

From Policy Learning to Autonomous Algorithm Design: AI for Network Optimization
Universidad Carlos III de Madrid

Automatic heuristic generation I: why heuristics, LLMs for code, FunSearch and EoH

2026-09-22

Jose A. Ayala Romero · NEC Laboratories Europe

Agenda

1. Heuristics in industrial systems
2. LLMs for code generation
3. FunSearch
4. Evolution of Heuristics (EoH)
5. Summary

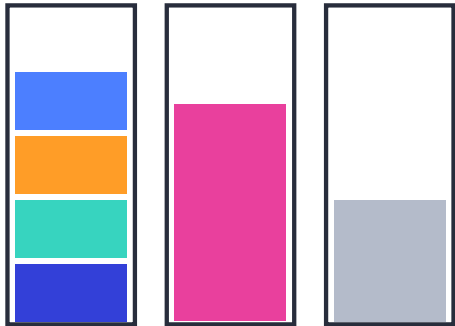
Heuristics in industrial systems

Why production schedulers are still rules, and what that costs

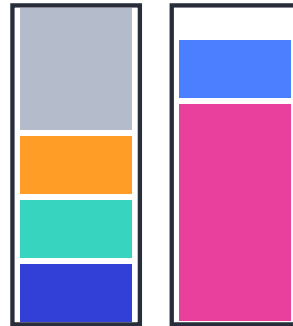
A heuristic is a decision rule you can read

Input, a score for each option, pick the best: three lines that run in microseconds

first fit (first bin with room): 3 bins



best fit (bin with least room left): 2 bins

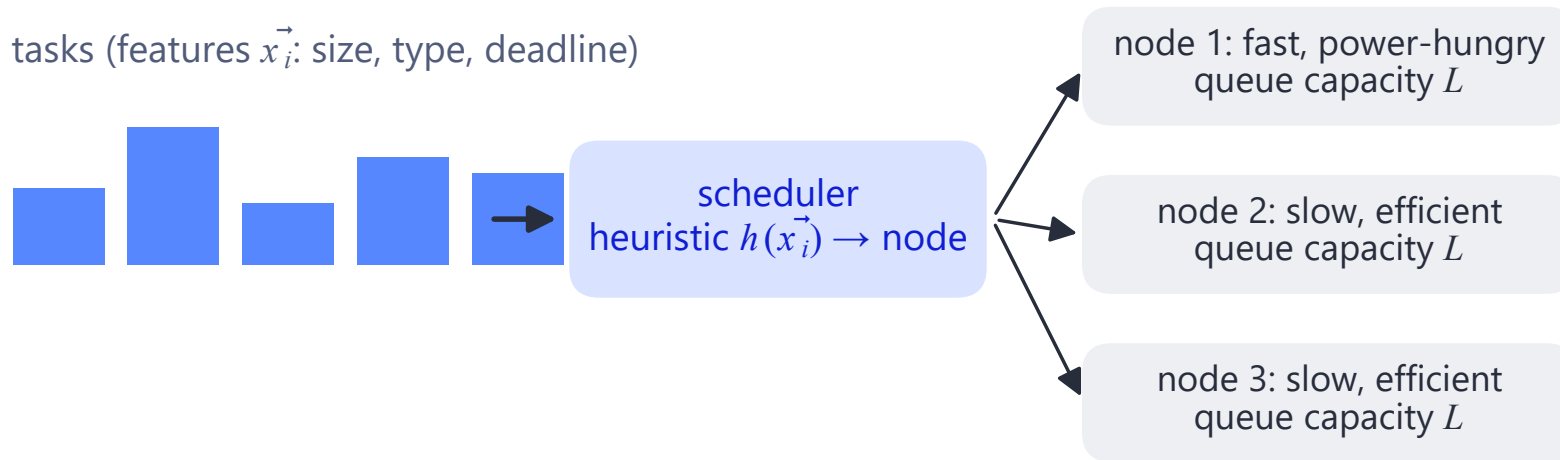


items arrive one at a time: 0.2, 0.2, 0.2, 0.7, 0.2, 0.4 (bin capacity 1)

- Online bin packing: place each arriving item into a bin without knowing the future
- **First fit** takes the first bin with room; **best fit** the bin that leaves the least room
- A scheduler's FIFO rule, a router's shortest queue: same shape, different score

