Neurosymbolic AI: Solvers and LLMs

Stefan Szeider

Algorithms and Complexity Group, TU Wien

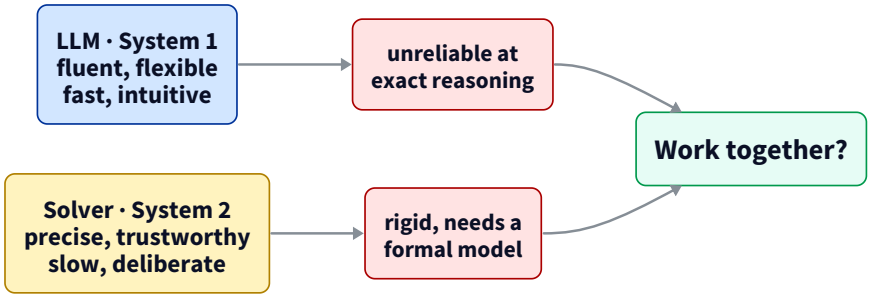
ESSAI 2026 Tutorial

ac.tuwien.ac.at/people/szeider/essai26

Part 1

Neural meets Symbolic

Two paradigms, opposite strengths



Guiding question: can fast, intuitive generation and slow, exact reasoning cover each other's weaknesses?

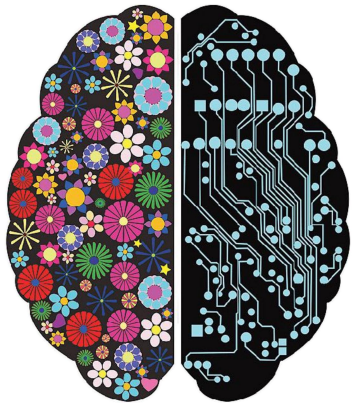
Neither paradigm alone is enough. This whole tutorial is about the bridge.

System 1 and System 2 (Kahneman)

Fast Thinking

System 1

LLMs



Slow Thinking

System 2

Solvers

Creativity and NLP interface

generate streamliners
generate encodings

Trust and Precision

provide exact reasoning/solving

Two complementary ways of thinking: intuition (LLMs) and exact reasoning (solvers).

Where LLMs fail (1): hallucination

- Air Canada's chatbot invented a refund policy; the airline was held liable
- A New York lawyer filed a brief citing **fictional** court cases from ChatGPT
- Even retrieval-augmented legal tools still mislead **1 in 6** times

Magesh et al. 2025, study of legal retrieval-augmented systems
Fluent does not mean correct: LLMs state falsehoods with full confidence.

Where LLMs fail (2): shaky reasoning

- LogicAsker: logical-rule errors from **29%** to **90%** across models
- Models fall back on remembered patterns instead of reasoning (Wan et al. 2024)
- “The Illusion of Thinking”: accuracy **collapses** past a complexity threshold (Shojaee et al. 2025)

On exact, compositional tasks the fluency breaks down.

The other side: solvers are astonishingly strong

- Modern SAT solvers routinely handle formulas with **millions** of variables
- A quiet revolution that reshaped verification, planning, and optimization

Before we lean on solvers: the exact-reasoning side of the bridge is genuinely powerful.

Where solvers fail (1): stating the problem

> “The user states the problem, the computer solves it.”
>
> — Freuder, 1997, on the promise of constraint programming

The wall is the **first half**, stating it *formally*:

- a non-expert knows what they want, but not how to write it as constraints
- requirements get **lost in translation** into a formal model
- if an expert must sit in the loop, the feedback cycle is slow

Everyone can describe a fair timetable in words; almost no one can write the constraints that pin down “fair”.

Getting the problem in is the first barrier, before the solver even runs.

Where solvers fail (2): making it fast

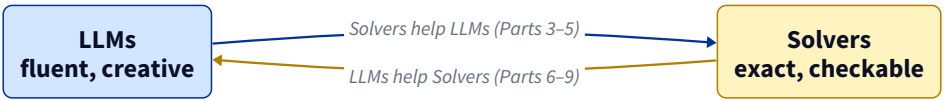
Even a **correct** model can be hopelessly slow: performance depends on how you write it, not just what it says.

- Miss a **symmetry** and the solver re-explores millions of equivalent configurations; one extra constraint can cut solving from **days to seconds**
- More generally, a naive encoding runs **orders of magnitude** slower than an expert's (Heule et al. 2023)

Choosing the fast formulation is expert work most users cannot do.

Automating exactly this is Part 8.
Two walls, not one: even once stated, a model needs an expert to run fast.

The bridge: two directions of synergy



One principle, both ways: **let the LLM propose, let the solver check.**

Keep this picture in mind. We walk the top arrow first, then the bottom.

Part 2

LLM Basics

Making of a frontier model

- **Pre-training:** predict the next token over a large corpus
- **Post-training:** fine-tuning and RLHF turn it into an assistant
- **Evaluation:** benchmarking and red-teaming before release

No background assumed: at a high level, this is all an LLM is.

