimization Problem" (COP) extends the CSP by introducing an objective function. Formally, a COP can be defined as

$$\langle X, D, C, f \rangle,$$

where $f : S \to \mathbb{R}$ is an objective function defined over the set S of feasible assignments. The goal is to compute an optimal solution

$$s^* = \arg\min_{s \in S} f(s) \quad \text{or} \quad s^* = \arg\max_{s \in S} f(s),$$

depending on whether the problem is a minimization or maximization task.

Different solver families address CSPs and COPs using distinct mathematical abstractions and algorithmic techniques.

Constraint Programming (CP) solvers

Constraint Programming solvers operate directly on variables with finite or structured domains and heterogeneous constraints. Their fundamental mechanisms are constraint propagation, which removes inconsistent values from variable domains, and systematic search, typically implemented via backtracking guided by variable- and value-ordering heuristics. CP solvers natively support high-level global constraints equipped with specialized filtering algorithms that exploit combinatorial structure [61]. A representative example is Gecode [40].

Linear and Mixed-Integer Linear Programming solvers

Linear Programming (LP) and Mixed-Integer Linear Programming (MILP) solvers address optimization problems in which the objective function and the constraints are linear. A standard LP formulation is

$$\min\{c^\top x \mid Ax \leq b,\ x \in \mathbb{R}^n\},$$

where $x \in \mathbb{R}^n$ is a vector of continuous decision variables, $c \in \mathbb{R}^n$ defines the linear objective function, and $Ax \leq b$ represents a system of linear inequalities.

LP solvers assume continuous domains and rely on algorithms such as the simplex method and interior-point methods [6]. They do not natively support discrete decision variables or high-level combinatorial constraints; such constructs must be reformulated as linear expressions over continuous variables.

Mixed-Integer Linear Programming extends this framework by restricting some or all variables to integer domains, i.e., $x_i \in \mathbb{Z}$ for selected indices. MILP solvers typically combine LP relaxations with tree-based search procedures such as branch-and-bound and branch-and-cut [8, 9]. This hybrid approach enables the treatment of discrete decisions within a linear modeling framework, although global or non-linear constraints must generally be encoded through auxiliary variables and additional linear constraints. Examples of solvers in this family include HiGHS [7] and Gurobi [60].

Portfolio Solvers

Solving combinatorial search problems is hard, and there exist nowadays plenty of techniques and constraint solvers for performing this task. It has become clear that different solvers are better when solving different problem instances, even within the same problem class. It has also been shown that a single, arbitrarily efficient solver can be significantly outperformed by using a portfolio of possibly on-average slower solvers [10].

Algorithm portfolios [11] can be seen as instances of the more general Algorithm Selection problem [12] where, as reported in [13], the algorithm selection is performed case-by-case for each problem to solve. Within the context of constraint solving, a portfolio approach enables to combine a number $m > 1$ of different constituent solvers $s_1, \ldots, s_m$ in order to create a globally better constraint solver, dubbed a portfolio solver. When a new, unseen problem p comes, the portfolio solver tries to predict the best constituent solver(s) $s_{i_1}, \ldots, s_{i_k}$ (with $1 \leq i_j \leq m$ for $j = 1, \ldots, k$) for solving p and then runs them on p. Properly selecting and scheduling the solvers is a crucial step for the performance of a portfolio solver, and it is usually performed by exploiting Machine Learning techniques based on features extracted from the problem p to solve.

In the course of this research thesis, starting from the idea behind portfolio solvers, we will try to evaluate the performances of a pre-trained LLM to understand if this could be a viable solution to the algorithm selection problem [12], avoiding the cost of training for portfolio solvers, following the new paradigm proposed in [14].

1.1.2 MiniZinc

MiniZinc is a simple but expressive CP modelling language which is suitable for modelling problems for a range of solvers and a reasonable compromise between many design possibilities.

It was born from the necessity of a standard modelling language for constraint programming problems, it is also making solver benchmarking simpler [4].

For these reasons all of the problems employed for testing in the course of this research paper were written using MiniZinc.

A MiniZinc Example

A MiniZinc problem specification has two parts: (a) the model, which describes the structure of a class of problems; and (b) the data, which specifies one particular problem within this class. The pairing of a model with a particular data set is a model instance (sometimes abbreviated to instance).

The model and data may be in separate files. Data files can only contain assignments to parameters declared in the model. A user specifies data files on the command line, rather than naming them in the model file, so that the model file is not tied to any particular data file [4].

Each MiniZinc model is a sequence of items, which may appear in any order. Consider the MiniZinc model and example data for a restricted job shop scheduling problem in Figure 1.1 and Figure 1.2.

Line 0 is a comment, introduced by the "%" character.

Lines 1-5 are "variable declaration items". Line 1 declares `size` to be an integer parameter, i.e. a variable that is fixed in the model. Line 20 (in the data file) is an "assignment item" that defines the value of `size` for this instance. Variable declaration items can include assignments, as in line 3. Line 4 declares `s` to be a 2D array of "decision variables". Line 5 is an integer variable with a restricted range. Decision variables are distinguished by the `var` prefix.

Lines 7-8 show a user-defined "predicate item", `no_overlap`, which constrains two tasks given by start time and duration so that they do not overlap in time.

Lines 10-17 show a "constraint item". It uses the built-in `forall` to loop over each job, and ensure that: (line 12) the tasks are in order; (line 13) they finish before end; and (lines 14-16) that no two tasks in the same column overlap in time. Multiple constraint items are allowed, they are implicitly conjoined.

Line 19 shows a `solve item`. Every model must include exactly one solve item. Here we are interested in minimizing the end time. We can also maximize a variable or just look for any solution (`solve satisfy`).

There is one kind of MiniZinc item not shown by this example: "include items". They facilitate the creation of multi-file models and the use of library files [4].

MiniZinc Challenge

Comparing constraint programming systems is fraught with difficulty. The reason is that there are so many components to a modern CP system, only some of which are implemented by some systems. To claim that one CP system is "better" than another is a bold claim, since

```
0   % (square) job shop scheduling in MiniZinc
1   int: size;                                  % size of problem
2   array [1..size,1..size] of int: d;          % task durations
3   int: total = sum(i,j in 1..size) (d[i,j]);  % total duration
4   array [1..size,1..size] of var 0..total: s; % start times
5   var 0..total: end;                          % total end time
6
7   predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
8       s1 + d1 <= s2 \/ s2 + d2 <= s1;
9
10  constraint
11      forall(i in 1..size) (
12          forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
13          s[i,size] + d[i,size] <= end /\
14          forall(j,k in 1..size where j < k) (
15              no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
16          )
17      );
18
19  solve minimize end;
```

Figure 1.1: MiniZinc model (`jobshop.mzn`) for the job shop problem(the portrayed example was taken from [4]).

```
20  size = 2;
21  d = [ 2,5,
22        3,4 ];
```

Figure 1.2: MiniZinc data (`jobshop2x2.dzn`) for the job shop problem (the portrayed example was taken from [4]).

there is almost certainly some problem for which the "worse" system allows a stronger model, or a better search, and performs better.

MiniZinc makes a good attempt to handle the most obvious obstacle: there are hundreds of potential global constraints, most handled by few or no systems. A standard input language, gives us the capability to compare different solvers. Hence, every year since 2008 the MiniZinc Challenge has been comparing different solvers that support MiniZinc [32].

The aim of the challenge is to compare various constraint solving technology on the same problems sets. The focus is on finite domain propagation solvers. An auxiliary aim is to build up a library of interesting problem models, which can be used to compare solvers and solving technologies.

Challenge participants provide a FlatZinc or MiniZinc solver and global constraint definitions specialized for their solver. Each solver is run on 100 MiniZinc model instances. For FlatZinc solvers, the MiniZinc compiler runs on the MiniZinc model and instance using the provided global constraint definitions to create a FlatZinc file. The resultant FlatZinc file is then given as input to the provided FlatZinc solver. For MiniZinc solvers, the MiniZinc model and data are input to the provided solver.

Points are awarded for solving problems, speed of solution, and goodness of solutions (for

optimization problems) [32], however, the MiniZinc challenge scoring system is not actually utilized in this specific research, since it is comparative scoring system, while an absolute one would be best suited for this specific research.

1.2 Large Language Models

Large Language Models (LLMs) are neural network models trained to process and generate natural language by learning statistical patterns over large text corpora.

LLMs are the culmination of decades of progress in natural language processing (NLP) and machine learning research, and their development is largely responsible for the explosion of artificial intelligence advancements across the late 2010s and 2020s. Popular LLMs have become household names, bringing generative AI to the forefront of the public interest. LLMs are also used widely in enterprises, with organizations investing heavily across numerous business functions and use cases [15].

LLMs are built on a type of neural network architecture called a transformer [16], which employs self-attention mechanisms to model dependencies among tokens in a sequence. Formally, given an input sequence $x_1, \ldots, x_n$, the model estimates the conditional distribution

$$P(x_{n+1} \mid x_1, \ldots, x_n),$$

and text generation proceeds autoregressively by repeatedly sampling from this distribution.

Training is typically divided into a large-scale pretraining phase followed by alignment stages. During pretraining, the model learns general linguistic and semantic representations through self-supervised objectives. Alignment techniques, such as supervised fine-tuning and reinforcement learning from feedback, adapt the model to follow instructions, respect constraints, and produce outputs that better match user expectations [15].

To better explain the concept, we can say LLMs are, at their core, autoregressive statistical models that generate text by iteratively predicting the most probable subsequent token in a sequence, based on patterns learned from their training data.

1.2.1 Inference and Decoding

At inference time, the model produces logits l_k over the vocabulary, which are transformed into probabilities using a softmax function [17]. Decoding strategies determine how the next token is selected from this distribution.

Modern LLMs typically generate text in a left-to-right, token-by-token fashion. For each prefix, the model computes a probability distribution of the next token over a fixed vocabulary. A decoding method defines how the generated token sequence is derived from these probability estimations. Deterministic approaches, select the most probable token at each step, while stochastic approaches introduce controlled randomness [18].

Some notable deterministic approaches are: "Greedy Search", which is limited to selecting the token with highest probability at each time step, or "Beam Search" [19], which maintains a beam of the k most probable sequences at each time step, where the hyperparameter k is referred to as the beam width.

Some notable stochastic methods are: Temperature Sampling samples tokens from the estimated next-token distributions. The skewness of distributions can be controlled using a temperature hyperparameter τ, or Top-p Sampling, which only considers the minimal set of most probable tokens that cover a specified percentage p of the distribution.

A central practical constraint is the context window, which limits the total number of tokens that can be processed jointly as input and output. This constraint directly influences prompt design, the amount of contextual information that can be provided, and the feasibility of maintaining conversational state.

1.2.2 Prompting and Interaction

LLMs are typically controlled through prompting, where task instructions and contextual data are encoded in the input.

Prompt engineering has emerged as an indispensable technique for extending the capabilities of LLMs. This approach leverages task-specific instructions, known as prompts, to enhance model efficacy without modifying the core model parameters. Rather than updating the model parameters, prompts allow seamless integration of pre-trained models into downstream tasks by eliciting desired model behaviors solely based on the given prompt.

Prompts can be natural language instructions that provide context to guide the model or learned vector representations that activate relevant knowledge. This rapidly growing field has enabled success across various applications, from question-answering to common sense reasoning [21].

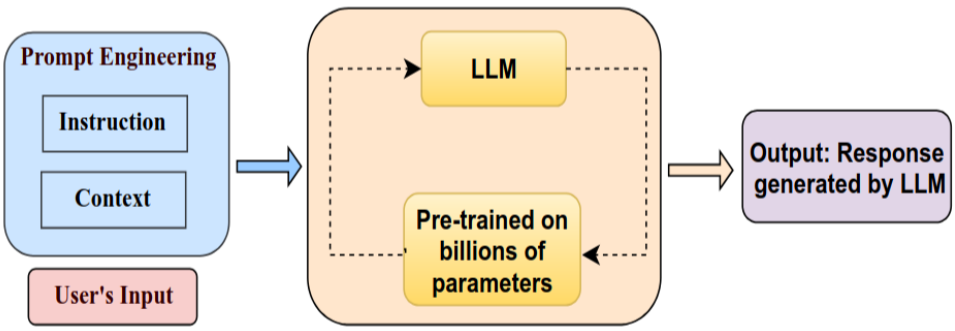

Figure 1.3: Visual breakdown of prompt engineering components: LLMs trained on extensive data, instruction and context as pivotal elements shaping the prompt, and a user input interface (image was taken from [21]).

Some notable prompt engineering techniques are: "Zero-Shot Prompting" which offers a

paradigm shift in leveraging large LLMs. This technique removes the need for extensive training data, instead relying on carefully crafted prompts that guide the model toward novel tasks, or "Few-Shot Prompting" which provides models with a few input-output examples to induce an understanding of a given task, unlike zero-shot prompting, where no examples are supplied [22].

Interaction may occur in a single-request mode, in which each task is processed independently, or in a multi-turn mode, where previous exchanges remain within the context window.

Prompt structure plays a significant role in model performance, particularly in technical tasks involving structured inputs, strict output formats, or long problem descriptions. The representation of information affects both the interpretability of the prompt and the effective use of the context window.

1.2.3 Application Programming Interfaces

In practical settings, LLMs are accessed through Application Programming Interfaces (APIs). An application programming interface, or API, enables commercial, military or private entities to make the data and functionality of their applications or systems available to external third-party developers, commercial partners, and internal departments within their own organizations. The usage of a defined interface enables services and products to interact with one another and benefit from each other's information and features. This interface is manipulated and programmed by developers to interact with other software, services or systems; they are not required to understand how a specific API is developed, and neither are the software's end-users. In short, an API is a contract between pieces of applications serving the main software once integrated into the source code of the main application [23].

An LLM API provides a standardized mechanism for submitting input messages and receiving model outputs, together with configurable inference parameters such as temperature. API-based access introduces operational constraints, including limits on the number of requests or tokens processed within a given time interval, as well as maximum context sizes per interaction. But enables the use of complex LLMs without the need of high computational power.

1.2.4 Relevance to Solver Selection

In this research, the aim is to test the potential of LLMs when used for solver selection. Where the prompt is engineered in order to help the LLM evaluate all the characteristics of a problem, to make a decision that is as functional as possible.

When applied to solver selection, an LLM functions as a decision component that maps problem representations to solver choices. Its performance depends on both theoretical aspects, such as its ability to model complex input-output relationships, and practical factors, including prompt formulation, context management, and API-level parameters.

Chapter 2

Methodology

In this chapter, we outline the decisions for an initial benchmark, which serves as the basis for refining subsequent experiments and assessing both the current capabilities and future potential of the solver.

The chapter is structured into six sections, each addressing a key preliminary choice. The first section discusses the provider selection. The second describes the selection of large language models used for testing. The third details the initial prompt-engineering strategy needed to define a clean, consistent prompt format for evaluation. The fourth presents the rationale behind the selection of benchmark problems used to test model performance. The fifth explains the metrics adopted to evaluate the testing results. The sixth describes the experiment setup and the pre-processing methods adopted to make automatic testing possible.

2.1 Provider Choice

First of all, we needed to understand which LLMs we could use for the algorithm selection. Given the limited computational resources available during the testing phase, we had to rely on externally hosted LLMs accessed through usage-based APIs. The selection prioritized generous free tiers, permissive rate limits, and straightforward integration. This led to the choice of the following providers:

- `Gemini API v1, GeminiAPI`, offered by Google DeepMind. Gemini is a family of LLMs with multiple sizes and capabilities. This provider selected for its strong reasoning abilities, robust tool-use features, and overall high-quality text generation. For the purpose of this research, the version v1 [25] was prefered as it is more stable and our only concern is its text generation capability.

- `Groq API, GroqAPI`, provided by Groq. Groq offers high-performance inference solutions through its specialized hardware architecture. The Groq API exposes a selection of LLMs through a simple and lightweight interface, enabling fast and low-latency experimentation.

Both APIs were selected for their ease of use, flexibility, overall performance, and, critically, their comparatively generous rate limits relative to competing services. In Table 2.2 and Table 2.1 are displayed rate limits of both APIs.

From the leftmost column of the table, there are: Model containing the names of each one of the available LLMs (for the purpose of this paper, only text generation models were selected), moving to the right *RPM* contains the maximum number of requests in a minute, *RPD* contains the maximum number of requests per day, *TPM* contains the maximum number of requested tokens per minute, and finally *TPD* contains the maximum number of tokens per day.

Model	RPM	RPD	TPM	TPD
allam-2-7b	30	7000	6000	500000
deepseek-r1-distill-llama-70b	30	1000	6000	100000
gemma2-9b-it	30	14400	15000	500000
groq/compound	30	250	70000	–
groq/compound-mini	30	250	70000	–
llama-3.1-8b-instant	30	14400	6000	500000
llama-3.3-70b-versatile	30	1000	12000	100000
meta-llama/llama-4-maverick-17b-128e-instruct	30	1000	6000	500000
meta-llama/llama-4-scout-17b-16e-instruct	30	1000	30000	500000
meta-llama/llama-guard-4-12b	30	14400	15000	500000
meta-llama/llama-prompt-guard-2-22m	30	14400	15000	500000
meta-llama/llama-prompt-guard-2-86m	30	14400	15000	500000
moonshotai/kimi-k2-instruct	60	1000	10000	300000
moonshotai/kimi-k2-instruct-0905	60	1000	10000	300000
openai/gpt-oss-120b	30	1000	8000	200000
openai/gpt-oss-20b	30	1000	8000	200000
playai-tts	10	100	1200	3600
playai-tts-arabic	10	100	1200	3600
qwen/qwen3-32b	60	1000	6000	500000

Table 2.1: Rate limits - Groq models [27]:
This table shows all the offered models from Groq API, in leftmost column and each relative rate limit.

2.2 Large Language Models Selection

Both providers offer a broad set of LLMs with varying capabilities and constraints, so an initial filtering step was required. Several options were excluded immediately because they are not designed for text generation. In particular, `playai-tts` and `playai-tts-arabic` are text-to-speech LLMs, and therefore unsuitable for testing.

Model	RPM	RPD	TPM	TPD
`gemini-2.5-pro`	5	100	250000	–
`gemini-2.5-flash`	10	250	250000	–
`gemini-2.5-flash-lite`	15	1000	250000	–
`gemini-2.0-flash`	15	200	1000000	–
`gemini-2.0-flash-lite`	30	200	1000000	–

Table 2.2: Rate limits - Gemini models [28]
This table shows all the offered models from Gemini API, in leftmost column and each relative rate limit.

Additional LLMs were removed because they are currently decommissioned or unavailable: `deepseek-r1-distill-llama-70b`, `gemini-2.0-flash-lite`, and `gemma2-9b-it`.

Two more LLMs were excluded due to insufficient context window size. Although their rate limits were acceptable, their token capacity was too small to accommodate even a single full MiniZinc model as input: `meta-llama/llama-prompt-guard-2-22m` and `meta-llama/llama-prompt-guard-2-86m`.

Finally, `allam-2-7b` was removed because it failed to follow instructions consistently, often producing incomplete, inconsistent, or unreadable outputs.

After this filtering stage, 18 LLMs remained as a stable base for the evaluation phase.

