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 2024/2025

Introduction

This thesis presents a systematic investigation of the use of Large Language Models (LLMs) [33] within the domain of Constraint Programming (CP) [24], specifically in the task of algorithm selection [46], referred to as "solver selection" in the context of NP-complete problems solved by constraint solvers. 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 instance prior to execution. This issue arises in many domains because no single algorithm performs optimally across all problem instances. Traditional algorithm selection and portfolio construction methods typically treat the problem as a supervised classification or regression task [42].

LLMs are statistical language models that estimate probability distributions over token sequences. In practice, these distributions are parameterized by deep neural network architectures trained on large text corpora. They result from 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 largely unexplored.

Constraint Programming is a declarative paradigm for solving combinatorial problems, particularly those arising in planning, scheduling, and resource allocation [24]. 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, which leads to performance variability across problem classes. Selecting an appropriate solver for a given instance is therefore a central challenge [22].

This thesis investigates the problem of algorithm selection by using LLMs as decision-making components within a solver portfolio. This idea is inspired by [23] and builds on 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.

Although the study is exploratory in scope, its experimental design is organized around a set of explicit working assumptions that motivate the progression of experiments. It is hypothesized that structured representations of problem instances will be more informative for solver selection than raw scripts. In addition, feature-based numerical summaries capturing structural properties of each instance are expected to provide stronger discriminative signals than text alone. Furthermore, the inclusion of solver descriptions in the prompt is expected to reduce biases arising from solver name recognition, enabling the model to reason on the basis of capability rather than prior association. Finally, sampling temperature is treated as a control variable regulating the trade-off between exploration and exploitation in solver choice, with lower values expected to favor stable, high-probability selections and higher values enabling broader exploration of the solver space.

The experimental evaluation relies on benchmark instances from the 2025 MiniZinc Challenge [51, 10]. A set of general-purpose LLMs was selected for evaluation and 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 experiments revealed substantive limitations in the ability of LLMs to support this task, producing only marginal improvements relative to individual baseline solvers. To mitigate these limitations, structured representations of the problem instances were incorporated through the feature extraction tool `mzn2feat` [17], which derives a set of static and dynamic numerical features from each instance, such as the number of variables and constraints. While this representation yielded the highest performance observed in our experiments, the overall improvement remained limited. Furthermore, prompts constructed from complete model-and-data encodings or from extensive feature vectors may become prohibitively large with respect to practical context-window constraints. This consideration motivated the development of `fzn2nl` [47], a tool designed to translate problem code into compact natural language descriptions, which were subsequently employed in additional experiments.

Overall, the results indicate that LLM-based solver selection is promising but fragile: the best configurations achieve only marginal gains over the single best solver in the full portfolio, and performance decreases in the restricted portfolio where finer discrimination is required. The `fzn2nl` representation shows mixed effects, not improving top-1 selection over the strongest feature-based variants but improving diversity and achieving the best parallel score in the restricted setting. The reasoning analysis further shows that LLMs can produce coherent solver-level and problem-level explanations, yet their final choices are often driven by prior associations and do not consistently align with empirical performance.

Beyond the empirical findings, this thesis contributes a reproducible evaluation protocol for LLM-based solver selection, a structured comparison of input representations, the deterministic `fzn2nl` pipeline for compact natural-language abstractions of FlatZinc models, and an interpre-

tative framework based on controlled perturbation tests (including description swapping and solver anonymization) to analyze decision patterns in addition to predictive accuracy.

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 temperature tuning. Chapter 4 introduces a complementary tool developed during this research, `fzn2nl`, which translates FlatZinc models into natural language descriptions, and evaluates LLM performance when operating directly on such generated representations. Chapter 5 describes an experimental investigation aimed at understanding the level of comprehension that LLMs exhibit with respect to solvers and problem characteristics. Chapter 6 presents the conclusions.

Contents

Chapter 1

Technical Background

This chapter introduces the theoretical background required for the remainder of the thesis.

The first section presents the fundamentals of Constraint Programming (CP), including CP solvers and the MiniZinc modeling language used in the experimental evaluation. The second section introduces Large Language Models (LLMs), focusing on their probabilistic formulation, inference mechanisms, and prompting strategies.

1.1 Constraint Programming

Constraint Programming (CP) is a paradigm for declaratively modeling and solving combinatorial problems, particularly in domains such as planning and scheduling [24].

A constraint is a logical relation defined over a set of variables, each associated with a domain of possible values. Constraints restrict the combinations of values that variables may simultaneously assume and therefore represent partial information about the system being modeled. An important feature of constraints is their declarative nature: they specify relationships that must hold without prescribing the procedure used to enforce them [24].

CP studies computational systems that solve problems by enforcing sets of such constraints and searching for assignments of variable values that satisfy them.

1.1.1 Solvers

In this thesis, a solver denotes a system that computes assignments satisfying a set of constraints and, when required, optimizes an objective function.

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

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

where X is a finite set of decision variables, $D(x)$ is the domain of 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 [25].

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

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

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

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

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

Constraint Programming solvers

Constraint Programming solvers operate directly on variables with finite or structured domains and heterogeneous constraints. Their core mechanisms are constraint propagation, which removes inconsistent values from variable domains, and systematic search, typically implemented through backtracking guided by heuristics. CP solvers also support high-level global constraints equipped with specialized filtering algorithms that exploit combinatorial structure [27]. A representative example is Gecode [40].

Portfolio solvers

Solving combinatorial search problems is computationally challenging, and different solvers often perform better on different instances of the same problem class. Algorithm portfolios exploit this observation by combining multiple solvers and selecting the most appropriate one for each instance [19].

Algorithm portfolios are a specific case of the broader Algorithm Selection problem [46]. In a portfolio solver, a set of constituent solvers $s_1, \ldots, s_m$ is available. Given a problem instance p, the system predicts which solver (or subset of solvers) is most suitable for solving it.

Solver selection is typically performed using machine learning models trained on features extracted from the problem instance [34]. In this work, we investigate whether a pre-trained LLM can serve as the decision component for this selection process, potentially avoiding the need for dedicated training [23].

1.1.2 MiniZinc

MiniZinc is a high-level modeling language for constraint programming that supports a wide range of solvers and enables solver-independent problem specification [41]. It was designed to provide a standard modeling language for CP and to facilitate solver benchmarking.

For these reasons, all problems used in the experimental evaluation were modeled in MiniZinc.

A MiniZinc example

A MiniZinc specification consists of a model, describing the structure of a class of problems, and a data component specifying a particular instance. Data files contain assignments to parameters declared in the model and are typically provided separately.

A MiniZinc model is composed of a sequence of items, including variable declarations, constraints, predicates, and a single `solve` item specifying whether the problem is a satisfaction or optimization task.

Figure 1.1 and Figure 1.2 show an example MiniZinc model and data file for a job shop scheduling problem [41]. The model defines parameters, decision variables, and constraints that enforce task ordering and prevent resource conflicts, while the data file provides the specific instance parameters.

```
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 (adapted from [41]).

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

Figure 1.2: MiniZinc data (`jobshop2x2.dzn`) for the job shop problem (adapted from [41]).

MiniZinc Challenge

Comparing constraint programming systems is inherently difficult because different solvers support different modeling features and optimization strategies. MiniZinc addresses this issue by providing a standardized modeling language that allows multiple solvers to be evaluated on the same problem specifications.

Since 2008, the MiniZinc Challenge has provided a yearly benchmark competition comparing solvers that support MiniZinc [51]. Participants submit solvers capable of handling MiniZinc or FlatZinc models, which are then evaluated on a standardized set of benchmark instances. Performance is assessed based on solution quality, number of solved instances, and solving time.

Although the MiniZinc Challenge provides a comparative scoring system, this thesis employs alternative evaluation metrics better suited to absolute performance assessment in solver-selection experiments.

1.2 Large Language Models

Large Language Models (LLMs) are statistical language models that estimate probability distributions over sequences of tokens by learning patterns from large text corpora.

Modern LLMs are typically implemented using transformer architectures [53], which rely on self-attention mechanisms to model dependencies between tokens in a sequence. Given an input sequence $x_1, \ldots, x_n$, the model estimates the conditional probability

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

Text generation proceeds autoregressively by repeatedly sampling tokens from this distribution.

Training generally involves two stages. During large-scale pretraining, models learn general linguistic and semantic representations through self-supervised objectives. Alignment techniques, such as supervised fine-tuning and reinforcement learning from feedback, subsequently adapt the model to follow instructions and produce outputs consistent with user expectations [33].

1.2.1 Inference and decoding

During inference, the model produces logits over the vocabulary, which are converted into probabilities using the softmax function [38]. Decoding strategies determine how the next token is selected from this distribution.

Deterministic decoding methods include greedy search, which selects the most probable token at each step, and beam search [28], which maintains multiple candidate sequences during generation.

Stochastic methods introduce controlled randomness into the generation process. Temperature sampling modifies the probability distribution using a temperature parameter τ, while top-p sampling restricts token selection to the smallest subset of tokens whose cumulative probability exceeds a threshold p [48].

A practical constraint is the context window, which limits the total number of tokens that can be processed simultaneously. This limitation influences prompt design and the amount of

contextual information that can be provided to the model.

1.2.2 Relevance to solver selection

In this work, LLMs are used as decision components for solver selection. Given a representation of a problem instance, the model is prompted to recommend the most appropriate solver from a predefined portfolio.

The effectiveness of this approach depends both on the model's capacity to interpret structural properties of constraint models and on practical aspects such as prompt formulation, representation choice, and inference parameters.

Chapter 2

Methodology

This chapter presents the design choices underlying the initial benchmark, which serves as the basis for refining subsequent experiments and assessing the current capabilities and potential of the proposed solver-selection approach.

The chapter is organized into six sections, each addressing a key design decision. The first section discusses the provider selection. The second describes the selection of large language models used for testing. The third section presents the initial prompt-engineering strategy used to define a 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

The first step was to determine which LLMs could be used for the algorithm-selection task. Given the limited computational resources available during testing, we relied 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` [1], offered by Google DeepMind. Gemini is a family of LLMs with multiple sizes and capabilities. This provider was 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 [6] was preferred as it was more stable and the primary concern of this work was its text-generation capability.

- `Groq API` [5], 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. The rate limits of both APIs are reported in Table 2.2 and Table 2.1.

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 for models available through the Groq API [8]. Each row corresponds to a distinct model, identified by its API name. The columns report the enforced usage constraints: requests per minute (RPM), requests per day (RPD), tokens per minute (TPM), and tokens per day (TPD). A dash (–) indicates that no explicit daily token limit is enforced for that model.

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 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.

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

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 for models available through the Gemini API [7]. All the columns are organized as in Table 2.1.

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 was 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:

- Minimise prompt-induced bias: An excessively descriptive or overly complex prompt could negatively influence model behaviour, since long prompts may lead to phenomena such as "context rot" [32]: a progressive decay in response quality as prompt length increases.
- Reduce token usage: Since the testing setup is constrained by API rate limits, keeping the prompt compact minimises 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.

Taking all of these considerations into account, the resulting prompt structure is as follows:

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 selection of benchmark problems. 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, including 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.

Based on these criteria, the selected benchmark was the problem set from the MiniZinc Challenge 2025 [51, 9, 10]. 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 consists of a `.mzn` file containing the MiniZinc model [41], which defines the decision variables, constraints, and objective function. Every problem is further accompanied by five data instances, each stored in either a `.dzn` or a `.json` file providing the problem-specific parameters and constants, yielding a total of 100 diverse and challenging test scenarios.

2.5 Test Metrics

In order to evaluate LLM performance rigorously, it is necessary to select a standard metric for assessing the quality of each model's output. In addition, a comparative metric is required to evaluate how well each LLM performs relative to the Single Best Solver (SBS).

Before analysing the evaluation metrics, the systems to which these metrics apply must be defined, namely, the solvers. In this context, a solver is a program that takes as input a MiniZinc description of a computational problem together with the associated instance data, and returns an observable outcome consisting of zero or more solutions to the given problem.

For decision problems, the outcome may simply be "satisfiable" or "unsatisfiable", while for optimisation problems the outcome of interest is typically the best objective value found within the allotted time. An evaluation metric, or performance metric, is a function that maps the outcome of a solver on a given instance to a numerical value quantifying the solver's performance on that 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 impose a `timeout` τ on the solver's execution when termination within a reasonable time cannot be guaranteed (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 every solver $s \in \mathcal{S}$ on every instance $i \in \mathcal{I}$ is always measured within 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.

To these ends, two separate metrics were adopted.

2.5.1 Metric for Solver Score

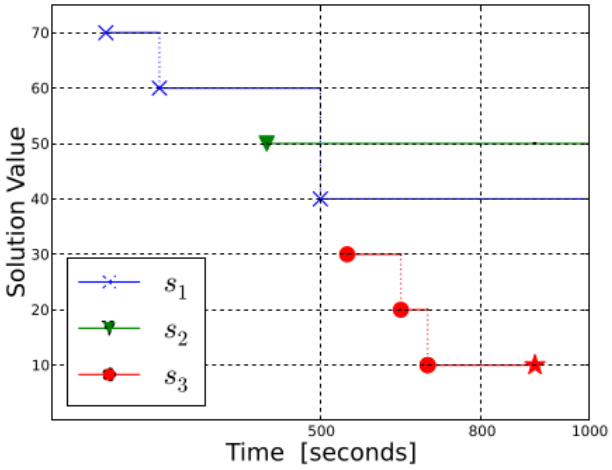

Figure 2.1: Solver performances example

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 (henceforth, the score) on instance i at a given time t as a function $\text{score}_{\alpha,\beta}$ [22, 18] 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, MIN(i) and 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 parameters used for score calculation in the experiments are: $T = 1{,}200{,}000$ ms (i.e., 20 minutes, the time limit adopted in the MiniZinc Challenge for solver evaluation), $\alpha = 0.25$, and $\beta = 0.75$.

2.5.2 Closed Gap

Once the solver score metric has been defined, a comparative metric is also required to assess the overall performance of a meta-solver relative to the reference baselines. To this end, the closed-gap (CG) [22] measure was adopted as the evaluation metric. CG is a relative, meta-solver-specific measure that was employed in the 2015 ICON and 2017 OASC [37] challenges to handle the disparate nature of the scenarios.

CG assigns a value to a meta-solver 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 a "meta-metric", defined in terms of a base evaluation metric m to minimise, which in this context is the scoring metric introduced in Section 2.5.1. Formally, if (I, S, τ) is a scenario then

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

and

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

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 $m_{\text{VBS}} > m_{\text{SBS}}$ 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 $m_{\text{SBS}} - m_{\text{VBS}}$, which makes little sense for individual solvers, as it wouldn't reflect their actual performance overall.

2.6 Experiment Setup

Having defined both the prompt format used to query the LLMs (Section 2.3) and the benchmark problem set on which they are evaluated (Section 2.4), the next step 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 instances, particularly their associated data files, still exceeded 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 was 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 of removing all non-essential elements, such as comments (starting with `%` [41]), tabs, and unnecessary whitespace. While this reduced token usage and standardised script formatting, it proved insufficient on its own. The principal contributor to token overflow was the presence of large data arrays, which not only increased message length but could also introduce irrelevant information into the context, thereby "distracting" the LLM from the most pertinent content [49].

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 did not account for the internal complexity of individual elements and performed poorly on more structured data. For this reason, a second truncation mechanism was introduced based on raw character count. Arrays exceeding 120 characters were truncated accordingly, using the same inline annotation to preserve information about the original size.

2.6.2 Custom Delays

Since each experiment involved multiple problems and multiple sequential requests per LLM, rate limits could still be exceeded even when individual requests were within bounds. To address 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
```