Using an LLM is text completion

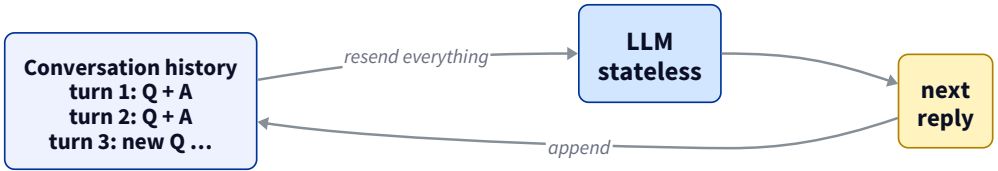
“What is the capital of France? And of Spain?”

. . . the model completes the text: Paris. Madrid.

The LLM is stateless: it keeps no memory between calls.

Everything the model knows about the conversation is in the text you send it.

Statefulness is simulated



Each turn resends the **full conversation history**; the model re-reads it and continues.

The context window is the memory. This matters once tools start filling it (Part 3).

A consequence: swap models mid-chat

- Start a conversation with one model, continue it with another
- The history is just text, so any model can pick it up

The context is the state, not the model.

This is also why a shared, cross-vendor tool interface (MCP) is even possible.

Part 3

Solvers help LLMs

From autocomplete to agents

1. **Single-shot** one input $\rightarrow$ one output
2. **Workflows** fixed chains of calls (RAG)
3. **Agentic** the model decides when to call a tool
4. **Context engineering** sub-agents, skills
5. **Persistent memory** state across sessions

*Level 3 is the jump: an LLM that reaches outside itself becomes an **agent**. Everything below Level 3 is fixed; at Level 3 the model itself decides when to act.*

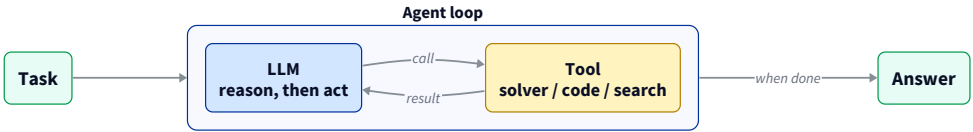
What a solver interface needs

- **Dynamic:** back-and-forth, immediate feedback, interactive refinement
- **Reusable:** any solver, any LLM, any agent framework
- **Validated:** an independent check that the model stays correct

These three requirements point straight at tool calling and a shared protocol.

Design goals first; the mechanism on the next slides follows from them.

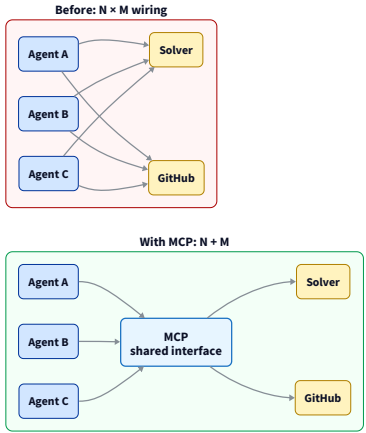
Tool calling: the model decides to act



All major LLMs are trained to call external functions. **The model decides if and when.** Interleaving reason and act is the **ReAct** loop (Yao et al. 2023).

This is the neurosymbolic bridge: the neural model left, a precise tool right.

The Model Context Protocol (MCP)



Open standard
(Anthropic, Nov 2024)

- $N \times M$ integrations become $N + M$
- OpenAI, Google, Microsoft, AWS, GitHub
- over **1000** servers

The “USB port” for tools: any solver drops into any agent.

Resources: Foundations

- **The Silent (R)evolution of SAT.**
  Fichte, Le Berre, Hecher, Szeider. *Comm. ACM* 66(6), 2023.
  `cacm.acm.org/research/the-silent-revolution-of-sat`
- **Model Context Protocol** (background).
  `modelcontextprotocol.io`

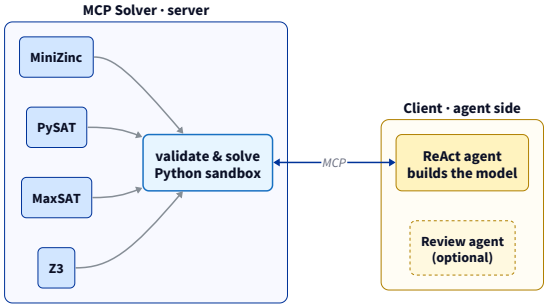
Article

Video (CACM)

Part 4

The MCP Solver

An MCP server for solving



A **server** that exposes SAT, CP, and SMT solving as validated tools; any MCP client (here a ReAct agent) drives it, one backend per session.

Four solvers, one uniform interface: the agent never touches solver internals.

A real session: touring Austria

User "Plan the shortest tour of Austria's nine province capitals, starting in Vienna." (a 9×9 distance table follows)

Agent writes a MiniZinc model, then calls the solver:

```
array[1..n] of var 1..n: tour;
constraint alldifferent(tour);
constraint tour[1] = 1; % start in Vienna
solve minimize total_distance;
```

`clear_model` → `add_item` → `solve_model`

solve_model returns:

```
{'satisfiable': True,
 'objective': 1564,
 'tour': [1,3,5,6,8,9,7,4,2],
 'optimal': True}
```

Agent "Shortest route: Vienna, Eisenstadt, Graz, Klagenfurt, Innsbruck, Bregenz, Salzburg, Linz, St. Pölten, back to Vienna — total **1564 km**."

The abstract loop, for real: the agent decides to call the solver, gets a proven optimum, and answers. The six tools it used: next.

