

Koi: A Causal, Self-Calibrating Slow-Loop Planner for Cluster-Level LLM Inference

Tandemn Team

Technical White Paper

Contents

1	Abstract	2
2	Introduction	2
3	Background	5
4	Problem Formulation	5
5	Why a Predictor Alone Is Not a Cluster Scheduler	6
5.1	Why Global Search Is Not a Static ILP	7
5.2	The Surrogate Stack	7
6	The Causal Substrate	8
6.1	Decision Variables	8
6.2	Mediators	8
6.3	Outcomes	9
6.4	Edges	9
6.5	Mechanisms	9
6.6	The Beta–Bernoulli Confidence Model	9
7	Exploration and Exploitation	10
8	Exploitation: Augmented Tchebycheff	11
9	Exploration: Expected Information Gain	12
10	Risk: Distributionally Robust Prediction	12
11	The Cluster Objective	13
12	Closing the Loop: CUSUM, Quadrants, and ICP	14
12.1	CUSUM on Mediators and Outcomes	14
12.2	Quadrant Classification	14
12.3	ICP Invariance Testing	15
13	The Slow Loop	15

14 The Agentic Planner	16
15 Tick Algorithm	18
16 Assumptions	19
17 Discussion	20
18 Appendix A	21
19 Appendix B: Notation and Key Hyperparameters	22
20 Appendix C: Tracked Variables	23
21 Appendix D: Lifecycle: Seed, Learn, Promote	24

1 Abstract

Serving large language model (LLM) inference on a heterogeneous, multi-cloud GPU fleet requires deciding, for every job, how to deploy it: which accelerator, how to shard the model across devices, which engine and quantization settings, and at how many replicas. These choices move cost-per-token and SLO attainment by multiples, yet finding a well-optimized deployment plan is notoriously hard and cannot be known in advance. This is because a deployment’s performance emerges from the non-separable interaction of the whole compute stack and is practically impossible to predict accurately. Harder still, the optimal plan is non-stationary, drifting as hardware, prices, and the engine stack change. To solve this, we propose Koi, a self-optimizing inference cluster that configures LLM inference jobs and makes cluster scheduling decisions. As it serves jobs, Koi learns for itself the causal mechanisms behind their performance, how deployment knobs shape the cluster’s latent state and, in turn, its outcomes. Performance numbers alone do not reveal why a deployment performs as it does, which is what self-optimization requires and what prior approaches, by steering on observed outcomes, cannot recover. At each interval, an LLM agent, guided by what the cluster has learned, proposes a cluster-wide plan. This plan allocates capacity across jobs, assigning each a deployment and a candidate mechanism, which is a hypothesis for why it will perform. Koi confirms or falsifies each hypothesis against realized telemetry by cross-environment invariance testing. Over time, it guides the plan toward the operator’s objective, such as cost per token under an SLO. Knowledge compounds across intervals, and the cluster improves with use.

2 Introduction

High-scale operators are increasingly focused on balancing business SLOs with the rising cost of running inference. Existing frameworks are strong at executing deployments, but they become increasingly more effective when paired with a predictive planning layer that continuously determines which deployment should run under current conditions and updates that guidance as conditions change. In many cases, matching live job characteristics to a heterogeneous mix of accelerators produces better cost-performance than relying on a homogeneous pool of maximum-performance GPUs controlled by a naive orchestrator.

Deploying a model requires choosing how to run it: which accelerator, how to partition the model with tensor (TP) and pipeline (PP) parallelism, which inference engine and quantization to enable, and how many replicas to provision. This choice can change latency, throughput, resource utilization, and cost by multiples. For example, on NVIDIA-L40S GPUs, LLaMA-70B sustains 212 tok/s in its worst feasible configuration and 1,187 tok/s in its best, $5.6\times$ throughput gap on identical hardware. Across different instance types the cost ranking even inverts. NVIDIA-A10G at \$2.20/hr costing $3.5\times$ *more* per token than an NVIDIA-L40S at \$4.50/hr because the cheaper-by-the-hour GPU is far slower for the workload. Performance like this emerges from the interaction of the entire stack rather than from any component in isolation. Hence, the best choice cannot be made in advance and can only be learned by running it. At cluster scale, with many such jobs running concurrently, choosing wrong means overspending by multiples or missing deadlines. And without correctly reasoning what happens and why that happened, the problems compound.

Problem. Koi continuously serves a stream of inference jobs on a heterogeneous, changing pool of GPU capacity spread across clouds, regions, and market types. Each job specifies a model, a workload profile (input/output length distributions and arrival pattern), a class, either latency-bound *online* or deadline-bound *batch*, and an SLO. For every job, the system must decide *how to deploy it*: the GPU type, TP/PP, engine and quantization settings, replica count, etc. It must then revisit that decision as jobs arrive and complete, as prices and capacity move, and as the engine stack changes. The objective is multi-objective and set by the operator’s preference over cost, latency, and throughput. What makes this more than scheduling is that the map from a deployment to its performance is **unknown, expensive to probe** (each trial is a multi-minute deployment costing GPU-hours), **non-stationary** shifting with hardware, virtualization, and engine versions), and **confounded** (a deployment can perform well for reasons that will not recur). The system therefore cannot optimize a known objective; it must learn the dynamics that determine performance. Deciding between what is good now and learning the unknown dynamics that make a decision good are coupled, and at fleet scale, shape every outcome that follows.

Why the performance map is unknown. Inference throughput is not a property of any single layer of the stack but of their non-separable interaction: accelerator microarchitecture, hardware topology, the cloud’s virtualization layer, the engine’s scheduling and memory management, the model’s architecture, and the workload’s token-length distribution jointly determine it, and the binding constraint shifts among compute, memory bandwidth, and communication depending on the specific combination. No layer’s contribution can be predicted in isolation, and no off-the-shelf predictor recovers the whole. Analytical roofline models are useful for estimating upper bounds, but they overestimate achievable inference throughput. They capture peak hardware limits, not the full deployment path: batching, routing, KV-cache behavior, placement, contention, and runtime overheads. Higher-fidelity simulators can model more of these effects, but they often rely on operator timings measured on a particular cluster. Those timings may not transfer cleanly across hardware, topology, runtime stack, or workload mix.

Why optimizing outcomes is not enough. The natural response is to measure: try configurations, observe their outcomes, and steer toward the cheapest one that meets the SLO. Black-box configuration selectors, LLM-driven evolutionary search, and cluster schedulers all do this in different guises: fitting or sampling an outcome surface and treating the system as a black box to be probed. But outcomes are confounded. A configuration can post a good number because of a transient capacity window, a favorable batch, or a workload that happened to suit it. An outcome-optimizer

cannot tell a robust win from a lucky one. Worse, outcome knowledge does not transfer: a throughput surface fit on one cloud’s spot A100s says little about another cloud’s on-demand H100s. On a heterogeneous, drifting fleet, an optimizer that only learns from outcomes must repay the cost of exploration. Outcomes show that a deployment performed well or poorly, but they do not explain why.

Insight: understand the dynamics, and let heterogeneity be the experiment. The only thing that transfers across hardware, time, and workload is the *causal mechanism*: the reason a configuration performs as it does (“pipeline parallelism beyond a few stages bleeds throughput into bubbles unless enough micro-batches are in flight”). A system that understood the mechanisms governing its jobs could optimize correctly, carry what it learns from one deployment to the next, and recognize when a good outcome is luck. But a proposed mechanism is only a claim until it is tested, and the test is whether it still holds when conditions change. This is what turns the fleet’s heterogeneity, normally a curse for data reuse, into the central asset. A cluster spanning many clouds, GPUs, and markets is a standing *natural experiment*, and a mechanism that is real shows up as one that holds *invariantly* across those environments. Testing a proposed mechanism for cross-environment invariance is how we separate genuine dynamics from coincidence, and it requires no designed experiment, because the fleet is already running them.

Koi. Understanding a cluster’s dynamics has historically required a human expert, someone who knows that tensor parallelism strains bandwidth, that grouped-query attention softens KV-cache pressure, or that a deep pipeline needs enough micro-batches. Automating this was practically impossible because classical causal discovery fails over a massive 140-dimensional configuration space with sparse data. Koi makes this possible by using a reasoning model that already carries a working understanding of inference physics. At every planning interval, the agent proposes a deployment for each job, along with a causal explanation of why that setup should work. But an agent alone is dangerous: it hallucinates plausible-but-wrong physics, is overconfident, and its proposals are just hypotheses. To make this work safely, Koi wraps the agent in a deterministic control layer that handles the three things an agent cannot be trusted to do. First, it falsifies: it checks the agent’s proposed mechanism against actual cluster telemetry to see if the claimed physics occurred, and updates a calibrated confidence score. Second, it plans: it turns those validated beliefs into a concrete schedule that meets SLOs at minimum cost, carefully balancing the need to exploit known-good setups, explore new ones, and minimize cluster churn. Finally, it compounds: validated mechanisms accumulate over time, meaning the system continuously sharpens its understanding and requires fewer expensive probes to place new jobs. Ultimately, the agent provides the mechanistic intuition, but the control layer falsifies it against reality and turns it into safe, cost-effective decisions.

Contributions.