The error code was inspected. Errors 413 and 429 indicated that a rate limit had been exceeded. The error message was then parsed to identify the specific limit involved. If the limit concerned tokens per minute (TPM) or requests per minute (RPM), the system paused execution for 60 seconds before retrying. If the exceeded limit was tokens per day (TPD) or requests per day (RPD), the message was further analysed to extract the cooldown duration, expressed in the form $XX\text{h}\,XX\text{m}\,XX.XX\text{s}$, where h denotes hours, m minutes, and s seconds. The required delay was then computed from this value, after which the request was 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, to estimate the baseline performance of the 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 [17], in combination with the prompt strategies developed earlier. The fifth section analyzes the effect of sampling temperature tuning [54] by testing multiple temperature values on the five most effective configuration variants identified across the study.

Finally, the sixth section analyzes LLM performance when tested on a restricted set of solvers to choose from, excluding both dominant solvers and clearly underperforming ones.

3.1 Preliminary tests

The first experiments were conducted using the original problem instances without any preprocessing. 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 6.2, Table 6.1 and Table 6.3) for both

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 using raw scripts as input to all tested LLMs. The column "Model" lists the name of each tested LLM; "Single Score" is the sum of the performance of the best solver suggested by the LLM for each instance (as explained in Section 2.5.1); "Parallel Score" is the sum of scores across all instances in the parallel-solver setup, computed by taking the three best solvers suggested by the LLM, evaluating each, and retaining the maximum; and "Closed Gap" is the closed-gap score calculated from "Single Score" using the formula in Section 2.5.2.

single-solver evaluation and parallel-solver evaluation, all scores remain below 70, indicating lower performance than three of the single solvers in the free category, and four of the single solvers from the open category, and resulting in negative closed-gap values for all tested 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 (shown in Table 2.2 and Table 2.1). This issue is particularly pronounced for 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 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`. The considerable execution time required for the experiments could reach up to 48 hours.

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 generates a response 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 6.4, performance under parallel-solver evaluation remains 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 selection approaches and the remaining standalone solvers.

More pronounced variability is observed in the single-solver evaluation (Table 6.5). 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 6.18 and Figure 6.5. 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 6.6.

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 may become counterproductive, potentially distracting the model and degrading performance rather than improving it [32, 49]. 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 textual 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 shown by the results, all closed-gap values remain negative 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 has been discussed so far indicates that the contextual information provided to the LLMs is either insufficient or, in some cases, not beneficial. 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 [44]. In the context of LLMs, messages are explicitly associated with roles, typically `system`, `user`, and `assistant`. This 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 are reported in Table 6.10. 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 solver 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 6.11), providing only solver descriptions does not lead to systematic improvements. The sole exception is `gemini-2.5-flash`, although its score remains below 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 6.12. 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 corresponds to 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, the scores of the other LLMs are slightly lower than in the previous setups, and all scores remain below 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 compared with the remaining solvers, these primitive configurations still hold fairly competitive results.

To better display the performance of individual LLMs, all variant scores are compared, using histograms for better visualization in Figure 6.3 for parallel-solver evaluation, Figure 6.4 for single-solver evaluation and Figure 3.1 for closed gap evaluation.

To provide an alternative perspective on the results, the performance of each configuration is displayed against the single solvers of the corresponding category. In Table 6.17 and Figure 6.6, the performance of all configurations under parallel-solver evaluation is compared against the single solvers from the open category. In Table 6.18 and Figure 6.5, the single-solver evaluation scores of all variants are compared against all solvers from the free category.

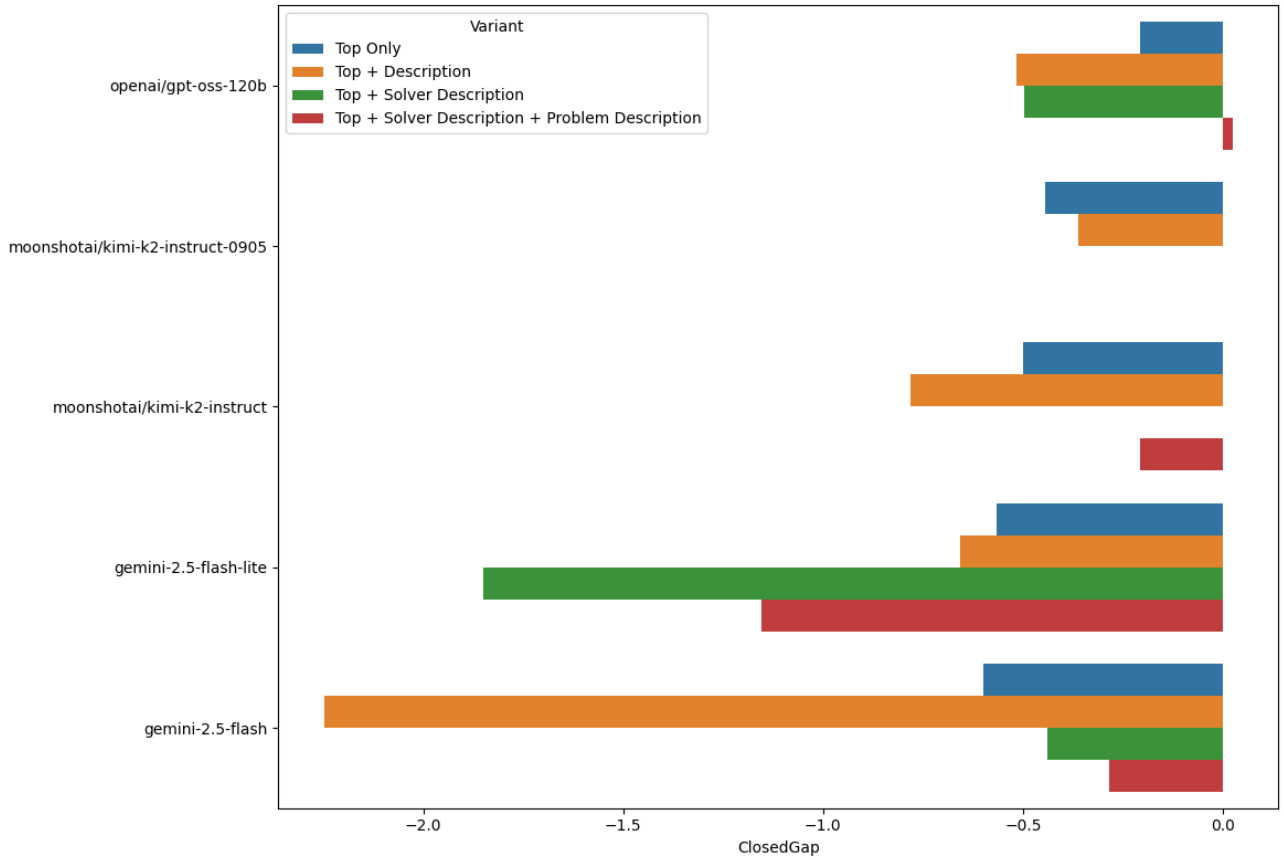

Figure 3.1: Histograms displaying the performances of all the solver variants in "Closed Gap" evaluation, as calculated in Table 3.1.

3.4 Feature Extraction

The preceding sections demonstrated the importance of textual information and of providing a richer context to the LLM rather than relying solely 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 significant limitation of the approach, given that such data must be supervised, if not entirely rewritten by hand, due to the high variability, especially when extracted on new problems. This limitation highlights the need for a method to extract information automatically, in a controlled and predictable way.

A further issue concerns script length: programs with longer scripts require techniques such as sanitization, as discussed in Section 2.6.1. While such techniques make it possible to process longer scripts and context data, they remain detrimental for longer problems, increasing the risk of hallucinations [36] and context rot [32] due to the forced removal of data elements, which leaves incomplete arrays as input.

For these tasks, we employed a feature extractor [17] capable of extracting an extensive set of 95 features from a Constraint Satisfaction/Optimization Problem defined in various modelling languages: MiniZinc, FlatZinc, or XCSP. The tool is designed to be independent of the particular machine on which it is run and of the specific global constraint redefinitions of

any given solver. The version employed here had previously been used in other work [16, 20, 21].

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 primitive 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 [30]. 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 [17].

For a better understanding of all of the features, refer to Table 6.19 for the description by category, and to Listing 6.1 for an example output.

3.4.2 Testing and Results

To evaluate the effectiveness of feature-based representations, the pretty-printed `mzn2feat` output (e.g., 6.1) 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` [17], all experiments were conducted using `gpt-oss-120b` as it is the only model that produced a positive closed gap up to 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 likely due to the issues discussed earlier such as context rot [32].

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 sampling-temperature tuning to study an exploration–exploitation trade-off in solver selection: lower temperatures bias the model toward more conservative, high-probability solver choices, whereas higher temperatures encourage exploration of less likely alternatives. 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 [14].

At each step of the 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 modifies the resulting probability distribution. 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 [31, 54].

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 [55]. Increased stochasticity, however, can also raise the likelihood of incoherent or factually incorrect outputs [36]. 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 [5]. The tested values were based on the preset recommendations in the provider documentation [12] (Table 3.7).

3.5.2 Testing and Results

As anticipated, all tests were conducted using `gpt-oss-120b` as the LLM. Due to rate-limit constraints, temperature tuning experiments were restricted to the five best-performing configuration variants identified earlier, which we will refer to using the following numbering:

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.