Six item-based tools

The agent edits the model through a small tool set:

- `add_item`, `replace_item`, `delete_item` edit one item
- `get_model` view the current numbered model
- `solve_model` run the solver with a timeout
- `clear_model` reset the model

Fewer tools work better: a smaller tool set is less cognitive load for the model.

An earlier version had more tools; trimming them raised reliability.

Item-based editing keeps the model valid

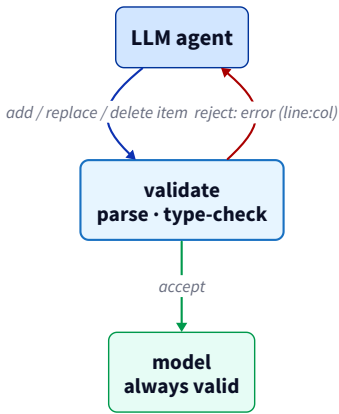
An **item** is one independent part of the encoding. Here, 10-queens in MiniZinc:

#	item
1	`int: n = 10;`
2	`array[1..n] of var 1..n: queens;`
3	`constraint alldifferent(queens);`
4	`constraint alldifferent([queens[i]+i | i in 1..n]);`
5	`constraint alldifferent([queens[i]-i | i in 1..n]);`
6	`solve satisfy;`

Any invalid item is rejected, so the model stays well-formed. Item-based beats line-based for MiniZinc.

The unit of editing is a constraint, not a text line: no half-broken models.

Incremental validation



Every edit is checked **before** it is applied:

- valid → item accepted, model returned with indices
- invalid → **precise error** (line:col) back to the agent

The agent *reads the error and corrects it*, usually with `replace_item`.

Errors are caught early, inside the loop: the encoding is never left broken.

What each backend validates

MiniZinc

- parse and type-check
- model-level consistency
- errors with exact line:col

PySAT / **Z3** (sandboxed)

- AST analysis for logic slips
- restrictive import policy
- checks solver and export calls

Each backend runs in an isolated subprocess with a hard timeout.
Validation is solver-specific, but the agent-facing contract is the same.

Resources: The MCP Solver

- **Bridging Language Models and Symbolic Solvers via MCP.**
  S. Szeider. *SAT 2025*, LIPIcs 341.
  doi.org/10.4230/LIPIcs.SAT.2025.30
- **MCP-Solver** (open source). arXiv:2501.00539.
  github.com/szeider/mcp-solver

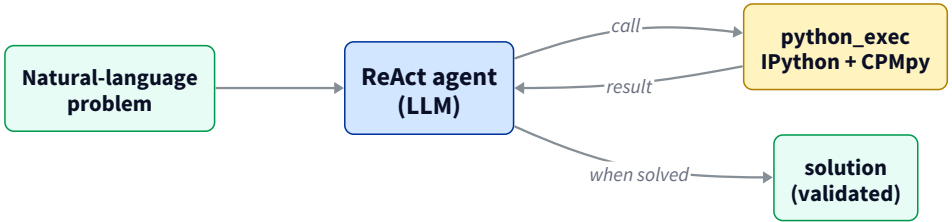
Paper

Code

Part 5

Fully Agentic Modelling

A fully agentic modeller



- A coding agent that **writes and tests** Python; no fixed workflow
- One tool: `python_exec` in a persistent IPython kernel (no bash, no file edits)
- Only the prompt is domain-specific: swap it to target CP, ASP, . . .

Minimal scaffolding: an LLM, a Python kernel, and a prompt, nothing else.

CP-Agent = coder + a short prompt

- The domain part is an **under-50-line** CPMpy prompt: use CPMpy, output JSON, test the solution
- Ablation: an 800-line detailed prompt does **not** beat the 50-line one
- Reusable: the same agent targets ASP by swapping only the prompt

Minimal guidance beats detailed procedural scaffolding.

The CP-Bench benchmark

- **101** natural-language CP problems: **71 decision** + **30 optimization**
- Sources: CSPLib, CPMpy, Kjellerstrand, KU Leuven; each with a reference model
- Our cleanup: 31 ambiguous specs clarified, 19 ground-truth models fixed

A clean, checkable testbed: every problem has a validated reference solution.

Results: every problem solved

Approach	Accuracy
Direct prompting (best fixed workflow)	$\sim$70%
CP-Agent (agentic workflow)	**100%**

- Claude Sonnet 4.5, all 101 problems, averaged over 3 runs
- Per problem: $\sim$52 s, 37K in / 3K out tokens, 1–28 `python_exec` calls
- Every solution validated against the reference model

Execution feedback closes the gap from 70% to perfect.

Why it works, and how far

It adapts to solver feedback (not a fixed script):

- Cryptarithmetic: 20-digit integers overflow $\rightarrow$ the agent redesigns with carry variables

The architecture is model-agnostic (10 problems, 3 runs each):

Model	Success
Claude Sonnet 4.5	100%
Claude Haiku 4.5	87%
DeepSeek v3.2 / Grok	80–83%

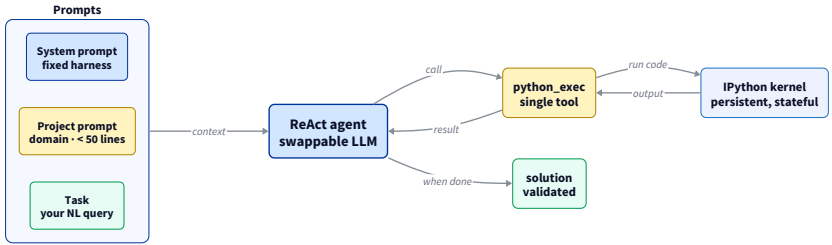
The agent debugs itself; stronger models just do it more reliably.

