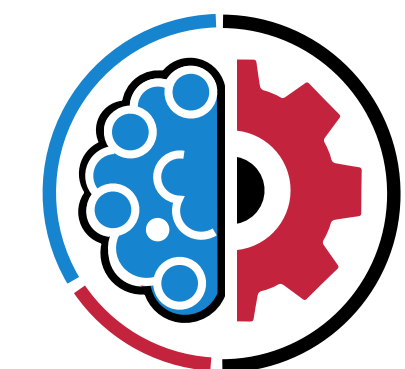




# MCP-Solver: Integrating Language Models with Constraint Programming Systems

**Stefan Szeider**

Algorithms and Complexity Group  
TU Wien, Vienna, Austria



# Motivation and Background

## Motivation:

- LLMs excel in natural language tasks but struggle with precise formal reasoning and specification.
- There is a need for systematic integration between LLMs and constraint programming systems to support reliable constraint model refinement.

## Background:

- Traditional methods lack dynamic interaction, while the MCP provides a standardized protocol for interfacing LLMs with external systems.

By exploiting MCP, the system bridges informal language input and formal constraint models, thereby enhancing the reliability and flexibility of the model refinement process.



# Related Work and the MCP Protocol

## Related Approaches:

- Previous efforts have paired LLMs with verifiers, SMT solvers, and symbolic reasoning tools (e.g., SATLM, LOGIC-LM, GenCP).
- Most solutions are fixed pipelines optimized for specific tasks.

## MCP Protocol:

- Provides a universal, standardized interface to integrate diverse tools and data sources.
- Widely adopted by platforms such as *Zed*, *Replit*, *Codeium*, and notably, *Cursor*—an LLM-based coding tool that also leverages MCP for integration.

The MCP standard is becoming a cornerstone for integrating LLMs with formal systems, as evidenced by its adoption in multiple industry tools including Cursor, which enhances coding workflows through standardized protocols.