Running example: dynamic task scheduling

incoming tasks (features $\vec{x}_i$: size, type, deadline)



cost per task: latency or energy if processed, a large penalty Γ if dropped (queue full, deadline missed)

- Streams of tasks onto a limited set of processing nodes: cloud orchestrators, base-station processing, packet processing in routers
- Inefficient assignments cost latency, energy and dropped tasks
- The nodes are heterogeneous, and their behaviour is usually not documented

The classical formulation rarely matches production

- The literature calls it a dynamic job-shop scheduling problem and solves it with rules, learning, or graph-based policies
- Standard formulations assume **infinite queues**, **time as the only objective**, and **deterministic, known processing times**
- Real systems have finite buffers, energy and reliability objectives, and processing functions nobody wrote down
- The formulation is not wrong; it is a different problem

The learned-policy route and its four deployment problems

Deep reinforcement learning schedulers reach strong performance in simulation, yet rarely reach production, because they:

- **lack generalisation**: an agent trained for one topology is retrained from scratch for the next
- **are black boxes**: no intuition for why a task went to that node
- **are brittle**: under sim-to-real distribution shift, performance can collapse
- **are slow**: neural inference does not fit sub-millisecond decision budgets (5G vRAN, hardware switches)

Why rule-based schedulers survive

Interpretable, auditable, predictable and fast, but blind to non-linear workloads

- Industrial schedulers (for instance the FIFO scheduler of Hadoop YARN) stay rule-based for interpretability and latency
- The rules cannot capture modern non-linear workloads: heterogeneous hardware, energy profiles, bursty arrivals
- Better rules exist, but designing them by hand takes experts weeks per deployment
- The gap: **high-performance, auditable heuristics** whose manual design cost is prohibitive

Learned policy versus heuristic

learned policy (deep RL)

strong in simulation

retrained for every new topology
black box: no intuition for its decisions
brittle under sim-to-real shift
inference in milliseconds on accelerators

rule-based heuristic

interpretable and auditable

portable across deployments
microsecond decisions on any CPU
predictable, easy to debug
but: hand-designed, misses non-linear workloads

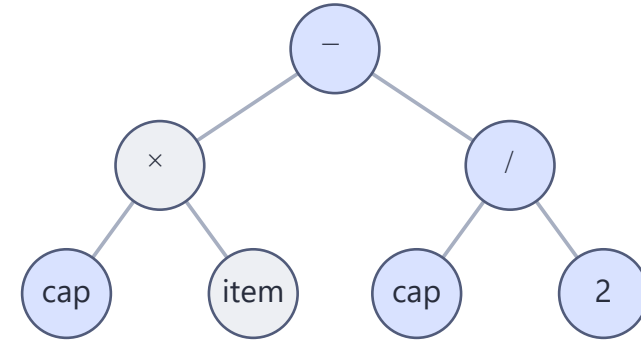
The goal of this day: auditable heuristics at low design cost

Automatic heuristic design is the middle way: keep the deployable artefact (a short program), but let a machine search for it.

- Yesterday: learn a **policy** online, on the system
- Today: generate a **program** offline, against an evaluator, and deploy the program
- The two meet at the end of the day

Classical automatic heuristic design

- **Hyper-heuristics:** search over a space of heuristics rather than solutions; select among given ones or combine given components ([Burke et al. 2013](#))
- **Genetic programming:** evolve expression trees over a fixed set of primitives
- Decades of results, one limitation: the search space is **predefined by a human**, and choosing the primitives is itself the hard part



score = cap × item − cap / 2: operators and terminals fixed in advance by a human

What an LLM could add

It brings

- an open-ended space: any program, not a tree over primitives
- world knowledge: it has read every textbook heuristic
- code that looks like human code, with comments

It lacks

- any notion of how well its program performs
- reliability: plausible but wrong is its failure mode
- novelty on demand: it retrieves more than it invents

We need the LLM's creativity and someone else's judgement

Exercise 1: would you deploy it

For each scenario, vote **learned policy** or **heuristic**, and name the criterion that decides.