2.3 Prompt General Structure

To determine which LLM would be best suited for algorithm selection, it was necessary to design a consistent prompt format to query each model. The primary objective was to define a structure that was as short and clean as possible, for two main reasons:

- Minimize prompt-induced bias: A highly descriptive or too long and complex prompt could influence LLMs negatively. As we could encounter problems as “context rot” [29] - a progressive decay in accuracy as prompts grow longer.

- Reduce token usage: Since the testing setup depends on API limits, keeping the prompt compact minimizes token consumption.

2.3.1 Output Structure

Ensuring a standardized output format was equally important: Automated testing requires that model outputs follow a strict and predictable format. Any deviation introduces ambiguity during parsing and prevents reliable extraction of solver selections. Maintaining this structure is therefore essential to ensure consistent and fully automated evaluation.

Large or verbose responses also impose practical limitations on the available context window. Because each message contributes to the total token count, excessively long outputs reduce the room available for subsequent turns and larger prompts.

For these reasons, the output format was fixed as an array of three strings:

$$[\text{“}1^{st}\text{Solver}\text{”},\ \text{“}2^{nd}\text{Solver}\text{”},\ \text{“}3^{rd}\text{Solver}\text{”}]$$

Selecting the top three solvers enables two forms of evaluation:

- Single-solver evaluation: Measures whether the solver chosen by the LLM is the single best solver for the given instance. If it is not, the evaluation can quantify how close its performance is to the optimal solver.
- Parallel-solver evaluation: Measures the effectiveness of running the top three solvers selected by the LLM in parallel. The best result among the three is considered, allowing assessment of whether any of them corresponds to the single best solver for the instance, or, if not, how close the best among the three comes to the optimal performance.

The metrics used for these evaluations will be detailed in section 2.5.

After all of this considerations, the resulting prompt structure is the following:

Prompt Structure

MiniZinc model:

`...MiniZinc problem model (.mzn content) ...`

MiniZinc data:

`...Instance relative data (.dzn or .json content) ...`

The goal is to determine which constraint programming solver would be best suited for this problem, considering the following options:

- s_1,
- s_2,
- ...
- s_n where $s_1 \dots s_n \in SolverList$.

Answer only with the name of the 3 best solvers inside square brackets separated by comma and nothing else.

2.4 Problem Selection

A crucial component of the testing pipeline is the problem selection. Consistent and meaningful evaluation requires a set of benchmark problems that are reliable, diverse, and representative of real solver behavior. To meet these requirements, the problem set should satisfy the following criteria:

- Extensive prior testing: The problems must be validated and associated with reliable solver performance data, preferably obtained from recent evaluations of state-of-the-art solvers.

- Diversity: The set must include a varied mix of problem types-combinatorial problems, real-world applications, and puzzle-like tasks-covering all major categories: Maximization, Minimization and Satisfaction.

  This ensures that LLM performance can be assessed across different solving paradigms.

- Complexity: The problems must be sufficiently challenging so that solver selection is non-trivial and the LLM's reasoning abilities are meaningfully tested.

Following these criteria, the selected benchmark was the problem set from the MiniZinc Challenge 2025[32, 30, 31]. These problems are specifically curated to benchmark the strongest solvers of the year and therefore represent an ideal test bed for algorithm selection.

The problem set contains twenty problems: 1 satisfaction problem, 3 maximization problems, 16 minimization problems.

Each problem is a combination of a `.mzn` file containing the Minizinc [4] model, containing variables, constraints and objective function. Every problem is also accompanied by five corresponding data instances, each of them contained either in a `.dzn` or a `.json` file containing specific parameters and constants, yielding a total of 100 testable, diverse, and complex scenarios.

2.5 Test Metrics

In order to actually evaluate LLM performance, it is necessary to chose a standard metric for answer evaluation, other than that, it is necessary to have a metric to evaluate how each given LLM would perform against the current Single Best Solver (SBS).

Before analyzing the evaluation metrics, we must first define the systems to which these metrics will be applied. Namely, the solvers. In our context, a solver is a program that takes as input the description of a computational problem in a MiniZinc, with relative data, and returns an observable outcome providing zero or more solutions for the given problem.

For example, for decision problems, the outcome may be simply "yes" or "no" while for optimization problems, we might be interested in the best solutions found along the search. An evaluation metric, or performance metric, is a function mapping the outcome of a solver on a given instance to a number representing "how good" the solver is on this instance.

An evaluation metric is often not just defined by the output of the solver. Indeed, it can be influenced by other actors, such as the computational resources available, the problems on which we evaluate the solver, and the other solvers involved in the evaluation. For example, it is often unavoidable to set a `timeout` τ on the solver's execution when there is no guarantee

of termination in a reasonable amount of time (e.g. NP-hard problems). Timeouts make the evaluation feasible but inevitably couple the evaluation metric to the execution context. For this reason, the evaluation of a meta-solver should also consider the scenario that encompasses the solvers to evaluate, the instances used for the validation, and the timeout. Formally, at least for the purposes of this paper, we can define a scenario as a triple $(\mathcal{I}, \mathcal{S}, \tau)$, where: $\mathcal{I}$ is a set of problem instances, $\mathcal{S}$ is a set of individual solvers, $\tau \in (0, +\infty)$ is a timeout such that the outcome of solvers $s \in \mathcal{S}$ Solver instance $i \in \mathcal{I}$ is always measured in the time interval $[0, \tau]$.

Evaluating meta-solvers over heterogeneous scenarios $(\mathcal{I}_1, \mathcal{S}_1, \tau_1)$, $(\mathcal{I}_2, \mathcal{S}_2, \tau_2)$, ..., is complicated by the fact that the sets of instances $\mathcal{I}_k$, the sets of solvers $\mathcal{S}_k$ and the timeouts τ_k can be very different. Scenarios that involve optimization problems may even present additional complexities.

For those objectives two separate metrics were chosen.

2.5.1 Metric for Solver Score

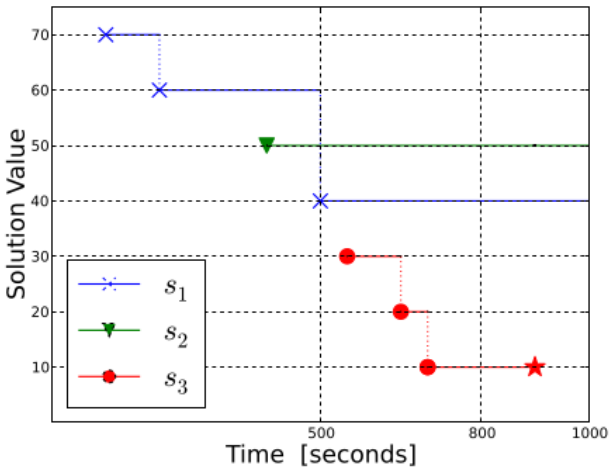

Figure 2.1: Solver performances example

We can now proceed to associate, for each instance i and solver s, a weight that quantitatively represents how well s performs on instance i within time T. We define the scoring value of s (shortly, score) on the instance i at a given time t as a function $\text{score}_{\alpha,\beta}$ [33, 34] defined as follows:

$$\text{score}_{\alpha,\beta}(s,i,t) = \begin{cases} 0, & \text{if } \text{sol}(s,i,t) = \text{unk}, \\ 1, & \text{if } \text{sol}(s,i,t) \in \{\text{opt}, \text{uns}\}, \\ \beta, & \text{if } \text{sol}(s,i,t) = \text{sat} \\ & \text{and } \text{MIN}(i) = \text{MAX}(i), \\ \max\left\{0,\ \beta - (\beta-\alpha)\dfrac{\text{val}(s,i,t) - \text{MIN}(i)}{\text{MAX}(i) - \text{MIN}(i)}\right\}, & \text{if } \text{sol}(s,i,t) = \text{sat} \\ & \text{and } i \text{ is a minimization problem}, \\ \max\left\{0,\ \alpha + (\beta-\alpha)\dfrac{\text{val}(s,i,t) - \text{MIN}(i)}{\text{MAX}(i) - \text{MIN}(i)}\right\}, & \text{if } \text{sol}(s,i,t) = \text{sat} \\ & \text{and } i \text{ is a maximization problem}. \end{cases}$$

Here, $\text{MIN}(i)$ and $\text{MAX}(i)$ denote the minimal and maximal objective function values found by any solver s at the time limit T.

As an example, consider the scenario in Figure 2.1 showing three different solvers on the same minimization problem. Let $T = 500$, $\alpha = 0.25$, $\beta = 0.75$. Solver s_1 finds the optimal value (40), therefore it receives score 0.75. Solver s_2 finds the maximal value (50), hence score 0.25. Solver s_3 does not find a solution in time, giving score 0. If instead $T = 800$, the value of s_1 becomes 0.375 and s_3 gets 0.75. If $T = 1000$, since s_3 improves the objective to 10 (marked with a star in the figure), it receives the highest score.

The parameter used for score calculation in testing are: $T = 1200000$ (1200000ms = 20 minutes, which is the time limit used solver evaluation in the MiniZinc Challenge) $\alpha = 0.25$ $\beta = 0.75$.

2.5.2 Closed Gap

Once the evaluation metric for solver score has been defined, we also need a comparative metric after score calculation. For this objective, we have chosen to use closed-gap (CG) [33] as the evaluation metric. CG is a relative and meta-solver-specific measure, adopted in the 2015 ICON and 2017 OASC [35] challenges to handle the disparate nature of the scenarios.

CG assigns to a meta-solver a value in $(-\infty, 1]$ proportional to how much it closes the gap between the best individual solver available, or single best solver (SBS), and the virtual best solver (VBS), i.e., an oracle-like meta-solver always selecting the best individual solver. The closed gap is actually a "meta-metric", defined in terms of another evaluation metric m to minimize, which in this case is the scoring metric defined earlier. Formally, if (I, S, τ) is a scenario then

$$m(i, \text{VBS}, \tau) = \min\{m(i, s, \tau) \mid s \in S\} \quad \text{for each } i \in I,$$

and

$$\text{SBS} = \arg\min_{s \in S} \sum_{i \in I} m(i, s, \tau).$$

With these definitions, CG can be defined as follows: Let $(\mathcal{I}, S, \tau)$ be a scenario and

$$m : \mathcal{I} \times \big(S \cup \{S, \mathrm{VBS}\}\big) \times [0, \tau] \to \mathbb{R}$$

an evaluation metric to minimize for that scenario, where S is a meta-solver over the solvers of S. Let

$$m_\sigma \;=\; \sum_{i \in \mathcal{I}} m(i, \sigma, \tau) \qquad \text{for } \sigma \in \{S, \mathrm{SBS}, \mathrm{VBS}\}.$$

The CG of S with respect to m on that scenario is

$$\frac{m_{\mathrm{SBS}} - m_S}{m_{\mathrm{SBS}} - m_{\mathrm{VBS}}}.$$

The assumption $m_{\mathrm{VBS}} > m_{\mathrm{SBS}}$ is required, i.e., no single-solver can be the VBS (otherwise, no algorithm selection would be needed, given that its objective is to reach the VBS). Unlike other scores, CG is designed specifically for meta-solvers. Applying it to individual solvers would assign 0 to the SBS and a negative score to the remaining solvers, proportional to their performance difference with respect to the SBS and the gap $m_{\mathrm{SBS}} - m_{\mathrm{VBS}}$, which makes little sense for individual solvers, as it wouldn't reflect their actual performance overall.

2.6 Experiment Setup

We have defined both the structure of the queries posed to the LLMs (Section 2.4) and the way in which these queries are formulated (Section 2.3). The aim now is to evaluate them automatically over the full set of selected instances. To this end, we designed an automated testing pipeline that parallelizes execution by assigning one thread per LLM. For each model, requests are issued sequentially, with five requests per problem, each containing a MiniZinc model and a single instance.

Despite preliminary prompt engineering and model filtering, several MiniZinc models, particularly their associated data files, still exceed the providers' rate limits. Given this limitation, additional mechanisms were required to prevent limit violations while still allowing evaluation over the complete instance set.

2.6.1 Script Sanitization

The most direct way to address oversized requests is to reduce their length. As the prompt itself was already minimal, this required direct manipulation of the MiniZinc model (`.mzn`) and data files.

A first step consisted in removing all non-essential elements, such as comments (starting with `%` [4]), tabs, and unnecessary whitespace. While this helps reduce token usage and standardizes script formatting, it is insufficient on its own. The main contributor to token overflow is the presence of large data arrays, which not only increase message length but may also pollute the context, "distracting" the LLM from the most relevant information [36].

To mitigate this issue, data arrays were truncated to a fixed maximum length of 30 elements, with an inline comment indicating the original size:

$$[e_1, e_2, \ldots, e_{30}\texttt{// array too long to display, dimensions: (150)}]$$

While effective for simple arrays of scalar values, this approach does not account for the complexity of individual elements and performs poorly on more structured data. For this reason, a second truncation mechanism was introduced based on raw character length. Arrays exceeding 120 characters were truncated accordingly, using the same annotation to preserve information about the original size.

2.6.2 Custom Delays

Since each experiment involves multiple problems and multiple sequential requests per LLM, rate limits can still be exceeded even when individual requests are within bounds. To handle this, custom delays were introduced into the experiment orchestration logic.

When an error message is received, for example:

```
Error code: 413 - Request too large for model `openai/gpt-oss-120
    b` in organization `org_01k9qqesvte4d9h5jnhmzvbmy4` service
    tier `on_demand` on tokens per minute (TPM): Limit 8000,
    Requested 8939, please reduce your message size and try again.
     Need more tokens? Upgrade to Dev Tier today at https://
    console.groq.com/settings/billing

Error code: 429 - Rate limit reached for model `openai/gpt-oss
    -120b` in organization `org_01k9qqesvte4d9h5jnhmzvbmy4`
    service tier `on_demand` on tokens per day (TPD): Limit
    200000, Used 193047, Requested 10632. Please try again in 26
    m29.328s. Need more tokens? Upgrade to Dev Tier today at https
    ://console.groq.com/settings/billing
```

Its code is inspected. Errors 413 and 429 indicate that a rate limit has been exceeded. The error message is then parsed to identify the specific limit involved. If the limit concerns tokens per minute (TPM) or requests per minute (RPM), the system pauses execution for 60 seconds before retrying. If the exceeded limit is tokens per day (TPD) or requests per day (RPD), the message is further analyzed to extract the cooldown duration, expressed in the form $XXh, XXm, XX.XXs$ where h stands for hours, m for minutes and s for seconds. The required delay is then computed from this value, after which the request is retried.

Chapter 3

Experimental Evaluation

This chapter presents the initial experimental study aimed at assessing the potential of LLMs for algorithm selection and identifying effective prompting strategies and contextual representations.

The first section reports results obtained with the most basic configuration, using only raw scripts and instance data, intended to estimate the baseline performance of all candidate LLMs. These results are used to select the five best-performing models for subsequent analysis. The second section examines refined prompt formulations based on sanitized scripts and the inclusion of problem descriptions, using a single-request setup in which each instance is handled independently.

The third section investigates multi-turn experiments, where prompts are enriched with solver descriptions and combined problem-solver descriptions. This setup leverages conversational state to reduce redundancy and mitigate rate-limit constraints inherent to single-request configurations. From this stage onward, experiments are conducted only with the best-performing model.

The fourth section evaluates the use of structured contextual representations derived from a feature extraction tool [39], in combination with the prompt strategies developed earlier. The fifth section analyzes the effect of sampling temperature tuning [47] by testing multiple temperature values on the five most effective configuration variants identified across the study.

Finally, the sixth section analyzes LLM performance whe tested on a restricted set of solvers to choose from, avoiding the best performing, dominant solvers and the worst performing ones.

3.1 Preliminary tests

The first experiments were conducted using the unedited problems. Each LLM was provided with the original MiniZinc model (`.mzn`) together with the corresponding instance data (in either `.dzn` or `.json` format), following the prompt structure defined in Section 2.3.

As reported in Table 3.1, (and more in depth in Table 4.4, Table 4.3 and Table 4.5) for both single-solver evaluation and parallel-solver evaluation, the scores are all under 70, meaning a

Model	Single Score	Parallel Score	Closed Gap
gemini-2.5-flash-lite	64.363	69.040	-1.047
gemini-2.5-flash	60.962	69.426	-1.330
moonshotai/kimi-k2-instruct-0905	59.680	66.186	-1.436
moonshotai/kimi-k2-instruct	58.609	65.816	-1.525
openai/gpt-oss-120b	58.166	64.508	-1.562
openai/gpt-oss-20b	57.154	63.329	-1.646
meta-llama/llama-4-maverick-17b-128e-instruct	56.297	63.549	-1.717
meta-llama/llama-4-scout-17b-16e-instruct	54.305	57.815	-1.883
gemini-2.0-flash	43.413	53.748	-2.788
qwen/qwen3-32b	42.117	48.018	-2.895
gemini-2.5-pro	37.641	41.594	-3.267
llama-3.1-8b-instant	36.082	56.228	-3.397
groq/compound-mini	25.423	29.623	-4.282
llama-3.3-70b-versatile	7.455	9.496	-5.775
groq/compound	5.197	8.246	-5.963

Table 3.1: Initial tests giving plain scripts to all the LLMs. In this table: column "Model" contains the names of each tested LLM, "Single Score" was calculated by summing the performance of what was suggested to be the best solver (as explained in Section 2.5.1) on each of the instances by the given LLMs, "Parallel Score" represents the sum of the score reached in every instance in parallel-solver setup, so taking the 3 best solvers given by the LLM, calculate the score of all the 3, and take the maximum out of the three, and finally "Closed Gap" displays the closed gap score calculated over "Single Score" by using the formula explained in Section 2.5.2.

lower performance than 3 of the single solvers from free category, and 4 of the single solvers from open category, and clearly a negative closed gap for all of the LLMs.

While part of this outcome can be attributed to the deliberately simple formulation of the requests, a major limiting factor is the presence of strict rate limits, which prevent many instances from being processed by some, if not all, of the LLMs, due to the script length alone exceeding TPM (showed in Table 2.2 and Table 2.1). This problem is predominant in problems with large instance data, such as: `ihtc-2024-marte` or `gt-sort`

These constraints motivated both the adoption of script manipulation techniques, as described in Section 2.6, and, simple time issues due to tests taking even up to 48 hours, the decision to restrict subsequent experiments to the five best-performing LLMs identified in this preliminary phase, namely: `gpt-oss-120b`, `gemini-2,5-flash`, `gemini-2,5-flash-lite`, `kimi-k2-instruct-0905` and `kimi-k2-instruct`.

3.2 Single Request Experiments

After establishing a stable experimental pipeline, we repeated the evaluation on the full set of 100 instances. All experiments in this phase adopt a single-request setup, where each prompt is processed independently: the LLM receives the input and produces an answer without access to any prior interaction history or retained context.

3.2.1 Base Setup

As a baseline, the LLMs were first evaluated under the same conditions as the preliminary experiments, using only the raw MiniZinc and data scripts.

Model	Single Score	Parallel Score	Closed Gap
`openai/gpt-oss-120b`	74.488	82.227	-0.206
`moonshotai/kimi-k2-instruct-0905`	71.623	82.657	-0.444
`moonshotai/kimi-k2-instruct`	70.939	83.268	-0.501
`gemini-2.5-flash-lite`	70.145	80.741	-0.567
`gemini-2.5-flash`	69.763	79.105	-0.598

Table 3.2: Tests on sanitized scripts given to the 5 best performing LLMs, columns content is calculated as in Table 3.1.