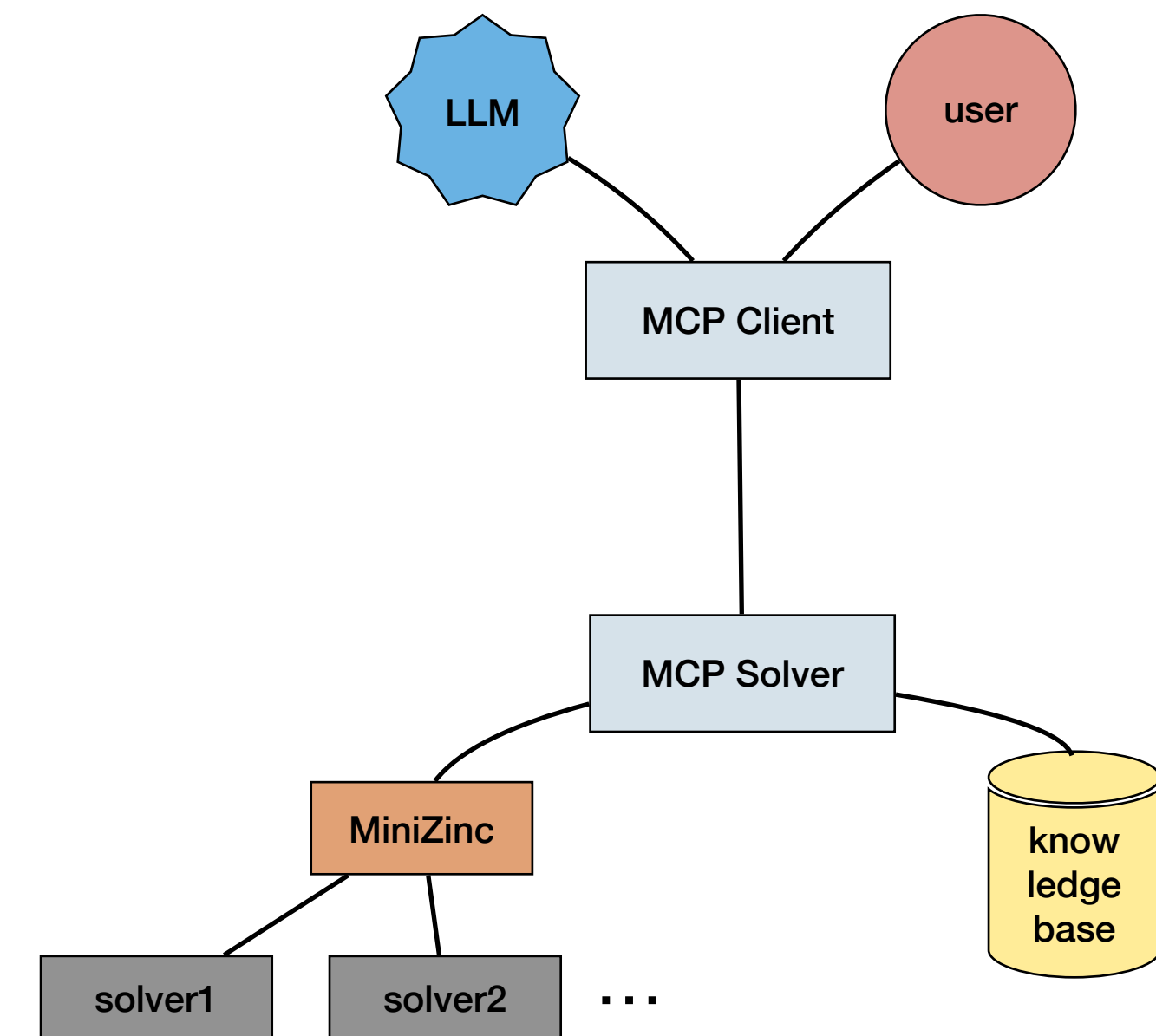
# System Overview – Architecture

## Key Components:

- MCP Client: Interfaces with LLMs (e.g., Claude Desktop) to interact with the system.
- MCP Solver (Server): Manages model editing, validation, and session state.
- MiniZinc Backend: Converts models into FlatZinc and leverages solvers (default: Chuffed) for constraint solving.

## Integration:

- The architecture enables item-based editing, persistent knowledge accumulation, and concurrent solving sessions.



The architecture clearly delineates the roles of each component, ensuring that natural language inputs are reliably translated into valid MiniZinc models while maintaining robust solver integrity.