- We reframe cluster-level inference optimization as understanding dynamics rather than fitting outcomes, and show why outcome-optimization is confounded and non-transferable on a heterogeneous, non-stationary fleet (Section 3).
- We develop a design in which an LLM agent proposes causal mechanisms that are made falsifiable by cross-environment invariance testing over the production fleet, grounded in the structural features of inference scheduling that make this both necessary and tractable (Section 11).
- We build Koi, a cluster control plane which is based on underlying causal mechanisms. It realizes a loop that proposes mechanisms, falsifies them against reality, calibrates belief, and acts, serving

online and batch jobs under multi-objective SLOs. It fences the agent within a deterministic planning machine, and improving with use (Section 15).

- We evaluate Koi on [models, workloads, GPU types], cost and SLO against existing state-of-the-art LLM inference cluster orchestration system.

3 Background

LLM inference throughput is not a single static number. It emerges from how multiple layers of the stack interact. To understand why optimization is difficult, we have to look at the dimensions of the configuration space. First, hardware varies wildly. Cloud GPUs like the L40S, A100, and H100 have different memory bandwidths, compute limits, and interconnects like PCIe or NVLink. Even the same GPU type can have different virtualized topologies depending on the cloud provider, which directly affects multi-GPU scaling. Second, models must be split using tensor parallelism (TP), pipeline parallelism (PP), and data parallelism (DP). The optimal split depends entirely on the hardware interconnect and the model’s architecture. Third, engine and model configurations shift the performance landscape. Engines like vLLM expose knobs for chunked prefill, prefix caching, and memory utilization. When combined with model properties like quantization or group query attention, the feasible deployment options change completely. Finally, workload shape dictates the bottleneck. If a job has long inputs and short outputs, it is prefill-dominated and compute-bound. If it has short inputs and long outputs, it is decode-dominated and memory-bandwidth-bound.

Because of all these variables, the dimensions interact non-linearly. For example, running a Qwen2.5-72B model on an L40S will result in vastly different throughputs just by changing the TP and PP split. Move that exact same configuration to an A100, and the throughput changes again due to bandwidth differences. The optimal setup will even reverse depending on the workload’s input-to-output ratio. The bottleneck constantly shifts between the compute units, the memory bandwidth, and the KV cache. Ultimately, there is no single configuration that works best for everything.

4 Problem Formulation

We formalize the decision Koi makes at each planning interval (a *tick*). It is a cluster-wide, multi-objective program over all jobs.

Jobs and resources. At tick t the cluster holds a set of active jobs $\mathcal{A}_t$ and a queue of pending jobs $\mathcal{Q}_t$. A job i is described by a model $\mathcal{M}_i$, a workload profile W_i (input/output length distributions and arrival pattern), a class $\kappa_i \in \{\text{online}, \text{batch}\}$, and a service-level objective. Capacity is a time-varying resource map R_t indexed by *environment* $e = (\text{cloud}, \text{region}, \text{zone}, \text{market}, \text{gpu_type})$, zone and market matter because they govern spot availability and price, and the environment is also the unit across which Koi later tests causal invariance.

Decision: a cluster-wide plan. A *chain* is one model copy sharded across GPUs by tensor (TP) and pipeline (PP) parallelism within a single environment. A *rank* pairs a chain specification $c = (e, x)$, where $x \in \mathcal{X}$ is a high-dimensional configuration (~ 140 knobs: parallelism, engine and version, quantization, batching and KV-cache settings), with a replica count n . A *ladder* $L_i = [(c_i^1, n_i^1), \dots, (c_i^k, n_i^k)]$ is the ordered, possibly heterogeneous deployment for job i , with data-parallel width $\sum_r n_i^r$. At each tick the planner emits a plan $P_t = \{L_i\}_{i \in \mathcal{A}_t \cup \mathcal{Q}_t}$, a ladder for every

active and pending job. *Keeping* the current ladder ($L_i = L_i^{(t-1)}$) is admissible for $i \in \mathcal{A}_t$, and *deferring* ($L_i = \emptyset$) for $i \in \mathcal{Q}_t$.

A multi-objective objective. Running a ladder produces a per-job outcome vector

$$y_i = (\text{cost-per-token}, p99 \text{ TTFT}, p99 \text{ TPOT}, \text{throughput}, \text{SLO margin}) \in \mathbb{R}^m, \quad (1)$$

with cost, TTFT, and TPOT minimized and throughput and SLO margin maximized. “Good” is generally multi-objective and differs by job: an online chatbot cares about p99 latency, while a batch evaluation about throughput per dollar. The operator encodes this as a per-job preference, a weight vector w_i on the simplex together with a set of objectives promoted to hard constraints. Koi scalarizes the preference with an augmented Tchebycheff value $J_i(y_i; w_i, z^*)$ against an ideal point z^* (the best achievable value per objective); we use Tchebycheff rather than a weighted sum because it is Pareto-complete.

The per-tick program. The planner maximizes cluster-wide deployment value while accounting for the cost of reconfiguration. Mathematically, we represent this as

$$\max_{\{L_i\}} \sum_{i \in \mathcal{A}_t \cup \mathcal{Q}_t} J_i(y_i(L_i); w_i, z^*) - \lambda_{\text{swit}} \sum_{i \in \mathcal{A}_t} \text{SwitchCost}(L_i^{(t-1)}, L_i) \quad (2)$$

For every job, this is subject to: **(C1)** resource feasibility, the chains requested across all ladders fit the free capacity in each environment under R_t ; **(C2)** per-chain physical feasibility, each chain fits VRAM, TP divides the attention and KV head counts, and PP divides the layer count; **(C3)** the job’s SLO, expressed as a chance constraint $P(\text{breach}) \leq \tau$ under prediction uncertainty; **(C4)** a swap budget $|\{i \in \mathcal{A}_t : L_i \neq L_i^{(t-1)}\}| \leq B_t$; and **(C5)** admission control, a pending job is deferred only if no feasible ladder exists. SwitchCost prices the real cost of churning a live deployment (cold start, brief A/B overlap, teardown, and transition risk), so a marginally better ladder does not trigger a disruptive migration.

Why it cannot be solved directly. Equation (2) is not a standard program, because the map $L_i \mapsto y_i(L_i)$ is unknown. Looking at the config space, y_i is the non-separable product of the full stack; it is also too expensive to probe exhaustively, non-stationary, and confounded. Koi therefore does not optimize a known objective. It estimates y_i from the best available evidence, deploys, measures the realized outcome, and corrects, learning the cluster’s dynamics while it decides.

5 Why a Predictor Alone Is Not a Cluster Scheduler

An intuitive chain of thought after reading the problem statement is to use a trained predictive model for shape/performance prediction, like AIConfigurator/DynoSim, or a nearest-neighbour database over profiled data for different models. The answer is that most predictive models are **local predictors**. These operate in isolation under a table of previous assumptions of how models perform. However, what we are trying to develop is a cluster scheduler, which is a **global sequential decision-maker**. A prediction model answers if given this candidate deployment, what behaviour should I expect? A cluster scheduler must answer, which deployments should exist for all jobs, given shared finite capacity?

Suppose each submitted job has a model name and an SLO requirement. A predictor can evaluate candidate deployments for each job independently. A tempting scheduler will then, a) for

each job, enumerate candidate configurations. b) use the predictor to score each candidate, c) assign the locally best feasible configuration.

This is greedy, because it fundamentally ignores the opportunity cost of scarce resources. As an example, consider a cluster with one H100 node and one A100 node serving two jobs:

Job	Workload	H100 prediction	A100 prediction
Job 1	online chat	TPOT SLO satisfied	TPOT SLO violated
Job 2	batch generation	cheaper \$/token	slightly higher \$/token, deadline still met

For Job 1, the H100 is not simply the better choice; it is the only feasible choice. For Job 2, however, the H100 is only marginally better than the A100. If Job 2 is processed first, a greedy per-job optimiser may assign it to the H100 because the predictor correctly identifies the H100 as its best local configuration. But doing so consumes the only resource capable of making Job 1 feasible.

In isolation, both jobs perform better on H100s, and thus the predictor was not necessarily wrong. The failure, however, is that local prediction does not encode global opportunity cost. The scheduler must compare not only the value of a resource to a job, but the value of denying that resource to every other job operating within the same resource pool.

5.1 Why Global Search Is Not a Static ILP

A naive way to make a global optimiser is to enumerate all candidate deployments, use the simulator to score them, and solve an ILP or mixed-integer program. However, the candidate space is very big and dynamic, not a small static table. It depends on a) heterogeneous GPU types and instance pools, b) multiple clouds, regions, zones, and markets, c) tensor, pipeline, data, expert, sequence, and context parallelism degrees, d) quantization choices. e) engine versions and scheduler/router policies.

We also note that a predictor alone is susceptible to bias and drift. For example, a predictor trained specifically on profiling data from one particular inference-stack version, and for one family of models, has to go through re-training when a new version or a new model architecture appears.

5.2 The Surrogate Stack

Even though we cannot use a predictor as a global cluster optimiser, we can still get a good estimate of an individual job’s performance as a starting point. We define a mixture of predictors whose outputs are learnable over time as a **Surrogate Stack**. For a candidate configuration and workload context X , and environment e , a surrogate gives predictions:

$$S(X, e) \rightarrow (\hat{V}, \hat{Y}),$$

where $\hat{V}$ denotes predicted intermediate system behaviours, such as memory pressure, communication overhead, KV-cache behaviour, pipeline bubbles, queueing effects, and other measurable physical mediators of performance. $\hat{Y}$ denotes predicted outcome-level metrics such as p99 TTFT, p99 TPOT, throughput, cost per token, and SLO margin. These are the quantities the scheduler ultimately cares about when comparing candidate deployments.