As reported in Table 3.2 and Table 4.6, performance under the parallel-solver evaluation is consistently high. All tested LLMs outperform every individual solver except `or-tools_cp-sat-par`, which constitutes the single best solver (SBS) in the open category and remains clearly dominant, maintaining a substantial margin over both the LLM-based algorithm selection approaches and the other standalone solvers.

More pronounced variability is observed in the single-solver evaluation (Table 4.7). In this setting, only `gpt-oss-120b` consistently surpasses all individual solvers other than the SBS in the free category (`or-tools_cp-sat-free`), as further detailed in Table 4.20 and Figure 4.7. The remaining LLMs still achieve competitive results relative to most standalone solvers; however, a non-negligible performance gap persists, as reflected in the closed-gap scores and analyzed in greater detail in Table 4.8.

3.2.2 Problem Description

The results of the baseline experiments indicate that further improvements are necessary. A natural approach is to provide the LLM with additional contextual information. However, as discussed previously, excessively large contexts can be counterproductive, potentially distracting the model and degrading performance rather than improving it [29, 36]. This trade-off motivates

a more careful investigation into which types of information are most beneficial for LLM-based solver selection.

As a first step, we augmented the prompt with a concise problem description (PD): a short textual summary of the MiniZinc model's semantics. These descriptions were automatically generated using another LLM (`GPT-5.1`) and subsequently refined manually to correct minor inaccuracies, e.g.:

- `atsp`: "Scheduling and resource allocation problem involving moulds, colors, and production jobs. The goal is to minimize makespan, tardiness, and waste while respecting compatibility and demand constraints."

- `black-hole`: "A constraint model for solving the Black Hole Patience solitaire game. Cards must be arranged so that the sequence follows game rules using global constraints."

The PD was incorporated into the prompt structure as follows:

Prompt Structure

Problem description:
`...Textual problem description ...`
MiniZinc model:
`...MiniZinc problem model (.mzn content) ...`

MiniZinc data:
`...Instance relative data (.dzn or .json content) ...`

The goal is to determine which constraint programming solver would be best suited for this problem, considering the following options:
- s_1,
- s_2,
- ...
- s_n where $s_1 \dots s_n \in SolverList$.

Answer only with the name of the 3 best solvers inside square brackets separated by comma and nothing else.

As shown in Table 3.3, single-solver performance consistently degrades, while parallel-solver evaluation shows modest improvements, with three out of the five tested LLMs achieving better results than in the base configuration. This divergence suggests that additional context can aid diversification in solver selection, even if, in this case, it does not reliably improve the choice of an optimal solver.

As can be deduced from the results, all the closed gap scores are negatives even after improving the context.

Model	Single Score	Parallel Score	Closed Gap
`moonshotai/kimi-k2-instruct-0905`	72.605	83.182	-0.362
`openai/gpt-oss-120b`	70.741	83.250	-0.517
`gemini-2.5-flash-lite`	69.043	82.621	-0.658
`moonshotai/kimi-k2-instruct`	67.555	81.890	-0.782
`gemini-2.5-flash`	49.897	59.154	-2.249

Table 3.3: Test on sanitized scripts combined with textual problem description. Columns are calculated as in Table 3.1.

3.3 Multi-turn Experiments

What we discussed so far indicate that the contextual information provided to the LLMs is either insufficient or, in some cases, not beneficial overall. A natural next step is therefore to explore alternative forms of context. However, this introduces two practical issues. First, the single-request setup already operates close to the maximum allowed tokens per minute (TPM), making it infeasible to simply add more information without violating rate limits. Second, the existing setup is inefficient in terms of token usage, as it repeatedly supplies redundant information.

More specifically, for each problem we evaluate five different instances, each with distinct data, while the underlying MiniZinc model and the associated problem description remain unchanged. Re-sending this invariant information with every request unnecessarily consumes tokens. Given the limited API resources available, addressing both constraints is essential. To this end, we transitioned to a multi-turn (chat-like) experimental setup.

3.3.1 Setup Explanation

The core idea of the multi-turn setup is to partition the interaction using role-based formatting [37]. In the context of LLMs, messages are explicitly associated with roles, typically `system`, `user`, and `assistant`. Which helps the LLM distinguish between instructions, inputs, and generated outputs, while also maintaining conversational state across turns.

In our experiments, the `system` role is used to convey all invariant and high-level information, namely the MiniZinc model (`.mzn`) content, the textual descriptions, and the expected format of the answers. The `user` role is then reserved for instance-specific inputs, containing the data associated with each instance (in `.dzn` or `.json` format). This separation allows us to avoid repeatedly transmitting redundant context, significantly reducing token consumption per instance.

An additional advantage of the multi-turn setup is that it enables the handling of larger instance data by distributing content across multiple messages, while relying on the LLM's

ability to retain previously supplied information within the same conversation. As a result, the system can accommodate longer and more complex inputs without exceeding rate limits.

3.3.2 Solvers Description

The increased efficiency of the multi-turn setup also makes it possible to enrich the contextual information with new textual data: solver descriptions, examples can be found in: Table 4.12. For each solver under consideration, a short textual description was generated and provided to the LLM within the `system` prompt, with the aim of improving the model's awareness of the available solver options and their respective characteristics.

To better understand the importance of this information, we experimented with two different configurations: one in which solver descriptions were combined with all previously provided contextual elements, and another in which the solvers descriptions constituted the only additional textual information alongside the MiniZinc model and the instance data. This design allows us to assess the contribution of solver-specific knowledge to the overall performance of LLMs.

Model	Single Score	Parallel Score	Closed Gap
`moonshotai/kimi-k2-instruct`	**76.964**	82.236	**0.000**
`moonshotai/kimi-k2-instruct-0905`	**76.964**	82.489	**0.000**
`gemini-2.5-flash`	71.687	83.137	-0.439
`openai/gpt-oss-120b`	70.974	78.800	-0.498
`gemini-2.5-flash-lite`	54.714	77.747	-1.849

Table 3.4: Test with sanitized scripts combined with solvers description in a multi turn setup. Columns are calculated as in Table 3.1.

In the parallel-solver evaluation (Table 4.13), providing only solver descriptions does not lead to systematic improvements. The sole exception is `gemini-2.5-flash` although its score is still under that of the single best solver (SBS) in the open category.

On the other hand, the effects are more pronounced in the single-solver evaluation, displayed in Table 3.4 and Table 4.14. Two models, `moonshotai/kimi-k2-instruct-0905` and `moonshotai/kimi-k2-instruct`, exhibit a substantial improvement, reaching the SBS score and thus achieving a closed-gap value of zero for the first time. A closer inspection of their outputs, however, reveals that this result is achieved by consistently selecting the same solver, namely `or-tools_cp-sat-free`, which is itself the SBS. This behavior effectively bypasses the decision-making role of the LLM, thereby undermining the intended purpose of employing an LLM in the first place.

3.3.3 Solver Description and Problem Description

Following this observation, we evaluated the configuration combining both forms of textual context: solver descriptions and problem descriptions. In this setup, each LLM is provided with the largest amount of contextual information until now.

Model	Single Score	Parallel Score	Closed Gap
`openai/gpt-oss-120b`	**77.261**	80.296	**0.025**
`moonshotai/kimi-k2-instruct-0905`	76.964	82.219	0.000
`moonshotai/kimi-k2-instruct`	74.464	80.417	-0.208
`gemini-2.5-flash`	73.552	83.651	-0.284
`gemini-2.5-flash-lite`	63.063	78.148	-1.155

Table 3.5: Test with sanitized scripts combined with both solvers description, and problem description in a multi turn setup. Columns are calculated as in Table 3.1.

In the parallel-solver evaluation, this configuration yields the highest scores observed so far, again driven primarily by `gemini-2.5-flash`. However, other LLMs score is slightly lower than in the previous setups, and all of the reached scores are still under the open-category SBS.

In the single-solver evaluation, `moonshotai/kimi-k2-instruct-0905` continues to default to selecting the SBS exclusively. Nevertheless, improvements emerge for two other models, namely `gemini-2.5-flash` and `gpt-oss-120b`. In particular, `gpt-oss-120b` achieves the first strictly positive closed-gap score, surpassing `or-tools_cp-sat-free`.

3.3.4 Basic Tests Evaluation

In this initial testing phase, a positive closed gap has been achieved, showing an algorithm selection process with better performance than always choosing the single best solver. Moreover, when we put the performance respect to the "non-best solvers", these primitive configurations still hold fairly competitive results.

To better display single LLMs performances, all variants scores are put together against one another, using histograms for better visualization in Figure 4.5 for parallel-solver evaluation, Figure 4.6 for single-solver evaluation and Figure 3.1 for closed gap evaluation.

To give results another point of view, the performance of each configuration is displayed against the single solvers of the corresponding category. In Table 4.19 and Figure 4.8 are shown the performance of all the configurations, scored with parallel-solvers evaluation, when put agains single solvers from the open category. On the other hand, looking at Table 4.20 and Figure 4.7, the resulting score of single-solver evaluation of all the variants is put against all the single solvers from free category.

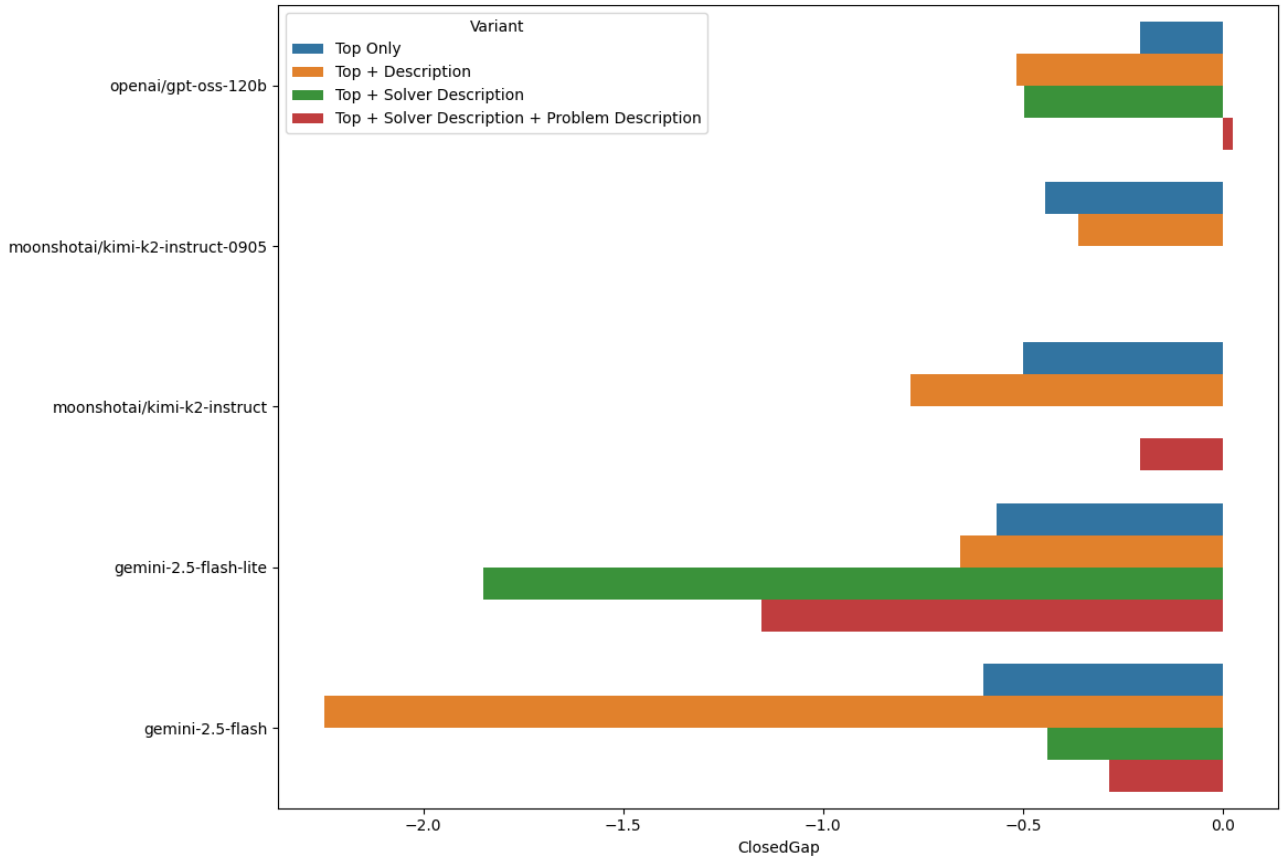

Figure 3.1: Histograms displaying the performances of all the solver variants in "Closed Gap" evaluation, as calculated in Table 3.1.

3.4 Feature Extraction

In the last sections, we exposed the importance of textual information, and giving a richer context to the LLM instead of relying on its understanding of a `.mzn` script. Even though a positive closed gap was already reached in previous experiments, relying on context information such as the problem description is a strong limitation for the solver, given those data need to be supervised, if not entirely rewritten by hand, due to the high variability, especially when extracted on new problems. This limitation highlights the necessity of a mean to extract information automatically, in a controlled and predictable way.

Another problem is the one concerning script dimensions: programs with longer scripts need some techniques like sanitization, as previously stated in Section 2.6.1, these type of technique, makes it possible to work with longer scripts and context data. But is still penalizing towards longer problems, raising the risk of hallucination [38] and context rot [29], due to the forced removal of elements in data, leaving incomplete arrays as input.

For these tasks we decided to employ a feature-extractor [39], which allows to extract an extensive set of 95 features from a Constraint (Satisfaction/Optimization) Problem defined in possibly different modelling languages: MiniZinc, FlatZinc or XCSP. Designed to be independent from the particular machine on which it is run as well as from the specific global redefinitions

of a given solver. The employed version was already used in other projects [42, 43, 44].

3.4.1 Tool Description

The tool mzn2feat is designed to extract a set of 155 features from a MiniZinc model. Of these, 144 are static features derived through syntactic analysis of the source problem instance, while 11 are dynamic features obtained via a short execution of the Gecode solver. Due to the complexity of the MiniZinc language, particularly the presence of control-flow constructs, direct extraction of syntactic features from MiniZinc models is nontrivial. To address this, models are first compiled into FlatZinc, a lower-level language whose syntax is largely a subset of MiniZinc. This translation is performed using the `mzn2fzn` tool provided within the MiniZinc toolchain.

The compilation step employs Gecode-specific [40] redefinitions of global constraints. This preserves information about the presence and type of global constraints without decomposing them into primitive constraints. Such preservation is relevant because, in the absence of solver-specific redefinitions, certain global constraints (e.g., `alldifferent`) when decomposed into sets of simpler constraints, from which the original high-level constraint cannot be uniquely reconstructed.

Static feature extraction from the resulting FlatZinc model is performed using `fzn2feat`, a parser implemented with Flex and Bison [41]. Dynamic features are obtained by executing the Gecode FlatZinc interpreter (`fz`) for a fixed time budget of two seconds on the compiled model.

In summary, given a MiniZinc model M, the mzn2feat workflow consists of three stages. First, M is translated into a FlatZinc model F_M using Gecode global constraint redefinitions. Second, static features are extracted from F_M via `fzn2feat`. Third, dynamic features are extracted from F_M through a bounded run of the Gecode interpreter. The static feature extraction stage is applicable to any FlatZinc model, although solver-specific redefinitions that are not recognized may be ignored. The static and dynamic feature extraction procedures are independent, allowing them to be executed in parallel or in arbitrary order. For example, static feature computation may be omitted if the instance is solved during the dynamic feature collection phase [39].

For a better understanding of all of the features, refer to Table 4.21 for the description by category, and to Listing 4.1 as an example output.

3.4.2 Testing and Results

To evaluate the effectiveness of feature-based representations, the pretty-printed `mzn2feat` output (e.g., 4.1) was incorporated into the prompt under several configurations:

- Only `mzn2feat` output.
- `mzn2feat` output and problem text description.

- `mzn2feat` output, problem text description and solvers text description.

Each configuration was further extended by incrementally adding the `.mzn` model and, subsequently, instance data (`.dzn`/`.json`).

Variant	Single Score	Parallel Score	Closed Gap
Features + Solver Description + Problem Description	75.659731	83.731582	-0.108399
Features + Solver Description	75.390197	82.994278	-0.130793
Features + Problem Description	74.829813	83.289246	-0.177354
Features	73.600955	82.259760	-0.279455
Features + .mzn + Problem Description	71.916389	78.530605	-0.419420
Features + .mzn	70.368365	78.371276	-0.548041
Features + .mzn + Instance Data	68.422708	74.803213	-0.709699
Features + .mzn + Instance Data + Solver Description	63.585459	66.346631	-1.111610
Features + .mzn + Instance Data + Problem Description	62.516550	69.284744	-1.200422
Features + .mzn + Solver Description	61.687990	66.874762	-1.269264
Features + .mzn + Solver Description + Problem Description	58.576543	64.774159	-1.527784
Features + .mzn + Instance Data + Solver Description + Problem Description	54.188182	59.410873	-1.892398

Table 3.6: Test with all the possible combinations involving features extracted using `mzn2feat` [39], all the tests have been made using `gpt-oss-120b` as it's the only model that produced a positive closed gap until this point. Columns are calculated as in Table 3.1.

Results are reported in Table 3.6. Configurations combining feature representations with textual descriptions consistently achieve higher scores than those including raw `.mzn` scripts or instance data. The addition of full model code or data is associated with systematic score reductions across both single and parallel evaluations, most probably because of some problems explained earlier such as context rot [29].

The highest-performing configurations in this group are those combining features with solver and/or problem descriptions. Although these variants do not exceed the best-performing configuration identified in earlier experiments, the top three feature-based setups achieve higher scores than the second-best configuration among the variants proposed in Section 3.3.3.

3.5 Sampling Temperature Tuning

The testing of this initial phase involved the use of sampling temperature tuning. Sampling temperature is a hyperparameter of an LLM used in a temperature-based sampling process. It controls the randomness of the model's output at inference time [45].

During each step of an LLM's decoding process, the LLM uses the previous tokens to choose the next output token. The final layer of the LLM uses a softmax function to convert raw scores (logits) into probabilities.

In greedy sampling, the model will always choose the most likely next token. However, for probabilistic sampling, the next token is selected from a probability distribution.

Temperature sampling is a modification to the softmax function, which adjusts the resulting probability mass functions. In this modified softmax function, v_k is the k-th vocabulary token, l_k is the token's logit, and τ is a constant temperature:

$$\Pr(v_k) = \frac{e^{l_k/\tau}}{\sum_i e^{l_i/\tau}}$$

A lower temperature makes the output of the LLM more deterministic, thus favoring the most likely predictions. This conservativeness is captured by the model's tendency to produce more repetitive, focused, and less diverse output based on the patterns most commonly seen in the training data. A higher temperature increases the randomness of the output, thus favoring more "creative" predictions. This creativity is captured by the model's willingness to explore more unconventional and less likely outputs. Higher temperatures can lead to novel text, diverse ideas, and creative solutions to problems [46, 47].

In the context of problem-solving, temperature can be seen as a trade-off between exploring possible solutions within the solution space and exploiting probable solutions; lower temperatures tend to exploit probable solutions, whereas higher temperatures explore the solution space more broadly.

3.5.1 Choosing Sampling Temperatures

In practical applications, lower temperatures are typically associated with tasks emphasizing consistency and structural correctness, whereas higher temperatures are used in contexts where output diversity is beneficial [48]. Increased stochasticity, however, can also raise the likelihood of incoherent or factually incorrect outputs [38]. Temperature selection therefore constitutes a trade-off between determinism and diversity.