Part 5 · the tool

The Agentic Python Coder

a reusable, open-source harness

One harness, any domain



Open-source ReAct coding agent (Apache 2.0). Only the **project prompt** is domain-specific: swap it for CP, ASP, . . .

```
uv tool install agentic-python-coder
```

The harness is fixed; the domain lives entirely in the prompt you hand it.

Driving it from the command line

```
coder "Create a function that calculates factorial"

# swap the --project prompt to change domain
coder --with cpmpy  --project cpmpy.md  "Solve 8-queens"
coder --with clingo --project clingo.md "Model bird flight"

coder -i --project cpmpy.md --with cpmpy   # interactive
coder --model opus45 "..."                 # any of 9 models
```

Swap `--project` for a new domain (CP, ASP, regex, . . .), `--model` for a new LLM.

CP and ASP are the same command with a different prompt file.

Also an MCP server, and parallel-ready

`ipython_mcp` gives any MCP client (e.g. Claude Desktop) persistent Python execution:

```
"ipython": {
  "command": "uvx",
  "args": ["--from", "agentic-python-coder", "ipython_mcp"]
}
```

- Tools: `python_exec`, `python_reset`, `python_status`, `python_interrupt`
- **Multi-kernel**: each sub-agent gets an isolated kernel, so agents run in parallel

The same harness returns in Part 6 (streamliners) and Part 8 (improving CP models). One reusable harness underneath everything that follows.

Resources: Fully Agentic Modelling

- **CP-Agent: Agentic Constraint Programming.**
  S. Szeider. LLM4Code '26 — Workshop on Large Language Models for Code, ACM; to appear.
  arxiv.org/abs/2508.07468
- **Agentic Python Coder** (open-source harness, Apache 2.0).
  github.com/szeider/agentic-python-coder
  `pip install agentic-python-coder`
- **CP-Bench** benchmark.
  Michailidis, Tsouros, Guns 2025.

Paper

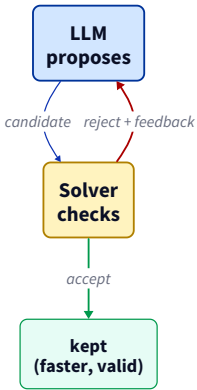
Code

Part 6

LLMs help Solvers

the check: does the model still solve, but faster?

Turning it around



What can an LLM give a solver? The **creativity** solvers lack.

The principle, in both directions:

let the LLM propose, let the solver check.

Bad guesses are simply discarded, so an unreliable generator becomes dependable.

This is the engine for the rest of the tutorial: propose, then verify.

What is a streamliner?

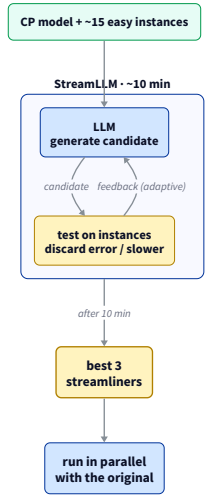
An **extra constraint that prunes the search space** (Gomes & Sellmann 2004). Example, n-queens:

```
constraint alldifferent(queens);
constraint alldifferent([queens[i]+i | i in 1..n]);
constraint alldifferent([queens[i]-i | i in 1..n]);
constraint queens[1] <= ceil(n/2);
```

The last line, `queens[1] <= ceil(n/2)`, is the streamliner. **Streamliners need not be sound:** in general they *may* cut valid solutions — unlike symmetry-breaking or implied constraints, which never do. That freedom is the point: it lets them prune aggressively. We keep completeness by running in parallel with the original.

This n-queens one is in fact sound (a symmetry-breaker); a streamliner in general need not be. Either way a cheap bet: the original runs too, so we never lose a solution.

StreamLLM: generate and test



- Input: the CP model + ∼**15 tiny** training instances (solvable in seconds)
- For ∼10 min: prompt the LLM, test each candidate, discard errors and slower-than-baseline ones
- Adaptive: feed results back; every 3rd round without feedback, for diversity
- Output the best 3, run in parallel with the original

Traditionally hand-crafted by an expert, or built over CPU-days.

Whole streamliners synthesised in seconds, not days.