This distinction is important as an outcome prediction alone can tell us that a surrogate was numerically close, but it does not tell us whether the deployment worked for the expected reason. By also predicting $\hat{V}$, Koi can compare observed behaviour against the surrogate’s internal explanation of why a candidate should perform well. Without such a surrogate stack, a global optimiser would

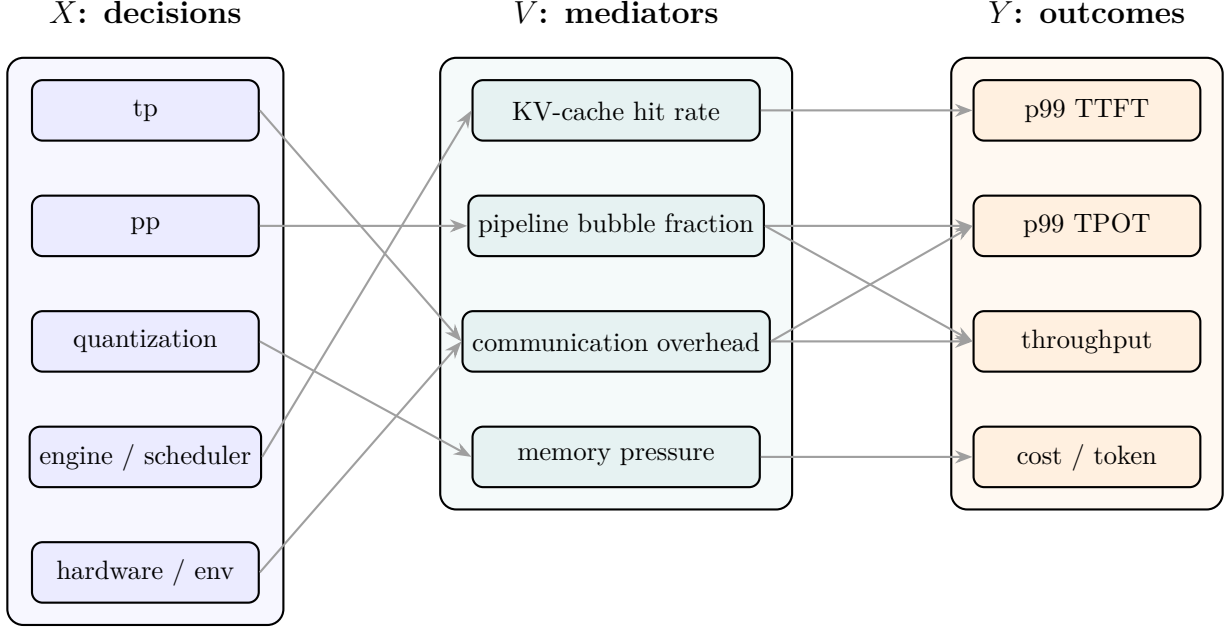


Figure 1: Koi’s causal substrate. Decisions X affect measurable mediators V , which in turn affect outcomes Y .

have to test every deployment directly on real hardware. For Koi, however, this map is only a forward evaluator and estimates what is likely to happen for a given X in environment e . It does not specify which X should be proposed, how many candidates should be compared, which job should receive scarce capacity, or whether a locally strong candidate should be rejected because it harms the cluster.

6 The Causal Substrate

Koi represents inference behaviour through a closed-world graph:

$$X \rightarrow V \rightarrow Y,$$

where X are controllable decisions, V are measurable mediators, and Y are outcomes. The directed acyclical graph of X , V , Y is visualized in Figure 2.

6.1 Decision Variables

X represents the workload characteristics, and it contains what Koi can directly configure: hardware, environment, engine, parallelism (**tp**, **pp**, **dp**, **ep**, **sp**, **cp**), quantization, batching limits, scheduler policy, routing policy, engine version, and memory/KV options.

6.2 Mediators

V contains the intermediate physical variables that impact the job performance, such as KV-cache hit rate, pipeline bubble fraction, communication overhead, memory-used fraction, KV pressure, etc. These are the mediators between the decision variables and an outcome, and cannot be tweaked directly by Koi.

6.3 Outcomes

Y contains the quantities which Koi aims to optimize on a per-job basis, such as throughput, p99 TTFT, p99 TPOT, cost per token, and SLO margin.

6.4 Edges

An edge is a causal relation

$$e = (src \rightarrow dstn),$$

where either $(src \in X, src \in V)$ or $(dstn \in V, dstn \in Y)$. Koi intentionally avoids direct $X \rightarrow Y$ edges so that every claimed effect passes through a measurable mediator.

6.5 Mechanisms

A mechanism is a scoped causal story

$$M = (E_M, \mathcal{S}_M, \eta_M, B_M),$$

where E_M is a set of edges forming one or more $X \rightarrow V \rightarrow Y$ paths; $\mathcal{S}_M$ is the scope where the mechanism applies; η_M is a natural-language of the relationship; and B_M is the mechanism confidence state. For example, a given mechanism can be represented as:

```
{
  "edge_ids": [
    "pp->pipeline_bubble_fraction",
    "max_num_batched_tokens->pipeline_bubble_fraction",
    "num_hidden_layers->pipeline_bubble_fraction",
    "pipeline_bubble_fraction->throughput_token_per_sec",
    "pipeline_bubble_fraction->p99_tpot_ms"
  ],
  "scope": {
    "x": ["pp", "max_num_batched_tokens", "num_hidden_layers"],
    "v": ["pipeline_bubble_fraction"],
    "workload_type": "any",
    "model_type": "dense_large",
    "conditions": [ {"feature": "pp", "op": ">", "value": 1} ]
  },
  "narrative": "Pipeline parallelism leaves fill/drain bubbles...",
  "status": "active"
}
```

6.6 The Beta–Bernoulli Confidence Model

Each edge carries a belief about whether the corresponding causal link will hold under deployment. We model this belief as a Bernoulli success probability p : when Koi validates an edge, the link either behaves as expected or it does not. The natural Bayesian model for this probability is the **Beta distribution**, so Koi represents each edge’s belief as

$$p \sim \text{Beta}(\alpha, \beta).$$

Here, α and β are evidence counts. α accumulates **pseudo-successes**, meaning cases where the observed behaviour supports the edge’s claimed relationship. The parameter β accumulates **pseudo-failures**, meaning cases where the observed behaviour contradicts or fails to support that relationship. At initialisation, we seed each edge using domain knowledge, profiling data, and past

deployment telemetry, giving an initial confidence $c_0 \in [0, 1]$. An edge initialised with $c_0 = 0.9$ starts with a strong belief that the link will hold, while an edge initialised with $c_0 = 0.1$ starts with a weak belief. After deployment, each validation updates the posterior:

$$\alpha = \alpha + 1 \quad \text{if the link holds,} \quad \beta = \beta + 1 \quad \text{if the link fails.}$$

We model an edge confidence with a posterior mean, c , where

$$c = \mathbb{E}[p] = \frac{\alpha}{\alpha + \beta}.$$

We also present a posterior variance which captures uncertainty, given by

$$\text{Var}[p] = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}.$$

The normalised uncertainty kernel is

$$U(\alpha, \beta) = 12 \cdot \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}.$$

The factor of 12 is a normalisation constant. Multiplying by 12 makes the uncertainty of an untested edge equal to one:

$$U(1, 1) = 1.$$

As more evidence accumulates, $\alpha + \beta$ grows and the variance decreases, so $U(\alpha, \beta)$ moves toward zero. Thus, U should be read as a relative uncertainty score: values near 1 indicate little evidence, while values near 0 indicate that the edge has been repeatedly tested.

7 Exploration and Exploitation

We have a causal model with calibrated but uncertain beliefs, and a simulator that turns a config into predicted $(\hat{V}, \hat{Y})$. To actually *plan*, Koi needs to pick between exploring uncertainty or defaulting to well-performing configurations. We define these notions as **exploitation** and **exploration**.

1. **Exploit.** For each job right now, among candidate configs, select the one that best serves that job’s objectives. This requires a score that collapses multiple objectives (throughput, TTFT, TPOT, cost) into one comparable number, respecting per-job priorities (Section 8, Tchebycheff).
2. **Explore.** Koi’s beliefs are uncertain, and a purely greedy planner would keep re-deploying the same well-performing configurations and never reduce that uncertainty. It would be permanently hostage to a possibly-wrong surrogate. We need to sometimes deploy configs specifically because they would teach us the most about uncertain causal links (Section 9, EIG).

This is the classic explore/exploit trade-off, but with a causal twist: exploration is targeted at reducing variance on a specific set of edges or mechanisms, as opposed to random actions. We define two further considerations, defined as risk and churn, which will round out the decision later. **Risk** means an SLO-violating config is bad even if its mean looks good, and **churn** means constantly swapping deployments is expensive. In the following sections, we formally define each and finally, in (Section 11), combine the notions of exploit, explore, risk, and churn into a single scoring objective.

8 Exploitation: Augmented Tchebycheff

A job is evaluated along several objectives at once. For example, Koi may care about cost per token, p99 TTFT, p99 TPOT, throughput, and SLO margin. The relative importance of these objectives is encoded by a weight vector w_t on the simplex:

$$\sum_j w_j = 1.$$

A natural first attempt is to collapse the objective vector into a weighted sum,

$$\sum_j w_j y_j, \quad y_j \in Y.$$

However, this approach is not sufficient. A linear weighted sum can only select points on the convex hull of the Pareto frontier. Inference trade-offs, however are often non-convex: for example, the relationship between throughput and latency as batch size changes can contain both concave and convex regions. As a result, a weighted-sum objective is structurally blind to parts of the achievable frontier. It may miss useful configurations that are Pareto-efficient but not exposed by any linear scalarization. We use Tchebycheff because it is Pareto-complete on a non-convex, per-job multi-objective frontier; a weighted sum would silently hide most of the achievable configs.