An ablation study on temperature was conducted only for `gpt-oss-120b`, the model accessed through the Groq API [26]. The tested values were based on the preset recommendations in the provider documentation [49] (Table 3.7).

Scenario	Temp	Comments
Data extraction (JSON)	0.0	Deterministic keys/values
Factual Q&A	0.2	Keeps dates & numbers stable
Long-form code	0.3	Fewer hallucinated APIs
Brainstorming list	0.7	Variety without nonsense
Creative copywriting	0.8	Vivid language, fresh ideas

Table 3.7: Temperature setting suggestions from Groq documentation [49]. In the leftmost column is presented the basic scenario in wich the following settings are more indicated, followed by the sampling temperature value, and finally a short comment on the expected behaviour from the LLM with the given setting.

3.5.2 Testing and Results

As anticipated, the tests were all made only using `gpt-oss-120b` for LLM. Due to rate-limit constraints, temperature tuning experiments was restricted to the five best-performing configuration variants identified earlier, which we will later refer to using this numbers:

1. `.mzn` scripts, instance data and text description for both solvers and problems.
2. Features extracted through `mzn2feat` and text description for both solvers and problems.
3. Features extracted through `mzn2feat` and text description for solvers.
4. Features extracted through `mzn2feat` and text description for problems.
5. `.mzn` scripts and instance data

Variant	Temperature	Single Score	Parallel Score	Closed Gap
Features + Solvers Descripton	**0.3**	**78.032**	82.873	**0.089**
Features + Solvers Descripton	**0.7**	**77.896**	83.767	**0.077**
Scripts + Solvers Descripton + Problem Description	0.0	76.971	82.042	0.001
Features + Solvers Descripton + Problem Description	0.3	76.529	82.959	-0.036
Features + Solvers Descripton	0.2	76.326	84.044	-0.053
Features + Problem Description	0.2	76.298	83.363	-0.055
Features + Solvers Descripton + Problem Description	0.2	76.156	83.763	-0.067
Features + Solvers Descripton + Problem Description	0.0	75.860	83.146	-0.092
Features + Problem Description	0.8	75.676	83.009	-0.107
Features + Solvers Descripton	0.0	75.477	83.444	-0.124
Features + Solvers Descripton + Problem Description	0.8	74.909	82.419	-0.171
Scripts + Solvers Descripton + Problem Description	0.3	74.907	80.042	-0.171
Features + Solvers Descripton	0.8	74.464	84.421	-0.208
Scripts	0.8	74.460	83.222	-0.208
Scripts	0.2	74.456	83.119	-0.208
Scripts	0.7	73.459	82.540	-0.291
Scripts	0.3	73.456	84.261	-0.291
Scripts	0.0	73.269	83.602	-0.307
Features + Solvers Descripton + Problem Description	0.7	73.224	80.845	-0.311
Features + Problem Description	0.0	73.114	81.456	-0.320
Scripts + Solvers Descripton + Problem Description	0.8	72.894	80.875	-0.338
Features + Problem Description	0.3	72.678	82.393	-0.356
Scripts + Solvers Descripton + Problem Description	0.2	71.717	82.042	-0.436
Features + Problem Description	0.7	71.235	81.896	-0.476
Scripts + Solvers Descripton + Problem Description	0.7	69.890	79.042	-0.588

Table 3.8: Sampling-temperature sweep on the best-performing variants using `gpt-oss-120b`. "Scripts" in "Variant" column refers to `.mzn` script and instance data script. Score columns are calculated as in Table 3.1.

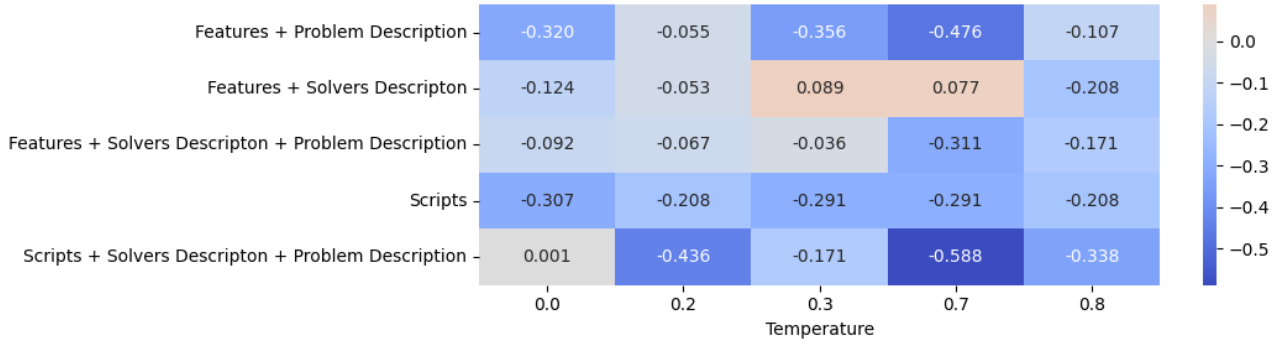

Figure 3.2: Heatmap displaying the performance of all the combinations of temperatures with the five best performing variants in "Closed Gap" evaluation calculated as in Table 3.1, all tests were performed using `gpt-oss-120b`.

As displayed in Table 3.8 and in Figure 4.9, in parallel-solver evaluation, temperature variation produces only moderate changes. Although some configurations yield higher parallel scores than their counterparts with no temperature configuration, all remain below the performance of `or-tools_cp-sat-par` (displayed in Table 4.19).

In the single-solver evaluation (Figure 4.10), temperature has a more pronounced effect. Configurations based on raw scripts show limited sensitivity to temperature, with only marginal improvements with configuration 5, and a decrease in scores for configuration 1. In contrast, feature-based configurations exhibit clearer trends: temperatures in the range 0.2-0.3 are associated with improved single-solver scores. Notably, configuration 3 achieves positive closed-gap values at τ=0.3 and τ=0.7, displayed in Figure 3.2, indicating performance above the SBS baseline under those settings, displaying highest scores yet.

3.6 Restricted Solver Portfolio

In the previous experimental setting, the best-performing configuration achieved a closed gap of 0.08, corresponding to a marginal improvement over the single best solver (SBS), namely `or-tools_cp-sat-free` (cf. Table 3.8). However, this improvement must be interpreted in light of the strong dominance of the SBS across the benchmark set. In practice, consistently selecting `or-tools_cp-sat-free` constitutes a highly competitive strategy, as shown in Table 3.4. This dominance reduces the effective difficulty of the solver selection task.

To obtain a more discriminative evaluation setting, we constructed a restricted solver portfolio excluding both highly dominant solvers and clearly underperforming ones. The objective was to create a scenario in which solver selection requires finer-grained differentiation among comparably performing systems, thereby allowing a more meaningful assessment of the LLMs' decision process.

To guide this selection, we computed, for each solver, the number of instances for which it achieved an optimal score (i.e., score equal to 1 according to the metric defined in Section 2.5).

The complete counts are reported in Table 4.22. Based on these results, we retained only solvers whose number of optimal solutions lies between 10 and 40. This filtering removes both outliers at the top end, which would trivialize selection, and solvers with consistently weak performance, which would introduce noise without contributing meaningful alternatives. The resulting portfolio is reported in Table 3.9.

Solver	Score	Optimal Count
gurobi-free	55.384518	38
pumpkin-free	58.543229	33
choco-solver__cp-sat_-free	60.808176	32
cplex-free	48.514529	30
choco-solver__cp_-free	58.404323	29
cp_optimizer-free	50.992682	26
izplus-free	58.672093	25
jacop-free	44.549443	25
sicstus_prolog-free	44.592608	24
scip-free	36.902766	20
highs-free	34.418506	18
cbc-free	22.615259	11

Table 3.9: Solvers retained after portfolio restriction, ordered by the number of optimal solutions (score equal to 1 as defined in Section 2.5).

We then evaluated the three previously best-performing LLM configurations (i.e., those achieving a positive closed gap in the full portfolio setting) on this restricted solver set. Results are reported in Table 3.10. In the "Single Score" evaluation, all configurations perform below the SBS of the restricted portfolio, which is now `choco-solver__cp-sat_-free` with a score of 60.808. The corresponding closed gaps are strictly negative.

Variant	Temperature	Single Score	Parallel Score	Closed Gap
Features + Solvers Description	0.7	55.843	73.042	-0.190
Features + Solvers Description	0.3	54.339	72.662	-0.254
Scripts + Problem Description + Solvers Description	default	49.776	74.736	-0.449

Table 3.10: Performance of the best LLM configurations on the restricted solver portfolio. Columns are computed as in Table 3.1.

A significant portion of the performance degradation is attributable to a systematic tendency of the LLM to select `cp_optimizer-free` for a large fraction of instances, despite its mid-range performance, placing itself as sixth within this restricted portfolio.

Regarding the "Parallel Score", comparison with `or-tools_cp-sat-par` is no longer meaningful, as its score (88.11) exceeds the virtual best solver (VBS) score achievable within the restricted portfolio (83.85). Therefore, even perfect selection over the reduced set would not reach that value. When compared against solvers whose performance is attainable within the restricted portfolio, the LLM-based variants remain below the second-best solver in the open category, `chuffed-free`, which achieves a score of 74.81.

Chapter 4

A FlatZinc Parser: fzn2nl

This chapter describes the rationale and development of `fzn2nl` [52], a tool designed to generate a deterministic natural language description of a constraint problem starting from its FlatZinc representation.

The first section outlines the motivations that led to the development of `fzn2nl`. The tool was conceived both as a supporting component of the present research and as an independent contribution. In particular, we discuss the limitations of raw FlatZinc code as input for LLM prompting, and motivate the need for a structured textual abstraction that preserves semantic content while improving interpretability.

The second section provides a concise overview of the MiniZinc compiler, with emphasis on the FlatZinc intermediate representation. We then describe the modifications introduced to the compiler infrastructure in order to avoid indirect suggestions for an LLM, in accordance with the design objectives presented earlier.

The third section details the internal architecture of `fzn2nl`. We describe the core data structures, the parsing procedure, and the mapping strategy used to translate variables, domains, constraints, and objective components into natural language. The exposition follows a running example to illustrate each transformation step and to clarify the correspondence between FlatZinc constructs and their textual representation.

The final section reports the experimental evaluation of LLM performance when prompted with the generated natural language descriptions. In order to assess the effectiveness of the textual abstraction introduced by `fzn2nl`.

4.1 Motivation

From the study explained in the course of Chapter 3 some important insights emerged. First and foremost, the limitation over token per request and over token per context window is a problem that we only managed to work around with the techniques explained in Section 2.6, before adding the use of a feature extractor (Section 3.4).

Another limitation in LLMs, as briefly explained in Section 1.2, is that they work only with

statistic prediction, based on their previous knowledge, this approach clearly creates gaps when the LLM is questioned with a completely new problem. To surpass this limitation, we needed a way to standardize all the problems so that an LLM can evaluate directly based on the problem characteristics.

The use of structured input, extracted from the problem scripts, proved to be a viable solution to these problems, showing the highest score yet. But while the results obtained with `mzn2feat` proved to be consistent, we still believed there could be a better way to formalize MiniZinc problems for an LLM given the portrayed limitations when confronting the LLM with a selected set of solver (Section 3.6).

The development of `fzn2nl` [52], was born from the idea to create a `FlatZinc` parser, that taken a file as input, produces a deterministic natural language description of the problem, exposing its main characteristics, in a way that is best suitable for an LLM.

LLMs encode vast amounts of pre-trained knowledge in their parameters, but updating them as real-world information evolves remains a challenge. Parametric knowledge of LLMs remains mostly static [54] after the pre-training stage, whereas knowledge in the world continues to change.

Even within the pre-training data, knowledge from recent years can conflict earlier knowledge. But, the auto-regressive training objective biases LLMs toward surfacing more frequent but not necessarily recent knowledge [55]. This problem surfaces primarily when the LLM is confronted with more indirect questions, given that updates in real world lead to nuanced and complex changings in model knowledge (Figure 4.1) [53].

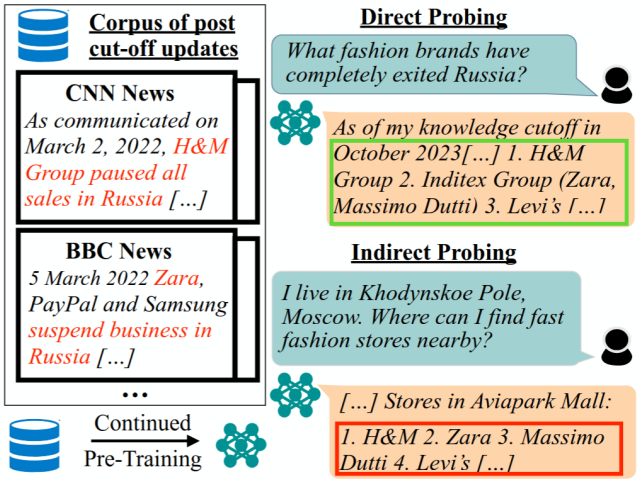

Figure 4.1: Example of LLM that is pre-trained on updated knowledge surfacing updates in the direct probing but failing under indirect probing settings (example image was taken from [53]).

In our setup, the LLM acts as the reasoning brain behind solver selection, so we need a way to give it a description of the problem that is as direct as possible, to actively avoid favoring "old" problems.

The most simple solution under those constraints is to create a deterministic description of

each problem, providing enough knowledge for reasonable choice while hiding most recognizable traits of well-known problems (retrievable from the code).

Large Language Models (LLMs) have demonstrated remarkable capabilities in natural language understanding and generation. However, they often struggle with complex reasoning tasks and are prone to hallucination. And while structured data, rich in logical and relational information, has the potential to enhance the reasoning abilities of LLMs. Still, its integration poses a challenge due to the risk of overwhelming LLMs with excessive tokens and irrelevant context information [56].

Those information lead to the development of `fzn2nl`, a parser that taken as input a `FlatZinc` file, directly translates its parameters to a structured natural language description of the given problem.

4.2 FlatZinc Compiler

In order to compile FlatZinc files, it is necessary to use the "MiniZinc Compiler" [57].

Constraint problems formulated in MiniZinc are solved by translating them to a simpler, solver-specific subset of MiniZinc, called FlatZinc, with the use of the "MiniZinc Compiler" [57]. The complexities in the translation arise from the need to simultaneously (a) unroll array comprehensions (and other loops), (b) replace predicate and function applications by their body, and (c) flatten expressions.

The translation algorithm generates a flat model equivalent to the original model as a global set of constraints. The translation uses full reification to create the model [59]. Common subexpression elimination is implemented using a technique similar to hash-consing in Lisp [58].

4.2.1 FlatZinc Limitation

As previously stated, Flatzinc is solver-specific, which is a clear limitation in our use case. FlatZinc solvers specify the set of global constraints they handle through dedicated propagators. When a given global constraint (e.g., `alldifferent` or `circuit`) is supported natively by the target solver, it is preserved in the compiled FlatZinc model and processed using the solver's specialized filtering algorithms. Otherwise, the MiniZinc compiler replaces the global constraint with an equivalent decomposition into more primitive constraints expressed in the solver's supported constraint language.

As a consequence, CP solvers such as Gecode [40] typically retain many global constraints in their high-level form, exploiting dedicated propagation mechanisms. In contrast, solvers based on alternative paradigms, such as linear programming or mixed-integer linear programming (e.g., Gurobi [60]), require these constraints to be reformulated as sets of linear constraints, thereby losing the original global structure in favor of a representation compatible with their underlying solving technology [61].

Given that, by translating a FlatZinc file with a specific solver, and serving its analysis to an LLM, we are indirectly pushing the LLM towards the solver used for translation, or one of the same category, therefore invalidating the reasoning process.

4.2.2 Custom Compiler

In order to solve the problem of solver-specific translation, we had to modify the MiniZinc compiler, to produce a pseudo-FlatZinc that is not directly dependant on solvers, at the cost of not being actually solvable.

As stated in Section 4.2.1, the difference between the use of one solver over the other is relative to the different propagation of global constraint.

So, in order to eliminate this distinction, we simply eliminated propagation as a whole, changing MiniZinc compiler code to completely avoid the substitution of predicates by their body. For example:

```
include "arg_max.mzn";
predicate fzn_maximum_arg_bool_opt(array [int] of var opt bool: x, var int: z) =
    let {
        array[index_set(x)] of var 0..2: dx = array1d(index_set(x), [(xi + 1) default 0 | xi in x]);
    } in maximum_arg(dx, z);
```

Became:

```
include "arg_max.mzn";
predicate fzn_maximum_arg_bool_opt(array [int] of var opt bool: x, var int: z);
```

Clearly, with this change, the FlatZinc is no longer directly solvable, but since our only need is for it to be non solver-specific and understandable when "explained" to an LLM, this is not a problem.

For the sake of explanation, we'll use the job shop problem as an example, as defined in Figure 1.1 and Figure 1.2. Using the compiler as defined in this section, the resulting FlatZinc is the one displayed in Figure 4.2.

4.3 Tool Functioning

The production of a viable natural language description with `fzn2nl` is made through a single-pass pipeline: first of all, the program takes a FlatZinc file as input, then this file is parsed, in order to extract variables, arrays, constraints, solve directive, and search annotations (when present) into an internal model object. The program then computes descriptive statistics on both variables and constraints, and reconstruct the objective structure. All of the parsed artifacts are mapped into readable sentences and grouped summaries. Finally, the system produces a report as output, consisting of problem description, variables description and constraint description.

```
array [1..2] of int: X_INTRODUCED_8_ = [1,-1];
var 0..7: X_INTRODUCED_1_;
var 2..9: X_INTRODUCED_2_;
var 0..7: X_INTRODUCED_3_;
var 3..10: X_INTRODUCED_4_;
var 7..14: end:: output_var;
var bool: X_INTRODUCED_10_ ::var_is_introduced :: is_defined_var;
var bool: X_INTRODUCED_12_ ::var_is_introduced :: is_defined_var;
var bool: X_INTRODUCED_14_ ::var_is_introduced :: is_defined_var;
var bool: X_INTRODUCED_15_ ::var_is_introduced :: is_defined_var;
array [1..4] of var int: X_INTRODUCED_0_:: output_array([1..4]) = [X_INTRODUCED_1_,X_INTRODUCED_2_,X_INTRODUCED_3_,X_INTRODUCED_4_];
constraint int_lin_le(X_INTRODUCED_8_,[X_INTRODUCED_1_,X_INTRODUCED_2_],-2);
constraint int_lin_le(X_INTRODUCED_8_,[X_INTRODUCED_2_,end],-5);
constraint bool_clause([X_INTRODUCED_10_,X_INTRODUCED_12_],[]);
constraint int_lin_le(X_INTRODUCED_8_,[X_INTRODUCED_3_,X_INTRODUCED_4_],-3);
constraint int_lin_le(X_INTRODUCED_8_,[X_INTRODUCED_4_,end],-4);
constraint bool_clause([X_INTRODUCED_14_,X_INTRODUCED_15_],[]);
constraint int_lin_le_reif(X_INTRODUCED_8_,[X_INTRODUCED_1_,X_INTRODUCED_3_],-2,X_INTRODUCED_10_):: defines_var(X_INTRODUCED_10_);
constraint int_lin_le_reif(X_INTRODUCED_8_,[X_INTRODUCED_3_,X_INTRODUCED_1_],-3,X_INTRODUCED_12_):: defines_var(X_INTRODUCED_12_);
constraint int_lin_le_reif(X_INTRODUCED_8_,[X_INTRODUCED_2_,X_INTRODUCED_4_],-5,X_INTRODUCED_14_):: defines_var(X_INTRODUCED_14_);
constraint int_lin_le_reif(X_INTRODUCED_8_,[X_INTRODUCED_4_,X_INTRODUCED_2_],-4,X_INTRODUCED_15_):: defines_var(X_INTRODUCED_15_);
solve  minimize end;
```

Figure 4.2: FlatZinc resulting from the compilation of the program in Figure 1.1 with the data in Figure 1.2, using the compiler defined in Section 4.2

4.3.1 Parser

The architecture of the implemented parser for this project, follows a modular design, separating data representation from syntactic extraction and higher-level semantic enrichment.