A streamliner it actually found

On **Social Golfers**, the LLM proposed this constraint:

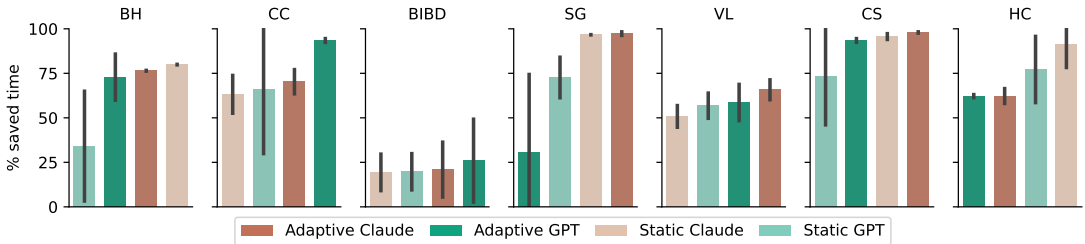
```
constraint forall(g in Golfer)
  (assign[g,2] = ((g-1) + n_per_group) mod n_groups + 1);
```

It fixes each golfer's group in week two, pruning the search — a genuine, non-obvious guess the solver then checks.

instances unsolved in **10 hours** $\longrightarrow$ solved in **3 minutes** (a $>$99% cut)

The LLM proposes a real constraint; the solver validates it. Propose, then check.

Results: solving time saved

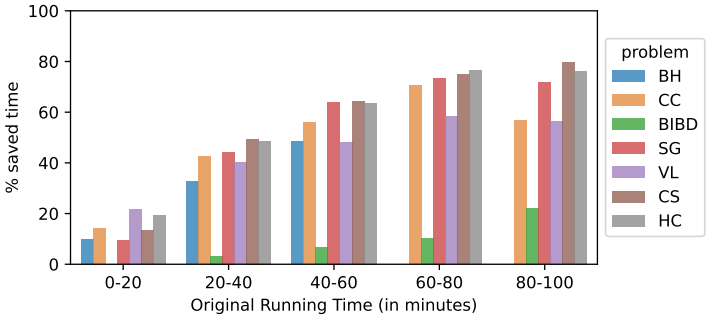


BH Black Hole · **CC** Carpet Cutting · **BIBD** Balanced Incomplete Block Design · **SG** Social Golfers · **VL** Vessel Loading · **CS** Car Sequencing · **HC** Hypergraph Coloring

Across seven problems, streamliners cut solving time by up to $>$**90%** (Social Golfers, Car Sequencing); little help where structure is thin (BIBD).

Big wins on structured problems, both with Claude and GPT.

Bigger wins on harder instances



The savings grow with the original running time: the harder the instance, the more a streamliner helps.

Exactly where you need it: streamliners pay off most on the slow instances.

Interlude: does the LLM understand, or just recall?

A worry worth pausing on. Maybe this only works because these are **textbook problems** the LLM has seen online, and it would fail on a genuinely new one. A controlled test: feed one problem in three versions.

- **Original:** the Social Golfers problem, named as usual
- **Disguised:** the same problem renamed (golfers $\rightarrow$ zoo animals, “Zoo Keepers”)
- **Obfuscated:** semantic names stripped to abstract symbols

If the LLM merely recalled streamliners tied to the familiar name, the rename would break it.

A side question, off the main thread: is it reasoning, or retrieving from training?

Interlude: semantics matter

Version	Surface language	Streamliner quality
Original	natural names	baseline
Disguised	renamed, still meaningful	**preserved / better**
Obfuscated	abstract symbols	**worse**

So it is not blind recall: a renamed, effectively unseen problem still works. But it **needs meaningful semantics** to reason; strip them and it degrades. A fully novel, cue-free problem remains the open risk.

End of interlude; back to optimization next.
Not decoration: meaning is what the LLM reasons over. Novelty without cues is untested.

Resources: LLMs help Solvers

- **Generating Streamlining Constraints with Large Language Models.**
  Voboril, Peruvemba Ramaswamy, Szeider. *JAIR* 84, 2025.
  doi.org/10.1613/jair.1.18965
- **Code & data** (Zenodo).
  zenodo.org/doi/10.5281/zenodo.13331048

Paper

Code

Part 7

Streamlining for Optimization

same engine, new check: does the *objective* improve?

From deciding to optimizing

So far a streamliner was good if the model still found *a* solution, faster. In an **optimization** problem there is always a solution; only its quality varies.

- Judge each candidate by the **objective it reaches** in a short run, not by yes/no
- Empirical fact: most of the improvement happens in the first few minutes
- So screen many streamliners cheaply, keep the ones that improve the objective

A streamliner is worth keeping if it reaches a better objective in minutes.

The verify step changes from “still solvable?” to “does the objective improve?”

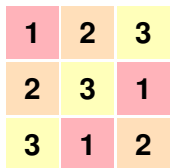
Case study: Balanced Latin Rectangles

A benchmark optimization problem from **experimental design** (Díaz et al. 2017).

- Lay out treatments in a field so that no treatment sits systematically near another, to avoid spatial-correlation bias (agriculture, drug trials, psychology)
- Formally: arrange symbols in a grid so every pair of symbols is equally **spread out**; the objective is to minimise the **imbalance**
- Optimal values are known for only a few sizes; a hard optimization benchmark

Real motivation: designing fair agricultural field trials. Also a tough solver benchmark.

What is a balanced Latin rectangle?



3×3, imbalance 0

A $k \times n$ **grid** of symbols $1..n$: no symbol repeats in a row or a column.

- **Distance** of a symbol pair = how far apart they sit, summed over rows
- **Balanced**: every pair has the same distance

Here every pair is at distance $4 \Rightarrow$ perfectly **balanced**.

Colour = symbol. Balance means no pair is systematically closer than any other.