Instead, Koi measures each candidate’s **weighted max-norm distance** from z^* , a reference point maintained by the slow loop that records the best value seen so far for each objective. For a candidate’s predicted $\hat{y}$, we represent this distance is given by J where

$$J = - \left[\max_j w_j g_j + \rho \sum_j w_j g_j \right],$$

and

$$g_j = \begin{cases} \frac{z_j^* - \hat{y}_j}{\text{range}_j}, & j \in \mathcal{Y}_{\max}, \\ \frac{\hat{y}_j - z_j^*}{\text{range}_j}, & j \in \mathcal{Y}_{\min}. \end{cases}$$

Here $\mathcal{Y}_{\max} = \{\text{throughput}, \text{slo_margin}\}$ and $\mathcal{Y}_{\min} = \{\text{TTFT}, \text{TPOT}, \text{cost}\}$. $g_j > 0$ means “worse than ideal”; the sign is flipped so **larger J (closer to 0) is better**, matching the convention that the planner maximizes. Gaps are normalized by range_j , initialized over past seen deployment trends, so weights stay comparable across milliseconds, tokens/sec, and dollars.

- The max term recovers non-convex regions and improving a config means improving its worst weighted objective, and sweeping w_t across the simplex sweeps the entire (possibly non-convex) Pareto front which a weighted sum cannot do and the augmentation term ($\rho \approx 10^{-3}$) breaks ties on flat max-norm ridges.

Here are two examples of Tchebycheff weights for two different scenarios, using the objective ordering

$$(\text{cost}, \text{p99 TTFT}, \text{p99 TPOT}, \text{throughput}, \text{SLO margin}).$$

For an online serving job, latency dominates:

$$w_{\text{online}} = (0.10, 0.35, 0.35, 0.10, 0.10).$$

For a batch or offline generation job, cost and throughput dominate:

$$w_{\text{batch}} = (0.35, 0.05, 0.05, 0.35, 0.20).$$

The optimization equation is unchanged; only the weights change. Thus the same candidate deployment may be judged differently depending on the job class: an online job treats TPOT or TTFT as the bottleneck, while a batch job treats cost and throughput as the primary axes.

9 Exploration: Expected Information Gain

If we only maximized J , we would keep re-using whatever the surrogate currently rates highest and never learn where it is wrong. We want a term that rewards deploying configs that would teach us the most, even utilizing the most uncertain, least-sampled, yet relevant causal links.

Why not ε -greedy or upper confidence bound (UCB)? Random exploration wastes time on deployments for whose links we already understand; a bandit UCB explores high-*mean* arms. We want to target known uncertainty on the causal graph itself. Exact Bayesian EIG would need outcome distributions and mechanism posteriors we do not have, so Koi uses a **deterministic proxy with the same intent**: reward a candidate in proportion to the Beta-posterior *variance* of the edges and mechanisms it would exercise. For a ladder L' ,

$$\text{EIG}(L') = \sum_{e \in \text{edges}(L')} a_e U(\alpha_e, \beta_e) + w_M \sum_{M \in \text{mechs}(L')} a_M U(\alpha_M, \beta_M),$$

The masks $a_e, a_M \in \{0, 1\}$ are **eligibility gates**: a link counts only if the ladder actually exercises a testable $X \rightarrow V \rightarrow Y$ path through it (the child V is observable, enough samples exist). Across a plan, an edge tested by several ranks is not double-counted. We represent this by **saturation aggregation**,

$$A_e(P) = 1 - \prod_i (1 - a_e(L'_i)),$$

which drives the marginal value of re-testing an already-tested edge to zero. EIG enters the score weighted by β_t , the exploration incentive the slow loop raises while learning and lowers as the cluster converges (Section 13).

10 Risk: Distributionally Robust Prediction

The surrogate stack gives Koi a point prediction for each candidate deployment, but a point prediction is not enough for scheduling. Koi does not only need to know what performance is expected under the surrogate; it needs to know how much risk it takes by acting on that prediction.

There are two sources of risk. First, the surrogate can be wrong. A candidate may lie in a part of the configuration space where Koi has limited telemetry, causing predicted latency, throughput, or cost to be biased. Second, the deployment characteristics themselves are not fixed. Request lengths, arrival rates, batching patterns, KV-cache reuse, GPU contention, network conditions, and placement can shift between the data used to calibrate the surrogate and the next deployment.

To address this, we use distributionally robust optimization (DRO). Rather than treating the empirical residual distribution as ground truth, Koi considers a neighborhood of plausible residual distributions around it. The size of this neighborhood is controlled by a Wasserstein radius ε . Larger ε means a larger uncertainty band around each prediction. Conversely, a smaller ε means

the surrogate has been well calibrated under similar conditions, so the band can tighten. The Wasserstein radius is bounded by an upper and lower bound, given by

$$\text{upper}_j = \hat{y}_j + Q_q(r_j) + 2\varepsilon\hat{\sigma}_j, \quad \text{lower}_j = \hat{y}_j + Q_{1-q}(r_j) - 2\varepsilon\hat{\sigma}_j,$$

where $Q_q(r_j)$ is the empirical q -quantile (default $q = 0.95$) of the residuals for objective j , $\hat{\sigma}_j$ their empirical std, and the $2\varepsilon\sigma$ term is a practical Wasserstein-1 inflation (tight for near-symmetric residuals). Before any residuals exist, the band falls back to the conservative $\hat{y} \pm \varepsilon$.

From the band, the chance-constraint estimate of violating a per-objective SLO threshold τ_j is the empirical exceedance plus a robustness inflation, with a union bound across objectives:

$$\Pr_{\text{DRO}}[g_j > 0] \leq \hat{P}_{\text{emp},j} + \frac{\varepsilon \text{Lip}_j}{\hat{\sigma}_j}, \quad \Pr_{\text{DRO}}^{\text{any}} = 1 - \prod_j (1 - \Pr_{\text{DRO}}[g_j > 0]).$$

This $\Pr_{\text{DRO}}$ is the risk penalty in σ . The radius ε is an adaptive coverage controller in the slow loop grows ε when realized outcomes fall outside the band too often (band too tight) and shrinks it when they're comfortably inside (band too loose), holding empirical coverage near a target (default 90%):

$$\varepsilon \leftarrow \begin{cases} \varepsilon(1 + \eta_\varepsilon) & \text{coverage} < \text{target} - \text{deadband} \\ \varepsilon(1 - \eta_\varepsilon) & \text{coverage} > \text{target} + \text{deadband} \\ \varepsilon & \text{otherwise} \end{cases}$$

With this framing, Koi's SLO promises are honest about its own prediction error and self-tighten as the surrogate calibrates.

11 The Cluster Objective

Koi does not optimize one ladder in isolation, but rather it optimizes placement for a job for all jobs in the whole cluster. For a single job i 's candidate ladder L'_i , we compute a per-candidate score:

$$\sigma_i(L'_i) = J_i + \beta_t \cdot \text{EIG}(L'_i) - \gamma \cdot \Pr_{\text{DRO},i} - \lambda_{\text{swit}} \cdot \text{SwitchCost}_i.$$

- J_i , exploit (Tchebycheff, Section 8).
- $\beta_t \cdot \text{EIG}$, explore, weighted by the annealed exploration incentive (Section 9, Section 13).
- $\gamma \cdot \Pr_{\text{DRO},i}$, risk: SLO-violation probability under the DRO band ($\gamma = \text{GAMMA_SLO}$, Section 10).
- $\lambda_{\text{swit}} \cdot \text{SwitchCost}_i$, **churn penalty** for changing *this* job's deployment. Switching is computed by several components: **cold-start** (spin up new chains), **parallel running** (briefly pay for old+new during an A/B canary), **kill** (tear down old chains), and **risk** (chance the new config underperforms inside its DRO band). λ_{swit} is the slow loop's churn price.

The planner's primary goal is to optimize the cluster objective. Within a given plan P , an action is assigned to each job. The total objective value is calculated by summing σ_i exclusively for ladder-bearing actions (place, swap, retry, resume). Since non-deploying actions (keep, defer, preempt, terminate, diagnose) do not introduce new ladders, they contribute exactly zero to the total. Because Koi is optimizing placement for all jobs in the cluster, we represent this as

$$\Sigma(P) = \sum_{i \in \text{ladder actions}} \sigma_i(L'_i) = \underbrace{\sum_i [J_i + \beta_t \text{EIG}(L'_i) - \gamma \text{Pr}_{\text{DRO},i}]}_{\text{total quality across all ladders}} - \underbrace{\lambda_{\text{swit}} \sum_i \text{SwitchCost}_i}_{\text{total cluster churn}}$$

$\Sigma(P)$ is the single comparable currency that lets a multi-objective, uncertain, risk-and-churn-aware decision over the whole cluster which is the total deployment quality summed over all ladders.

Total churn is also hard-bounded by the swap budget B_t (Section 13) which caps how many active jobs may change in one tick, so λ_{swit} *prices* churn while B_t *limits* it.

12 Closing the Loop: CUSUM, Quadrants, and ICP

A plan is only half the system and we need to find out whether reality matched the story, and feed that back into the Beta confidences. After a deployment runs for a tick, Koi collects sub-tick **trajectories** of each V and Y and computes three statistic tests.