1. A CPU scheduler inside a 5G virtual base station with a 100 μ s decision budget that will be deployed in production
2. A job placer for cloud data centres, one model per data centre, retrained on each new hardware generation
3. A research prototype of a very complex problem evaluated only in simulation

- (1) heuristic: latency and auditability are non-negotiable
- (2) either; the retraining cost per deployment and the black-box audit are the arguments for a generated heuristic
- (3) a learned policy is fine; nothing of the four deployment problems applies in simulation

Agenda

1. Heuristics in industrial systems
2. **LLMs for code generation**
3. FunSearch
4. Evolution of Heuristics (EoH)
5. Summary

LLMs for code generation

What they do well, and why one query does not give a useful heuristic

How an LLM writes code



sample a token, append it, repeat: temperature controls how often the less likely tokens are taken

- Trained on text and code to predict the **next token**; the prompt conditions the distribution
- Sampling one token at a time produces a program; the temperature controls diversity
- Nothing in the process executes the code or measures anything

What they are good at

- Programming competitions and interview problems
- Debugging and explaining existing code
- Translating between languages and libraries
- Filling in a function from a signature and a docstring

All four share a property: the answer is **already in the training distribution**, or can be verified by the person asking.

Ask for a bin-packing heuristic and you get one

```
1 def heuristic(item, bins):
2     """Score each bin; higher is better."""
3     remaining = bins - item
4     remaining[remaining < 0] = np.inf    # bins that cannot fit the item
5     # prefer the tightest fit
6     return -remaining
```

- It runs, it is readable, and it is **best fit**: the textbook answer
- Ask again and you get first fit, or best fit with a different sign convention
- Ask for "a better one" and you get best fit with a comment saying it is better

Why one shot is not enough

- **Confabulation**: plausible statements that are wrong, and code that looks right but is not
- **Retrieval, not invention**: the model reproduces known ideas; surpassing them needs something it has not seen
- **No knowledge of the target system**: it does not know that node 1 is fast but power-hungry
- **No verification**: nothing checks correctness
- **No feedback**: nothing tells it how the heuristic performed, so nothing improves

Sampling more does not fix it

Ten thousand independent samples lose to two thousand queries inside a loop

Method	LLM queries	Gap to the lower bound (bin packing)
Random sampling of the same LLM	10 000	2.44%
Evolution of Heuristics (EoH)	2 000	0.66%

- Same model, same problem; the difference is what happens between queries ([Liu et al. 2024](#))
- Independent samples keep re-drawing from the same distribution; a loop moves the distribution

The missing ingredients

An evaluator

Runs the program on instances and returns a score. Ground truth against confabulation.

A memory

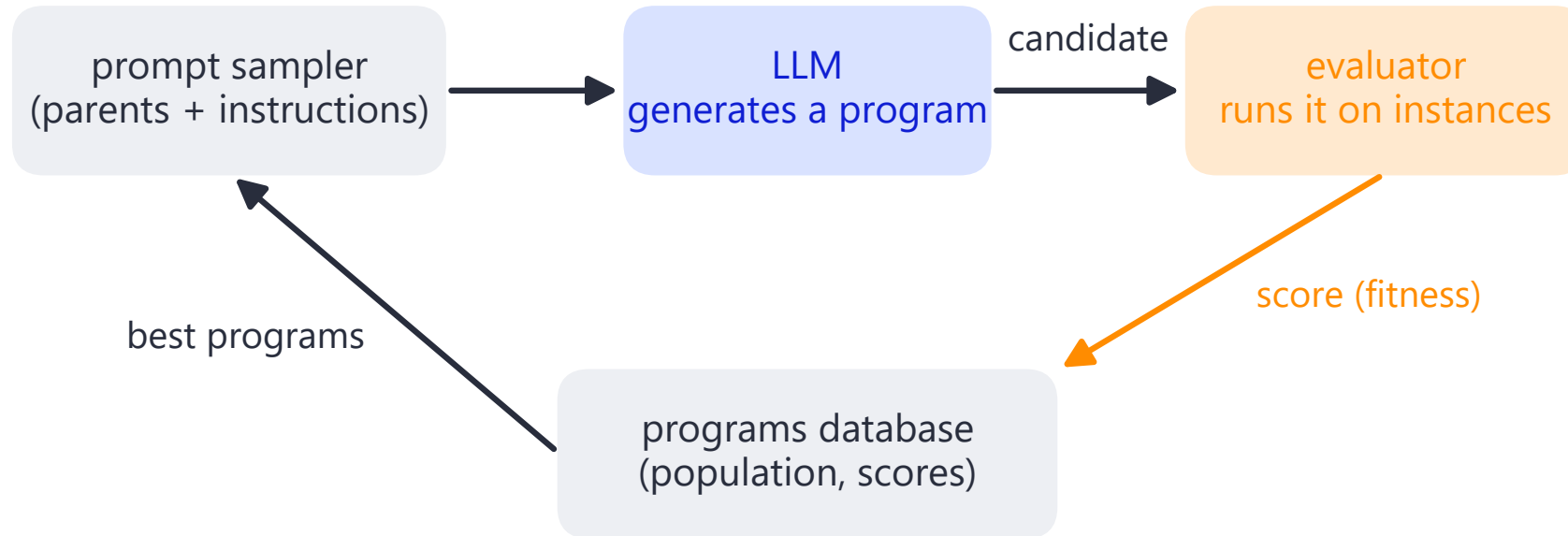
A population or database of programs with their scores. Keeps what worked.