Core Data Structures

The central abstraction is the `FlatZincModel` class, which acts as a container for all extracted elements. It maintains dictionaries for scalar variables and arrays, a list of constraints, a map of definition dependencies, and metadata concerning the problem type, objective function, and search annotation.

Scalar variables are stored in a dictionary indexed by name. Each entry records the declared type (integer or Boolean), an optional domain, and an origin flag distinguishing user-declared variables from compiler-introduced ones. Domains are represented by an immutable `Domain` data class containing minimum and maximum bounds, together with a derived mean value.

Arrays are stored separately from scalar variables. For each array, the parser records element type, whether elements are decision variables, index range length (when statically inferable), and the list of items when explicitly provided.

Constraints are stored as structured dictionaries containing the constraint type, argument string, annotation string, reconstructed textual rendering, and any variables declared via `defines_var` annotations, preserving sufficient information for both textual reporting and dependency reconstruction.

Parsing Strategy

The parser operates in a single pass over the file contents, using regular expressions for high-level pattern detection and auxiliary scanning procedures for balanced parenthesis handling. The overall workflow can be described as follows.

First, scalar variable declarations are extracted using line-anchored regular expressions. The parser distinguishes between standard integer and Boolean declarations and explicit domain specifications, such as interval domains (e.g., `1..10`) or set domains (e.g., `{1,3,5}`). Domains are interpreted conservatively: when a set is provided, only its minimum and maximum values are retained.

Second, array declarations are parsed independently. The index range is analyzed to determine the array length when specified as a closed interval. If an initializer is present, its elements are collected in textual form. This design avoids premature semantic interpretation while preserving structural information.

Third, constraints are parsed using a hybrid approach. A regular expression identifies occurrences of the keyword `constraint` followed by a constraint symbol. Since FlatZinc arguments may contain nested parentheses, a dedicated balanced-call extractor is used to recover the full argument list safely. Trailing annotations are preserved, and any `defines_var` markers are recorded. A secondary pass constructs a definition map from variables to the constraints that define them.

Finally, the `solve` statement is parsed to determine whether the problem is a satisfaction or optimization instance. If a search annotation is present, a recursive routine interprets common patterns such as `int_search` and `seq_search`. The resulting structure encodes variable selection strategy, value selection strategy, completeness mode, and, in the case of sequential search, the list of phases.

A heuristic mechanism is implemented to distinguish user-defined variables from those introduced during compilation. The detection combines name-based patterns (e.g., identifiers containing `INTRODUCED`) with annotation-based indicators (e.g., `is_defined_var`). Although not formally guaranteed to be complete, this approach is robust across common FlatZinc toolchains and enables analyses restricted to original model variables.

Beyond syntactic parsing, the architecture includes lightweight analytical utilities. For example, a degree computation function estimates, for each scalar variable, the number of constraints in which it appears. This is achieved by tokenizing constraint argument strings and counting occurrences of known variable identifiers. While approximate, this metric provides a structural indicator of variable centrality in the constraint graph.

The architecture deliberately avoids constructing a full abstract syntax tree of the FlatZinc language. Instead, it adopts a pragmatic intermediate representation that captures structural information sufficient for meta-analysis and feature extraction. Balanced-parenthesis scanning is used selectively to overcome the limitations of purely regular-expression-based parsing, particularly in constraint arguments and search annotations.

Example of Parser Execution on a FlatZinc Instance

To better explain the functioning of the parser, we'll explain how it would behave when given the FlatZinc script defined in Figure 4.2 as input. The parser first processes scalar variable declarations. The following integer decision variables are identified in the lines 1-6:

- `X_INTRODUCED_1_` with domain $[0, 7]$,
- `X_INTRODUCED_2_` with domain $[2, 9]$,
- `X_INTRODUCED_3_` with domain $[0, 7]$,
- `X_INTRODUCED_4_` with domain $[3, 10]$,
- `end` with domain $[7, 14]$.

In addition, four Boolean variables are declared in the lines 7-10:

`X_INTRODUCED_10_`, `X_INTRODUCED_12_`, `X_INTRODUCED_14_`, `X_INTRODUCED_15_`.

All variables whose identifiers contain the substring `INTRODUCED` are classified as compiler-introduced by the heuristic detection mechanism. The Boolean variables are also annotated with `var_is_introduced` and `is_defined_var`, which reinforces this classification. The variable `end`, although not matching the naming pattern, is treated as user-defined since no introduction annotation is present.

Each integer domain expressed as an interval is converted into a `Domain` object storing the minimum and maximum bounds. Boolean variables are internally represented with domain $[0, 1]$.

Two array declarations are parsed.

The first, in line 1,

```
array [1..2] of int:  X_INTRODUCED_8_ = [1,-1];
```

is recognized as a fixed integer array of length 2. Its elements are stored as the literals 1 and -1. Since its name matches the introduction pattern, it is classified as compiler-introduced.

The second, in line 11,

```
array [1..4] of var int:  X_INTRODUCED_0_ = [...],
```

is recognized as an array of integer decision variables of length 4. The stored items correspond to the scalar variables

`X_INTRODUCED_1_`, `X_INTRODUCED_2_`, `X_INTRODUCED_3_`, `X_INTRODUCED_4_`.

The array is marked as containing decision variables (`is_var = true`) and as compiler-introduced.

The parser then extracts all constraint declarations. Each constraint is represented internally by its type, argument string, annotations, and any variables declared through `defines_var`.

Four linear inequality constraints of type `int_lin_le` are identified, in lines 12,13,15 and 16. Each uses the coefficient array `X_INTRODUCED_8_` and a pair of integer variables. For example,

```
int_lin_le(X_INTRODUCED_8_, [X_INTRODUCED_1_, X_INTRODUCED_2_], -2)
```

would result as:

```
{
 "type" : "int_lin_le",
 "args" : "X_INTRODUCED_8_,                                          ,
          [X_INTRODUCED_1_, X_INTRODUCED_2_],
          -2"
 "ann" : "", (empty string, since no trailing annotation is present)
 "text" : "int_lin_le(X_INTRODUCED_8_,                               ,
          [X_INTRODUCED_1_, X_INTRODUCED_2_],
          -2)"
 "defines" : [] (empty since the constraint does not contain defines_var annotation)
}
```

Two `bool_clause` constraints are parsed, respectively at lines 14 and 17. Each clause contains a list of Boolean variables and an empty negative literal list, corresponding to a disjunction over the given positive literals.

Four reified linear constraints of type `int_lin_le_reif` are also extracted, at lines 18-21. Each includes a trailing annotation of the form

```
::  defines_var(X_INTRODUCED_k_).
```

During post-processing, the parser builds a definition map associating each of the Boolean variables

```
X_INTRODUCED_10_, X_INTRODUCED_12_, X_INTRODUCED_14_, X_INTRODUCED_15_
```

with the corresponding reified constraint that defines it.

The final statement,

```
solve minimize end;
```

is parsed as an optimization directive. The model is classified as a minimization problem, with objective variable `end`. No search annotation is provided; therefore, the `search` field in the internal representation remains `None`.

4.3.2 Mapping to Natural Language

Once the FlatZinc instance has been parsed into the internal `FlatZincModel` representation, `fzn2nl` performs a deterministic mapping step that turns the extracted structures into a natural language report.

The mapping stage is implemented as a set of specialized formatters, each targeting a distinct part of the FlatZinc model. Rather than generating free-form text, each formatter follows fixed templates and uses controlled vocabularies defined in dedicated mapping tables.

Constraint descriptions and categorization FlatZinc constraints are reported primarily by type. For each constraint type, the mapper aggregates the total number of occurrences in the instance, and the constraint arity. Since FlatZinc arguments can contain arrays, the arity estimate expands known arrays of decision variables into their element variables when possible, and otherwise falls back to array length when statically inferable.

To attach a human-readable explanation to each constraint type, `fzn2nl` uses a layered description lookup, adding a description corpus extracted from MiniZinc/FlatZinc documentation and an optional categorized version of the same corpus, which allows constraint types to be grouped under conceptual families such as scheduling, graph, counting, and packing constraints.

The resolver is robust to common naming variations introduced by compilation and library predicates. For example, when a constraint is prefixed by `fzn_` or carries suffixes such as `_reif`, the resolver progressively strips prefix and trailing tokens to find a meaningful base description.

Search-annotation mapping When the `solve` item includes an explicit search annotation, the parser extracts it into a structured object (e.g., `int_search` or `seq_search`). The natural language mapper then translates the search parameters using controlled vocabularies. For sequential searches, the report enumerates phases and describes each phase with the same template.

When the internal model object is available, `fzn2nl` enrich search descriptions, mentioning a domain range and providing simple structural context such as whether the target is a scalar or an array and how many elements are involved.

Objective-function reconstruction FlatZinc explicitly specifies only the optimization direction (minimize or maximize) and an objective variable. The high-level expression for that variable is typically not present as a single expression, but it can be reconstructed in a best-effort way using compilation annotations such as `defines_var`. `fzn2nl` builds a shallow expression tree by following the chain of defining constraints for the objective variable and rendering common arithmetic constructs (e.g., `int_plus`, `int_minus`, ...). If reconstruction is not possible, the report falls back to describing the objective variable alone.

The objective expression is additionally rewritten into an abstract form, by replacing variable names with placeholders (e.g., $a, b, c, \ldots$). This preserves the algebraic structure of the

objective while hiding FlatZinc-specific naming patterns (such as `X_INTRODUCED_k_`). The abstract expression is also length-limited to ensure the report remains concise.

Variable statistics Finally, `fzn2nl` maps variables and arrays into summary statistics. The mapper separates user-declared and compiler-introduced variables, distinguishes between integer and Boolean variables, and summarizes known finite integer domains via simple bucketed distributions. Arrays of decision variables are treated as collections of element variables when their elements are explicitly listed; otherwise the mapper falls back to counting by array length when available.

Following the example, the resulting natural language description extracted from Figure 4.2, is:

Example output from `fzn2nl` with Figure 4.2 as input.

Problem:
This is a minimization problem. The objective is to minimize an objective variable with domain [7, 14] (size 8, mean 10.50) and degree 2. The objective function is in the form: minimize a. No explicit search strategy is specified.

Variables:
The model contains 9 variables (5 integer, 4 Boolean): 8 compiler-introduced and 1 user-introduced. Among 5 integer variables with known finite domains, 100.0% have domain size in [8, 8] (avg size 8.00).

Constraints:
Bool FlatZinc builtins: 1 type, 2 constraints (avg arity 2.00)
bool_clause: 2 constraints with average arity 2.00 (constrains $\bigvee_i as[i] \vee \bigvee_j \neg bs[j]$)
Integer FlatZinc builtins: 2 types, 8 constraints (avg arity 2.50)
int_lin_le: 4 constraints with average arity 2.00 (constrains $\sum$ as[i] * bs[i] $\leq$ c)
int_lin_le_reif: 4 constraints with average arity 3.00 (constrains $r \leftrightarrow (\sum$ as[i] * bs[i] $\leq$ c))

The constraints definition are written in LaTeX, in order to include mathematical expressions. In the example output we are showing the compiled version for better understanding.

4.4 Testing and Results

To evaluate the impact of `fzn2nl`, we tested two configurations: (i) the natural language report generated by `fzn2nl` alone, and (ii) the same report combined with solver descriptions as introduced in Section 3.3.2. Both configurations were evaluated over the restricted solver

portfolio defined in Section 3.6, and for all sampling temperatures considered in Section 3.5. Results are reported in Table 4.1.

Variant	Temperature	Single Score	Parallel Score	Closed Gap
fzn2nl	0.7	52.843	73.601	-0.318
fzn2nl	0.2	52.002	75.143	-0.354
fzn2nl + Solver Description	0.7	51.305	73.374	-0.384
fzn2nl	0.0	50.761	72.702	-0.407
fzn2nl + Solver Description	0.3	50.576	73.893	-0.415
fzn2nl + Solver Description	0.8	49.826	73.545	-0.447
fzn2nl	0.8	48.582	74.979	-0.500
fzn2nl	0.3	49.423	71.380	-0.464
fzn2nl + Solver Description	0.0	49.098	72.374	-0.478
fzn2nl + Solver Description	0.2	49.076	73.624	-0.479

Table 4.1: Sampling-temperature sweep for configurations including `fzn2nl` output, evaluated with `gpt-oss-120b`. Columns are computed as in Table 3.1.

As shown in Table 4.1, all configurations based on `fzn2nl` yield lower "Single Score" values compared to the best-performing `mzn2feat`-based configurations combined with solver descriptions. However, in terms of "Parallel Score", the configuration using only `fzn2nl` output with temperature 0.2 achieves the highest value observed within the restricted solver portfolio. This score exceeds that of all individual solvers in the open category except for `or-tools_cp-sat-par`, whose performance remains unattainable under the current portfolio.

4.4.1 Performance with GPT-5.2

To further assess the impact of model capacity on solver selection, we conducted additional experiments using `GPT-5.2` [62], a recent large-scale LLM reported to achieve state-of-the-art performance across multiple evaluation benchmarks, including GDPval [63]. The model is designed to support extended context handling, agentic tool use, and improved reasoning capabilities.

Testing was conducted by hand, through the official chatbot interface [64], which imposes daily usage limits [65]. Due to these constraints, a reduced evaluation protocol was adopted. Instead of prompting the model on all instances of each problem, a single representative instance per problem was selected. The solver recommendation obtained for that instance was then applied uniformly to all instances of the corresponding problem for scoring purposes.

As reported in Table 4.2, `GPT-5.2` does not improve the "Single Score" relative to the strongest previously tested configurations, and the resulting closed gap remains negative. The obtained value is lower than 3 of the `fzn2nl`-based variants reported in Table 4.1, as well as

Model	Single Score	Parallel Score	Closed Gap
GPT-5.2	51.539	76.630	-0.373

Table 4.2: Evaluation of GPT-5.2 under the reduced protocol (one instance per problem). Columns are computed as in Table 3.1.

below the 2 best-performing configurations evaluated on the restricted portfolio (Table 3.10).

In contrast, the "Parallel Score" achieved by GPT-5.2 under this limited setup is the highest observed within the restricted solver set. Although the evaluation protocol differs from the full-instance experiments, this result indicates that increased model capacity may positively affect parallel-selection performance. A systematic evaluation under uniform experimental conditions would be required to quantify this effect more precisely.

Appendix

Model	Total Score	Instances Covered	Average Score
gemini-2.0-flash	53.748	67	0.802
gemini-2.5-flash	69.426	86	0.807
gemini-2.5-flash-lite	69.040	85	0.812
gemini-2.5-pro	41.594	51	0.815
allam-2-7b	0.0	10	0.0
groq/compound	8.246	9	0.916
groq/compound-mini	29.623	35	0.846
llama-3.1-8b-instant	56.228	72	0.780
llama-3.3-70b-versatile	9.496	10	0.949
meta-llama/llama-4-maverick-17b-128e-instruct	63.548	72	0.882
meta-llama/llama-4-scout-17b-16e-instruct	57.814	69	0.837
meta-llama/llama-guard-4-12b	0.0	77	0.0
moonshotai/kimi-k2-instruct	65.816	75	0.877
moonshotai/kimi-k2-instruct-0905	66.186	75	0.882
openai/gpt-oss-120b	64.508	74	0.871
openai/gpt-oss-20b	63.328	75	0.844
qwen/qwen3-32b	48.018	65	0.738

Table 4.3: Initial tests giving plain scripts to all the LLMs, In this table: column "Model" contains the names of each tested LLM, “Total Score” is the equivalent of “Parallel Score” as explained in Table 3.1, “Instances Covered” contains the amount of instances that gave a viable result so the ones that did not exceed the token limitations portrayed in Table 2.1 and Table 2.2, “Average Score” represents $\frac{TotalScore}{InstancesCovered}$ to show the solver performance in the evaluated instances.