12.1 CUSUM on Mediators and Outcomes

For each variable, Koi compares the observed trajectory to the surrogate’s prediction and runs a **two-sided CUSUM** (cumulative-sum) change detector on the residual stream $r_t(\text{observed}_t - \text{predicted}_t)$:

$$S_t^+ = \max(0, S_{t-1}^+ + r_t - \delta), \quad S_t^- = \min(0, S_{t-1}^- + r_t + \delta),$$

firing **DIVERGED** when $S_t^+ > h$ or $S_t^- < -h$, else **MATCHED**. The slack δ and threshold h are set per variable as multiples of that variable’s residual std ($\delta = 0.5\sigma$, $h = 4\sigma$) so the test is **unit-aware** and works for milliseconds and for tokens/sec without hand-tuning.

We chose CUSUM and not a t -test on means because CUSUM is sequential and accumulates small persistent drifts, so it catches a config that is slowly degrading. This catches the failure mode of a deployment that looked fine at deploy time and checks whether predicted latent values are drifting apart from actual results, rather than basing the verdict on outcomes alone. We perform CUSUM on V and not just Y because if we only watched outcomes (Y), we could never tell a good outcome for the right reason from a lucky one. By also watching the mediators (V), we can know if the mechanism’s claimed intermediate physics actually happened. A mechanism M validates over its own bundle of V s and Y s as the trajectories are filtered through that mechanism’s edges.

12.2 Quadrant Classification

The two CUSUM verdicts (did the V bundle match? did the Y bundle match?) form a 2×2 quadrant that we define as the entire learning signal. Each region in the quadrant is represented as

	Y matched	Y diverged
V matched	Q1 , replicable success (mechanism <i>and</i> outcome held)	Q2 , sound mechanism, unlucky outcome
V diverged	Q3 , <i>lucky arm</i> (good outcome, mechanism did not hold $\rightarrow$ punish)	Q4 , falsified (neither held)

These labels are *per mechanism*: one deployment row feeds N Beta updates, one per applicable mechanism where evidence compounds.

12.3 ICP Invariance Testing

A causal claim that only holds in one environment is not really causal. **Invariant Causal Prediction (ICP)**, tests, per edge, whether the residual distribution is invariant across environments (cloud/region/GPU). Koi groups an edge’s residuals by environment and runs an F-test (or permutation test). with enough environments and samples ($\geq \text{n_env_min} = 3$ environments, each $\geq \text{n_b} = 15$ samples) it returns:

- **accept** ($p > 1 - \alpha_{\text{ICP}}$) classifying as invariant where the edge earns α even on weaker quadrants.
- **reject** ($p < \alpha_{\text{ICP}}$) classifying as not invariant and the edge loses confidence (β) regardless of the local outcome boosting our understanding for the edge as environment-specific and not a general law.
- **undecided** or insufficient environments/samples. We only apply small-magnitude updates (never freeze learning).

ICP modulates the edge update magnitude (the accept/reject/undecided rows of the edge update table) and the mechanism update uses the quadrant alone. Together, CUSUM answers if the system behaved as predicted, the quadrant answers if it was skill or luck, and ICP answers if it generalized. All three flow into the Beta confidences that drive the next tick’s planning and exploration.

13 The Slow Loop

The σ objective also has some adaptive variables, the weights w_t , the exploration incentive β_t , the swap budget B_t , the churn price λ_{swit} , the DRO radius ε , and the ideal point z^* . These should not be fixed constants because early on we want aggressive exploration and re-planning but as the cluster converges we want to kep exploiting good placements and stop churning. The slow loop is a small control layer that adapts all of them each tick, driven by **regret** across **three different timescales**:

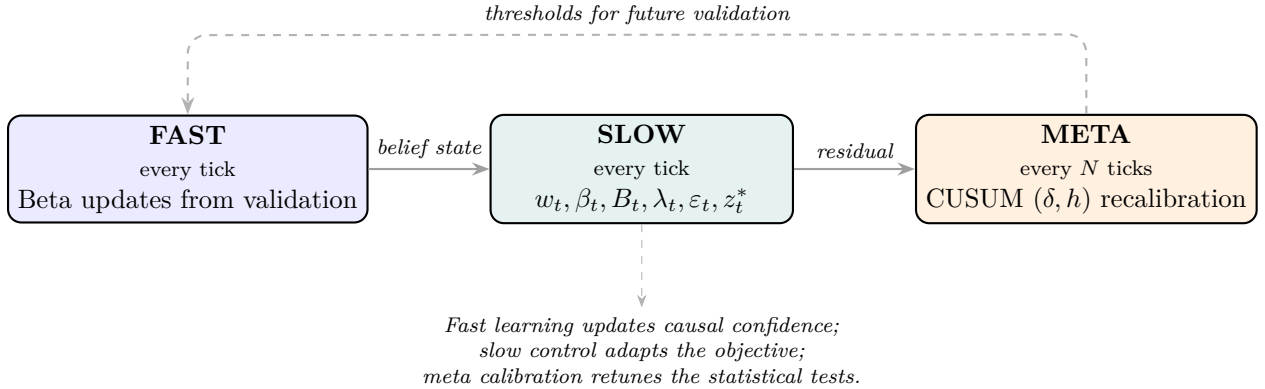


Figure 2: Koi’s three-timescale learning loop. Fast updates change causal confidence, slow updates adapt planning knobs, and meta updates recalibrate validation thresholds.

Regret is derived from the **Q1 rate**, the fraction of validated deployments that landed in Q1 quadrant (replicable success). High instantaneous regret showcases we are still often wrong and we should keep exploring. The slow loop’s updates are multiplicative-weights or mirror-descent controllers. (Appendix A Section 18)

14 The Agentic Planner

We have mentioned the important components we need but we need an executor that *uses* those to produce a plan. We believe this executor is an **LLM agent**, an RLM (recursive language model) running the S4 planning state. The design choice here that the per-tick decision is judgment-heavy (reconcile many jobs, many objectives, finite capacity, partial beliefs) and benefits from a broad world-model, and must remain steerable but must also be grounded in the real math. So the agent does not free-form a plan but reasons in code against a set of deterministic tools.

Instead of using standard function calling, the agent acts by writing code. It writes Python in a sandboxed REPL where the live cluster state and the tool registry exist as ordinary functions. The agent inspects the state, calls the tools, builds a plan, and commits it by calling `FINAL_VAR(plan)`. We chose this RLM style over native tool-calling because it makes the system completely model-agnostic. It works out of the box with frontier APIs or open models like Llama, Qwen, and DeepSeek over any standard endpoint. Because the tools are just regular Python functions, it doesn't matter if a model has weak tool-calling support, and we can handle models without a system role simply by folding the prompt. Here are the agent's inputs and the pipeline it drives to propose, validate, and deploy:

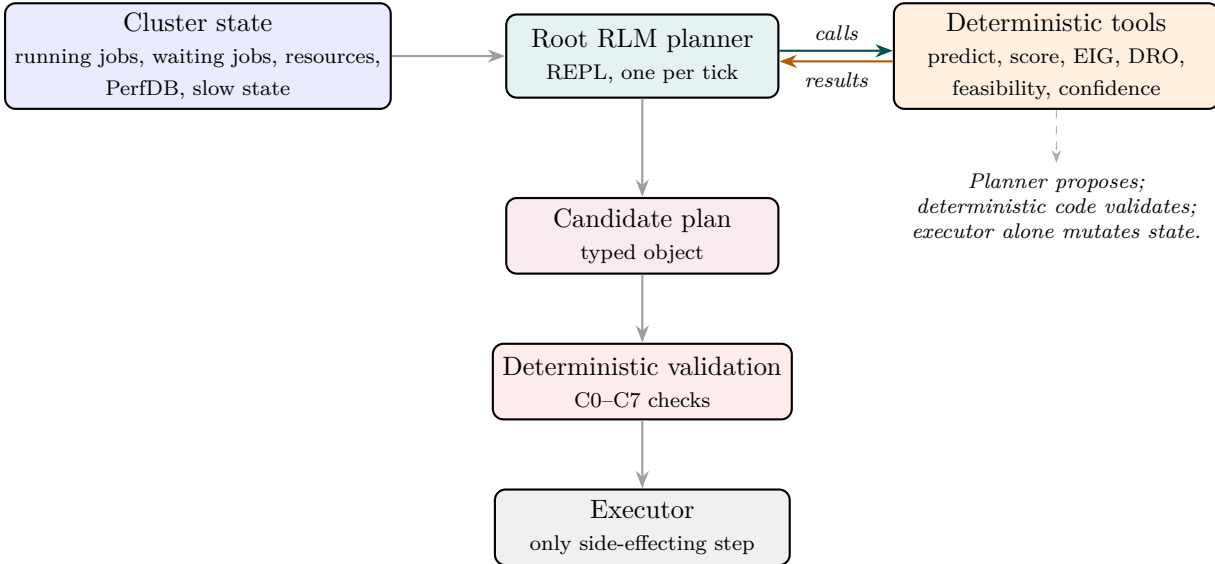


Figure 3: Koi’s agentic planning boundary. The RLM planner searches over plans using deterministic tools, but only the validated plan reaches the executor.