Measuring imbalance

1	2	3	4
2	4	1	3
3	1	4	2

3×4, imbalance 4 (best possible)

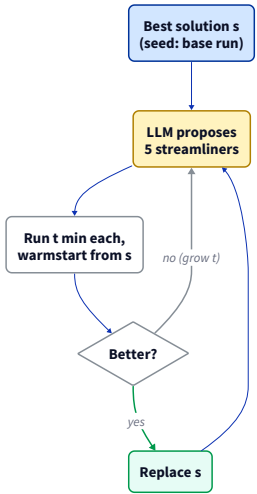
Ideal distance $Z = k(n+1)/3$; imbalance of a pair $= |\,\text{dist} - Z\,|$; imbalance of the grid $=$ their sum.

- Perfect balance (0) is **impossible** when $n \equiv 1 \pmod 3$
- Goal: given k, n, find the grid of **minimum imbalance**

Prior record via a SAT encoding (Peruvemba Ramaswamy & Szeider, CP 2023).

The objective to minimise. For most sizes the true optimum is still unknown.

Orchestrating streamliners: warmstart

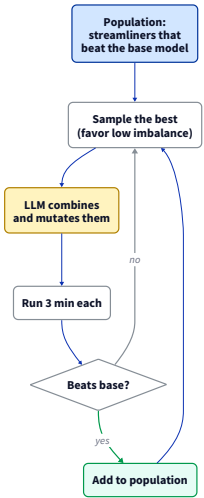


Keep improving one solution.

- Seed with a quick base run; keep the best solution s
- Each round the LLM proposes 5 streamliners, run briefly, warmstarted from s
- Any better result replaces s; enlarge the time budget when progress stalls

Exploit the best solution found so far, over four hours.

Orchestrating streamliners: evolution



Breed a pool of streamliners.

- Every streamliner that beats the base model joins a population
- Sample the best (favouring low imbalance); the LLM combines and mutates them
- Winners are added back; the best three run longer at the end

Evolve a diverse population of streamliners, over five hours.

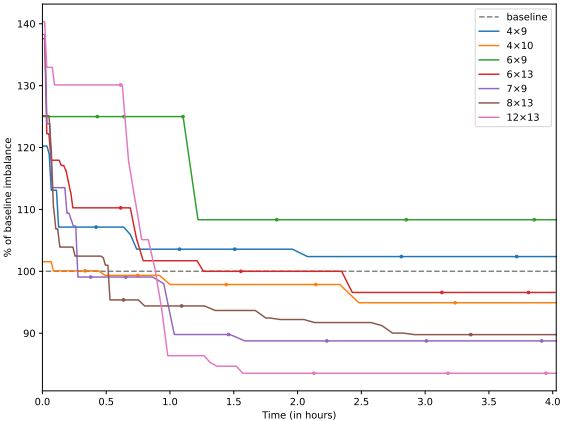
New records across the board

$n\backslash k$	3	4	5	6	7	8	9	10	11	12
9	56	56	52.6	48	51.3	53.3				
10	80	78.6	66.6	70	72	82.6	70	40		
11	114	112	116	108	118	114	120	110	0	
12	154	154.6	145.3	120	164	148	154	174.6	147.3	0
13	198	211.3	210.6	204	203.3	220.6	214	240.6	224	274

Purple $=$ a strictly better imbalance than any previously known. Improved **32 of 44** open instances; beat a 4-hour Gurobi run on **40 of 44**.

Each approach improves about 24 instances alone; together they push 32 records.

Records in minutes, not days



Combining both strategies (warmstart seeded from evolution's best) sets the records.

Some streamliners are so effective that **3 minutes** of the streamlined model beats 4 hours of the original.

Imbalance drops fastest early, then flattens, which is exactly what the short-run screening exploits.

The payoff: LLM-generated streamliners find record Latin rectangles in minutes.

Resources: Streamlining for Optimization

- **Balancing Latin Rectangles with LLM-generated Streamliners.**
  Voboril, Peruvemba Ramaswamy, Szeider. *CP 2025*, LIPIcs 340, art. 40.
  doi.org/10.4230/LIPIcs.CP.2025.40
- **Model, code & record rectangles** (Zenodo).
  doi.org/10.5281/zenodo.15616074

Paper

Code

Part 8

Improving CP Models

now propose a whole new model; check: *equivalent*, and faster?

Modeling makes or breaks a solver

The same problem has **many correct CP models**, and their solving times differ by orders of magnitude. The usual levers:

- symmetry-breaking and implied constraints
- high-level global constraints (specialised propagation)
- reformulation and alternative variable representations

Choosing them well needs a modeling expert. **Goal: let an agent improve the model automatically.**

Same solver, same problem: a better model can be orders of magnitude faster.

The task: rewrite the model, keep its meaning

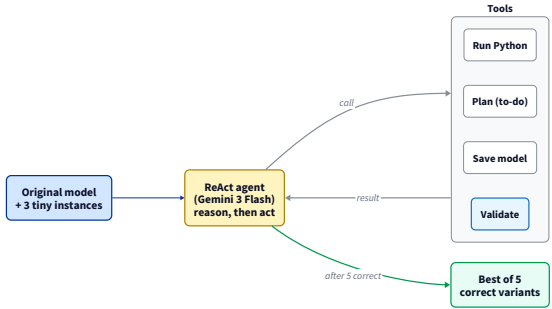
- **Input:** an existing CPMpy model + just **3 tiny training instances**
- **Output:** a semantically equivalent model that solves better (higher objective quality within the time limit)

Not solver selection, not parameter tuning: the agent **rewrites the model itself**.