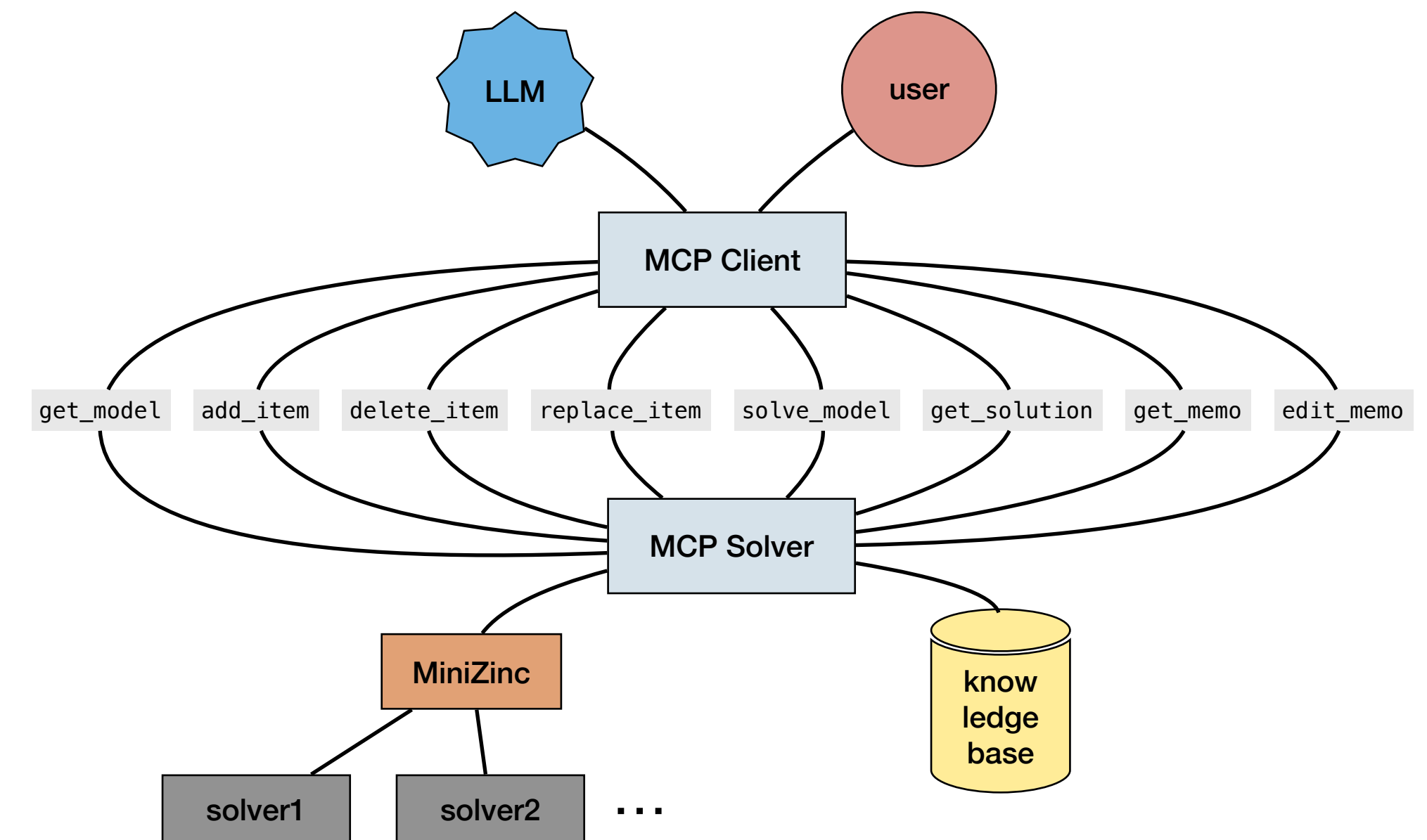
# Design Principles and MCP Tools

## Design Principles:

- Maintain solver integrity through rigorous validation of every model modification.
- Separate model editing from solving operations to ensure consistency and resource cleanup.

## MCP Tools:

- Then operations called *tools* (aka functions)
- These tools enable precise, atomic modifications that are validated before application.



By decomposing the editing process into discrete, validated operations, MCP-Solver ensures that every interaction is both robust and reliable, adhering strictly to the MCP specification.