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 [12]. The leftmost column presents the scenario for which the corresponding settings are most appropriate, followed by the sampling temperature value, and a short comment on the expected behaviour of the LLM under the given setting.

4. Features extracted through `mzn2feat` and text description for problems.

5. `.mzn` scripts and instance data

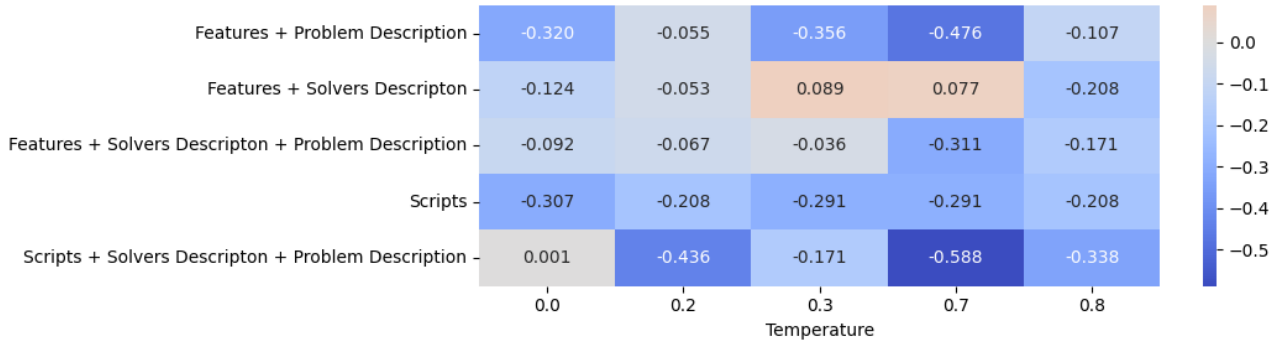

Figure 3.2: Heatmap displaying the performance of all the combinations of temperatures with the five best performing variants in "Closed Gap" evaluation calculated as in Table 3.1, all tests were performed using `gpt-oss-120b`.

As displayed in Table 3.8 and in Figure 6.7, 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 6.17).

In the single-solver evaluation (Figure 6.8), 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 $\tau = 0.3$ and $\tau = 0.7$, displayed in Figure 3.2, indicating performance above the SBS baseline under those settings, and representing the highest scores obtained thus far.

Variant	Temperature	Single Score	Parallel Score	Closed Gap
Features + Solvers Description	**0.3**	**78.032**	82.873	**0.089**
Features + Solvers Description	**0.7**	**77.896**	83.767	**0.077**
Scripts + Solvers Description + Problem Description	0.0	76.971	82.042	0.001
Features + Solvers Description + Problem Description	0.3	76.529	82.959	-0.036
Features + Solvers Description	0.2	76.326	84.044	-0.053
Features + Problem Description	0.2	76.298	83.363	-0.055
Features + Solvers Description + Problem Description	0.2	76.156	83.763	-0.067
Features + Solvers Description + Problem Description	0.0	75.860	83.146	-0.092
Features + Problem Description	0.8	75.676	83.009	-0.107
Features + Solvers Description	0.0	75.477	83.444	-0.124
Features + Solvers Description + Problem Description	0.8	74.909	82.419	-0.171
Scripts + Solvers Description + Problem Description	0.3	74.907	80.042	-0.171
Features + Solvers Description	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 Description + Problem Description	0.7	73.224	80.845	-0.311
Features + Problem Description	0.0	73.114	81.456	-0.320
Scripts + Solvers Description + Problem Description	0.8	72.894	80.875	-0.338
Features + Problem Description	0.3	72.678	82.393	-0.356
Scripts + Solvers Description + Problem Description	0.2	71.717	82.042	-0.436
Features + Problem Description	0.7	71.235	81.896	-0.476
Scripts + Solvers Description + 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.

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 6.20. 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 it 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` [47], 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 presented in Chapter 3, several important insights emerged. First and foremost, the limitations on tokens per request and on the context window represent a significant constraint that we were only able to partially mitigate using the techniques described 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

statistical prediction, based on their previous knowledge; this approach may lead to limitations when the model is confronted with entirely novel problems. To address this limitation, we needed a way to standardize all the problems so that the LLM could evaluate instances directly based on their structural 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. While the results obtained with `mzn2feat` proved to be consistent, it was hypothesized that a more effective representation might exist to formalize MiniZinc problems for an LLM, given the limitations observed when evaluating the LLM against a selected set of solvers (Section 3.6).

The development of `fzn2nl` [47] originated from the idea of constructing a FlatZinc parser that, given a model as input, produces a deterministic natural language description of the problem exposing its main characteristics, in a form suitable for LLM processing.

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 [29] after the pre-training stage, whereas knowledge in the world continues to change.

Even within the pre-training data, knowledge from recent years may conflict with earlier information. Moreover, the auto-regressive training objective biases LLMs toward surfacing more frequent but not necessarily recent knowledge [26]. This problem surfaces primarily when the LLM is confronted with more indirect questions, given that updates in the real world lead to nuanced and complex changes in model knowledge (Figure 4.1) [35].

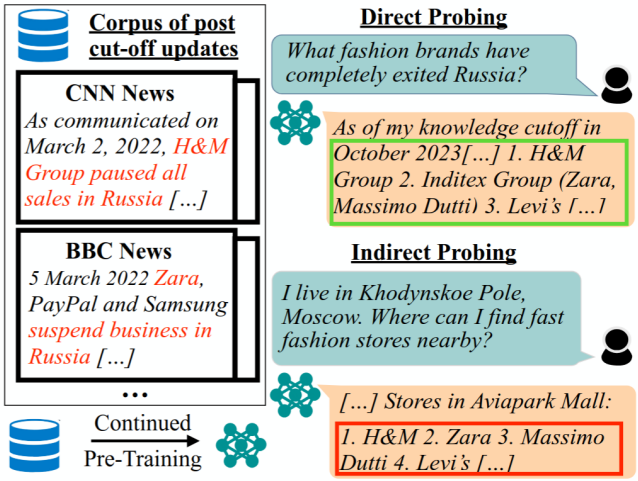

Figure 4.1: Example of LLM that is pre-trained on updated knowledge surfacing updates in the direct probing but failing under indirect probing settings (example image was taken from [35]).

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

The simplest solution under these constraints is to create a deterministic description of

each problem, providing enough knowledge for a reasonable solver selection 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. While structured data, rich in logical and relational information, has the potential to enhance the reasoning abilities of LLMs, its integration poses a challenge due to the risk of overwhelming LLMs with excessive tokens and irrelevant context information [56].

These considerations led to the development of `fzn2nl`, a parser that takes as input a `FlatZinc` file and translates its structural elements into 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" [39].

Constraint problems expressed in MiniZinc are solved by translating them to a simpler, solver-specific subset of MiniZinc, called FlatZinc, using the "MiniZinc Compiler" [39]. The complexity of the translation arises from the need to simultaneously (a) unroll array comprehensions (and other loops), (b) replace predicate and function applications by their bodies, 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 [50]. Common subexpression elimination is implemented using a technique similar to hash-consing in Lisp [15].

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 [2]), 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 [27].

Consequently, translating a FlatZinc model using a specific solver implicitly introduces solver-specific structural information, and by exposing this representation to an LLM, one indirectly biases the model toward the solver used during compilation, or one of the same category, thereby invalidating the reasoning process.

4.2.2 Custom Compiler

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

As stated in Section 4.2.1, the difference between solvers concerns the propagation of global constraints.

To eliminate this distinction, propagation was effectively disabled by modifying the MiniZinc compiler code to avoid the substitution of predicates by their bodies. 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 illustration, we will 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 generation of a natural language description in `fzn2nl` follows a single-pass pipeline: the program takes a FlatZinc file as input, then this file is parsed in order to extract variables, arrays, constraints, the solve directive, and search annotations (when present) into an internal model object. The program then computes descriptive statistics on both variables and constraints and reconstructs the objective structure. All parsed elements are then mapped into readable sentences and grouped summaries. Finally, the system produces a report as output, consisting of a problem description, a variable description, and a 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 the element type, whether elements are decision variables, the 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 proceeds 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 illustrate the operation of the parser, we describe how it processes 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, incorporating 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 the 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` enriches 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 also rewritten into an abstract form, by replacing variable names with placeholders (e.g., $a, b, c, \dots$). 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`, two configurations were tested: (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` [11], a recent large-scale LLM reported to achieve state-of-the-art performance across multiple evaluation benchmarks, including GDPval [45]. The model is designed to support extended context handling, agentic tool use, and improved reasoning capabilities.

Testing was conducted manually, through the official ChatGPT interface [43], which imposes daily usage limits [13]. Under the full protocol, this setup would have required manually submitting 100 separate queries (one per instance). Due to these constraints, a reduced evalua-

tion 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.

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.

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 three of the `fzn2nl`-based variants reported in Table 4.1, as well as below the two 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.

Chapter 5

LLM Reasoning Analysis

Across the experiments conducted thus far, several limitations have emerged in the use of general-purpose LLMs for solver selection. Although certain configurations yielded positive outcomes, the observed improvements remain marginal, particularly when considered in light of the intrinsic non-determinism of LLM-generated responses. These limitations become more evident when the models are evaluated under a restricted solver set, as discussed in Section 3.6.

In this context, the objective of the present chapter is not to further optimize selection performance, but rather to provide a more precise assessment of the extent to which LLMs effectively interpret the problems included in the prompts and comprehend the characteristics of the candidate solvers. The focus is therefore shifted from outcome-oriented evaluation to an analysis of the internal decision mechanisms underlying solver selection.

To this end, the first section investigates the impact of modifying the solver representation. This includes experiments in which solver descriptions are deliberately permuted across solvers, as well as configurations employing anonymized identifiers (e.g., `a`, `b`, `c`, ...) in place of descriptive names. The aim is to assess the sensitivity of the models to semantic content versus superficial labeling.

In the second section, we prompt `GPT-5.2` to explicitly articulate its reasoning process in the context of solver selection. The generated explanations are subsequently analyzed to extract decision criteria and broader insights into the model's interpretative behavior.

5.1 Analysis of Solver Understanding

The rationale underlying the experiments presented in this section is to determine whether the LLM effectively incorporates the provided solver descriptions into its decision process, or whether it primarily relies on prior knowledge associated with solver names. In other words, the objective is to assess whether the model's selections are driven by the semantic content of the descriptions included in the prompt, or by pre-existing associations learned during pre-training.

The relevance of this analysis lies in evaluating the model's potential to generalize to previously unseen solvers. If the LLM systematically favours well-known or extensively documented

solvers, particularly widely adopted industrial tools, over less prominent or open-source alternatives, independently of the information supplied in the prompt, then its applicability in dynamic or evolving solver ecosystems would be inherently constrained.

5.1.1 Experiments with Swapped Solver Descriptions

The first experiment designed to address this question consists of presenting the LLM with a solver list in which the textual descriptions are deliberately permuted across solvers. The aim is to determine whether the model's responses change in accordance with the swapped descriptions, or whether such descriptions are effectively disregarded in favor of prior knowledge linked to solver names.

To ensure internal consistency, all solver descriptions were edited to remove explicit references to any solver name, whether original or other linked solvers. This prevents the model from trivially recovering the correct association through lexical cues.

The experiment was conducted using several variants that explicitly incorporate solver descriptions across different setups. In particular, five configurations were selected among the best-performing variants identified in previous sections:

1. `fzn2nl` output combined with swapped solver descriptions, sampling temperature 0.7.
2. `fzn2nl` output combined with swapped solver descriptions, sampling temperature 0.2.
3. `mzn2feat` output combined with swapped solver descriptions, sampling temperature 0.7.
4. `mzn2feat` output combined with swapped solver descriptions, sampling temperature 0.3.
5. Original sanitized scripts combined with problem description and swapped solver descriptions, default sampling temperature.

For clarity of interpretation, the swapping procedure was applied to all solvers within the free category. The complete set of results is reported in Table 5.1.

To compute the evaluation scores, each solver suggested by the LLM was mapped back to the solver from which the associated description had been taken, thereby aligning the evaluation with the effective semantic content presented in the prompt.

As shown in Table 5.1, performance decreases substantially under swapped descriptions. The degradation is particularly pronounced when structured feature representations are used, and further accentuated when raw scripts are provided as input. Within each representation type, higher sampling temperatures consistently yield better results. This pattern may indicate that when confronted with representations more distant from natural language, the model exhibits limited capacity to integrate and reason over the altered solver descriptions.