Model	Total Score	Instances Covered	Average Score
gemini-2.5-flash-lite	64.363	85	0.794
gemini-2.5-flash	60.962	85	0.734
moonshotai/kimi-k2-instruct-0905	59.680	75	0.817
moonshotai/kimi-k2-instruct	58.609	75	0.837
openai/gpt-oss-120b	58.166	74	0.796
openai/gpt-oss-20b	57.154	75	0.828
meta-llama/llama-4-maverick-17b-128e-instruct	56.297	72	0.804
meta-llama/llama-4-scout-17b-16e-instruct	54.305	69	0.798
gemini-2.0-flash	43.412	67	0.700
qwen/qwen3-32b	42.116	65	0.779
gemini-2.5-pro	37.640	51	0.738
llama-3.1-8b-instant	36.082	72	0.546
groq/compound-mini	25.422	35	0.726
llama-3.3-70b-versatile	7.454	10	0.745
groq/compound	5.197	9	0.577
allam-2-7b	0.0	4	0.0

Table 4.4: Initial tests giving plain scripts to all the LLMs, in this table "Total Score" is the equivalent of "Single Score" as explained in Table 3.1. "Instances Covered" and "Average Score" are calculated the same way as in Table 4.3

model	InstancesCovered	AS	SBS	VBS	ClosedGap
gemini-2.5-flash-lite	85	64.363	76.964	89.000	-1.047
gemini-2.5-flash	85	60.962	76.964	89.000	-1.330
moonshotai/kimi-k2-instruct-0905	75	59.680	76.964	89.000	-1.436
moonshotai/kimi-k2-instruct	75	58.609	76.964	89.000	-1.525
openai/gpt-oss-120b	74	58.166	76.964	89.000	-1.562
openai/gpt-oss-20b	75	57.154	76.964	89.000	-1.646
meta-llama/llama-4-maverick-17b-128e-instruct	72	56.297	76.964	89.000	-1.717
meta-llama/llama-4-scout-17b-16e-instruct	69	54.305	76.964	89.000	-1.883
gemini-2.0-flash	67	43.413	76.964	89.000	-2.788
qwen/qwen3-32b	65	42.117	76.964	89.000	-2.895
gemini-2.5-pro	51	37.641	76.964	89.000	-3.267
llama-3.1-8b-instant	72	36.082	76.964	89.000	-3.397
groq/compound-mini	35	25.423	76.964	89.000	-4.282
llama-3.3-70b-versatile	10	7.455	76.964	89.000	-5.775
groq/compound	9	5.197	76.964	89.000	-5.963

Table 4.5: Initial tests giving plain scripts to all the LLMs, in this table: "Instances Covered" is the same as in Table 4.3, "AS" is the same as the "Single Score" explained in Table 3.1, "SBS" displays the sum of all the scores obtained by the single best solver (namely, `or-tools_cp-sat-free`) in every instance, and "VBS" displays the score obtained with the use of an ipothetical virtual best solver, giving the maximum obtainable score on every instance. Finally, "Closed Gap" is calculated as in Table 3.1

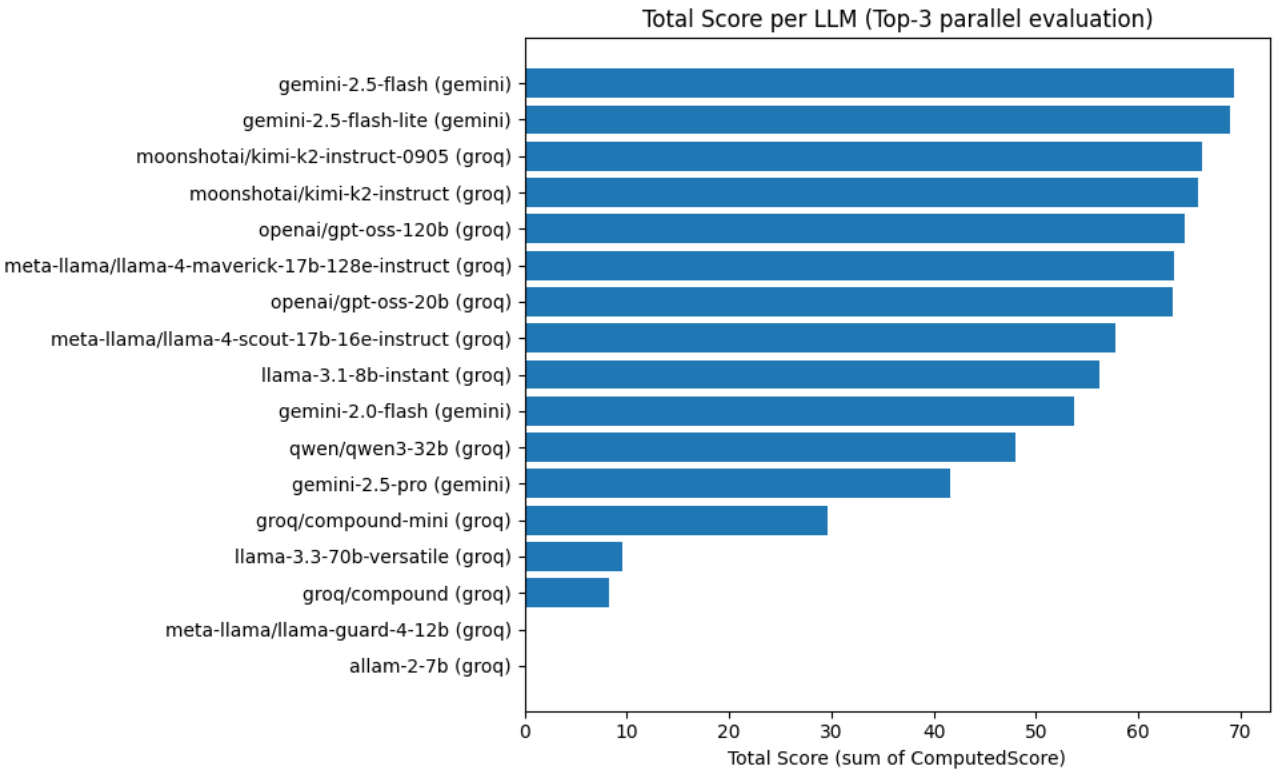

Figure 4.3: Scores obtained by all of the considered LLM from parallel-solver evaluation, "Total Score" is the equivalent of "Single Score" as explained in Table 3.1.

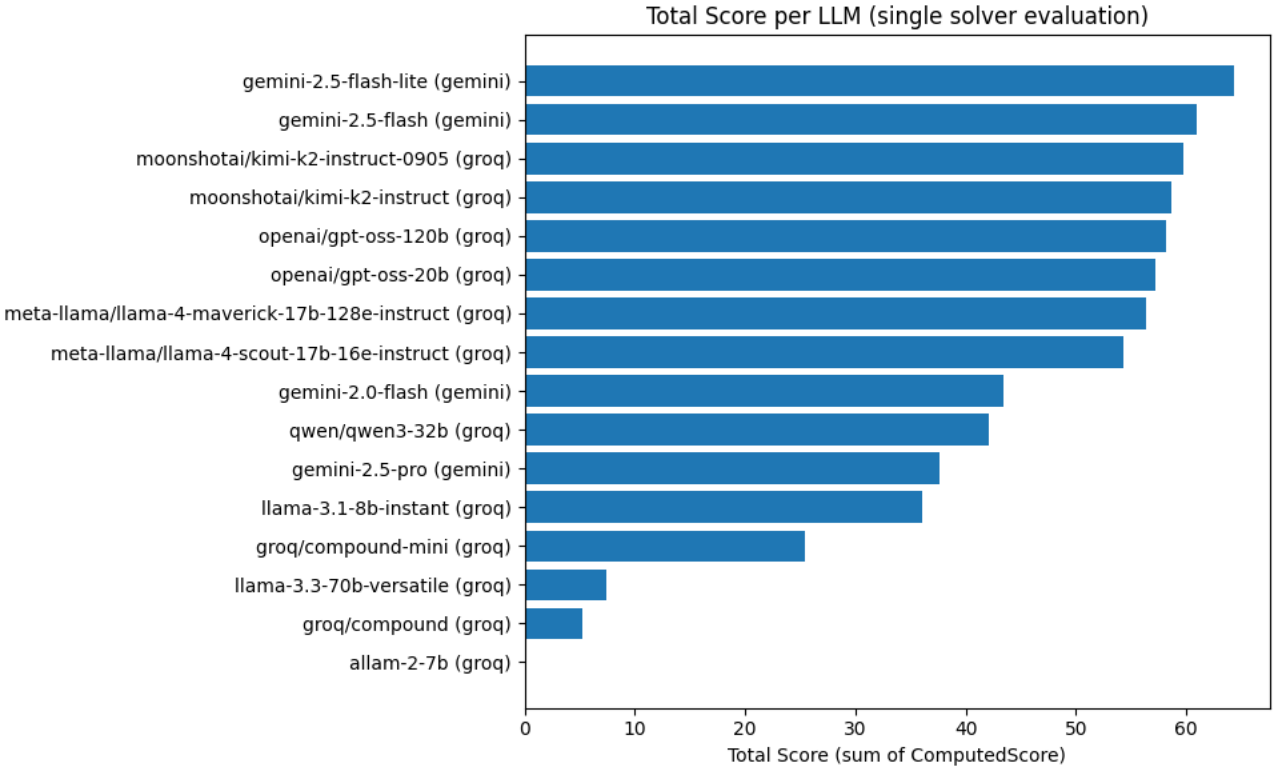

Figure 4.4: Scores obtained by all of the considered LLM from single-solver evaluation, "Total Score" is the equivalent of "Parallel Score" as explained in Table 3.1

Model	Total Score	Instances Covered	Average Score
gemini-2.5-flash	79.105	100	0.791
gemini-2.5-flash-lite	80.740	100	0.807
moonshotai/kimi-k2-instruct	83.268	100	0.832
moonshotai/kimi-k2-instruct-0905	82.656	100	0.826
openai/gpt-oss-120b	82.226	100	0.822

Table 4.6: Tests on sanitized scripts given to the 5 best performing LLMs, parallel-solver evaluation. Columns are calculated as in Table 4.3

Model	Total Score	Instances Covered	Average Score
openai/gpt-oss-120b	74.488	100	0.744
moonshotai/kimi-k2-instruct-0905	71.622	100	0.753
moonshotai/kimi-k2-instruct	70.939	100	0.723
gemini-2.5-flash-lite	70.145	100	0.738
gemini-2.5-flash	69.763	98	0.742

Table 4.7: Tests on sanitized scripts given to the 5 best performing LLMs, single-solver evaluation. Columns are calculated as in Table 4.4

Model	Instances Covered	AS	SBS	VBS	Closed Gap
openai/gpt-oss-120b	100	74.488	76.964	89.0	-0.205
moonshotai/kimi-k2-instruct-0905	100	71.622	76.964	89.0	-0.443
moonshotai/kimi-k2-instruct	100	70.939	76.964	89.0	-0.500
gemini-2.5-flash-lite	100	70.145	76.964	89.0	-0.566
gemini-2.5-flash	98	69.763	76.964	89.0	-0.598

Table 4.8: Tests on sanitized scripts given to the 5 best performing LLMs, closed gap. Columns are calculated as in Table 4.5

Model	Total Score	Instances Covered	Average Score
gemini-2.5-flash	59.154	75	0.788
gemini-2.5-flash-lite	82.620	100	0.826
moonshotai/kimi-k2-instruct	81.889	100	0.818
moonshotai/kimi-k2-instruct-0905	83.182	100	0.831
openai/gpt-oss-120b	83.249	100	0.832

Table 4.9: Test on sanitized scripts combined with textual problem description, parallel-solver evaluation. Columns are calculated as in Table 4.3

Model	Total Score	Instances Covered	Average Score
moonshotai/kimi-k2-instruct-0905	72.605	100	0.748
openai/gpt-oss-120b	70.740	100	0.721
gemini-2.5-flash-lite	69.042	100	0.719
moonshotai/kimi-k2-instruct	67.554	100	0.718
gemini-2.5-flash	49.896	73	0.723

Table 4.10: Test on sanitized scripts combined with textual problem description, single-solver evaluation. Columns are calculated as in Table 4.4

Model	Instances Covered	AS	SBS	VBS	Closed Gap
moonshotai/kimi-k2-instruct-0905	100	72.605	76.964	89.0	-0.362
openai/gpt-oss-120b	100	70.740	76.964	89.0	-0.517
gemini-2.5-flash-lite	100	69.042	76.964	89.0	-0.658
moonshotai/kimi-k2-instruct	100	67.554	76.964	89.0	-0.781
gemini-2.5-flash	73	49.896	76.964	89.0	-2.248

Table 4.11: Test on sanitized scripts combined with textual problem description, closed gap. Columns are calculated as in Table 4.5

Solver	Description
chuffed-free	A free, high-performance CP solver leveraging Lazy Clause Generation to combine SAT-like learning with traditional CP techniques, especially suited for rich scheduling and routing tasks.
cbc-free	COIN-OR Branch-and-Cut (CBC) is an open-source MILP solver specializing in integer and linear optimization, widely used as a robust and lightweight alternative to commercial engines.

Table 4.12: Examples of solver description for two solvers, namely COIN-OR Branch-and-Cut (CBC) [50], and Chuffed [51].

Model	Total Score	Instances Covered	Average Score
gemini-2.5-flash	83.137	100	0.831
gemini-2.5-flash-lite	77.746	100	0.777
moonshotai/kimi-k2-instruct	82.236	100	0.822
moonshotai/kimi-k2-instruct-0905	82.488	100	0.824
openai/gpt-oss-120b	78.799	100	0.787

Table 4.13: Test with sanitized scripts combined with solvers description in a multi turn setup, parallel-solver evaluation. Columns are calculated as in Table 4.3

Model	Total Score	Instances Covered	Average Score
moonshotai/kimi-k2-instruct	76.964	100	0.769
moonshotai/kimi-k2-instruct-0905	76.964	100	0.769
gemini-2.5-flash	71.686	100	0.716
openai/gpt-oss-120b	70.974	100	0.716
gemini-2.5-flash-lite	54.713	100	0.552

Table 4.14: Test with sanitized scripts combined with solvers description in a multi turn setup, single-solver evaluation. Columns are calculated as in Table 4.4

Model	Instances Covered	AS	SBS	VBS	Closed Gap
moonshotai/kimi-k2-instruct	100	76.964	76.964	89.0	0.0
moonshotai/kimi-k2-instruct-0905	100	76.964	76.964	89.0	0.0
gemini-2.5-flash	100	71.686	76.964	89.0	-0.438
openai/gpt-oss-120b	100	70.974	76.964	89.0	-0.497
gemini-2.5-flash-lite	100	54.713	76.964	89.0	-1.848

Table 4.15: Test with sanitized scripts combined with solvers description in a multi turn setup, closed gap. Columns are calculated as in Table 4.5

Model	Total Score	Instances Covered	Average Score
gemini-2.5-flash	83.650	100	0.836
gemini-2.5-flash-lite	78.147	100	0.781
moonshotai/kimi-k2-instruct	80.417	99	0.812
moonshotai/kimi-k2-instruct-0905	82.218	100	0.822
openai/gpt-oss-120b	80.295	100	0.802

Table 4.16: Test with sanitized scripts combined with both solvers description, and problem description in a multi turn setup, parallel-solver evaluation. Columns are calculated as in Table 4.3

Model	Total Score	Instances Covered	Average Score
openai/gpt-oss-120b	77.260	100	0.780
moonshotai/kimi-k2-instruct-0905	76.964	100	0.769
moonshotai/kimi-k2-instruct	74.464	99	0.752
gemini-2.5-flash	73.551	100	0.750
gemini-2.5-flash-lite	63.062	100	0.630

Table 4.17: Test with sanitized scripts combined with both solvers description, and problem description in a multi turn setup, single-solver evaluation. Columns are calculated as in Table 4.4

Model	Instances Covered	AS	SBS	VBS	Closed Gap
`openai/gpt-oss-120b`	100	77.260	76.964	89.0	0.024
`moonshotai/kimi-k2-instruct-0905`	100	76.964	76.964	89.0	0.0
`moonshotai/kimi-k2-instruct`	99	74.464	76.964	89.0	-0.207
`gemini-2.5-flash`	100	73.551	76.964	89.0	-0.283
`gemini-2.5-flash-lite`	100	63.062	76.964	89.0	-1.155

Table 4.18: Test with sanitized scripts combined with both solvers description, and problem description in a multi turn setup, closed gap. Columns are calculated as in Table 4.5

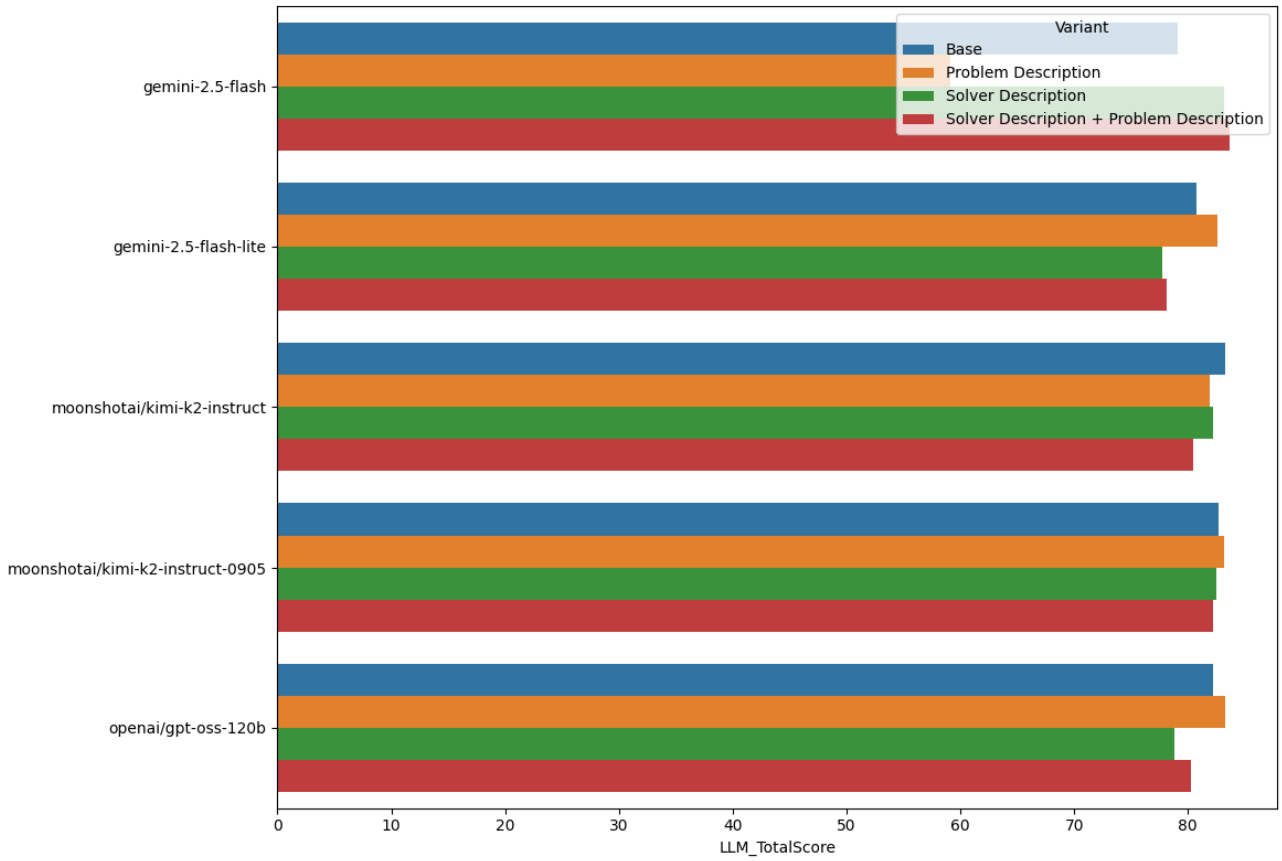

Figure 4.5: Histogram displaying the performances of all the solver variants in "Parallel Score" evaluation, as calculated in Table 3.1.