CPMpy is a Python constraint-modeling library; LLMs handle it well, having seen much Python in training.

A different task from Parts 6–7: not extra constraints, but a full re-modeling.

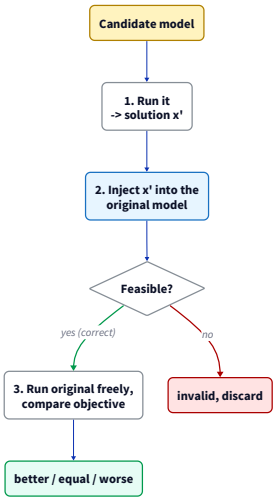
An agent that tries, tests, and iterates



The **agentic Python coder** (Gemini 3 Flash) in a ReAct loop: analyse the model, generate a variant, validate it, diagnose failures, repeat until **5 correct variants** (at most 200 steps), then return the best.

No hand-coded transformation rules: the agent decides which techniques to try.

Checking correctness without a proof



Every candidate is validated in three steps:

- Run it to get a solution x'
- **Inject x' back into the original model:** if the original accepts it, the candidate invented nothing
- Run the original freely to grade the objective

An empirical safeguard, not a formal equivalence proof, but nothing invalid slipped through.

Watching the agent recover

On **Quadratic Assignment**, the objective is an $O(n^4)$ sum over a boolean assignment matrix. The agent's plan: cut it to $O(n^2)$ with a permutation variable.

```
cost = sum(distance[p[f]][p[g]] ...) # IndexError, every try
```

It fails repeatedly, then diagnoses the cause mid-run:

> *CPMpy overloads indexing for its own variables but not for numpy arrays. . . Let's use the boolean variables, but simplify the sum.*

```
if flow[f,g] != 0 and distance[i,j] != 0: # skip zero pairs
    terms.append(flow[f,g]*distance[i,j] * (x[f,i] & x[g,j]))
```

The pivot beats the original on all three training instances (one solved optimally in 3.7 s).

The ReAct loop's value: it diagnoses its own failure and changes strategy.

Better models on 20 of 27 test instances

	Train 1	Train 2	Train 3	Test 1	Test 2	Test 3
Balanced Latin Rectangle	better	better	better	NS	NS	better
Golomb Ruler	better	better	better	better	worse	better
Job Shop Scheduling	better	better	better	better	better	ERR
Mentor Matching	better	better	better	better	better	better
Progressive Party	better	better	better	better	better	better
Quadratic Assignment	better	better	better	better	better	better
Set Cover	better	equal	better	better	better	equal
Traveling Salesman	better	worse	better	better	worse	worse
Warehouse Location	better	better	better	better	better	better

74% test wins ($p \approx 0.019$), ∼15 min per problem. Removing the loop hurts: the full agent wins **22/27** vs a single LLM call's **9/27** (12 failed to run).

The agent beats one-shot prompting; the iterate-and-validate loop is what matters.

Resources: Improving CP Models

- **Improving Constraint Models with LLM Agents.**
  Voboril, Szeider. To appear at *LLM-Solve 2026* (LLMs meet Constraint Solving), FLoC & CP 2026 workshop, Lisbon.
- Built on the **agentic Python coder** harness (Part 5) with **CPMpy** and the **Chuffed** solver.

Workshop paper; no public preprint yet.

Part 9

Proving Lower Bounds

Can the loop do mathematics?

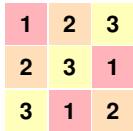
So far the loop improved **engineering artifacts**: streamliners, CP models. Can the same propose-and-check loop produce **genuine mathematics**?

A test problem: the **imbalance of Latin squares** (Part 7, but now $n \times n$).

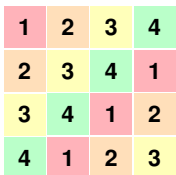
- Perfect balance ($I = 0$) is **impossible** when $n \equiv 1 \pmod 3$
- **Open question: what is the minimum possible imbalance?**

The turn from applying solvers to discovering a new theorem. (Xia, Gomes, Selman, Szeider.)

Recall: imbalance, now for $n \times n$ squares



3×3: $I = 0$



4×4: $I = 16/3$ (best)

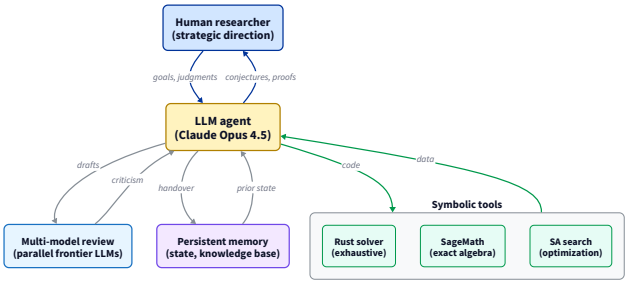
The **imbalance** measure is exactly the one from Part 7; only the grid is now square.

- $n = 3$: balanced, $I = 0$
- $n = 4$: here $n \equiv 1 \pmod 3$, so $I = 0$ is **out of reach**; best is $I = 16/3$

How far above zero must the minimum sit?