A more detailed inspection of the solver suggestions supports this interpretation. Even in the best-performing configurations, the scoring and the distribution of suggested solvers

Variant	Temperature	Single Score	Parallel Score	Closed Gap
fzn2nl + Solver Descriptions	0.7	60.509869	81.068252	-1.367150
fzn2nl + Solver Descriptions	0.2	58.827415	80.534470	-1.506940
Features + solver descriptions	0.7	50.797299	77.640982	-2.174135
Features + solver descriptions	0.3	47.712594	73.080124	-2.430433
Scripts + Problem Description + solver descriptions	default	46.154496	70.744078	-2.559890

Table 5.1: Analysis over configurations involving swapped solver descriptions, evaluated with `gpt-oss-120b`. Columns are computed as in Table 3.1, after mapping each suggested solver to the solver from which the description was taken.

reveal a limited sensitivity to the modified descriptions. In particular, `or-tools_cp-sat-free` remains the most frequently selected solver across all variants, despite having been assigned the description originally associated with `atlantis-free`, a solver with substantially lower standalone performance. The suggestion counts reported in Table 5.2, Table 6.21, Table 6.22, Table 6.23 and Table 6.24, each corresponding to one of the configurations, illustrate this tendency.

Suggested Solver	Description	Top1 Count	Total Count
`or-tools_cp-sat-free`	`atlantis-free`	62	98
`atlantis-free`	`cplex-free`	32	53
`chuffed-free`	`highs-free`	6	49
`highs-free`	`or-tools_cp-sat_ls-free`	0	68
`yuck-free`	`choco-solver__cp_-par`	0	31
`choco-solver__cp_-free`	`jacop-free`	0	1

Table 5.2: Solver suggestions from `gpt-oss-120b` under configuration 1. “Suggested Solver” is the solver named in the LLM output; “Description” is the solver whose description was assigned to it; “Top1 Count” is the number of instances for which it was recommended as the best solver; “Total Count” is the number of instances in which it appeared among the three best suggestions.

Overall, these results indicate that, even when provided with explicit and potentially contradictory information, the LLM continues to rely predominantly on pre-existing associations tied to solver names. The solver descriptions included in the prompt appear to exert limited influence on the final selection, suggesting that the model’s decision process is only marginally conditioned on the newly supplied semantic content [35].

5.1.2 Experiments with Anonymous Solvers

The experiments described in the previous section revealed a clear predilection of the LLM for certain solvers, based solely on their names rather than on their descriptions. In order to investigate this issue in greater depth, an anonymization procedure was introduced for the next

experiment.

Each solver name was replaced by an arbitrary identifier (e.g., `a`, `b`, `c`, ...), while preserving the exact textual descriptions previously used. After inference, the suggested identifiers were mapped back to the corresponding original solvers in order to compute the evaluation metrics reported in Table 5.3.

Variant	Temperature	Single Score	Parallel Score	Closed Gap
Scripts + Problem Description + solver descriptions	default	77.428630	83.162352	0.038573
fzn2nl + Solver Descriptions	0.7	75.066104	83.031082	-0.157721
fzn2nl + Solver Descriptions	0.2	74.793219	83.016420	-0.180394
Features + solver descriptions	0.7	73.177307	78.103986	-0.314655
Features + solver descriptions	0.3	70.409070	75.751372	-0.544658

Table 5.3: Analysis over configurations involving anonymous solvers, evaluated with `gpt-oss-120b`. Columns are computed as in Table 3.1, after mapping each suggestion back to the solver from which the description was taken.

The results indicate that anonymization does not substantially alter performance. Across configurations, both Single and Parallel Scores remain comparable to those obtained when solver names are visible. In particular, the configuration using raw scripts and problem description even achieves a positive Closed Gap value of 0.038.

A more detailed inspection reveals a highly concentrated selection pattern. The model predominantly assigns top-1 recommendations to two solvers, corresponding, after mapping back, to `or-tools_cp-sat-free` and `chuffed-free`. As shown in Table 5.4, these two solvers account for 100% of the top-1 selections in Configuration 1.

These findings support two observations. First, the LLM demonstrates the capacity to interpret solver descriptions independently of their names, as performance metrics remain stable under anonymization. Second, the model maintains a strong inductive bias toward a narrow subset of solvers, leading to limited diversification in its recommendations, even after the anonymization process.

5.2 Reasoning with GPT-5.2

The objective of this additional analysis is to examine the reasoning process articulated by an LLM when confronted with one of the benchmark problems considered in the experimental campaign. To this end, the same prompts described in Section 4.4 were submitted to the `GPT-5.2` chatbot [11], with the explicit request to provide a detailed explanation of the solver selection immediately after reporting the chosen solvers.

Given that the responses consist of unconstrained free text, the evaluation was restricted to a high-level assessment, comparing the stated reasoning with the empirical results of the

Suggested Name	Description	Top1 Count	Total Count
f	chuffed-free	78	100
n	or-tools_cp-sat-free	22	100
e	choco-solver__cp-sat_-free	0	62
g	cp_optimizer-free	0	29
k	huub-free	0	5
l	izplus-free	0	2
c	choco-solver__cp_-free	0	1
d	choco-solver__cp_-par	0	1

Table 5.4: Solver suggestions from `gpt-oss-120b` under configuration 1. Columns are computed as in Table 5.2.

MiniZinc Challenge [10]. All referenced problems and corresponding instances are available at [9].

An additional motivation for limiting the analysis to a qualitative level is epistemological in nature: it cannot be determined with certainty whether the reported reasoning reflects the actual internal decision process of the model, or whether the solver selection is produced first and subsequently rationalized in a post hoc manner [52]. Consequently, the following discussion focuses on consistency between the articulated arguments and the observed empirical performance, rather than on claims regarding the authenticity of the reasoning trace.

The following paragraphs present selected responses from the LLM (in the text box), followed by a critical comparison with benchmark outcomes from the MiniZinc Challenge [10].

mondoku/10-10-6

GPT-5.2 Answer:

`[choco-solver__cp_-free, cp_optimizer-free, choco-solver__cp-sat_-free]`

The model is a finite-domain constraint programming problem with 100 integer decision variables over a small domain $[1, 6]$, extensive use of global_cardinality constraints, value precedence (symmetry breaking), and numerous reified linear constraints. The structure is highly combinatorial and dominated by global and logical constraints rather than large-scale linear relaxations.

Pure MILP solvers (cbc, cplex, gurobi, highs, scip in MIP mode) are less suitable because the problem relies on global_cardinality and precedence constraints that propagate strongly in CP but translate weakly into linear relaxations. SAT-centric solvers (pumpkin) are less aligned due to significant integer-domain reasoning beyond Boolean structure.

Choco CP and CP-SAT configurations are well matched to finite-domain propagation, global constraints, and reification. IBM CP Optimizer is particularly strong for combinatorial propagation and symmetry handling. Therefore, the most appropriate solvers are the CP-

focused engines: standard Choco CP, IBM CP Optimizer, and Choco CP-SAT.

This assessment is partially aligned with the benchmark results. Both `cp_optimizer-free` and `choco-solver__cp-sat_-free` are indeed capable of finding an optimal solution. However, `choco-solver__cp_-free`, which is suggested as the most appropriate solver, does not reach optimality within the imposed limits. The critique of MILP solvers is supported by empirical evidence, as most of them fail to solve the instance effectively; an exception is `gurobi-free`, which in this case reaches optimality.

tower/tower_070_070_15_070-08

GPT-5.2 Answer:

`[cp_optimizer-free, choco-solver__cp-sat_-free, choco-solver__cp_-free]`
The model is dominated by Boolean-integer channeling (1120 bool2int, 1050 int_eq_reif) and reified linear constraints, with a relatively small objective domain and structured sequential search. This profile is characteristic of a constraint programming formulation with strong reliance on propagation over reified constraints rather than pure linear relaxations.
`cp_optimizer-free` is well suited due to advanced propagation on reified arithmetic constraints, efficient handling of large numbers of auxiliary variables, and robust search heuristics for structured discrete models.
`choco-solver__cp-sat_-free` is appropriate because the problem contains a high density of Boolean variables and reified constraints; hybrid CP/SAT reasoning can exploit clause learning and Boolean propagation effectively.
`choco-solver__cp_-free` is also competitive, given its native support for finite-domain propagation, customizable search strategies (including first-fail), and good performance on combinatorial models with mixed integer-Boolean structure.
MILP-oriented solvers (CBC, CPLEX, Gurobi, HiGHS, SCIP) are less aligned with the heavy reification and element constraints unless strong linear relaxations are present, which is not evident from the structure described.

In this case, both the solver ranking and the supporting reasoning are consistent with benchmark outcomes. `cp_optimizer-free` is the best-performing solver on this instance, and both it and `choco-solver__cp_-free` reach optimality. Moreover, MILP-oriented solvers are among the weakest performers, in agreement with the model's structural analysis. The justification provided aligns closely with the observed solver behavior.

is/BnMUR3YMRk

GPT-5.2 Answer:

`[cp_optimizer-free, choco-solver__cp-sat_-free, scip-free]`
The model is predominantly combinatorial, with a circuit constraint (`fzn_circuit`), numerous reified equalities/inequalities, table constraints, and Boolean clauses, indicating a

structured CP problem with graph and logical components rather than a purely linear MILP formulation. The objective is linear but simple, and the search strategy is explicitly sequential with first-fail branching, consistent with CP-oriented solving.
`cp_optimizer-free` is well suited due to strong propagation for global constraints (including circuit-like structures) and effective search control. `choco-solver__cp-sat_-free` is appropriate given the large number of reified and Boolean constraints, benefiting from SAT-style reasoning integrated with CP. scip-free is competitive because of its hybrid CP/MIP architecture, capable of handling discrete structures and reified logic while exploiting presolve and constraint integration.
Pure MILP solvers (cbc-free, cplex-free, gurobi-free, highs-free) are less aligned with the model's structural emphasis on global graph and logical constraints, and lighter CP systems (`jacop-free`, `izplus-free`, `sicstus_prolog-free`) are typically less competitive on large hybrid combinatorial instances of this scale. Pumpkin-free is specialized for SAT/Max-SAT and less suitable for the mixed integer and global constraint structure present here.

In this case, both the selection and the justification are inconsistent with empirical results. The solver suggested as the most appropriate, `cp_optimizer-free`, is in fact the only one among the considered candidates that fails to produce a solution for this instance. Contrary to the model's assessment, lighter CP systems exhibit the strongest performance: `sicstus_prolog-free` and `pumpkin-free` are the best-performing solvers, followed closely by the pure MILP solver `gurobi-free`.

fbd1/FBDk09

GPT-5.2 Answer:

`[cplex-free, gurobi-free, scip-free]`
The model is dominated by linear structure: the objective is linear ($a = 2b + c$), and the constraints consist primarily of linear equalities and inequalities (`int_lin_eq`, `int_lin_ne`, `int_lin_le`), together with a single large all-different constraint of arity 135. The search space over the 10 user variables (domain $[1, 735]$) is combinatorial but structurally regular, and the remaining variables are largely compiler-introduced and linearly defined.
Such a formulation is close to a mixed-integer linear programming (MILP) model with one global all-different constraint, which can be encoded through linear constraints. Therefore, high-performance MILP solvers with strong presolve, cutting planes, and branch-and-bound are expected to perform robustly.
`cplex-free` and `gurobi-free` provide state-of-the-art MILP technology, including advanced presolve reductions and highly optimized branching strategies, which are well suited for dense linear systems and medium-scale combinatorial domains of this type.
`scip-free` combines MILP and constraint programming techniques, offering native handling of integrality and additional flexibility for global constraints such as `all-different`.

Its hybrid architecture makes it particularly appropriate when linear structure coexists with discrete global constraints.
Pure CP solvers (e.g., Choco, JaCoP) are capable of handling the model, but the heavy linear algebraic component and limited number of decision variables favor MILP-based engines in this case.