An operator

Builds the next prompt from the best programs so that the LLM improves rather than restarts.

Together they turn sampling into **search**: an evolutionary algorithm whose mutation operator is a language model.

The LLM-based AHD loop



the LLM proposes, the evaluator judges, the database remembers

Where the loop works

- An **efficient evaluator**: scoring a candidate must be cheap enough to do thousands of times
- A **rich score**: a number that says how much better, not a pass/fail bit
- An **isolated part to evolve** inside a skeleton that encodes what is already known
- Counter-example: generating proofs of theorems, where the score is binary ([Romera-Paredes et al. 2024](#))

Evaluation is the hard part in systems

The evaluator defines what “good” means, and the search will exploit every gap in it

- Instances must represent production traffic; a heuristic tuned on toy instances will not transfer
- Noisy systems need repetitions, which multiply the cost
- Time and memory limits are part of the specification: a heuristic that hangs is discarded
- Score what you care about: energy and violations, not a proxy that is easier to compute

Exercise 2: design the evaluator

For the task-scheduling example: a heuristic maps a task's features to a node.

1. What score does the evaluator return?
 2. Which instances does it run, and how many?
 3. What happens with a heuristic that crashes, or takes a second per decision?
- A cost per task summed over a trace: latency or energy if processed, a large penalty if dropped; lower is better
 - Several traces sampled from the target workload, with the answer averaged; held-out traces to check generalisation
 - Discard it: syntax errors, runtime errors and time-outs never enter the database