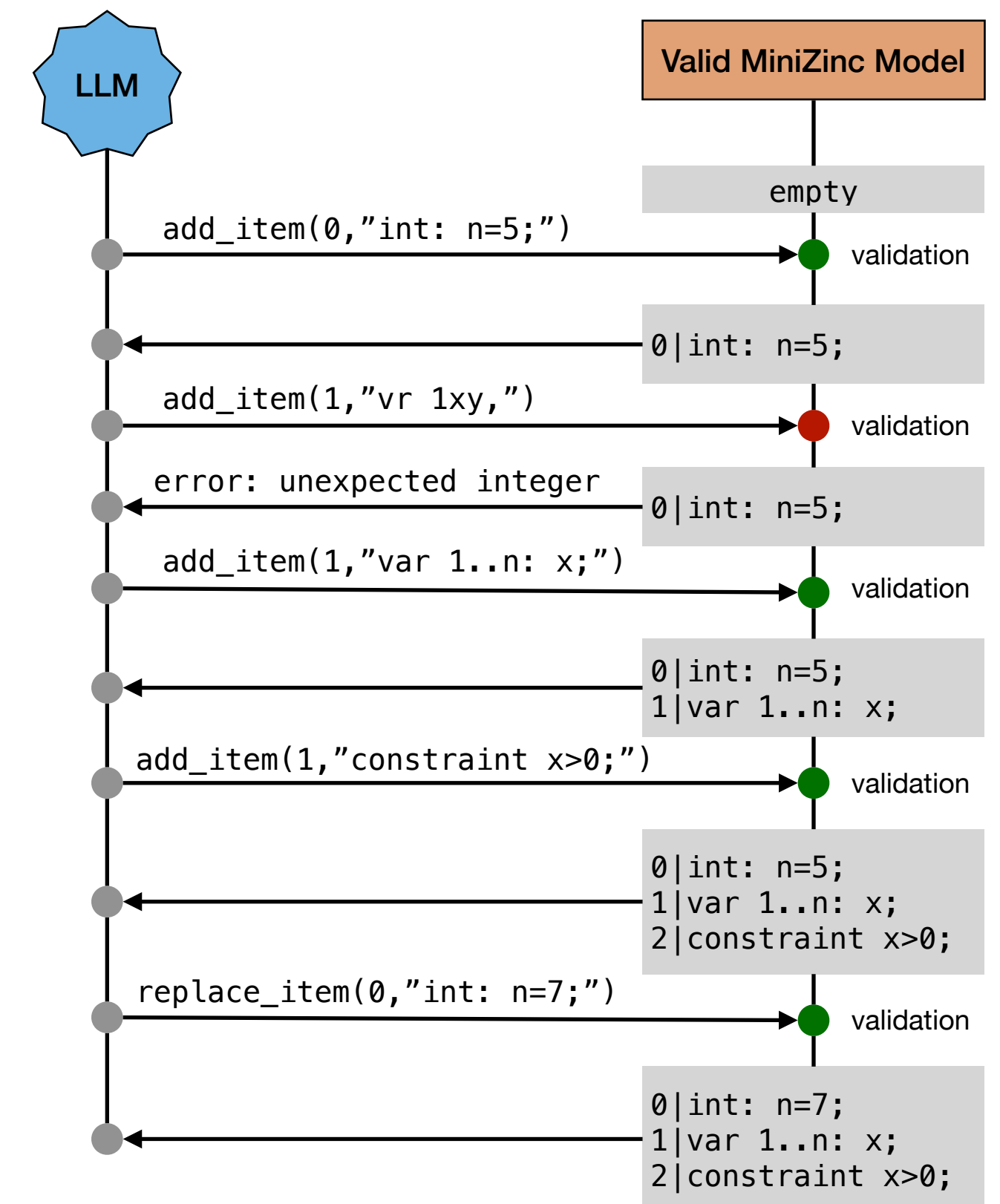
# Model Management – Item-Based Editing

## Item-Based Editing:

- Models are managed as a collection of numbered items (e.g., variable declarations, constraints).
- Each change is validated for syntax, type correctness, and instantiation before being accepted.

## Validation Example:

- An incorrect modification triggers a detailed error message, allowing for targeted correction.



This granular approach to model management ensures that every modification maintains the overall model's consistency, enabling iterative refinement without accumulating errors.

# Persistent Knowledge Base

## Memo System:

- A file-backed memo system captures successful modeling strategies and solver insights across sessions.
- The system uses `get_memo` and `edit_memo` tools to facilitate knowledge accumulation and error recovery.

## Benefits:

- Encourages reuse of proven modeling patterns and supports continuous improvement in the modeling process.

The persistent knowledge base acts as a central repository of insights, enabling the system to learn from past refinements and continuously enhance its performance.

# Preliminary Experimental Evaluation

## Scope:

- The system was tested on various problems (e.g., Casting, Traveling Salesperson, N-Queens, Timetabling) to assess its flexibility and robustness.

## Performance:

- Experiments demonstrate that the MCP-Solver can effectively translate natural language specifications into formal models and iteratively refine them based on user feedback.

Preliminary results validate that MCP-Solver can bridge the gap between natural language and formal constraint models, proving its practical viability for a range of moderately sized problems.