This analysis is not supported by the benchmark outcomes. All solvers suggested by the model fail to produce a solution for the instance, as do all pure MILP solvers more generally. In contrast, pure CP solvers consistently achieve the best objective values among the tested systems, the same one for all of them, even though none is able to prove optimality within the time limit.

proteindesign12/1UBI.13p.9aa.usingEref_self_x

GPT-5.2 Answer:

`[cplex-free, gurobi-free, scip-free]`
The model is a pure integer minimization problem with a linear objective of the form $a = b+ c+\ldots$, represented internally as a single high-arity `int_lin_eq` constraint (arity 92). There are no global CP constraints such as `all-different`, `cumulative`, or scheduling-specific propagators. The remaining 91 constraints are low-arity `fzn_table_int` constraints, which are extensionally defined relations of small scope (average arity 2.86). Structurally, this corresponds to a mixed-integer linear model with additional small discrete compatibility relations.
The search strategy defined in the FlatZinc model (four sequential integer phases, first-fail, complete exploration) is generic and does not exploit advanced CP-specific global propagation. Given that the objective is linear, the constraint graph is dominated by a single dense linear equality and numerous small table constraints, the problem is naturally amenable to MILP reformulation. Modern MILP solvers efficiently linearize table constraints and benefit from presolve, cutting planes, and branch-and-bound with strong LP relaxations.
Among the candidates:
`cplex-free` and `gurobi-free` provide state-of-the-art MILP performance, including aggressive presolve, strong cutting-plane generation, and parallel branch-and-bound. They are well suited to dense linear objectives combined with many medium-to-large integer domains.
scip-free integrates MILP and CP techniques, offering flexible handling of discrete structures and strong presolve, making it particularly appropriate when table constraints interact nontrivially with linear structure.
Pure CP solvers (e.g., Choco, JaCoP, CP Optimizer) would not gain significant advantage from specialized global constraints here, since the model is dominated by linear structure rather than combinatorial global patterns. SAT-oriented solvers (e.g., `pumpkin-free`) are

less natural given the large integer domains and linear objective. Highs and CBC are solid open-source MILP solvers but typically less competitive than CPLEX, Gurobi, or SCIP on structured mixed-integer models of this type.
Therefore, the three most suitable solvers for this instance are `cplex-free`, `gurobi-free`, and `scip-free`.

Empirical results contradict this assessment. The best-performing solvers for this instance are `choco-solver__cp-sat_-free` and `choco-solver__cp_-free`, which achieve identical objective values and correspond to CP and hybrid CP/SAT paradigms. These solver types are characterized in the model's reasoning as less suitable for the given problem.

Conversely, `cplex-free` and `gurobi-free` are among the weakest performers on this instance, and `scip-free` fails to return a solution.

5.2.1 Overall Analysis

As can be inferred from the preceding examples and their analysis, even when employing a state-of-the-art LLM such as `GPT-5.2` [11], the generated reasoning is consistently articulated in a technically coherent and internally consistent manner. However, in several cases, most notably `fbd1` and `is`, the conclusions drawn by the model diverge substantially from the empirical evidence.

Despite these inaccuracies, the responses demonstrate a structured understanding of solver paradigms. The model systematically distinguishes among major solver categories (e.g., CP, MILP, hybrid CP/SAT) and correctly associates individual solvers with their respective classes. Furthermore, the structural analyses of the benchmark problems are often well-formulated, highlighting relevant features such as global constraints, reification, linear structure, and search strategies. This further suggests that the textual representations produced by `fzn2nl` (Section 4.3) are interpretable and informative for high-level problem characterization.

The primary limitation appears to lie not in the recognition of solver categories or problem features per se, but in the mapping between these elements and actual solver performance. In particular, the model exhibits limited sensitivity to the nuanced distinctions that differentiate solvers within the same paradigm. As a consequence, selections frequently reflect a preference for widely recognized or institutionally prominent solvers rather than those empirically best suited to the instance under consideration. More specifically, `cp_optimizer-free` is systematically favored when a CP solver is deemed appropriate, and `cplex-free` is similarly prioritized within the MILP category; both solvers are developed by IBM, a widely recognized institution in computer science.

This tendency persists even though, within the restricted solver set, as explained in Section 3.6, these solvers are not consistently the top-performing options. The observed bias may plausibly be attributed to the prominence of certain commercial solvers in publicly available

documentation and training data, leading the model to associate them with higher general competence. In contrast, alternative solvers with stronger empirical performance in specific instances, such as `gurobi-free`, are selected less consistently.

Chapter 6

Conclusions

This thesis investigated the feasibility of employing LLMs as decision components for solver selection in CP. The central research question concerned whether a general-purpose LLM, without task-specific fine-tuning, can infer appropriate solver choices from structural characteristics of combinatorial models within a predefined solver portfolio.

The study focused on MiniZinc [41] models derived from benchmark instances of the MiniZinc Challenge [9]. Several input representations were evaluated, including raw MiniZinc scripts, structured feature vectors, natural-language descriptions, and natural-language abstractions derived from FlatZinc models. These representations were combined with different prompting strategies and inference configurations in order to assess their impact on solver selection performance. The objective was to determine whether structural information about a problem instance can be effectively interpreted by an LLM and translated into meaningful solver recommendations.

The experimental results indicate that general-purpose LLMs exhibit a limited but non-negligible capacity to interpret structural properties of constraint models. Initial experiments showed that naïve prompting strategies based on raw MiniZinc scripts produced results below the performance of the single best solver. Introducing structured natural-language descriptions produced modest improvements, suggesting that the representation of the input problem significantly influences the model's ability to interpret relevant structural information.

Further experiments explored more structured input formats. Feature vectors generated by the `mzn2feat` tool [17] were evaluated both independently and in combination with other representations. However, this approach did not yield improvements over the best configurations obtained with natural-language inputs. Additional tuning of inference parameters, particularly sampling temperature, produced small performance gains, indicating that controlled stochasticity can influence solver selection behaviour.

Despite these adjustments, the overall performance of the LLM-based approach remained below that of the best single solver. A recurring pattern emerged in which the model consistently selected a small subset of globally dominant solvers across most instances. While this strategy is not inherently ineffective given the strong empirical performance of those solvers, it limits

the discriminative capacity of the selection process and reduces sensitivity to instance-specific characteristics.

To further investigate this behaviour, additional experiments were conducted using a restricted solver portfolio. In this setting, accurate discrimination among solvers becomes necessary to achieve competitive performance. Under these conditions, the limitations of the approach became more apparent, as the LLM-based selection consistently performed below the best solver within the restricted set.

Motivated by the observed influence of natural-language representations, this thesis introduced `fzn2nl` [47], a tool that converts FlatZinc models into deterministic natural-language descriptions. The resulting representation preserves structural information while reducing syntactic noise and token usage. Experimental evaluation shows that this representation achieves performance comparable to other input modalities while requiring substantially fewer tokens, demonstrating improved efficiency in terms of context usage.

Additional experiments examined the internal decision dynamics of the LLM. Controlled perturbation tests revealed that solver names strongly influence the model's choices: when solver descriptions were permuted while preserving solver names, the resulting selections remained largely unchanged. Conversely, anonymizing solver names increased the influence of descriptive content, indicating that the model can partially associate solver characteristics with problem structure when name-based biases are removed. Nevertheless, solver choices remained concentrated on a small subset of candidates.

Qualitative analysis of generated explanations further indicates that the model possesses a coherent high-level understanding of constraint models and solver paradigms. The LLM correctly identifies structural aspects such as constraint types, objective structure, and variable domains, and it is generally capable of associating these properties with broad solver categories (e.g., CP, MILP, or CP-SAT). However, the model shows limited ability to discriminate between solvers within the same category or to relate fine-grained instance characteristics to empirically optimal solver choices.

The conclusions of this study must be interpreted in light of several limitations. The experimental campaign was constrained by API rate limits and token throughput restrictions imposed by commercial LLM services. These constraints limited the number of evaluated instances and prevented extensive repeated trials across different inference configurations.

A further limitation concerns the intrinsic non-determinism of LLM inference. Since model outputs depend on stochastic sampling mechanisms, identical prompts may produce different solver recommendations across runs. This variability affects the stability of individual predictions and complicates systematic evaluation.

In addition, the empirical analysis was restricted to benchmark instances derived from the MiniZinc Challenge. Although this benchmark suite represents a widely used evaluation standard in CP, it does not cover the full diversity of real-world modeling scenarios.

Future work may address these limitations by extending the evaluation to larger and more

heterogeneous benchmark collections and by assessing more advanced LLM architectures with larger context windows and improved reasoning capabilities. Another promising direction concerns the integration of task-specific learning. Fine-tuning LLMs on solver-performance datasets could enable models to internalize empirical solver competitiveness and strengthen the relationship between structural instance features and solver selection decisions.

Overall, the results indicate that current general-purpose LLMs are not yet competitive with specialized solver-selection methods. Nevertheless, they demonstrate a meaningful capacity to interpret structural properties of constraint models and solver paradigms. These findings suggest that, with appropriate methodological developments and domain-specific adaptation, LLM-based components may become a viable element of future CP workflows.

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 6.1: Parallel-solver evaluation results for all LLMs on unprocessed (plain) MiniZinc scripts. The column "Model" identifies each tested LLM by its API name. "Total Score" corresponds to the Parallel Score metric as defined in Table 3.1. "Instances Covered" reports the number of instances for which a valid response was obtained, i.e., those whose token count did not exceed the rate limits reported in Table 2.1 and Table 2.2. "Average Score" is computed as $^{\text{Total Score}}/_{\text{Instances Covered}}$ and reflects solver selection quality over 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 6.2: Single-solver evaluation results for all LLMs on unprocessed (plain) MiniZinc scripts. "Total Score" corresponds to the Single Score metric as defined in Table 3.1. The remaining columns are computed as described in Table 6.1.

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 6.3: Closed-gap evaluation results for all LLMs on unprocessed (plain) MiniZinc scripts. "Instances Covered" is defined as in Table 6.1. "AS" denotes the Aggregate Score, equivalent to the Single Score in Table 3.1. "SBS" reports the total score accumulated by the single best solver (`or-tools_cp-sat-free`) across all instances; "VBS" reports the score of the hypothetical Virtual Best Solver, which achieves the maximum attainable score on every instance. "Closed Gap" is computed as defined in Table 3.1.

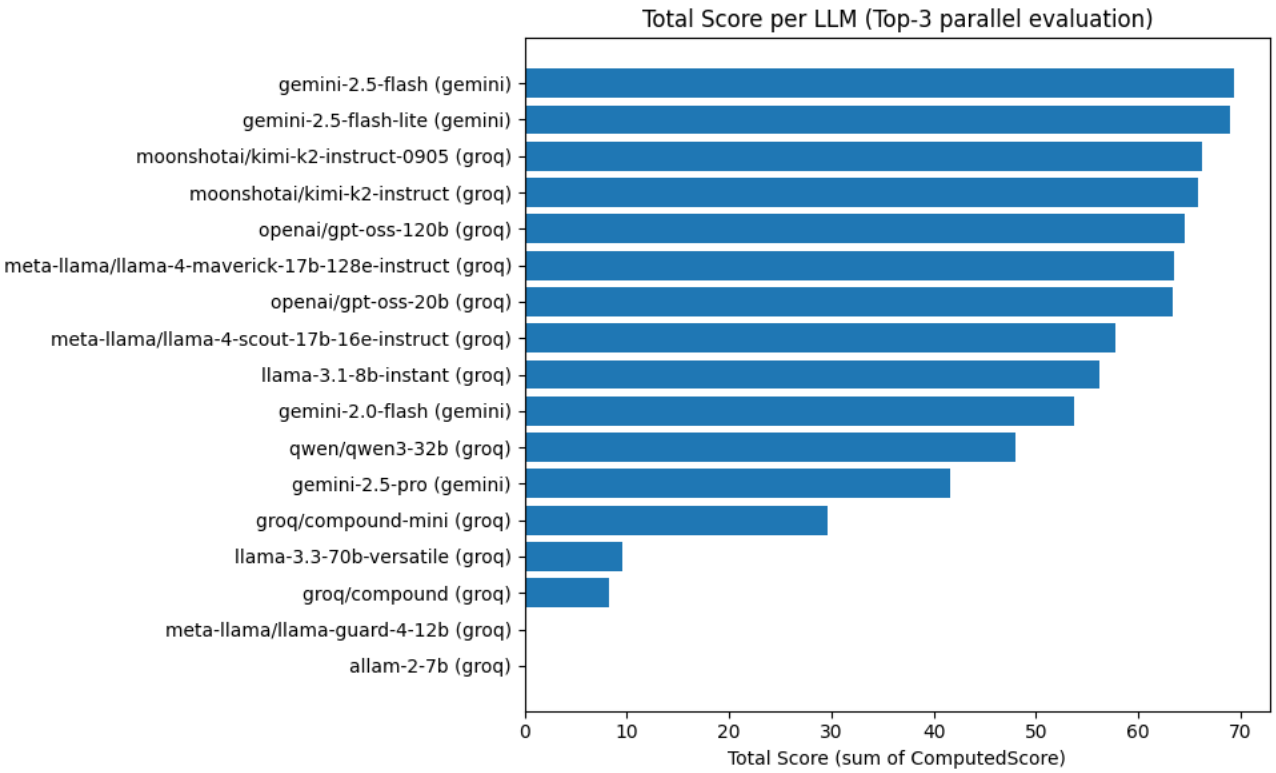

Figure 6.1: Total scores obtained by all evaluated LLMs under the parallel-solver evaluation scheme. "Total Score" corresponds to the Parallel Score metric as defined in Table 3.1.