The tools are all read-or-compute with no side effects and they fall into the following families:

Family	Tools
Cluster/context	<code>get_cluster_state</code> , <code>get_resource_map</code> , <code>get_active_jobs</code>
Tenant/budget	<code>build_tenant_envelopes</code> , <code>validate_budget_book</code> , <code>run_job_specialists</code>
Resource simulation	<code>simulate_allocation</code> , <code>enumerate_ladder</code> , <code>size_ladder</code>
Mechanism/confidence	<code>get_scope</code> , <code>get_edge_confidence</code> , <code>get_mechanism_confidence</code> <code>get_influencing_knobs</code> , <code>set_new_mechanisms</code>
Prediction/scoring	<code>predict_outcome</code> , <code>get_z_star</code> , <code>compute_tchebycheff</code> <code>optimize_config</code> , <code>compute_eig</code> , <code>compute_switching_cost</code> <code>compute_slo_dro</code> , <code>compute_sigma</code>

Every scoring path optimizes against actual deployed numbers, and a surrogate that was wrong on the last tick will self-correct as the performance database grows. Using a rank configuration and the target throughput, Koi computes the number of replicas based on per-replica throughput and available capacity. It is also regime-aware and for batch jobs, it plots batch sizes on a throughput-vs-deadline curve. For online jobs, it rejects configurations where the predicted p99 TTFT/TPOT exceeds the target and caps utilization to throttle expected throughput and prevent queue buildup. To prevent split-brain resource contention, we take a budget-first approach where single root planner handles all cluster-wide trade-offs and allocates budgets before running any per-job specialists. The flow goes from tenant envelopes, to a job-priority table, to a BudgetBook. Once validated, the per-job specialists run within their specific budget slices and report their fitness (e.g., starved, happy, over-provisioned, or blocked) instead of fighting for resources. The specialists only output proposals and the root then rescores these proposals using σ , reconciles them, and reallocates resources based on the fitness signals. This strict ordering prevents two different layers from trying to grab the same GPUs and specialists are limited to proposing place, keep, swap, or defer. Any cross-job or lifecycle actions like preempt, resume, retry, terminate, or diagnose are strictly handled by the root.

We enforce safety invariants entirely in code and not in the prompt. Before validation, the committed plan is parsed and shape-checked and if a plan is malformed, the system degrades to a safe original state. We also bound each agent trajectory by a maximum number of turns (K_{max}), a wall-clock timeout, and a consecutive-error limit. The agent runs K_p independent trajectories and returns the plan with the best σ score:

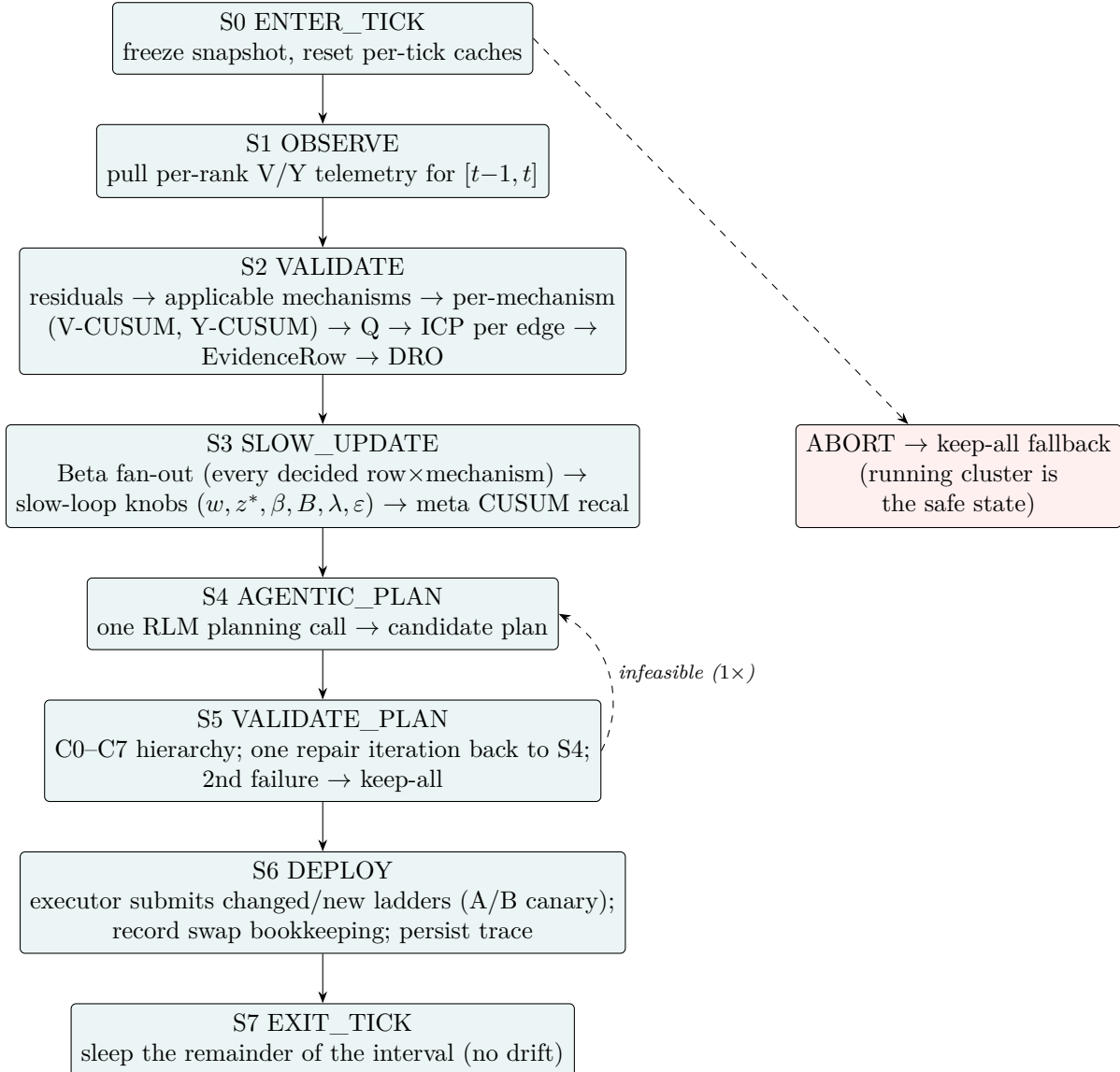
```
plan = {
  "tick_rationale": "<cluster-wide reasoning>",
  "actions": [
    {
      "job_id": "job_123",
      "type": "place",
      "tenant_id": "tenant_abc",
      "ladder": [
        {
          "role": "aggregate",
          "env": ["reserved", "aws", "us-east-1", "use1-az1", "H100"],
          "config": {
            "instance_type": "p5.48xlarge",
            "tp": 4,
            "pp": 1,
            "dp": 1,
            "ep": 1,
            "gpu_count": 4,
            "engine_name": "vllm"
          },
          "n_replicas": 2,
          "mechanism_id": "M_some_mechanism"
        }
      ],
      "target_tps": 1500.0,
      "mechanism_id": "M_some_mechanism",
      "swap_reason": "retune",
      "budget_ref": "slice_job_123",
      "rationale": "why this action was chosen"
    }
  ]
}
```

Each rank comes with a role, a 5-tuple environment definition (e.g., ["reserved", "aws", "us-east-1", "use1-az1", "H100"]), a config for the various hardware and parallelism knobs (instance_type, tp, pp, dp, ep, gpu_count, engine_name, etc.), a replica count, and a mechanism_id.

The action itself also includes metadata like the `tenant_id`, target throughput, a `swap_reason`, a `budget_ref`, and a plain-text `rationale`. If a weaker model outputs an incomplete plan, any omitted jobs automatically default to KEEP (running) or DEFER (waiting) so the cluster is always fully covered. The previous sections explain what we are optimizing and how we track state, but they don't explain how to actually search through this massive action space. Generating good deployment candidates requires real domain knowledge about hardware, model fit, parallelism, costs, and tenant constraints. Because of this, Koi uses a reasoning model to drive the search across deterministic tools. The model's only job is to propose candidate plans; it cannot execute side effects. All the actual mechanics predictions, scoring, feasibility checks, and state updates are strictly handled by deterministic tool calls.

15 Tick Algorithm

Koi runs continuously through a tick state machine.



The tick algorithm is a) Freeze the cluster snapshot, b) Collect rank-level telemetry for the previous interval, c) Produce evidence rows by computing residuals, CUSUM verdicts, quadrants, and invariance tests. Update edge and mechanism confidences, d) Update slow-loop variables, e) Generate candidate cluster plans, f) Validate the selected plan against feasibility constraints, g) Deploy only the validated plan, h) Repeat.

We strictly separate the system phases: observation is replayable, learning is explicit, planning is validated, and deployment is the only step that executes actual side effects. For evidence semantics, observation is scoped to the rank while verdicts are scoped to the mechanism. A single rank produces one set of V/Y trajectories. Every applicable mechanism filters these trajectories through its own bundle to generate its specific (V-verdict, Y-verdict) and Q. Because of this, one `EvidenceRow` drives N Beta updates. To ensure observation stays idempotent and replayable, S2 only writes evidence, while S3 reads it and applies the actual updates. The (δ, h) CUSUM thresholds are resolved first from the slow loop’s recalibrated table. As a cold-start fallback, they can also self-calibrate using the rank’s own residuals. Finally, our failure policy is strictly enforced: any unhandled exception between S0 and S6 immediately aborts the tick and triggers a keep-all fallback. A half-planned tick is dangerous, so leaving the running cluster exactly as it is serves as our safe state.