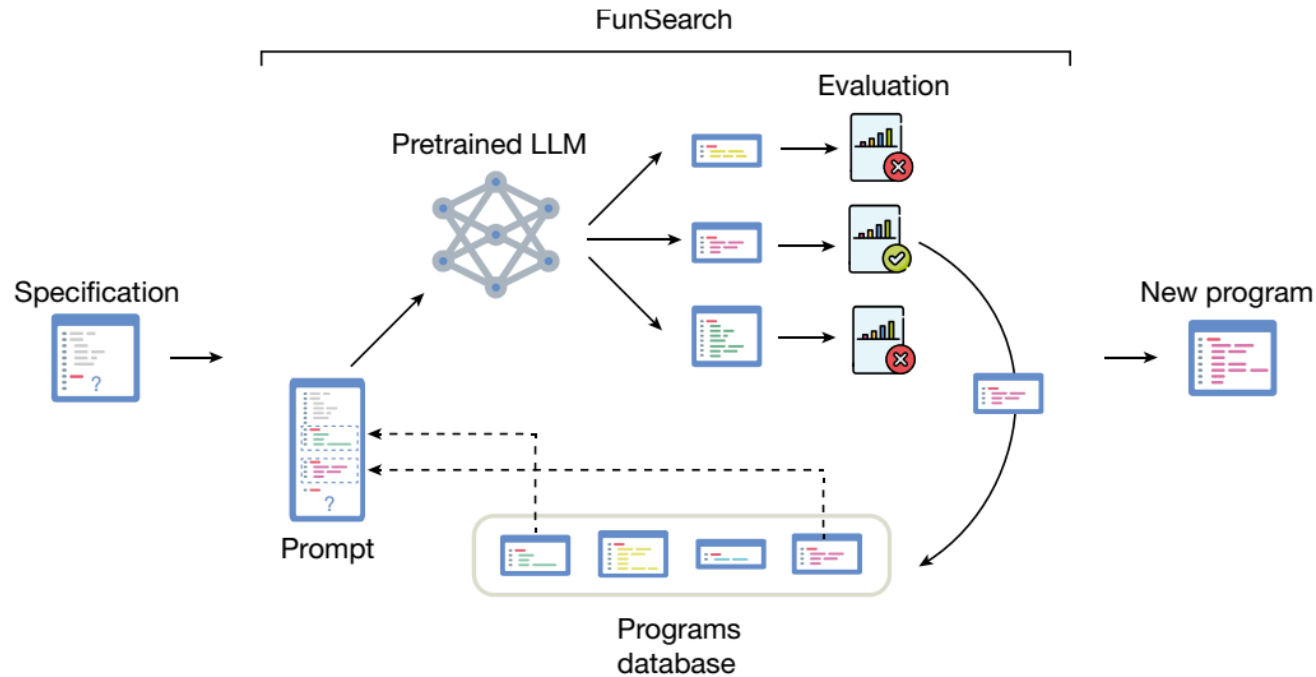
Agenda

1. Heuristics in industrial systems
2. LLMs for code generation
3. **FunSearch**
4. Evolution of Heuristics (EoH)
5. Summary

FunSearch

Searching in function space with a frozen LLM and a systematic evaluator

FunSearch: a frozen LLM plus a systematic evaluator



Source: [Romera-Paredes et al. 2024](#), Fig. 1

- Searches for **programs that describe how to solve** a problem, not for the solution itself
- Published with new results on an open problem in combinatorics and on online bin packing ([Romera-Paredes et al. 2024](#))

The specification: evaluate, solve and the function to evolve

```
1 def main(problem):                                # executed by FunSearch
2     bins = problem.bins
3     for item in problem.items:                    # online: one item at a time
4         valid = get_valid_bin_indices(item, bins)
5         best = solve(item, bins[valid])
6         bins[valid[best]] -= item
7     return evaluate(bins, problem)
8
9 def evaluate(bins, problem):                      # the score: fewer bins is better
10     if is_valid_packing(bins, problem):
11         return -count_used_bins(bins, problem)
12     return None                                  # invalid → discarded
13
14 def solve(item, bins):                           # the skeleton: pick the best-scored bin
15     return np.argmax(heuristic(item, bins))
16
17 def heuristic(item, bins):                       # the ONLY function FunSearch evolves
18     return -(bins - item)                       # initial version: best fit
```

Why a skeleton

- The LLM spends its samples on the **critical logic**, not on re-creating boilerplate it might break
- A fixed structure encodes what is already known about the problem: greedy construction, online processing, validity checks
- A skeleton restricts the space of programs, and the paper finds that this improves results
- Optional extras: docstrings, helper functions and imports the LLM may use

The LLM: fast, small, frozen

- No fine-tuning on the problem: a pretrained code model used as it is
- A **fast** model is preferred over a slower, higher-quality one: the loop needs volume
- Results are not very sensitive to the choice of model, as long as it was trained on enough code
- Total budget in the paper: on the order of 10^6 samples