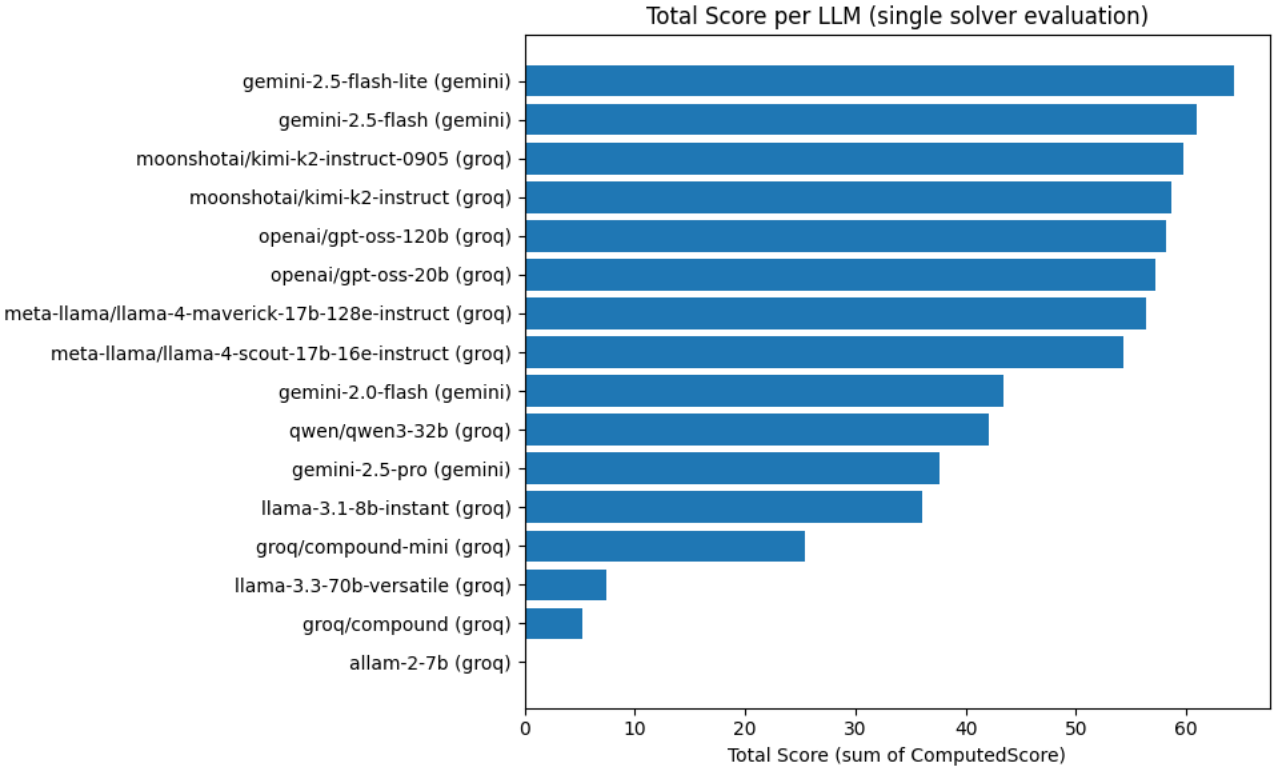

Figure 6.2: Total scores obtained by all evaluated LLMs under the single-solver evaluation scheme. "Total Score" corresponds to the Single Score metric as defined in Table 3.1.

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 6.4: Parallel-solver evaluation results for the five best-performing LLMs on sanitized MiniZinc scripts. Column definitions are as given in Table 6.1.

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 6.5: Single-solver evaluation results for the five best-performing LLMs on sanitized MiniZinc scripts. Column definitions are as given in Table 6.2.

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 6.6: Closed-gap evaluation results for the five best-performing LLMs on sanitized MiniZinc scripts. Column definitions are as given in Table 6.3.

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 6.7: Parallel-solver evaluation results for the five best-performing LLMs on sanitized scripts augmented with a textual problem description. Column definitions are as given in Table 6.1.

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 6.8: Single-solver evaluation results for the five best-performing LLMs on sanitized scripts augmented with a textual problem description. Column definitions are as given in Table 6.2.

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 6.9: Closed-gap evaluation results for the five best-performing LLMs on sanitized scripts augmented with a textual problem description. Column definitions are as given in Table 6.3.

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 6.10: Examples of solver description for two solvers, namely COIN-OR Branch-and-Cut (CBC) [3], and Chuffed [4].

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 6.11: Parallel-solver evaluation results for the five best-performing LLMs on sanitized scripts augmented with solver descriptions, using a multi-turn prompt setup. Column definitions are as given in Table 6.1.

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 6.12: Single-solver evaluation results for the five best-performing LLMs on sanitized scripts augmented with solver descriptions, using a multi-turn prompt setup. Column definitions are as given in Table 6.2.

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 6.13: Closed-gap evaluation results for the five best-performing LLMs on sanitized scripts augmented with solver descriptions, using a multi-turn prompt setup. Column definitions are as given in Table 6.3.

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 6.14: Parallel-solver evaluation results for the five best-performing LLMs on sanitized scripts augmented with both solver descriptions and textual problem descriptions, using a multi-turn prompt setup. Column definitions are as given in Table 6.1.

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 6.15: Single-solver evaluation results for the five best-performing LLMs on sanitized scripts augmented with both solver descriptions and textual problem descriptions, using a multi-turn prompt setup. Column definitions are as given in Table 6.2.

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 6.16: Closed-gap evaluation results for the five best-performing LLMs on sanitized scripts augmented with both solver descriptions and textual problem descriptions, using a multi-turn prompt setup. Column definitions are as given in Table 6.3.

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 6.17: Comprehensive comparison of all LLM configuration variants under the parallel-solver evaluation scheme against all individual solvers participating in the open category of the MiniZinc Challenge 2025 [10]. "Total Score" is computed as the Parallel Score defined in Table 3.1. LLM variant names are suffixed with the prompt configuration used (Simple, Problem Description, Solvers Description, or both).

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 6.18: Comprehensive comparison of all LLM configuration variants under the single-solver evaluation scheme against all individual solvers participating in the free category of the MiniZinc Challenge 2025 [10]. "Total Score" is computed as the Single Score defined in Table 3.1. The best-performing LLM variant is highlighted in bold.

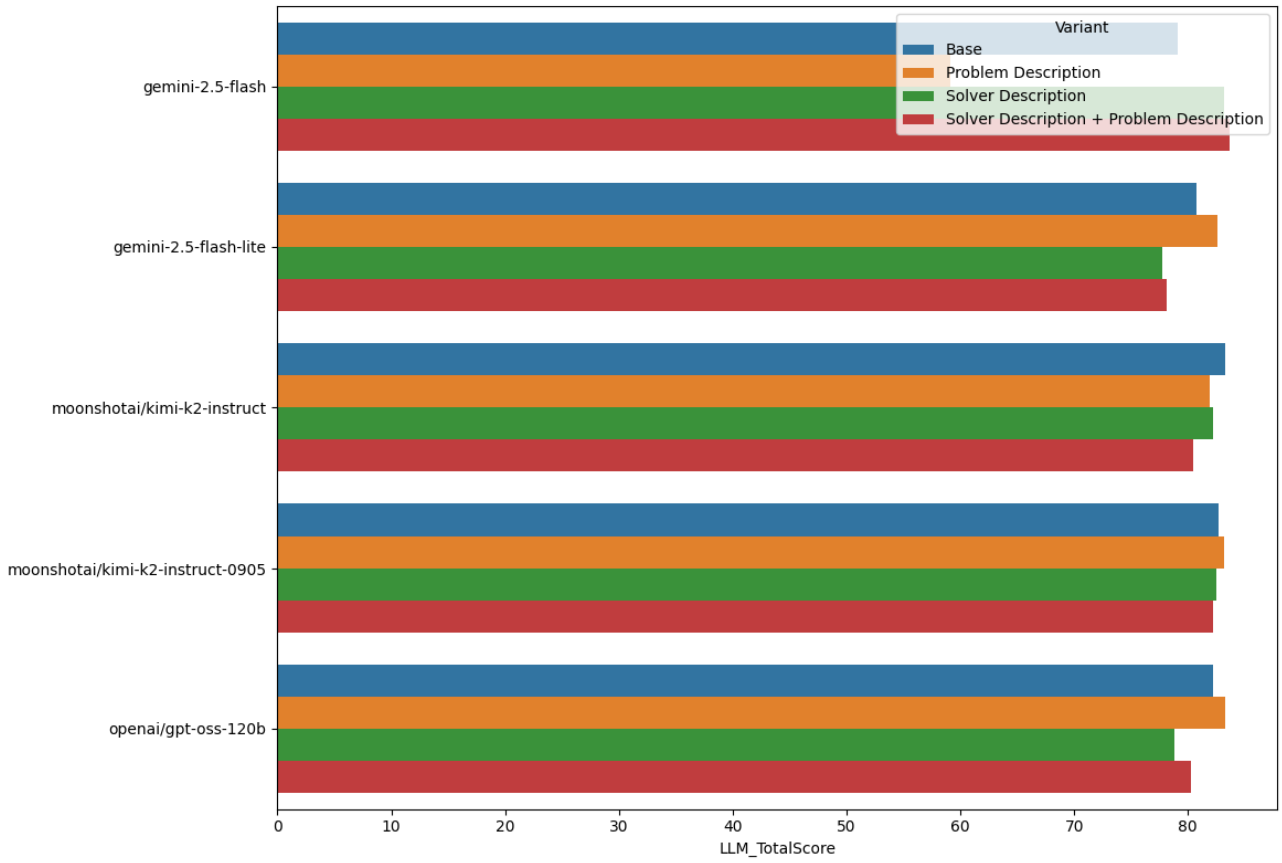

Figure 6.3: Bar chart displaying the Parallel Score of all LLM configuration variants, as defined in Table 3.1. Each bar corresponds to a distinct LLM–configuration pair.

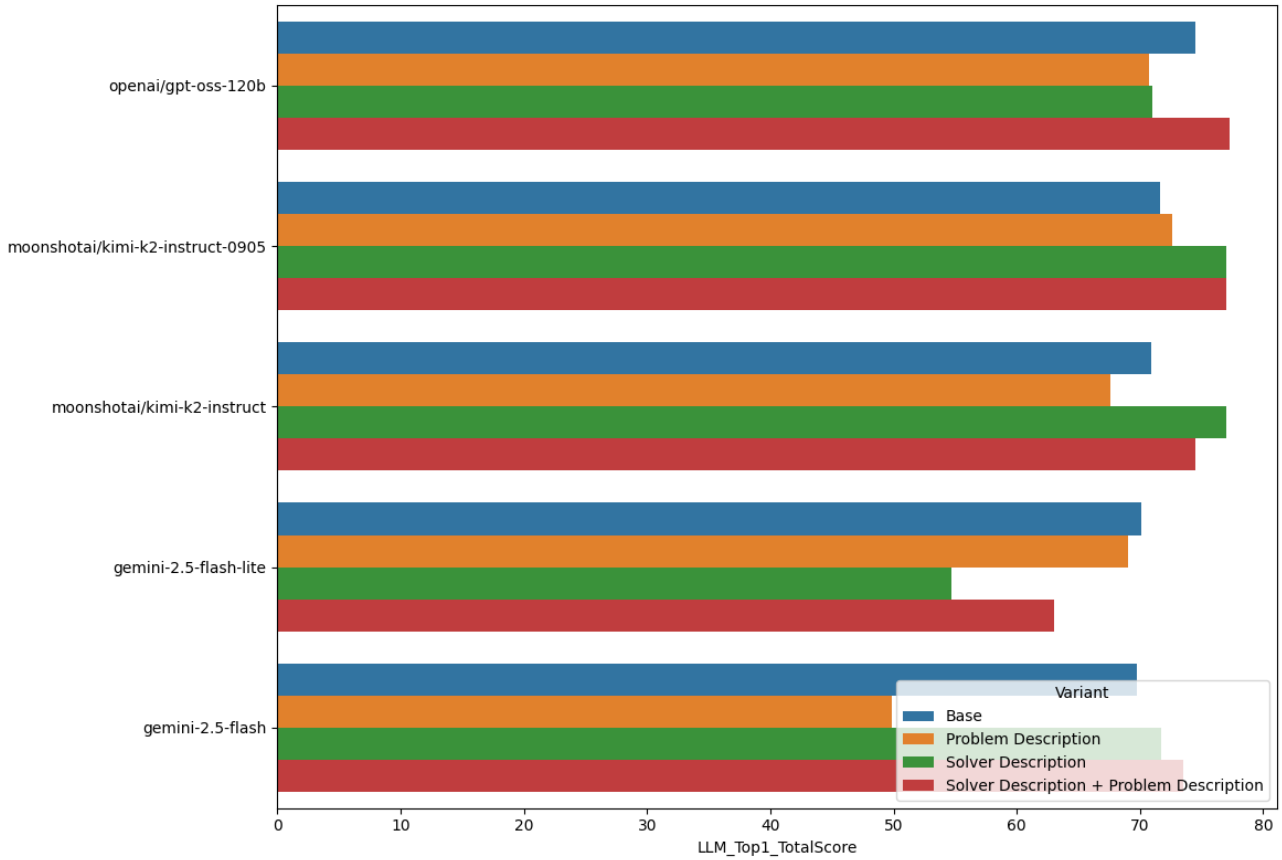

Figure 6.4: Bar chart displaying the Single Score, Parallel Score, and Closed Gap of all LLM configuration variants, as defined in Table 3.1. Each group of bars corresponds to a distinct LLM–configuration pair.