16 Assumptions

Assumption 1: Near-Optimal Proposal Coverage

We use an LLM to come up with potential cluster-level plans and placements. We hypothesize that the agent goes down a “ReAct-like” trajectory of multiple reasoning steps, tool calls, and observations to produce P_t , the cluster-level plan. We assume that if we let multiple such parallel trajectories run, among the proposed trajectories there exists at least one that is within ε -error of the optimal placement.

Intuitively, at tick t the proposal model generates multiple candidate trajectories. We do **not** require every sampled trajectory to be good. We only require that the proposal set is rich enough that, with sufficiently high probability, at least one sampled trajectory lands close to the true optimal trajectory.

Let $K_p \in \mathbb{N}$ be the number of candidate trajectories sampled, $\varepsilon > 0$ the allowed error tolerance, $\eta \in (0, 1]$ the minimum probability that the proposal set contains a near-optimal trajectory, P_t^* the true optimal trajectory at tick t , $P_1, \dots, P_{K_p}$ the sampled candidates, $d(P_j, P_t^*)$ the distance between candidate P_j and the optimum, and q_ϕ the **proposal distribution**. Then:

$$\Pr_{P_1, \dots, P_{K_p} \sim q_\phi^{\otimes K_p}} \left[\min_{1 \leq j \leq K_p} d(P_j, P_t^*) \leq \varepsilon \right] \geq \eta.$$

The core assumption is that among the K_p proposed trajectories, at least one has error from the optimal placement P_t^* of at most ε , with probability at least η .

Assumption 2: Convergence

We define Koi as having converged over a rolling window of W ticks when the cluster reaches a low-regret, low-churn fixed point. In this state, mechanism predictions validate against reality, exploration no longer finds materially better placements, and no active job has a swap where the expected gain outweighs the switching cost.

Let q_t denote the fraction of decided evidence–mechanism pairs in the last W ticks that were perfectly accurate (landing in Q1). Given an ideal rate of $q^* = 1$, we define the rolling mechanism-regret signal s_t as:

$$s_t = \frac{1}{W} \sum_{k=t-W+1}^t \max(0, q^* - q_k)$$

For a set of small tolerances $\epsilon_Q, \epsilon_R, \epsilon_{\text{swap}}, \epsilon_z, \epsilon_U, \epsilon_\varepsilon > 0$, convergence requires the following criteria to be met:

- **Mechanisms mostly validate:** Prediction accuracy is high and rolling regret is bounded.

$$q_t \geq 1 - \epsilon_Q, \quad s_t \leq \epsilon_R$$

- **Exploration and churn have annealed:** The system stops thrashing. Exploration (β_t) and budget (B_t) hit their minimums, and the fraction of active jobs swapped (ρ_t^{swap}) drops below tolerance.

$$\beta_t \approx \beta_{\min}, \quad B_t = B_{\min}, \quad \rho_t^{\text{swap}} \leq \epsilon_{\text{swap}}$$

- **The Pareto reference point has stabilized:** The observed target point stops shifting.

$$\|z_{t+1}^* - z_t^*\| \leq \epsilon_z$$

- **Causal uncertainty is small:** We have high statistical confidence in frequently used mechanisms.

$$U(\alpha, \beta) = 12 \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)} \leq \epsilon_U$$

- **The DRO radius is calibrated:** The Distributionally Robust Optimization bounds stabilize rather than simply shrinking to zero. Empirical coverage κ_t stays within a deadband d of the target κ^* , and the radius ε_t stops fluctuating.

$$|\kappa_t - \kappa^*| \leq d, \quad |\varepsilon_{t+1} - \varepsilon_t| \leq \epsilon_\varepsilon$$

17 Discussion

Koi isn’t just a basic simulator, a pure RL agent, or a standard combinatorial scheduler. It is a hybrid causal planning system. The surrogate estimates what might happen, the causal graph explains the underlying reasons, and the objective function balances utility, exploration, risk, and churn. Meanwhile, the validation loop checks if reality actually matches the causal story, and the slow loop updates future behavior based on real-world evidence. Because of this design, the planner is entirely self-calibrating. Its predictions naturally align with reality using residual evidence, and the objective itself evolves as the system learns. Metrics like exploration pressure, churn tolerance, risk radius, and objective weights all adapt based on what actually succeeds in production. More importantly, because Koi relies on deterministic tools and causal mapping rather than hardcoded heuristics, it is highly adaptive. While we have focused on inference, the exact same architecture scales directly to distributed training workloads. If a new type of accelerator hits the market, or if a new parallelization strategy emerges, the system doesn’t need a rewrite. You simply expose the new knobs as variables in the configuration. The agent will test them, observe the residuals, and automatically map out the performance physics of the new hardware or technique. Ultimately, the goal isn’t to force a model to discover the laws of distributed systems from scratch. It is to build a structured, auditable, and empirically corrected map of cluster physics that can confidently guide high-stakes scaling and scheduling decisions under uncertainty.

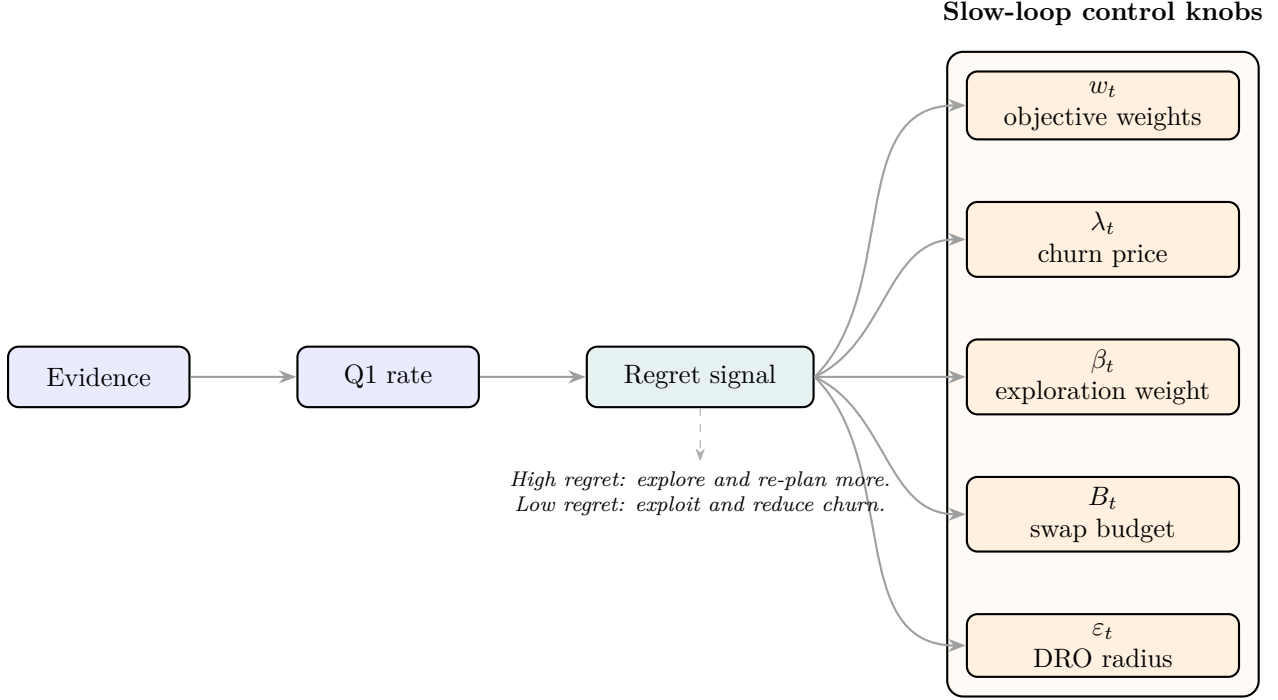


Figure 4: Koi’s slow loop. Evidence determines the Q1 rate, which induces a regret signal used to adapt exploration pressure, churn cost, swap budget, DRO radius, and objective weights.

18 Appendix A

- **Objective weights** w_t — entropic mirror descent along an R^2 Pareto-coverage gradient: $w_j \leftarrow w_j \exp(\eta_w \nabla R^2_j)$, renormalized to the simplex. Sweeping w_t over time sweeps the whole Pareto front (the property Tchebycheff gives us, Section 8).
- **Exploration incentive** β_t — mirror descent against the regret slope: $\beta \leftarrow \beta \exp(\eta_\beta(\text{slope} - \text{target}))$, clamped. Steep regret $\rightarrow$ more exploration; flat $\rightarrow$ exploit. Never zero, so the system stays sensitive to regime shifts.
- **Swap budget** B_t — a step gate: $B_{\max}$ while regret slope exceeds target (re-plan aggressively), $B_{\min}$ once converged (lock in). B_t caps how many *active* jobs may churn per tick.
- **Churn price** λ_{swit} — mirror descent against a target swap rate: if actual churn exceeds target, raise the price of switching.
- **DRO radius** ε — the coverage controller of Section 10.
- **Ideal point** z^* — recomputed each tick from the **performance database**: the best observed value per objective (over similar past deployments, with a small slack) is the reference Tchebycheff measures distance from. So “good” means “close to the best we’ve actually seen,” and it ratchets as we improve.

This is what makes Koi *self-calibrating*: the objective it optimizes is itself tuned by how well it has been doing.