Best-shot prompting

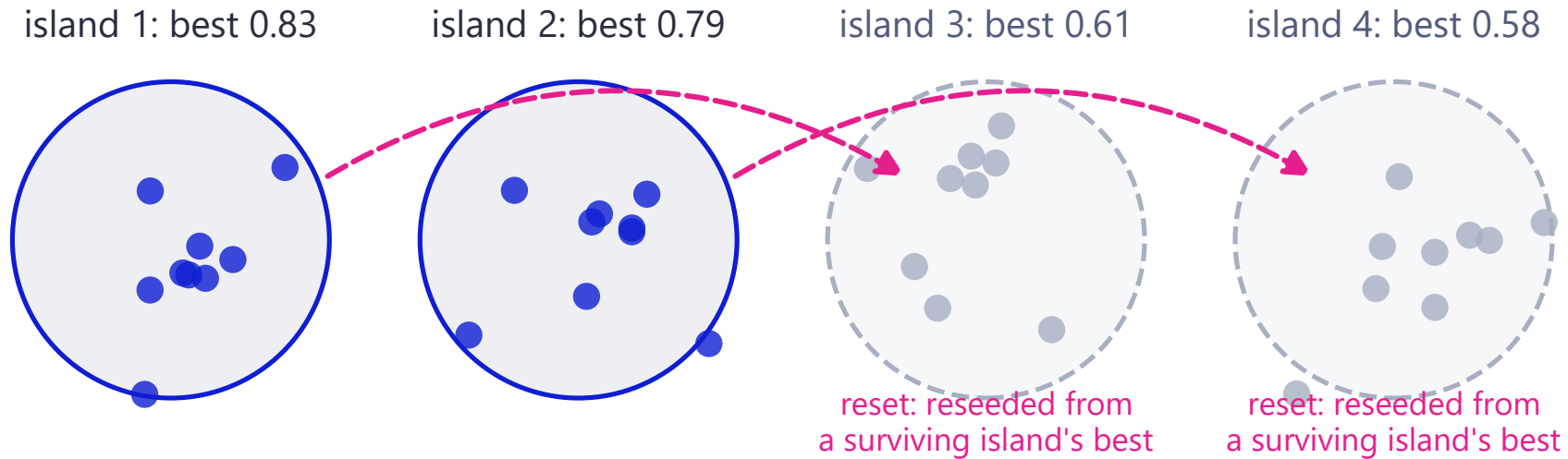
```
"""Finds good assignment for online 1d bin packing."""
def heuristic_v0(item, bins):      # score 0.71 (lower-scoring parent)
    return -(bins - item)

def heuristic_v1(item, bins):      # score 0.83 (higher-scoring parent)
    score = 1000 * np.ones_like(bins)
    score -= bins * (bins - item)
    return score

def heuristic_v2(item, bins):      # ← the LLM completes this one
    """Improved version of heuristic_v1."""
```

- Sample $k = 2$ programs from one island, sort by score, name them v0 and v1
- Append the header of v2 and let the LLM complete it
- Two parents let the model spot what the better one does differently; more than two brought little

The programs database keeps diversity with islands



islands evolve independently; periodically the worst half is discarded, keeping diversity while spreading good ideas

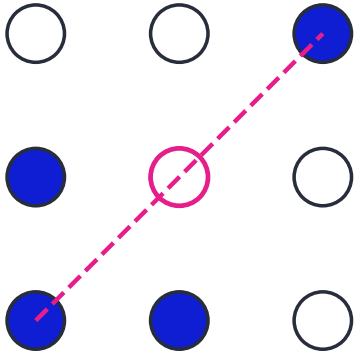
- Several populations evolve independently; prompts are built within an island
- Sampling favours **higher-scoring and shorter** programs
- Periodically the worst half of the islands is discarded and reseeded from the survivors' best programs

A distributed, asynchronous pipeline

- Three worker types: the programs database, **samplers** (LLM inference on accelerators) and **evaluators** (plain CPUs)
- Evaluation can take minutes per program, so many evaluators run in parallel: typically 15 samplers and 150 evaluators
- Sampling and evaluation overlap; nothing waits for anything else
- Cheap CPU evaluators keep the total cost and energy of an experiment low

Result 1: a new cap-set construction

$\mathbb{Z}_3^2$: 9 points, cap set of size 4



a cap set: no three points on a line
(lines wrap around modulo 3)

adding the centre would complete
the dashed line, so it stays out

exact maximum known only up to $n = 6$;
the space has about 3^{1600} candidates at $n = 8$

- Largest set of vectors in $\mathbb{Z}_3^n$ with no three on a line: an open problem in extremal combinatorics
- The skeleton adds vectors greedily by a **priority** function; FunSearch evolves the priority
- Dimension 8: a cap set of size **512**, larger than the best known (496)
- Also a new lower bound on the asymptotic capacity, the largest improvement in 20 years

Reading the program gave a new mathematical insight

The discovered **priority function** treated **groups of coordinates symmetrically**

- Inspecting the code showed a symmetry nobody had looked for
- Restricting the search to symmetric sets gave a much smaller space, so much larger instances became tractable, and the bound improved again
- A loop with humans inside: **search** → **inspect** → **restrict** → **search**
- Only possible because the output is a program, not a list of 200 000 vectors