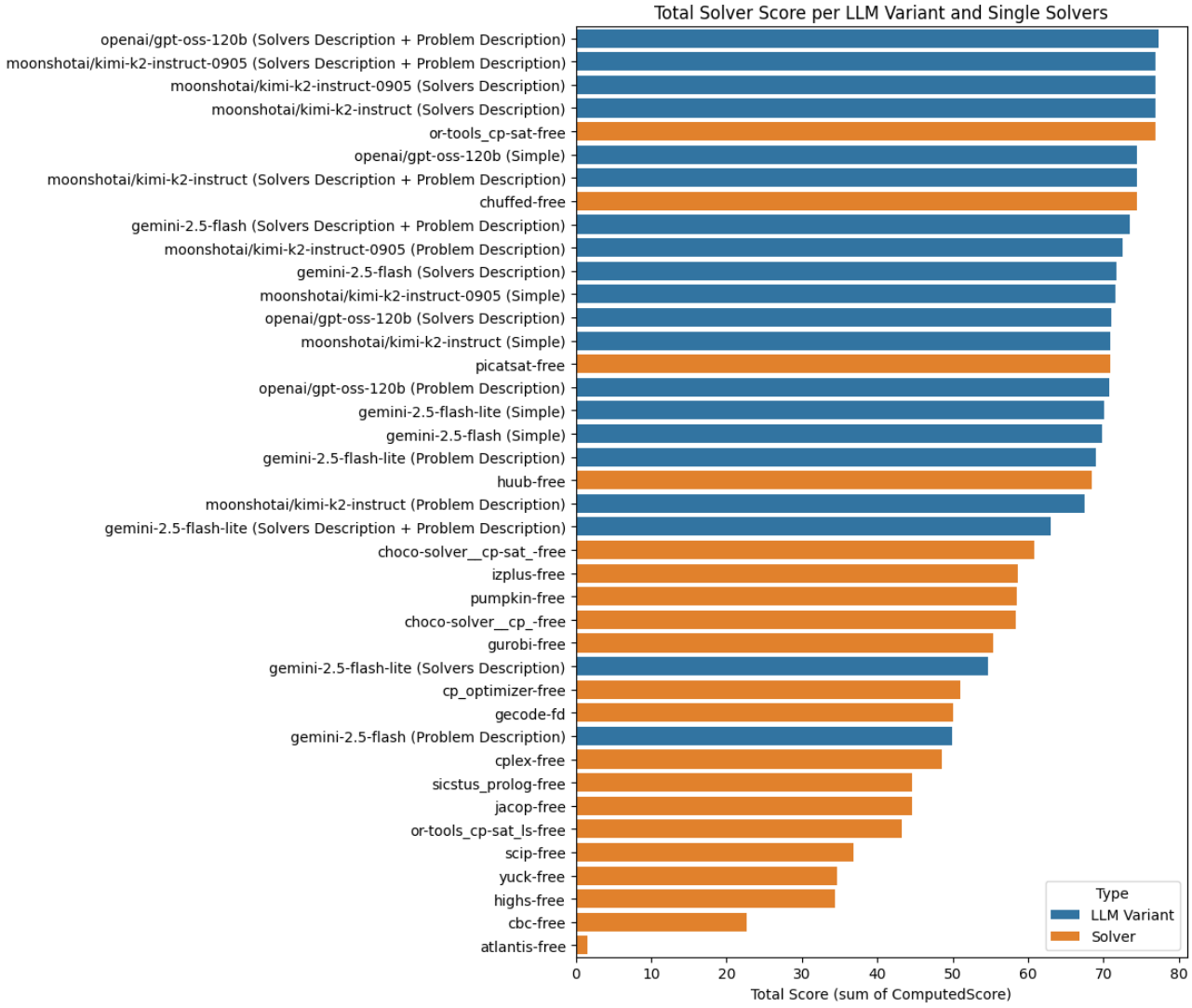

Figure 6.5: Bar chart comparing the Single Score of all LLM configuration variants against individual solvers from the free category of the MiniZinc Challenge 2025. “Total Score” corresponds to the Single Score metric as defined in Table 3.1.

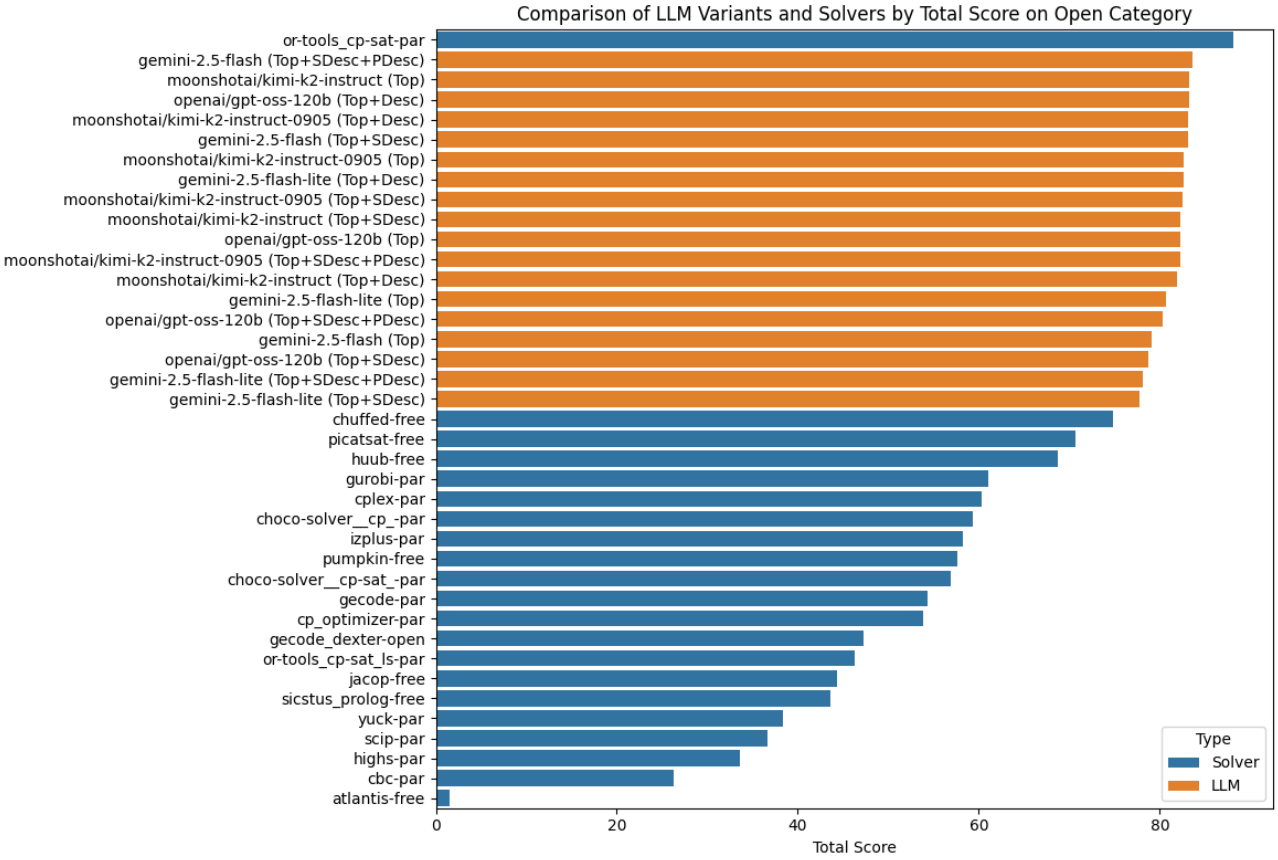

Figure 6.6: Bar chart comparing the Parallel Score of all LLM configuration variants against individual solvers from the open category of the MiniZinc Challenge 2025. “Total Score” corresponds to the Parallel Score metric as defined in Table 3.1.

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 6.19: Summary of the feature categories extracted by the `mzn2feat` tool [17]. Each row describes a feature category, reports the number of features it contains, and provides a brief description of the quantities measured. Static features are derived from syntactic and structural analysis of the FlatZinc model; dynamic features are collected at runtime from short Gecode executions.

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 6.1: Example output of the `mzn2feat` tool [17] (pretty-print mode) applied to the `EchoSched.mzn` model, instance `14-10-0-2_1`. Each row lists the feature identifier, its computed value, and a brief description.

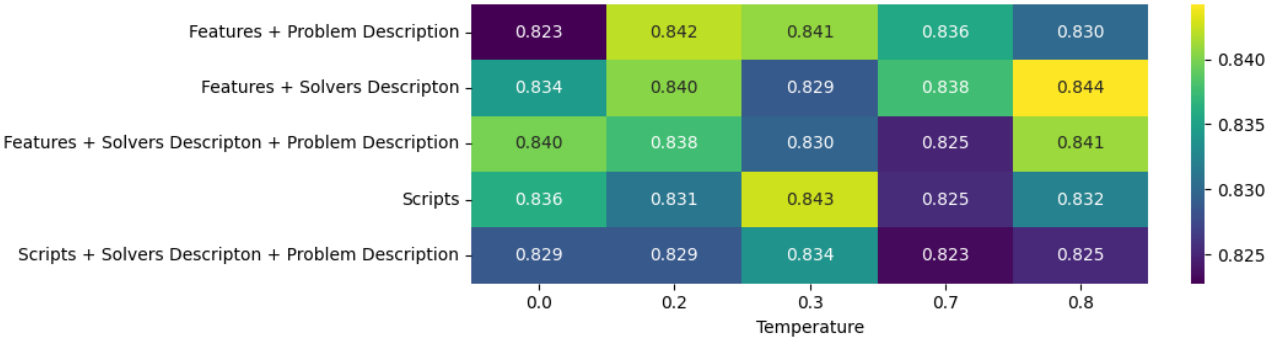

Figure 6.7: Heatmap of Parallel Score results across all tested temperature values and the five best-performing prompt configurations, as evaluated on `gpt-oss-120b`. The Parallel Score is computed as defined in Table 3.1.

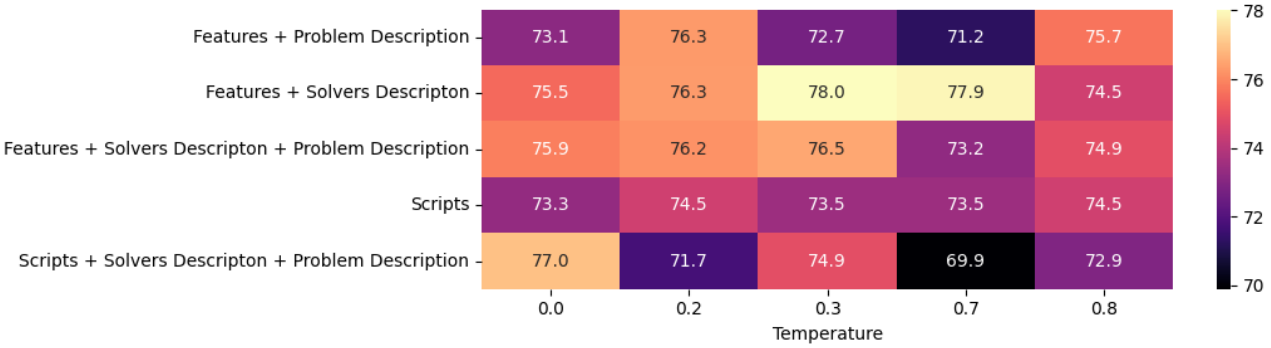

Figure 6.8: Heatmap of Single Score results across all tested temperature values and the five best-performing prompt configurations, as evaluated on `gpt-oss-120b`. The Single Score is computed as defined in Table 3.1.