Exploration pressure evolves as

$$\beta_{t+1} = \text{clip}(\beta_t \exp(\eta_\beta(s_t - s^*)), \beta_{\min}, \beta_{\max}).$$

The swap budget is

$$B_t = \begin{cases} B_{\max}, & s_t > s^*, \\ B_{\min}, & s_t \leq s^*. \end{cases}$$

The switch penalty evolves with observed swap rate ρ_t :

$$\lambda_{t+1} = \text{clip}(\lambda_t \exp(\eta_\lambda(\rho_t - \rho^*)), \lambda_{\min}, \lambda_{\max}).$$

The DRO radius evolves by coverage control:

$$\epsilon_{t+1} = \begin{cases} \epsilon_t(1 + \eta_\epsilon), & \kappa_t < \kappa^* - d, \\ \epsilon_t(1 - \eta_\epsilon), & \kappa_t > \kappa^* + d, \\ \epsilon_t, & \text{otherwise.} \end{cases}$$

Objective weights may be updated by entropic mirror descent:

$$w_{j,t+1} = \frac{w_{j,t} \exp(\eta_w \nabla_j R 2_t)}{\sum_k w_{k,t} \exp(\eta_w \nabla_k R 2_t)}.$$

The ideal point used in Tchebycheff scoring is computed from observed performance:

$$z_{j,t}^* = \begin{cases} \max_{r \in H_t} y_{r,j} + \Delta_j, & j \text{ maximized,} \\ \min_{r \in H_t} y_{r,j} - \Delta_j, & j \text{ minimized.} \end{cases}$$

19 Appendix B: Notation and Key Hyperparameters

Symbol	Meaning	Default
$c = \frac{\alpha}{\alpha + \beta}$	Edge / mechanism confidence, using the Beta posterior mean.	seeded
$U(\alpha, \beta)$	Normalized Beta variance used by EIG; equals 1 at Beta(1, 1).	—
J	Augmented Tchebycheff exploit score. Larger is better.	—
ρ_{aug}	Tchebycheff augmentation factor.	1e-3
$\sigma(L')$	Per-candidate score: $J + \beta \text{EIG} - \gamma \text{Pr}_{\text{DRO}} - \lambda \text{SwitchCost}$.	—
γ	SLO-risk penalty weight.	1.0
β_t	Exploration incentive in the slow loop.	$[\beta_{\min}, \beta_{\max}]$
B_t	Swap budget: active jobs allowed to churn per tick.	$B_{\min} = 1, B_{\max} = 10$
λ_{swit}	Churn price in the slow loop.	mirror descent
z^*	Ideal point: per-objective best observed value plus slack.	data-driven / tick
α_{ICP}	ICP significance level.	0.05
n_env_min, n_b	ICP power gate: minimum environments and samples per environment.	3, 15
$\delta_{\text{CUSUM}}, h_{\text{CUSUM}}$	CUSUM slack and threshold, measured as multiples of residual σ .	$0.5\sigma, 4\sigma$
ε_{DRO}	Adaptive Wasserstein radius for DRO bands.	init 0.15; cov. 0.90
$\eta_w, \eta_\lambda, \eta_\beta$	Slow-loop mirror-descent step sizes.	0.10, 0.05, 0.10
W_regret, W_q1	Regret and Q1-rate windows, in ticks.	20, 20
K_P, K_MAX	Plan samples per tick and REPL turns per trajectory.	1, 64
UTILIZATION	Online per-replica utilization cap.	0.8
tick interval	Slow-loop cadence.	~ 300 s

Mechanism confidence bins: CERTAIN (9, 1), STRONG (4, 1), LIKELY (3, 1), PLAUSIBLE (2, 1), EVEN (1, 1).

Edge Beta update ($\Delta\alpha, \Delta\beta$) indexed by $\text{ICP} \times \text{Q}$, and **Mechanism Beta update** indexed by Q : see the update tables in Section 12.

Koi: causal structure for interpretability and targeted learning; Bayesian confidence for calibrated belief and exploration; Tchebycheff for Pareto-complete multi-objective scoring; EIG for deliberate uncertainty reduction; CUSUM + ICP + quadrants to separate skill from luck and learn from every deployment; a slow loop to keep the objective honest; and one LLM agent to reason over all of it — every five minutes, across the whole cluster.

Symbol	Meaning
J_t	set of visible jobs at tick t
R_t	resource map at tick t
P_t	cluster plan
L_i	ladder for job i
x	configuration / decision vector
e	environment
W_i	workload context
S	surrogate predictor
X	decision variables
V	mediators
Y	outcomes
M	mechanism
c_e	edge confidence mean
c_M	mechanism confidence mean
J	Tchebycheff exploitation score
EIG	causal information gain proxy
Pr_{DRO}	robust SLO violation probability
σ_i	candidate ladder score
$\Sigma_t(P)$	cluster plan objective
β_t	exploration weight
B_t	swap budget
λ_t	churn penalty
ε_t	DRO radius
z_t^*	ideal point

20 Appendix C: Tracked Variables

Outcomes - Y

Variable	Unit	Direction	Notes
throughput_tokens_per_sec	tok/s	maximize	system output throughput
p99_ttft_ms	ms	minimize	online TTFT SLO
p99_tpot_ms	ms	minimize	online per-token latency SLO
cost_per_token	\$/tok	minimize	price $\div$ throughput
slo_margin	—	maximize	headroom to TTFT/TPOT targets

Mediators - V

Family	Variables
Cache/sequence	kvcache_hit_rate, input_length_observed output_length_observed
Memory pressure	gpu_mem_used_fraction, kv_cache_util vram_headroom_gb, kv_pressure_score
Parallelism overhead	pipeline_bubble_fraction, comm_overhead_pct per_tok_comm_bytes, pd_inbalance
Work budget	total_token_budget

If any identifier still overflows, change to for this table:

...

Knobs - X

Family	Examples
Parallelism	tp, pp, dp, ep, sp, cp
Engine	engine_name, engine_version, block_size
Batching / scheduler	max_num_seq, max_num_batched_tokens, preemption_policy, router_policy
Memory / KV	gpu_mem_util, prefix_cache_enabled, chunked_prefill_enable, kv_transfer_method
Quantization	weight_dtype, kvcache_dtype, weight_quantization_bits
Disaggregation (deferred)	pd_enabled, prefill_worker_count, decode_worker_count
MoE	is_moe, num_routed_experts, num_active_experts
Environment	market, cloud, region, availability_zone, gpu_type, instance_type, num_nodes_per_chain

Workload Context

Category	Variables
Model architecture	model_params_b, num_hidden_layers, hidden_size num_attn_heads, num_kv_heads, is_moe
Request shape	isl_token_avg, osl_token_avg, request_arrival_rate workload_prefix_concentration, shared_prefix_length_avg is_session_affinity
SLO targets	target_p99_ttft_ms, target_p99_tpot_ms total_token_budget, deadline_hours

21 Appendix D: Lifecycle: Seed, Learn, Promote

- **Seed.** Before any traffic, an LLM converts physics domain knowledge into the edge confidence table and mechanism confidence bins. Confident links start near Beta(9, 1); doubtful ones near Beta(1, 1)–Beta(2, 1).
- **Propose.** At runtime the agent may invent a new mechanism over *existing* edges (it never adds edges). It is admitted only after deterministic validation (every edge exists, topology legal, not a duplicate) and is seeded **neutral** Beta(1, 1) — the proposer does not grade its own theory.
- **Learn.** Every tick, validation verdicts move the Betas (Section 12).

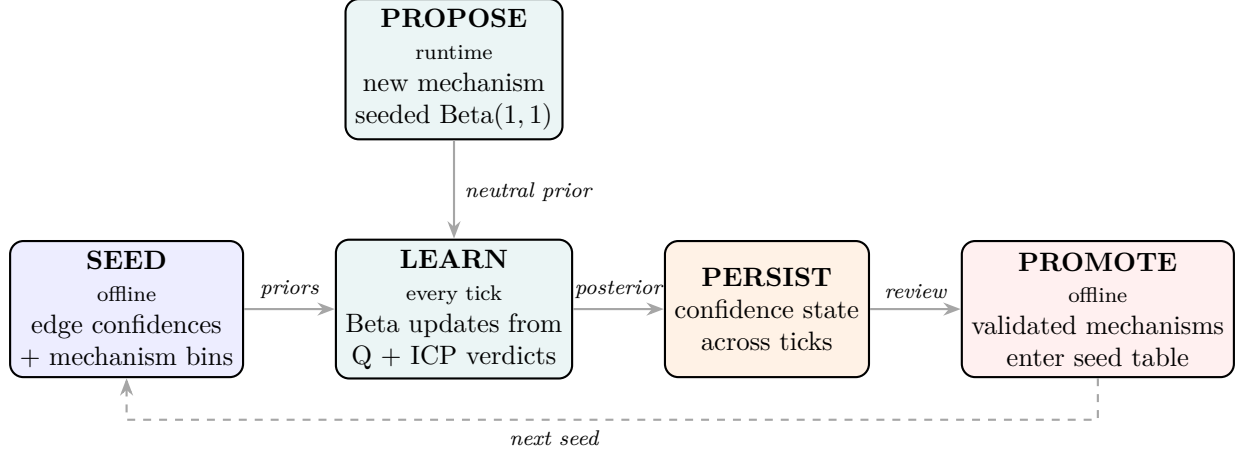


Figure 5: Mechanism lifecycle. Runtime-proposed mechanisms enter with neutral confidence, learn from evidence, persist across ticks, and may later be promoted into the offline seed table.

- **Promote.** Offline, an LLM reviews mechanisms that have accumulated enough evidence and high confidence and writes them back into the seed table for the next cold start.

This is the sense in which Koi is *evolutionary across ticks*: the population of mechanisms and their confidences is selected by reality over time.