Type	Solver	Total Score
Solver	or-tools_cp-sat-par	88.117
LLM Variant	gemini-2.5-flash (Solvers Description + Problem Description)	83.650
LLM Variant	moonshotai/kimi-k2-instruct	83.268
LLM Variant	openai/gpt-oss-120b (Problem Description)	83.249
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Problem Description)	83.182
LLM Variant	gemini-2.5-flash (Solvers Description)	83.137
LLM Variant	moonshotai/kimi-k2-instruct-0905	82.656
LLM Variant	gemini-2.5-flash-lite (Problem Description)	82.620
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Solvers Description)	82.488
LLM Variant	moonshotai/kimi-k2-instruct (Solvers Description)	82.236
LLM Variant	openai/gpt-oss-120b	82.226
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Solvers Description + Problem Description)	82.218
LLM Variant	moonshotai/kimi-k2-instruct (Problem Description)	81.889
LLM Variant	gemini-2.5-flash-lite	80.740
LLM Variant	openai/gpt-oss-120b (Solvers Description + Problem Description)	80.295
LLM Variant	gemini-2.5-flash	79.105
LLM Variant	openai/gpt-oss-120b (Solvers Description)	78.799
LLM Variant	gemini-2.5-flash-lite (Solvers Description + Problem Description)	78.147
LLM Variant	gemini-2.5-flash-lite (Solvers Description)	77.746
Solver	chuffed-free	74.819
Solver	picatsat-free	70.647
Solver	huub-free	68.784
Solver	gurobi-par	61.081
Solver	cplex-par	60.301
Solver	choco-solver__cp_-par	59.365
Solver	izplus-par	58.216
Solver	pumpkin-free	57.681
Solver	choco-solver__cp-sat_-par	56.896
Solver	gecode-par	54.283
Solver	cp_optimizer-par	53.877
Solver	gecode_dexter-open	47.304
Solver	or-tools_cp-sat_ls-par	46.292
Solver	jacop-free	44.373
Solver	sicstus_prolog-free	43.548
Solver	yuck-par	38.321
Solver	scip-par	36.675
Solver	highs-par	33.598
Solver	cbc-par	26.334
Solver	atlantis-free	1.555

Table 4.19: Table displaying all the different LLM variants results from parallel-solver evaluation, "Total Score" is calculated as "Parallel Score" in Table 3.1, and compared to all of the single solvers in open category of the MiniZinc Challenge [31]

Type	Solver	TotalScore
LLM Variant	openai/gpt-oss-120b (Solvers Description + Problem Description)	**77.260**
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Solvers Description + Problem Description)	76.964
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Solvers Description)	76.964
LLM Variant	moonshotai/kimi-k2-instruct (Solvers Description)	76.964
Solver	or-tools_cp-sat-free	76.964
LLM Variant	openai/gpt-oss-120b (Simple)	74.488
LLM Variant	moonshotai/kimi-k2-instruct (Solvers Description + Problem Description)	74.464
Solver	chuffed-free	74.456
LLM Variant	gemini-2.5-flash (Solvers Description + Problem Description)	73.551
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Problem Description)	72.605
LLM Variant	gemini-2.5-flash (Solvers Description)	71.686
LLM Variant	moonshotai/kimi-k2-instruct-0905 (Simple)	71.622
LLM Variant	openai/gpt-oss-120b (Solvers Description)	70.974
LLM Variant	moonshotai/kimi-k2-instruct (Simple)	70.939
Solver	picatsat-free	70.933
LLM Variant	openai/gpt-oss-120b (Problem Description)	70.740
LLM Variant	gemini-2.5-flash-lite (Simple)	70.145
LLM Variant	gemini-2.5-flash (Simple)	69.763
LLM Variant	gemini-2.5-flash-lite (Problem Description)	69.042
Solver	huub-free	68.497
LLM Variant	moonshotai/kimi-k2-instruct (Problem Description)	67.554
LLM Variant	gemini-2.5-flash-lite (Solvers Description + Problem Description)	63.062
Solver	choco-solver__cp-sat_-free	60.808
Solver	izplus-free	58.672
Solver	pumpkin-free	58.543
Solver	choco-solver__cp_-free	58.404
Solver	gurobi-free	55.384
LLM Variant	gemini-2.5-flash-lite (Solvers Description)	54.713
Solver	cp_optimizer-free	50.992
Solver	gecode-fd	50.073
LLM Variant	gemini-2.5-flash (Problem Description)	49.896
Solver	cplex-free	48.514
Solver	sicstus_prolog-free	44.592
Solver	jacop-free	44.549
Solver	or-tools_cp-sat_ls-free	43.222
Solver	scip-free	36.902
Solver	yuck-free	34.715
Solver	highs-free	34.418
Solver	cbc-free	22.615
Solver	atlantis-free	1.5

Table 4.20: Table displaying all the first LLM variants results given single-solver evaluation, "Total Score" is calculated as "Single Score" in Table 3.1, and compared to all of the single solvers in free category of the MiniZinc Challenge [31]

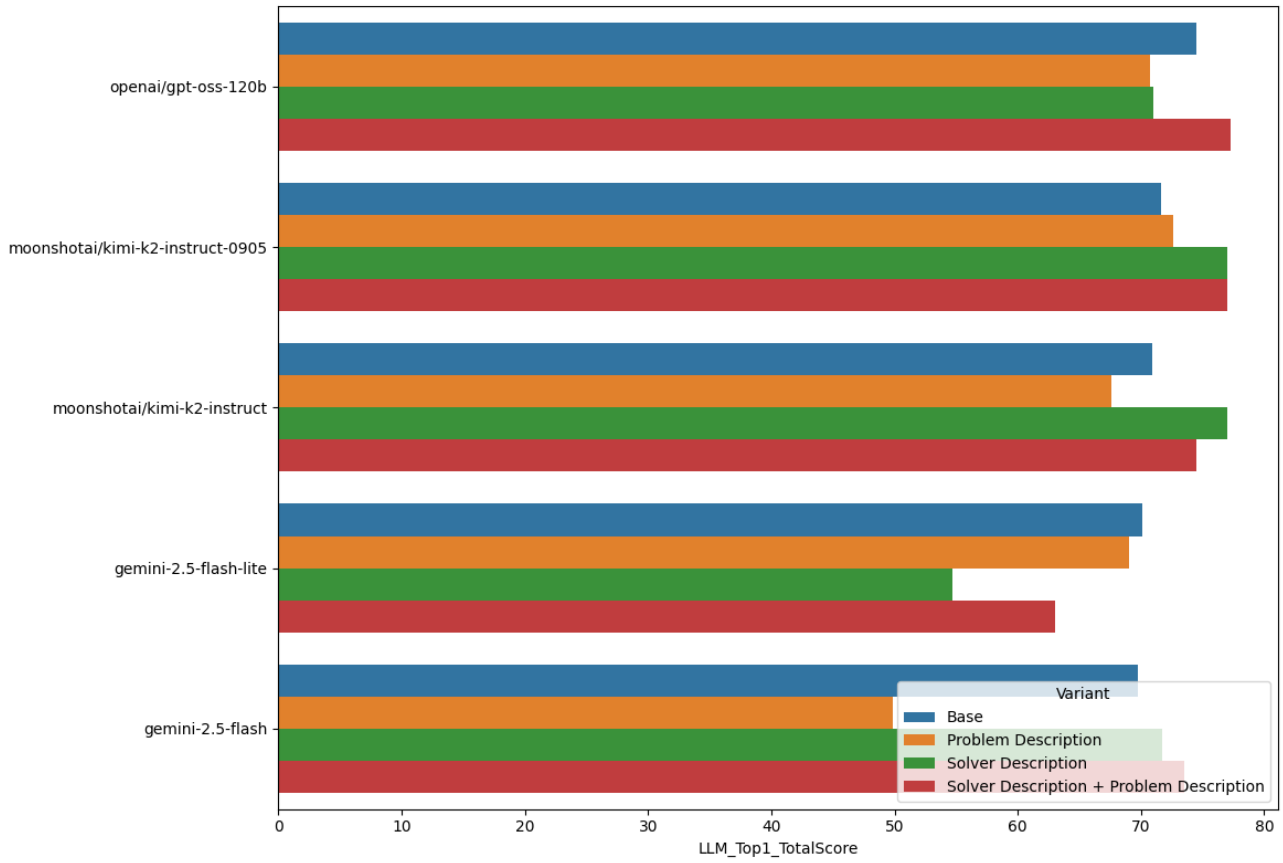

Figure 4.6: Histogram displaying the performances of all the solver variants in "Single Score" evaluation, "Parallel Score" evaluation and "Closed Gap" evaluation, as calculated in Table 3.1.

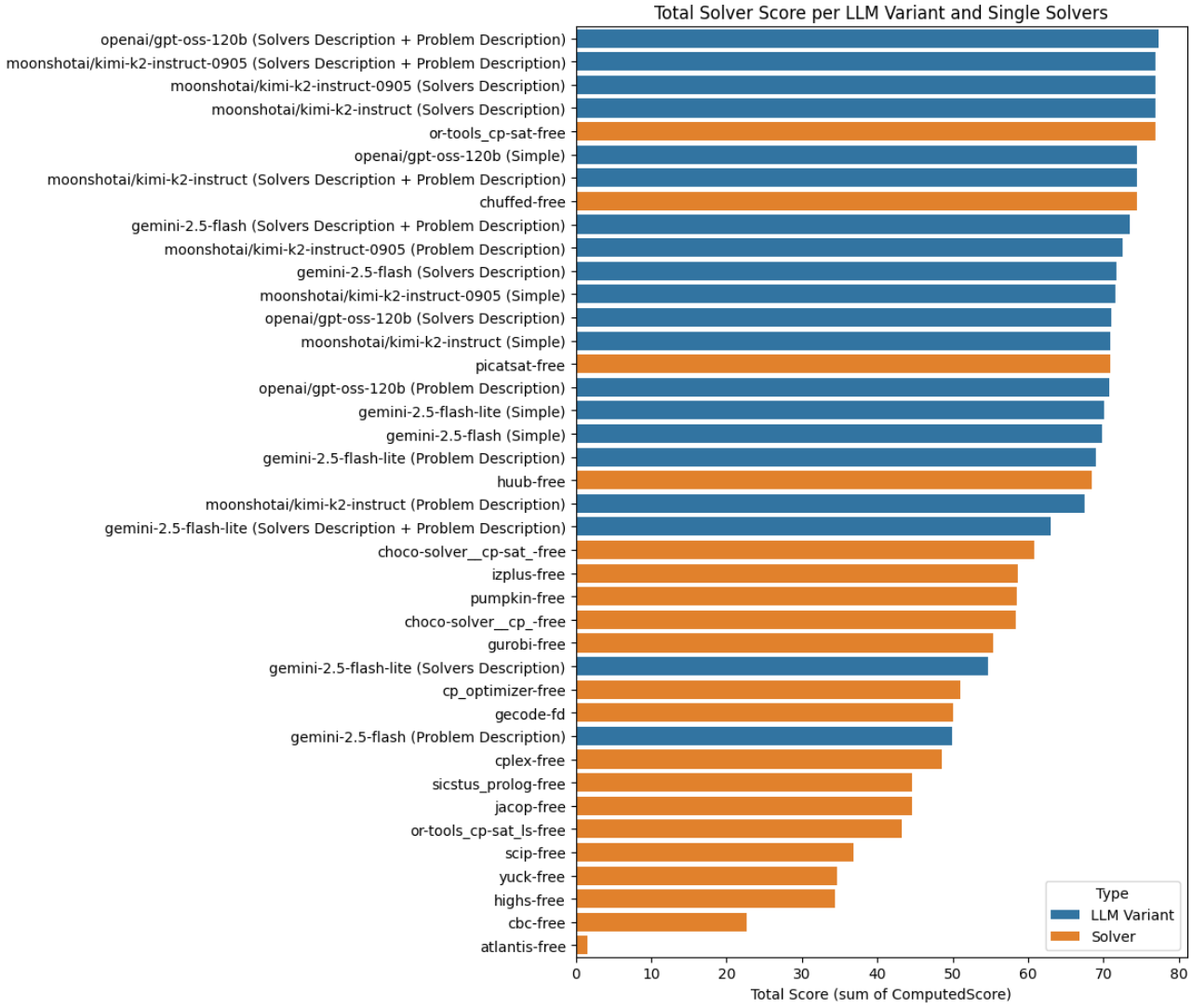

Figure 4.7: Histograms to visualize the difference between first LLM variants and single solvers free category performance, "Total Score" is the equivalent of "Single Score" as explained in Table 3.1.

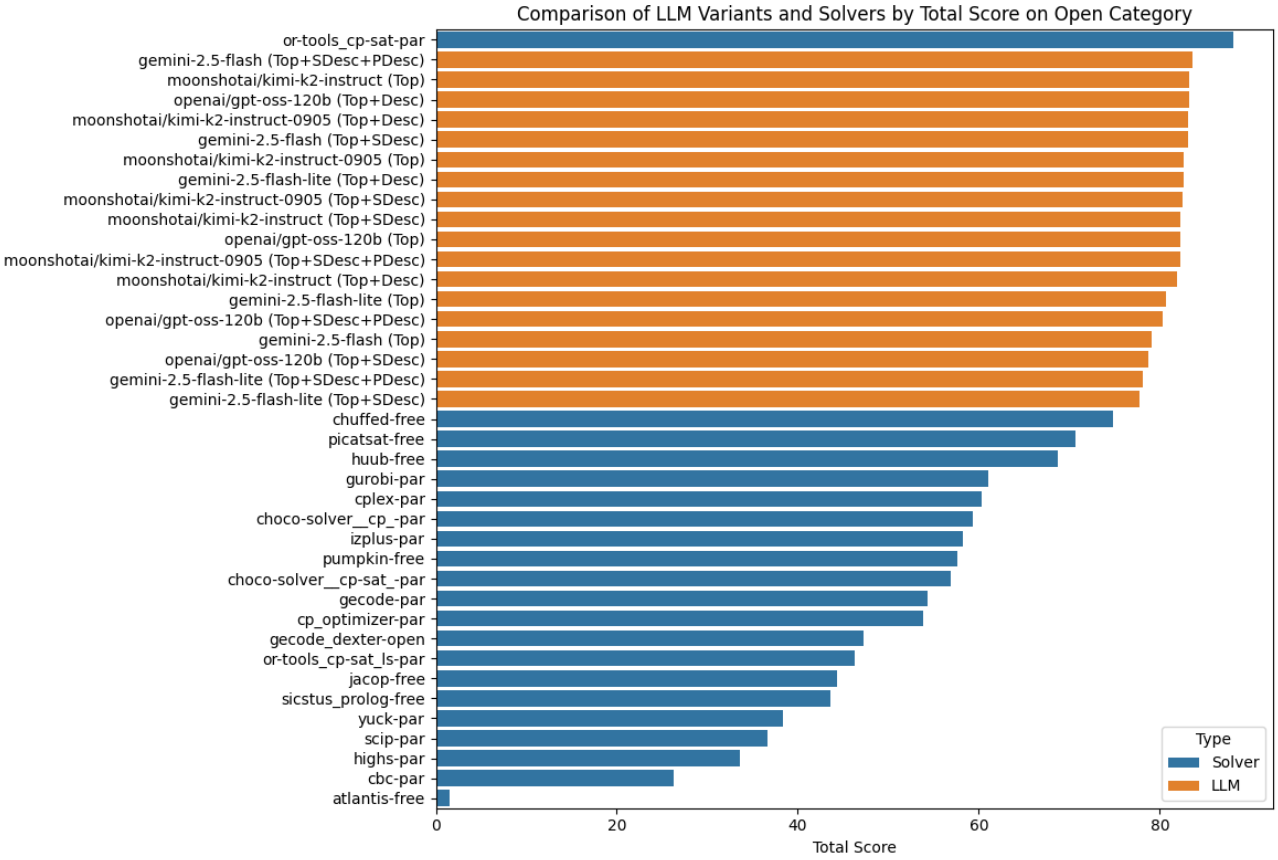

Figure 4.8: Histograms to visualize the difference between first LLM variants and single solvers from open category performance, "Total Score" is the equivalent of "Parallel Score" as explained in Table 3.1.

Category	Number	Description (summary)
Variables	27	Counts of variables (including constants, aliases, defined and introduced variables), ratios involving N_V and N_C, and statistics of domain size, degree, and domain/degree ratio.
Domains	18	Counts and ratios of variables by type (Boolean, integer, float, set) and constraints by type (Boolean, integer, float, set, array).
Constraints	27	Total N_C, ratios with N_V, annotation usage, and statistics of constraint domain, degree, and domain/degree ratio.
Global constraints	29	Total number and ratio of global constraints, plus counts per equivalence class of Gecode-supported globals.
Graphs	20	Statistics on structural properties of the constraint graph and variable graph (degree, clustering coefficient, diameter).
Solving	11	Information from the solve item, including labeled variables, goal type, and counts of search and heuristic annotations.
Objective	12	Domain, degree, and graph-related measures of the objective variable, including normalized and standardized forms relative to global domain and degree statistics.
Static total	144	Extracted from the FlatZinc model via syntactic and structural analysis.
Dynamic	11	Runtime indicators from short Gecode executions: solutions found, propagations, nodes, failures, search depth, memory usage, and timing measures for translation and feature extraction.

Table 4.21: The table shows a quick explanation of the features that can be extracted using the tool `mzn2feat` [39], divided per category, and counted.

IDENTIFIER	VALUE	DESCRIPTION
c_avg_deg_cons	3.11367	Average of the constraints degree
c_avg_dom_cons	20.3934	Average of the constraints domain
c_avg_domdeg_cons	5.85385	Average of the ratio constraints domain/degree
c_bounds_d	0	No of constraints using 'boundsD' annotation
c_bounds_r	0	No of constraints using 'boundsR' annotation
c_bounds_z	0	No of constraints using 'boundsZ' or 'bounds' annotation
c_cv_deg_cons	0.963493	Coefficient of Variation of constraints degree
c_cv_dom_cons	1.50739	Coefficient of Variation of constraints domain
c_cv_domdeg_cons	0.566571	Coefficient of Variation of the ratio constraints domain/degree
c_domain	0	No of constraints using 'domain' annotation
c_ent_deg_cons	1.60699	Entropy of constraints degree
c_ent_dom_cons	5.54008	Entropy of constraints domain
c_ent_domdeg_cons	2.58764	Entropy of the ratio constraints domain/degree
c_logprod_deg_cons	6021.66	Logarithm of the product of constraints degree
c_logprod_dom_cons	14249.3	Logarithm of the product of constraints domain
c_max_deg_cons	141	Maximum of the constraints degree
c_max_dom_cons	1724.64	Maximum of the constraints domain
c_max_domdeg_cons	12.2315	Maximum of the ratio constraints domain/degree
c_min_deg_cons	1	Minimum of the constraints degree
c_min_dom_cons	1	Minimum of the constraints domain
c_min_domdeg_cons	1	Minimum of the ratio constraints domain/degree
c_num_cons	3906	Total no of constraints
c_priority	0	No of constraints using 'priority' annotation
c_ratio_cons	1.49426	Ratio no of constraints / no of variables
c_sum_ari_cons	13031	Sum of constraints arity
c_sum_dom_cons	79656.6	Sum of constraints domain
c_sum_domdeg_cons	22865.2	Sum of the ratio constraints domain/degree
d_array_cons	1	No of array constraints
d_bool_cons	910	No of boolean constraints
d_bool_vars	1820	No of boolean variables
d_float_cons	0	No of float constraints
d_float_vars	0	No of float variables
d_int_cons	2591	No of integer constraints
d_int_vars	794	No of integer variables
d_ratio_array_cons	0.000256016	Ratio array constraints / total no of constraints
d_ratio_bool_cons	0.232975	Ratio boolean constraints / total no of constraints
d_ratio_bool_vars	0.696251	Ratio boolean variables / total no of variables
d_ratio_float_cons	0	Ratio float constraints / total no of constraints
d_ratio_float_vars	0	Ratio float variables / total no of variables
d_ratio_int_cons	0.663338	Ratio integer constraints / total no of constraints
d_ratio_int_vars	0.303749	Ratio integer variables / total no of variables
d_ratio_set_cons	0	Ratio set constraints / total no of constraints
d_ratio_set_vars	0	Ratio set variables / total no of variables
d_set_cons	0	No of set constraints
d_set_vars	0	No of set variables
gc_diff_globs	1	No of different global constraints
gc_global_cons	404	Total no of global constraints
gc_ratio_diff	0.00247525	Ratio different global constraints / no of global constraints
gc_ratio_globs	0.103431	Ratio no of global constraints / total no of constraints
o_deg	1	Degree of the objective variable
o_deg_avg	0.214932	Ratio degree of the objective variable / average of var degree
o_deg_cons	0.000256016	Ratio degree of the objective variable / number of constraints
o_deg_std	-0.442948	Standardization of the degree of the objective variable
o_dom	6685	Domain size of the objective variable
o_dom_avg	12.79	Ratio domain of the objective variable / average of var domain
o_dom_deg	6685	Ratio domain of the objective variable / degree of the obj var
o_dom_std	4.13516	Standardization of the domain of the objective variable

```
s_bool_search        0                  Number of 'bool_search' annotations
s_first_fail         1                  Number of 'int_search' annotations
s_goal               2                  Solve goal (1 = satisfy, 2 = minimize, 3 = maximize)
s_indomain_max       0                  Number of 'indomain_max' annotations
s_indomain_min       1                  Number of 'indomain_min' annotations
s_input_order        0                  Number of 'input_order' annotations
s_int_search         1                  Number of 'int_search' annotations
s_labeled_vars       1                  Number of variables to be assigned
s_other_val          0                  Number of other value search heuristics
s_other_var          0                  Number of other variable search heuristics
s_set_search         0                  Number of 'set_search' annotations
v_avg_deg_vars       4.65264            Average of the variables degree
v_avg_dom_vars       522.674            Average of the variables domain
v_avg_domdeg_vars    141.005            Average of the ratio variables domain/degree
v_cv_deg_vars        1.77237            Coefficient of Variation of variables degree
v_cv_dom_vars        2.85116            Coefficient of Variation of variables degree
v_cv_domdeg_vars     3.94885            Coefficient of Variation of the ratio variables domain/degree
v_def_vars           2334               Number of defined variables
v_ent_deg_vars       1.0029             Entropy of variables degree
v_ent_dom_vars       2.09955            Entropy of variables domain
v_ent_domdeg_vars    1.83264            Entropy of the ratio variables domain/degree
v_intro_vars         2753               Number of introduced variables
v_logprod_deg_vars   3665.47            Logarithm of the product of variables degree
v_logprod_dom_vars   6788.52            Logarithm of the product of variables domain
v_max_deg_vars       36                 Maximum of the variables degree
v_max_dom_vars       6685               Maximum of the variables domain
v_max_domdeg_vars    6685               Maximum of the ratio variables domain/degree
v_min_deg_vars       0                  Minimum of the variables degree
v_min_dom_vars       2                  Minimum of the variables domain
v_min_domdeg_vars    0.1                Minimum of the ratio variables domain/degree
v_num_aliases        700                Number of alias variables
v_num_consts         10                 Number of constant variables
v_num_vars           2614               Total no of variables variables
v_ratio_bounded      0.271614           Ratio (aliases + constants) / total no of variables
v_ratio_vars         0.669227           Ratio no of variables / no of constraints
v_sum_deg_vars       12162              Sum of variables degree
v_sum_dom_vars       1.36627e+06        Sum of variables domain
v_sum_domdeg_vars    368586             Sum of the ratio variables domain/degree
```

Listing 4.1: Example output using `mzn2feat` [39] pretty print option on `EchoSched.mzn`, instance `14-10-0-2_1`.