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 6.20: All solvers from the free category of the MiniZinc Challenge 2025, ranked by the number of instances on which they achieved an optimal score of 1 under the scoring metric defined in Section 2.5. The "Score" column reports the total accumulated score across all instances.

Suggested Solver	Description	Top1 Count	Total Count
or-tools_cp-sat-free	atlantis-free	72	99
atlantis-free	cplex-free	21	48
chuffed-free	highs-free	7	51
highs-free	or-tools_cp-sat_ls-free	0	78
yuck-free	choco-solver__cp_-par	0	24

Table 6.21: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 2, using the solver-name-swapping experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
or-tools_cp-sat-free	atlantis-free	76	95
chuffed-free	highs-free	24	100
cp_optimizer-free	or-tools_cp-sat-free	0	74
picatsat-free	cp_optimizer-free	0	21
choco-solver__cp_-free	jacop-free	0	6
scip-free	choco-solver__cp-sat_-free	0	3
or-tools_cp-sat_ls-free	yuck-free	0	1

Table 6.22: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 3, using the solver-name-swapping experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
or-tools_cp-sat-free	atlantis-free	78	100
chuffed-free	highs-free	20	100
cp_optimizer-free	or-tools_cp-sat-free	2	45
picatsat-free	cp_optimizer-free	0	52
choco-solver__cp_-free	jacop-free	0	3

Table 6.23: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 4, using the solver-name-swapping experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
or-tools_cp-sat-free	atlantis-free	56	95
chuffed-free	highs-free	40	100
atlantis-free	cplex-free	4	22
scip-free	choco-solver__cp-sat_-free	0	36
yuck-free	choco-solver__cp_-par	0	35
picatsat-free	cp_optimizer-free	0	6
cp_optimizer-free	or-tools_cp-sat-free	0	3
choco-solver__cp_-free	jacop-free	0	2
choco-solver__cp-sat_-free	sicstus_prolog-free	0	1

Table 6.24: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 5, using the solver-name-swapping experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
f	chuffed-free	69	100
n	or-tools_cp-sat-free	31	100
e	choco-solver__cp-sat_-free	0	79
g	cp_optimizer-free	0	15
l	izplus-free	0	3
c	choco-solver__cp_-free	0	1
d	choco-solver__cp_-par	0	1
k	huub-free	0	1

Table 6.25: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 2, using the anonymous (single-letter) solver-name experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
f	chuffed-free	52	96
n	or-tools_cp-sat-free	44	95
e	choco-solver__cp-sat_-free	0	83
l	izplus-free	0	5
o	or-tools_cp-sat_ls-free	0	5
d	choco-solver__cp_-par	0	3
k	huub-free	0	1

Table 6.26: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 3, using the anonymous (single-letter) solver-name experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
n	or-tools_cp-sat-free	59	89
f	chuffed-free	32	90
c	choco-solver__cp_-free	1	3
k	huub-free	1	2
e	choco-solver__cp-sat_-free	0	84
o	or-tools_cp-sat_ls-free	0	5
g	cp_optimizer-free	0	3
l	izplus-free	0	2
d	choco-solver__cp_-par	0	1

Table 6.27: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 4, using the anonymous (single-letter) solver-name experimental setup. Column definitions follow Table 5.2.

Suggested Solver	Description	Top1 Count	Total Count
f	chuffed-free	58	98
n	or-tools_cp-sat-free	38	96
c	choco-solver__cp_-free	2	4
g	cp_optimizer-free	1	54
k	huub-free	1	12
l	izplus-free	0	23
e	choco-solver__cp-sat_-free	0	9
d	choco-solver__cp_-par	0	2
o	or-tools_cp-sat_ls-free	0	2

Table 6.28: Frequency analysis of solver suggestions produced by gpt-oss-120b under configuration 5, using the anonymous (single-letter) solver-name experimental setup. Column definitions follow Table 5.2.

Bibliography

[1] Gemini API docs and reference. https://ai.google.dev/gemini-api/docs. Accessed Dec. 10, 2025.

[2] Gurobi optimizer. https://www.gurobi.com/solutions/gurobi-optimizer/.

[3] Cbc. https://github.com/coin-or/Cbc, May 2022.

[4] Chuffed, a lazy clause generation solver. https://github.com/chuffed/chuffed, December 2023.

[5] GroqCloud. https://console.groq.com/docs/overview, 2024. Accessed Dec. 10, 2025.

[6] API versions explained. https://ai.google.dev/gemini-api/docs/api-versions, 2025. Accessed Dec. 11, 2025.

[7] Rate limits. https://ai.google.dev/gemini-api/docs/rate-limits, 2025. Accessed Dec. 10, 2025.

[8] Rate limits – GroqDocs. https://console.groq.com/docs/rate-limits, 2025. Accessed Dec. 10, 2025.

[9] MiniZinc – list of problems and globals used in the MiniZinc challenge. https://www.minizinc.org/challenge/globals/, 2025. Accessed Dec. 11, 2025.

[10] MiniZinc – challenge 2025 results. https://www.minizinc.org/challenge/2025/results/, 2025. Accessed Dec. 11, 2025.

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

[12] Prompt basics – GroqDocs. https://console.groq.com/docs/prompting, 2026.

[13] GPT-5.2 in ChatGPT – OpenAI help center. https://help.openai.com/en/articles/11909943-gpt-52-in-chatgpt, 2026. Accessed Feb. 11, 2026.

[14] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1):147–169, January 1985. doi: 10.1016/S0364-0213(85)80012-4.

[15] J. Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, 1978.

[16] R. Amadini and Peter J. Stuckey. Sequential time splitting and bounds communication for a portfolio of optimization solvers. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, 2014.

[17] R. Amadini, M. Gabbrielli, and J. Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2014.

[18] R. Amadini, Maurizio Gabbrielli, and J. Mauro. Portfolio approaches for constraint optimization problems. In *Lecture Notes in Computer Science*, pages 21–35, January 2014. doi: 10.1007/978-3-319-09584-4_3.

[19] R. Amadini, M. Gabbrielli, and J. Mauro. Why CP portfolio solvers are (under)utilized? issues and challenges. In *Lecture Notes in Computer Science*, pages 349–364, 2015. doi: 10.1007/978-3-319-27436-2_21.

[20] R. Amadini, M. Gabbrielli, and J. Mauro. SUNNY-CP: a sequential CP portfolio solver. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2015.

[21] R. Amadini, M. Gabbrielli, and J. Mauro. A multicore tool for constraint solving. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[22] R. Amadini, Maurizio Gabbrielli, T. Liu, and J. Mauro. On the evaluation of (meta-)solver approaches. *Journal of Artificial Intelligence Research*, 76:705–719, March 2023. doi: 10.1613/jair.1.14102.

[23] Roberto Amadini and Simone Gazza. From portfolio solvers to agentic solvers. In Tias Guns, Serdar Kadioglu, Stefan Szeider, and Dimos Tsouros, editors, *LLM-Solve Workshop at CP 2025*, pages 569–585, Glasgow, United Kingdom, August 2025. doi: 10.5281/zenodo.17640236. hal-05446738.

[24] R. Barták. Constraint programming: In pursuit of the holy grail. Technical report, January 1999.

[25] F. Chalumeau, I. Coulon, Q. Cappart, and L.-M. Rousseau. SeaPearl: A constraint programming solver guided by reinforcement learning. arXiv:2102.09193, April 2021. URL `https://arxiv.org/abs/2102.09193`.

[26] J. Cheng, M. Marone, O. Weller, D. Lawrie, D. Khashabi, and V. Durme. Dated data: Tracing knowledge cutoffs in large language models. arXiv:2403.12958, March 2024.

[27] 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*, 6(1), March 2015. doi: 10.14807/ijmp.v6i1.262.

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

[29] Z. Gekhman et al. Does fine-tuning LLMs on new knowledge encourage hallucinations? arXiv:2405.05904, May 2024.

[30] D. Guo. Development of compiler based on flex and bison. *Ordnance Industry Automation*, January 2004.

[31] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. arXiv:1503.02531, March 2015. URL `http://arxiv.org/abs/1503.02531`.

[32] Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts LLM performance. `https://research.trychroma.com/context-rot`, 2025. Accessed Dec. 11, 2025.

[33] IBM. What are large language models (LLMs)? `https://www.ibm.com/think/topics/large-language-models`, November 2023.

[34] L. Kotthoff. Algorithm selection for combinatorial search problems: a survey. *AI Magazine*, 35(3):48–60, 2014.

[35] A. O. Li and T. Goyal. Memorization vs. reasoning: Updating LLMs with new knowledge. arXiv:2504.12523, April 2025.

[36] X. Lin et al. LLM-based agents suffer from hallucinations: A survey of taxonomy, methods, and directions. arXiv:2509.18970, September 2025.

[37] M. Lindauer, J. N. van Rijn, and L. Kotthoff. The algorithm selection competitions 2015 and 2017. *Artificial Intelligence*, 272:86–100, 2019.

[38] Y. Liu et al. Understanding LLMs: A comprehensive overview from training to inference. arXiv:2401.02038, January 2024.

[39] MiniZinc. GitHub – MiniZinc/libminizinc: The MiniZinc compiler. `https://github.com/MiniZinc/libminizinc`, January 2026. Accessed Feb. 04, 2026.

[40] M. Morara, J. Mauro, and M. Gabbrielli. Solving XCSP problems by using Gecode. arXiv:1112.6096, December 2011.

[41] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *Springer eBooks*, pages 529–543, October 2007. doi: 10.1007/978-3-540-74970-7_38.

[42] R. Oentaryo, S. D. Handoko, and H. C. Lau. Algorithm selection via ranking. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, February 2015. doi: 10.1609/aaai.v29i1.9466.

[43] OpenAI. ChatGPT. `https://chatgpt.com/`, 2025.

[44] Sumanth P. Agentic prompt engineering: A deep dive into LLM roles and role-based formatting. `https://www.clarifai.com/blog/agentic-prompt-engineering`, July 2025. Accessed Jan. 03, 2026.

[45] T. Patwardhan et al. GDPval: Evaluating AI model performance on real-world economically valuable tasks. arXiv:2510.04374, October 2025.

[46] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[47] Vittorio Rossetto. fzn2nl – GitHub repository. `https://github.com/VittorioRossetto/fzn2nl`, 2025. Accessed Feb. 06, 2026.

[48] C. Shi et al. A thorough examination of decoding methods in the era of LLMs. arXiv:2402.06925, February 2024.

[49] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. Chi, N. Schärli, and D. Zhou. Large language models can be easily distracted by irrelevant context. arXiv:2302.00093, 2023.

[50] P. J. Stuckey and G. Tack. MiniZinc with functions. In *Lecture Notes in Computer Science*, pages 268–283, 2013. doi: 10.1007/978-3-642-38171-3_18.

[51] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer. The MiniZinc challenge 2008–2013. *AI Magazine*, 35(2):55, June 2014. doi: 10.1609/aimag.v35i2.2539.

[52] P. Sychev, A. Goncharov, D. Vyazhev, E. Khalafyan, and A. Zaytsev. When an LLM is apprehensive about its answers – and when its uncertainty is justified. arXiv:2503.01688, March 2025.

[53] A. Vaswani et al. Attention is all you need. arXiv:1706.03762, 2017. URL `https://arxiv.org/abs/1706.03762`.

[54] P.-H. Wang et al. Contextual temperature for language modeling. arXiv:2012.13575, January 2020.

[55] 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:2305.13733, May 2023. URL `https://arxiv.org/abs/2305.13733`.

[56] X. Wu and K. Tsioutsiouliklis. Thinking with knowledge graphs: Enhancing LLM reasoning through structured data. arXiv:2412.10654, December 2024.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Roberto Amadini, and to AFORM – Settore Servizi Didattici Scienze Cittadella, for the opportunity to carry out the research underlying this thesis in Australia. I also wish to thank the staff of the OPTIMA – ARC Training Centre at Monash University, and especially Prof. Guido Tack, for hosting and supporting me during this research period.