# Example: TSP

User

A saleswoman based in Vienna needs to plan her upcoming tour through Austria, visiting each province capital once. Help find the shortest route. Distances in km: 1 = Vienna, 2 = St. Pölten, 3 = Eisenstadt, 4 = Linz, 5 = Graz, 6 = Klagenfurt, 7 = Salzburg, 8 = Innsbruck, 9 = Bregenz

|   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 0   | 65  | 60  | 184 | 195 | 319 | 299 | 478 | 631 |
| 2 | 65  | 0   | 125 | 119 | 130 | 254 | 234 | 413 | 566 |
| 3 | 60  | 125 | 0   | 184 | 157 | 281 | 261 | 440 | 593 |
| 4 | 184 | 119 | 184 | 0   | 208 | 252 | 136 | 315 | 468 |
| 5 | 195 | 130 | 157 | 208 | 0   | 136 | 280 | 459 | 629 |
| 6 | 319 | 254 | 281 | 252 | 136 | 0   | 217 | 391 | 566 |
| 7 | 299 | 234 | 261 | 136 | 280 | 217 | 0   | 188 | 343 |
| 8 | 478 | 413 | 440 | 315 | 459 | 391 | 188 | 0   | 157 |
| 9 | 631 | 566 | 593 | 468 | 629 | 566 | 343 | 157 | 0   |

LLM

I'll model this as a Traveling Salesperson Problem using a circuit constraint.

# Example: TSP cont'd

MCP add\_item (multiple)

```
include "globals.mzn";
int: n = 9;
array[1..n,1..n] of int: dist =
[| 0, 65, 60,184,195,319,299,478,631
 | 65, 0,125,119,130,254,234,413,566
 | 60,125, 0,184,157,281,261,440,593
 |184,119,184, 0,208,252,136,315,468
 |195,130,157,208, 0,136,280,459,629
 |319,254,281,252,136, 0,217,391,566
 |299,234,261,136,280,217, 0,188,343
 |478,413,440,315,459,391,188, 0,157
 |631,566,593,468,629,566,343,157, 0|];
array[1..n] of var 1..n: succ;
var int: total_dist = sum(i in 1..n)(dist[i,succ[i]]);
constraint circuit(succ);
solve minimize total_dist;
```

# Example: TSP cont'd

LLM

Optimal route found: Vienna → St. Pölten → Linz → Salzburg → Bregenz → Innsbruck → Klagenfurt → Graz → Eisenstadt → Vienna

Total distance: 1,564 km

Route verification:

- Each city visited exactly once: ✓ (circuit constraint)
- Distance calculation correct: ✓
- Starting/ending in Vienna (city 1): ✓

User

Assume the road between Vienna and Eisenstadt is blocked, what is now the shortest tour?

LLM

I'll modify the distance matrix to make the Vienna-Eisenstadt route impossible (infinite distance).

# Example: TSP cont'd

MCP edit\_model

```
array[1..n,1..n] of int: dist =  
[| 0, 65,9999,184,195,319,299,478,631  
 | 65, 0,125,119,130,254,234,413,566  
 |9999,125, 0,184,157,281,261,440,593  
 |184,119,184, 0,208,252,136,315,468  
 |195,130,157,208, 0,136,280,459,629  
 |319,254,281,252,136, 0,217,391,566  
 |299,234,261,136,280,217, 0,188,343  
 |478,413,440,315,459,391,188, 0,157  
 |631,566,593,468,629,566,343,157, 0|];
```

LLM

Optimal route: Vienna → Linz → St. Pölten → Salzburg → Bregenz → Innsbruck  
→ Klagenfurt → Graz → Eisenstadt → Vienna  
Total distance: 1,694 km (+130 km compared to unblocked route)

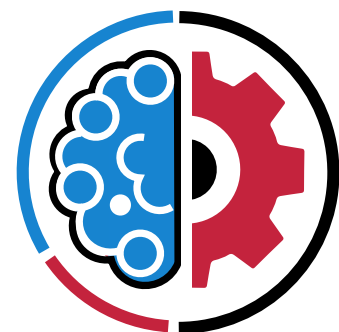
# Conclusion and Future Directions

## Summary:

- MCP-Solver demonstrates a novel integration of LLMs with constraint programming via the MCP standard.
- It enables structured, iterative refinement of constraint models while maintaining solver integrity.

## Future Work:

- Further improvements will target autonomous knowledge accumulation, enhanced resource management, and broader solver support.
- Expanding integration with the MCP ecosystem (as seen with tools like Cursor) may lead to comprehensive end-to-end problem-solving workflows.



Technical Report: <https://arxiv.org/abs/2501.00539>

Code: <https://github.com/szeider/mcp-solver>