Result 2: online bin packing beyond best fit

Heuristic	OR1	OR4	Weibull 5k	Weibull 100k
First fit	6.42%	5.23%	4.23%	4.00%
Best fit	5.81%	4.94%	3.98%	3.79%
FunSearch	5.30%	2.47%	0.68%	0.03%

- Excess bins over the offline lower bound; the heuristic was evolved on instances of OR1's size only
- It generalises to larger instances and, on Weibull-distributed sizes typical of scheduling workloads, nearly closes the gap

The discovered heuristic and its strategy

```
1 def heuristic(item, bins):
2     score = 1000 * np.ones(bins.shape)
3     score -= bins * (bins - item)          # penalise bins with large capacity
4     index = np.argmin(bins)                # the best-fit bin
5     score[index] *= item                  # scale it by item size
6     score[index] -= (bins[index] - item)**4 # penalise it if the fit is not tight
7     return score
```

- Best fit only when the fit is **tight**; otherwise leave the bin with more room
- Avoids small gaps that will never be filled
- Eight lines, deployable anywhere, and an explanation a scheduler engineer can check

Why search programs instead of solutions

- **Interpretable:** a domain expert can read, check and improve the result
- **Concise:** structured problems have short descriptions; short programs scale to large instances
- **Deployable:** code in a standard language needs no special hardware and can be verified against a specification
- The search implicitly prefers short programs, and the LLM prefers human-looking ones

Limits of FunSearch

Where it stops

- one function of 10 to 20 lines, in Python
- evaluation must be fast (minutes on one CPU)
- one scalar metric
- millions of samples from a small model
- minimal context: only previous solutions

What comes next

- richer representations and operators (EoH)
- feedback beyond a number (ReEvo, TRACE)
- a model that learns from the search (CALM)
- whole programs, many metrics, frontier models (AlphaEvolve)

The loop is the invention; everything after is about spending fewer samples and evolving more

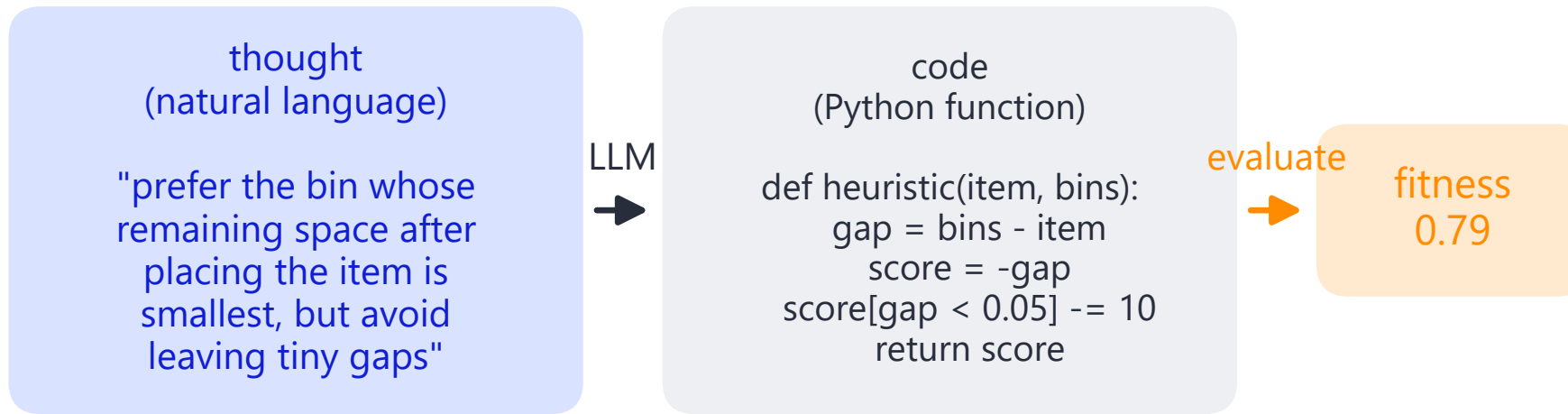
Agenda

1. Heuristics in industrial systems
2. LLMs for code generation
3. FunSearch
4. **Evolution of Heuristics (EoH)**
5. Summary

Evolution of Heuristics (EoH)