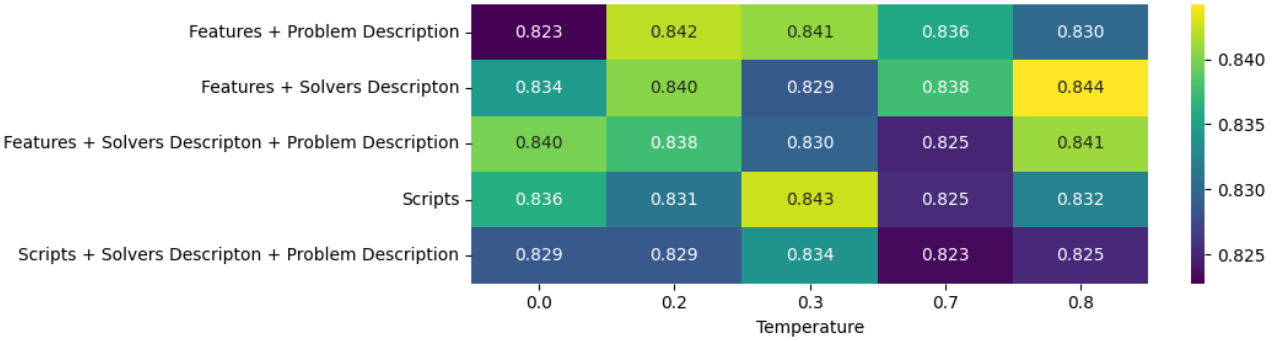

Figure 4.9: Heatmap displaying the performace of all the combinations of temperatures with the five best performing variants in "Parallel Score" evaluation calculated as in Table 3.1, all tests were performed using `gpt-oss-120b`.

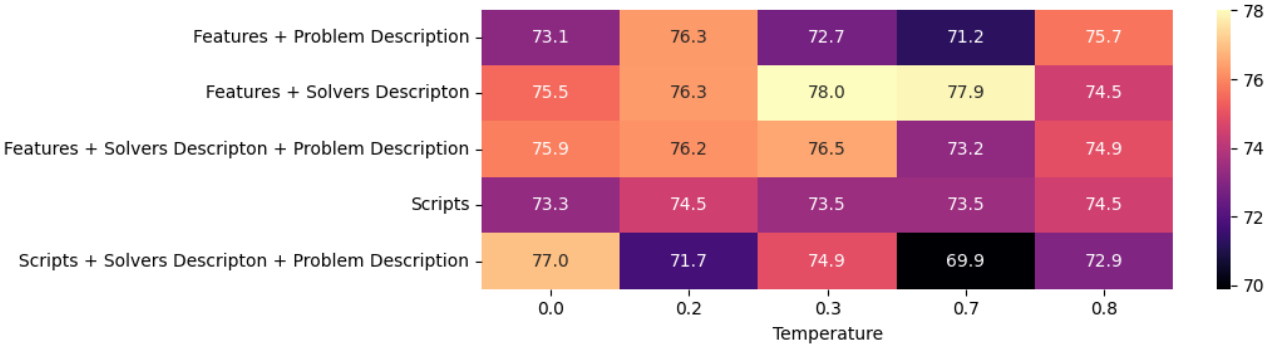

Figure 4.10: Heatmap displaying the performace of all the combinations of temperatures with the five best performing variants in "Single Score" evaluation calculated as in Table 3.1, all tests were performed using `gpt-oss-120b`.

Solver	Score	Optimal Count
or-tools_cp-sat-free	76.964375	55
picatsat-free	70.933307	53
chuffed-free	74.456334	52
huub-free	68.497987	48
gurobi-free	55.384518	38
pumpkin-free	58.543229	33
choco-solver__cp-sat_-free	60.808176	32
cplex-free	48.514529	30
choco-solver__cp_-free	58.404323	29
cp_optimizer-free	50.992682	26
izplus-free	58.672093	25
jacop-free	44.549443	25
sicstus_prolog-free	44.592608	24
scip-free	36.902766	20
highs-free	34.418506	18
cbc-free	22.615259	11
or-tools_cp-sat_ls-free	43.222031	7
yuck-free	34.715515	4
atlantis-free	1.500000	0

Table 4.22: Table displaying all the solvers from free catecory ordered by the number of optimal solutions they reach, intended as the number of instances where the given solver scores 1, following the scoring in Section 2.5.

Bibliography

[1] R. Oentaryo, S. D. Handoko, and H. C. Lau, "Algorithm Selection via Ranking", Proceedings of the AAAI Conference on Artificial Intelligence, vol. 29, no. 1, Feb. 2015, doi: https://doi.org/10.1609/aaai.v29i1.9466.

[2] R. Barták, "Constraint Programming: In Pursuit of the Holy Grail", Jan. 1999.

[3] E. C. Freuder, "In Pursuit of the Holy Grail", Constraints, vol. 2, no. 1, pp. 57-61, Apr. 1997, doi: https://doi.org/10.1023/a:1009749006768.

[4] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language", Springer eBooks, pp. 529-543, Oct. 2007, doi: https://doi.org/10.1007/978-3-540-74970-7_38.

[5] F. Chalumeau, I. Coulon, Q. Cappart, and L.-M. Rousseau, "SeaPearl: A Constraint Programming Solver guided by Reinforcement Learning", arXiv.org, Apr. 20, 2021. https://arxiv.org/abs/2102.09193

[6] Szabo, Zsuzsanna & Kovacs, Marta. "ON INTERIOR-POINT METHODS AND SIMPLEX METHOD IN LINEAR PROGRAMMING". Analele Stiintifice ale Universitatii Ovidius Constanta, Seria Matematica. 11. (2003).

[7] "HiGHS - High-performance parallel linear optimization software", Highs.dev, 2024. https://highs.dev/

[8] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning", Discrete Optimization, vol. 19, pp. 79-102, Feb. 2016, doi: https://doi.org/10.1016/j.disopt.2016.01.005.

[9] J. Wang and Y. Yang, "Branch-cut algorithm with fast search ability for the shortest branch-cuts based on modified GA", Journal of Modern Optics, vol. 66, no. 5, pp. 473-485, Nov. 2018, doi: https://doi.org/10.1080/09500340.2018.1548663.

[10] R. Amadini, M. Gabbrielli, and J. Mauro, "Why CP Portfolio Solvers Are (under)Utilized? Issues and Challenges", Lecture Notes in Computer Science, pp. 349-364, 2015, doi: `https://doi.org/10.1007/978-3-319-27436-2_21`.

[11] Gomes, C.P., Selman, B.: "Algorithm portfolios". Artif. Intell. 126(1-2), 43-62 (2001)

[12] Rice, J.R.: "The algorithm selection problem". Adv. Comput. 15, 65-118 (1976)

[13] Kotthoff, L.: "Algorithm selection for combinatorial search problems: a survey". AI Mag. 35(3), 48-60 (2014)

[14] Roberto Amadini, Simone Gazza. "From Portfolio Solvers to Agentic Solvers". LLM-Solve, Tias Guns; Serdar Kadioglu; Stefan Szeider; Dimos Tsouros, Aug 2025, Glasgow, United Kingdom, United Kingdom. pp.569 - 585, 10.5281/zenodo.17640236. hal-05446738

[15] IBM, "What are large language models (LLMs)?", Ibm.com, Nov. 02, 2023. https://www.ibm.com/think/topics/large-language-models

[16] A. Vaswani et al., "Attention Is All You Need", arXiv.org, 2017. https://arxiv.org/abs/1706.03762

[17] Y. Liu et al., "Understanding LLMs: A Comprehensive Overview from Training to Inference", arXiv (Cornell University), Jan. 2024, doi: https://doi.org/10.48550/arxiv.2401.02038.

[18] C. Shi et al., "A Thorough Examination of Decoding Methods in the Era of LLMs", arXiv (Cornell University), Feb. 2024, doi: https://doi.org/10.48550/arxiv.2402.06925.

[19] Markus Freitag and Yaser Al-Onaizan. 2017. "Beam search strategies for neural machine translation. In Proceedings of the First Workshop on Neural Machine Translation."

[20] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. "The curious case of neural text degeneration. In 8th International Conference on Learning Representations", ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.

[21] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications", arXiv (Cornell University), Feb. 2024, doi: https://doi.org/10.48550/arxiv.2402.07927.

[22] S. Rahman, S. Khan, and F. Porikli, "A Unified Approach for Conventional Zero-Shot, Generalized Zero-Shot, and Few-Shot Learning", IEEE Transactions on Image Processing, vol. 27, no. 11, pp. 5652-5667, Nov. 2018, doi: https://doi.org/10.1109/tip.2018.2861573.

[23] Farhan Nadim Iqbal, "A BRIEF INTRODUCTION TO APPLICATION PROGRAMMING INTERFACE (API)", zenodo.org, doi: https://doi.org/10.5281/zenodo.10198423.

[24] "Gemini API Docs and Reference", Google AI for Developers. https://ai.google.dev/gemini-api/docs (accessed Dec. 10, 2025).

[25] "API versions explained", Google AI for Developers, 2025. https://ai.google.dev/gemini-api/docs/api-versions (accessed Dec. 11, 2025).

[26] "GroqCloud", Groq.com, 2024. https://console.groq.com/docs/overview (accessed Dec. 10, 2025).

[27] "Rate Limits - GroqDocs", GroqDocs, 2025. https://console.groq.com/docs/rate-limits (accessed Dec. 10, 2025).

[28] "Rate limits", Google AI for Developers, 2025. https://ai.google.dev/gemini-api/docs/rate-limits (accessed Dec. 10, 2025).

[29] Kelly Hong, Anton Troynikov, Jeff Huber, "Context Rot: How Increasing Input Tokens Impacts LLM Performance", Trychroma.com, 2025. https://research.trychroma.com/context-rot?ref=blog.promptlayer.com (accessed Dec. 11, 2025).

[30] "MiniZinc - List of Problems and Globals used in the MiniZinc Challenge", Minizinc.org, 2025. https://www.minizinc.org/challenge/globals/ (accessed Dec. 11, 2025).

[31] "MiniZinc - Challenge 2025 Results", Minizinc.org, 2025. https://www.minizinc.org/challenge/2025/results/ (accessed Dec. 11, 2025).

[32] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, "The MiniZinc Challenge 2008-2013", AI Magazine, vol. 35, no. 2, p. 55, Jun. 2014, doi: https://doi.org/10.1609/aimag.v35i2.2539.

[33] R. Amadini, Maurizio Gabbrielli, T. Liu, and J. Mauro, "On the Evaluation of (Meta-)solver Approaches", Journal of Artificial Intelligence Research, vol. 76, pp. 705-719, Mar. 2023, doi: https://doi.org/10.1613/jair.1.14102.

[34] R. Amadini, Maurizio Gabbrielli, and J. Mauro, "Portfolio Approaches for Constraint Optimization Problems", Lecture notes in computer science, pp. 21-35, Jan. 2014, doi: https://doi.org/10.1007/978-3-319-09584-4_3.

[35] Lindauer, M., van Rijn, J. N., & Kotthoff, L. (2019). "The algorithm selection competitions" 2015 and 2017.Artificial Intelligence,272, 86-100.

[36] Shi, F., Chen, X., Misra, K., Scales, N., Dohan, D., Chi, E., Schärli, N., and Zhou, D. (2023). "Large Language Models Can Be Easily Distracted by Irrelevant Context" arXiv preprint arXiv:2302.00093. doi: https://doi.org/10.48550/arXiv.2302.00093

[37] Sumanth P, "Agentic Prompt Engineering: A Deep Dive into LLM Roles and Role-Based Formatting", Clarifai.com, Jul. 2025. https://www.clarifai.com/blog/agentic-prompt-engineering#title_1 (accessed Jan. 03, 2026).

[38] X. Lin et al., "LLM-based Agents Suffer from Hallucinations: A Survey of Taxonomy, Methods, and Directions", arXiv (Cornell University), Sep. 2025, doi: https://doi.org/10.48550/arxiv.2509.18970.

[39] R. Amadini, M. Gabbrielli, and J. Mauro. "An Enhanced Features Extractor for a Portfolio of Constraint Solvers". In SAC, 2014.

[40] M. Morara, J. Mauro, and M. Gabbrielli, "Solving XCSP problems by using Gecode", arXiv (Cornell University), Dec. 2011, doi: https://doi.org/10.48550/arxiv.1112.6096.

[41] D. Guo, "Development of Compiler Based on Flex and Bison", Ordnance Industry Automation, Jan. 2004.

[42] R. Amadini, and Peter J. Stuckey. "Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers". In CP, 2014.

[43] R. Amadini, M. Gabbrielli, and J. Mauro. "SUNNY-CP: a Sequential CP Portfolio Solver". In SAC, 2015.

[44] R. Amadini, M. Gabbrielli, and J. Mauro. "A Multicore Tool for Constraint Solving". In IJCAI, 2015.

[45] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines", Cognitive Science, vol. 9, no. 1, pp. 147-169, Jan. 1985, doi: https://doi.org/10.1016/S0364-0213(85)80012-4.

[46] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network", arXiv.org, Mar. 09, 2015. http://arxiv.org/abs/1503.02531

[47] P.-H. Wang et al., "Contextual Temperature for Language Modeling", arXiv (Cornell University), Jan. 2020, doi: https://doi.org/10.48550/arxiv.2012.13575.

[48] R. Wang, H. Wang, F. Mi, Y. Chen, R. Xu, and K.-F. Wong, "Self-Critique Prompting with Large Language Models for Inductive Instructions", arXiv.org, May 23, 2023. https://arxiv.org/abs/2305.13733 (accessed Mar. 05, 2024).

[49] "Prompt Basics - GroqDocs", GroqDocs, 2026. https://console.groq.com/docs/prompting

[50] "Cbc", GitHub, May 24, 2022. https://github.com/coin-or/Cbc

[51] "Chuffed, a lazy clause generation solver", GitHub, Dec. 07, 2023. https://github.com/chuffed/chuffed

[52] Vittorio Rossetto, "GitHub - VittorioRossetto/fzn2nl", GitHub, 2025. https://github.com/VittorioRossetto/fzn2nl (accessed Feb. 06, 2026).

[53] A. O. Li and T. Goyal, "Memorization vs. Reasoning: Updating LLMs with New Knowledge", arXiv (Cornell University), Apr. 2025, doi: https://doi.org/10.48550/arxiv.2504.12523.

[54] Z. Gekhman et al., "Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?", arXiv (Cornell University), May 2024, doi: https://doi.org/10.48550/arxiv.2405.05904.

[55] J. Cheng, M. Marone, O. Weller, D. Lawrie, D. Khashabi, and V. Durme, "Dated Data: Tracing Knowledge Cutoffs in Large Language Models", arXiv (Cornell University), Mar. 2024, doi: https://doi.org/10.48550/arxiv.2403.12958.

[56] X. Wu and K. Tsioutsiouliklis, "Thinking with Knowledge Graphs: Enhancing LLM Reasoning Through Structured Data", arXiv (Cornell University), Dec. 2024, doi: https://doi.org/10.48550/arxiv.2412.10654.

[57] MiniZinc, "GitHub - MiniZinc/libminizinc: The MiniZinc compiler", GitHub, Jan. 23, 2026. https://github.com/MiniZinc/libminizinc (accessed Feb. 04, 2026).

[58] Allen, J.: "Anatomy of LISP". McGraw-Hill, Inc., New York (1978)

[59] P. J. Stuckey and G. Tack, "MiniZinc with Functions", Lecture Notes in Computer Science, pp. 268-283, 2013, doi: https://doi.org/10.1007/978-3-642-38171-3_18.

[60] "Gurobi Optimizer", Gurobi Optimization. https://www.gurobi.com/solutions/gurobi-optimizer/

[61] R. M. e S. de Oliveira and M. S. F. O. de C. Ribeiro, "Comparing Mixed & Integer Programming vs. Constraint Programming by solving Job-Shop Scheduling Problems", Independent Journal of Management & Production, vol. 6, no. 1, Mar. 2015, doi: https://doi.org/10.14807/ijmp.v6i1.262.

[62] "Introducing GPT-5.2", Openai.com, Dec. 11, 2025. https://openai.com/index/introducing-gpt-5-2/

[63] T. Patwardhan et al., "GDPval: Evaluating AI Model Performance on Real-World Economically Valuable Tasks", arXiv (Cornell University), Oct. 2025, doi: https://doi.org/10.48550/arxiv.2510.04374.

[64] OpenAI, "ChatGPT", ChatGPT, 2025. https://chatgpt.com/

[65] "GPT-5.2 in ChatGPT OpenAI Help Center," OpenAI Help Center, 2026. https://help.openai.com/en/articles/11909943-gpt-52-in-chatgpt#h_1fadb43e65 (accessed Feb. 11, 2026).

Acknowledgements

Here you can thank whoever you want.