One-line recall, not a re-derivation: same measure as Part 7, square grids now.

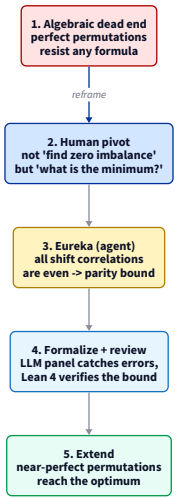
A neurosymbolic research team



An **LLM agent** (Claude Opus 4.5) orchestrates symbolic tools (a Rust enumerator, SageMath, simulated annealing), a multi-model review panel, and a two-tier persistent memory. The human sets direction.

The agent writes and runs the code; the tools give ground truth; the human steers.

The discovery, phase by phase



Thirty agent sessions over six days, in five phases:

- An algebraic approach **dead-ends**
- The human reframes the question
- The agent's **eureka** from data
- An LLM panel + Lean 4 verify it
- Simulated annealing extends the reach

21 sessions of dead ends, then a single **six-minute** session proved the bound.

The turning point is human; the eureka is the agent's. Neither alone suffices.

The eureka: parity from data

For $n = 13$: a naive count gives $I \geq 26$, yet simulated annealing **stuck at** 69.3.
Probing why, the agent hit a wall — then saw it, in its own notes:

```
f=61 NOT found in
100000 random permutations!
```

```
# f(d) is ALWAYS even for n=13!
# This is a parity constraint.
# ...
# min deviation per pair is 2, not 1.
# This changes the lower bound!
```

Confirmed on data: for $n = 4$, the minimum over all 576 Latin squares is exactly $16/3 = 4 \cdot 4 \cdot 3/9$.

Every distance is even, but the ideal value is non-integer, so each pair misses by ≥ 2, not 1: the bound **doubles**.

The 69.3 wall was the true optimum; a four-line parity argument, spotted by the agent, explained it.

A theorem, formally verified

For $n \equiv 1 \pmod 3$, every $n \times n$ Latin square has

$$I(L) \;\geq\; \frac{\boldsymbol{4n(n-1)}}{\boldsymbol{9}}.$$

The bound is **tight**: matched by a new class of near-perfect permutations (a circulant construction). Verified in **Lean 4** / **Mathlib** ($\sim$340 lines) via the LeanBack MCP server:

```
theorem LatinSquare.imbalance_lower_bound
    (L : LatinSquare n) (hn : n % 3 = 1) :
    3 * L.imbalance3 >= 4 * n * (n - 1)
```

A machine-checked proof: no trust in the LLM required for the final result.

How far it reaches, and one lesson

n	optimal $I^* = 4n(n-1)/9$
4	5.3
13	69.3
25	266.6
40	693.3

Near-perfect permutations **reach** I^*, verified computationally.

One lasting lesson about *where* each kind of reasoning is trustworthy:

- **multi-model review is reliable for *criticism*, not *construction***
- the agent even caught an error in its own draft proof
- every **negative** result came from the symbolic tools, not the LLM

"Theorem 1 claims all Latin squares, but the proof is circulant-only — all 4 models flag this."

The reach goes far; the durable takeaway is which component to trust for what.

Resources: Proving Lower Bounds

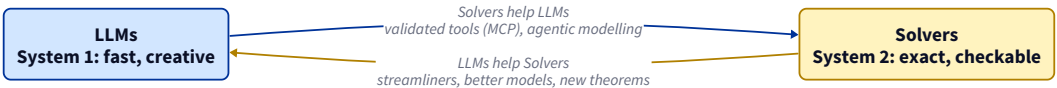
- **Agentic Neurosymbolic Collaboration for Mathematical Discovery: A Case Study in Combinatorial Design.**
  Xia, Gomes, Selman, Szeider. To appear at *NeSy 2026*.
  arxiv.org/abs/2603.08322
- **LeanBack** MCP server (Lean 4 verification).
  pypi.org/project/leanback

Paper (arXiv)

Part 10

Summary & Outlook

Two directions, one loop



We walked the top arrow (Parts 3–5) and the bottom arrow (Parts 6–9). The same **System 1** and **System 2** from the start, now working together.

Back to the opening picture: fast intuition and slow exact reasoning, combined.

The one principle

Let the LLM propose, let the solver check.

It works in both directions:

- a solver makes an LLM's output trustworthy (validated tools, tested models)
- an LLM gives a solver creativity (streamliners, reformulations, conjectures)

A creative-but-unreliable generator plus a rigorous checker becomes a **dependable system**.

One idea to take home. Everything in this tutorial is an instance of it.

Outlook: from checking to proving

Today the solver checks *each instance*. The next step is to **prove properties once, for all** n.

Our project **LeanCSP** certifies constraint reformulation and solving in **Lean 4**:

- prove a streamliner sound, so it helps on **unsatisfiable** instances too, not just finding solutions
- prove two models equivalent in general (arbitrary n), not just tested on three instances (Part 8)
- establish satisfiability or unsatisfiability **without trusting the solver**

Open source: github.com/leansolving/leancsp
The frontier: neurosymbolic systems that are not just effective, but certified.

Questions?

Neurosymbolic AI: Solvers and LLMs

Tutorial page

MCP Solver

CP-Agent