Evolving the idea together with the code

Evolve the idea, not only the code



one individual of the population = idea + implementation + score

- Each heuristic is a **thought** (a few sentences), a code block in a fixed format, and a fitness
- The LLM first writes the thought, then the code, mimicking how an expert reasons
- FunSearch evolves code only; EoH evolves both ([Liu et al. 2024](#))

The evolutionary framework

- Population of N heuristics, initialised by prompting the LLM with the task description
- Each generation: five prompt strategies, each applied N times, produce up to $5N$ candidates
- Every candidate is evaluated on a set of instances and added if it runs
- Keep the best N ; parents are selected with probability decreasing in their rank

Five prompt strategies

Strategy	Parents	Instruction to the LLM
E1 exploration	several	design a heuristic as different as possible from the parents
E2 exploration	several	identify the common idea, keep it, add new parts
M1 modification	one	modify it for better performance
M2 modification	one	change its parameters only
M3 modification	one	remove redundant components, simplify

- E strategies act like crossover; M strategies like mutation
- Every prompt asks for the thought first, then the code

An E2 prompt for bin packing

Task: design a heuristic that scores bins for an arriving item (function signature given).

Here are two existing heuristics with their thoughts and code: [heuristic 1] [heuristic 2]

Identify the common backbone idea in these heuristics. Then design a new heuristic that is based on the common idea but as different as possible from the given ones by introducing new parts.

First describe your new heuristic in one paragraph. Then implement it in Python as a function with the given signature.

Efficiency and what matters

Setting	LLM queries	Gap to the lower bound
FunSearch	about 10^6	0.97%
EoH, code only in the prompts	2 000	2.57%
EoH, thoughts only in the prompts	2 000	2.13%
EoH, thoughts and code	2 000	0.66%
EoH seeded with an expert heuristic	2 000	0.55%

- Three orders of magnitude fewer queries than FunSearch for a better heuristic
- Dropping either the thought or the code from the prompts costs most of the gain
- A known good heuristic in the initial population is inherited and improved

FunSearch versus EoH

FunSearch

- scale and simplicity: sample a lot, keep what scores
- one operator: complete v2 from v0 and v1
- diversity by islands

EoH

- representation: thought plus code
- five explicit operators for exploration and refinement
- diversity by the E1 strategy and rank-based selection

Better prompts are a cheaper lever than more samples

Exercise 3: pick the operator

A population of three bin-packing heuristics has not improved for ten generations. The best one is 40 lines long and mixes four ideas; the other two are variants of it.

Which of E1, E2, M1, M2, M3 would you apply first, and why?

- **E1** first: the population has collapsed onto one idea, so it needs something different, not a refinement
- **M3** next on the 40-line best: simplifying often removes parts that only added noise, and shorter programs are easier to improve
- **M2** last: tuning parameters of a stuck design rarely escapes a plateau

Agenda

1. Heuristics in industrial systems
2. LLMs for code generation
3. FunSearch
4. Evolution of Heuristics (EoH)
5. **Summary**

Summary

Checklist for an AHD problem

1. **Isolate** the decision function: a signature, a skeleton around it, a trivial initial version
2. **Write the evaluator**: instances, score, time limits, what to do with failures
3. **Choose instances** that represent production, and held-out ones for generalisation
4. **Decide the seed**: from scratch, or an expert heuristic to improve upon
5. **Choose the loop**: sample volume (FunSearch) or structured operators (EoH)

Next session

- **Better guidance:** reflections as verbal gradients (ReEvo)
- **A model that learns from the search:** reinforcement fine-tuning inside the loop (CALM)
- **Whole programs and many metrics:** AlphaEvolve, including a data-centre scheduling heuristic in production
- **Systems that are only partially observable:** bringing logs into the loop (TRACE), and where the two days meet

References

- Burke, Edmund K., Michel Gendreau, Matthew Hyde, et al. 2013. "Hyper-Heuristics: A Survey of the State of the Art." *Journal of the Operational Research Society* 64 (12): 1695–724.
- Liu, Fei, Xialiang Tong, Mingxuan Yuan, et al. 2024. "Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model." *International Conference on Machine Learning (ICML)*.
- Romera-Paredes, Bernardino, Mohammadamin Barekatain, Alexander Novikov, et al. 2024. "Mathematical Discoveries from Program Search with Large Language Models." *Nature* 625: 468–75.

NEC

\Orchestrating a brighter world