THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

AReaL

On the Opportunities of Exploiting LLM Agents for ML System Design and Implementation

Binhang Yuan

Assistant Professor @ CSE, HKUST

Leader of AReaL Open-Source Community

15.07.2026

Brief Bio: Binhang Yuan

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

1 Ph.D.

Rice University — Computer Science

2013 – 2020

Advisor: Prof. Chris Jermaine

2 Postdoc

ETH Zürich — Computer Science

2021 – 2023

With Prof. Ce Zhang

3 Assistant Professor

CSE, HKUST

2023 – present

Relaxed System Lab · AReaL community

Research focus as Assistant Professor

The last 3 years (2023 – 2026)

System for LLM

AReaL · HexGen · HexiScale — building systems for LLMs

The next 3 years (2026 – 2029)

LLM for System

ARGUS · Autopoiesis · AReaL AutoPilot — agents engineering systems

The Path To a Foundation Model

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

Data Preparation

Preparing a massive corpus for pretraining

A multi-stage pipeline:
- Acquisition (e.g. OCR)
- Cleaning;
- Filtering;
- De-duplication;
- Sampling;
- Shuffling.

Pre-training

Training a foundation model by fitting the large corpus by auto-regression

Core infra:
a large-scale parallel training system running over thousands of GPUs

Tuning & Alignment

Tune the model so it can follow user instructions effectively and safely.

Various methods:

- SFT
- RLHF
- RLAIF

Evaluation

Evaluate the model from different aspects:

- Comprehension
- Instruction following
- Expression
- Reasoning
- Coding
- Interaction
- Safety

Deployment

Deploy the model to serve users under different settings:
- Cloud
- Local devices

Require system efforts to optimize the performance, cost, and latency.

Representative works for at each stage:

[Trident: VLDB 2026] [HexiScale: MLSys 2026] [AReaL: NeurIPS 2025, AReaL-DTA: ICML 2026] [AtmosSciBench: NeurIPS 2025] [HexGen: ICML 2024, HexGen-2: ICLR 2025, HexGen-3: ICML 2026]

From Code Assistant to System Engineer

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

> ***Central claim:*** ***LLM agents can act as autonomous system engineers that own design, optimisation, validation, and operation — not merely as code-generation assistants that a human must supervise at every step.***

Conventional: Code assistant

- The human specifies the intent and the design.
- The agent drafts code; the human verifies it.
- Feedback is sparse: the tests pass, or they do not.
- The agent is stateless — nothing accumulates across deployments.
- The human owns the loop, so the loop runs at human speed.

What we want: Autonomous system engineer

- The agent owns design → optimize → validate → operate → adapt.
- The environment supplies dense, structured, verifiable feedback.
- Deterministic machinery — compilers, simulators — guarantees correctness.
- Experience accumulates in a curated, auditable substrate.
- ML system design and implementation could be done by autonomous agents.

Our goal is to: close the engineering loop without a human inside it.

The Infrastructure Bottleneck

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

> ***Observation.*** ***Machine-learning systems are increasingly constrained not by model architecture alone, but by the difficulty of designing, optimising, and continuously adapting the software infrastructure beneath them.***

1 The deployment surface explodes

- Heterogeneous accelerators, shifting workloads, evolving models — each moving on its own schedule.
- The cross-product of hardware × workload × model is not enumerable offline.

2 Hand-engineering cannot keep pace

- Vendor assembly libraries take months of expert effort and trail yearly hardware refreshes.
- Serving policies are tuned per deployment and go stale as conditions drift.

3 Why now

- Frontier coding models make agentic system engineering practical for the first time.
- The missing ingredient is not capability — it is a feedback channel the agent can act on.

The Central Lesson

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

> *Key insight.* ***Autonomous agents become effective system engineers when the environment manufactures dense, structured, verifiable, and timely feedback to guide the optimization.***

ARGUS	**Compile-time data-flow-invariant analysis** Every violation names the offending thread, data element, and program point — a defect report, not a verdict.
Autopoiesis	**Calibrated simulator + per-candidate cost breakdown** Each candidate returns N, Σt_stale, Σt_reconfig, Σt_serve — the agent sees which axis its edit moved.
AReaL-AutoPilot	**Validated deployment plans + runtime telemetry** The plan is checked before launch; execution is monitored and fed back as policy pressure.

Roadmap

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

1 ARGUS

Offline kernel synthesis

Agent-generated GPU kernels that reach hand-tuned assembly performance.

Data-flow invariants · compile-time counterexamples · ICRL planner

GEMM · flash attention · MoE, AMD MI300X

2 Autopoiesis

Online inference adaptation

Serving policies that an agent rewrites while the system is live.

Two-plane architecture · LLM-driven program synthesis · hot-swap

Homogeneous · heterogeneous · elastic cloud clusters

3 AReaL-AutoPilot

Full lifecycle for RL tasks (in progress)

An agentic control layer for distributed RL post-training.

Spec → validated plan → launch → monitor → adapt

Built on AReaL; automatic RL deployment.

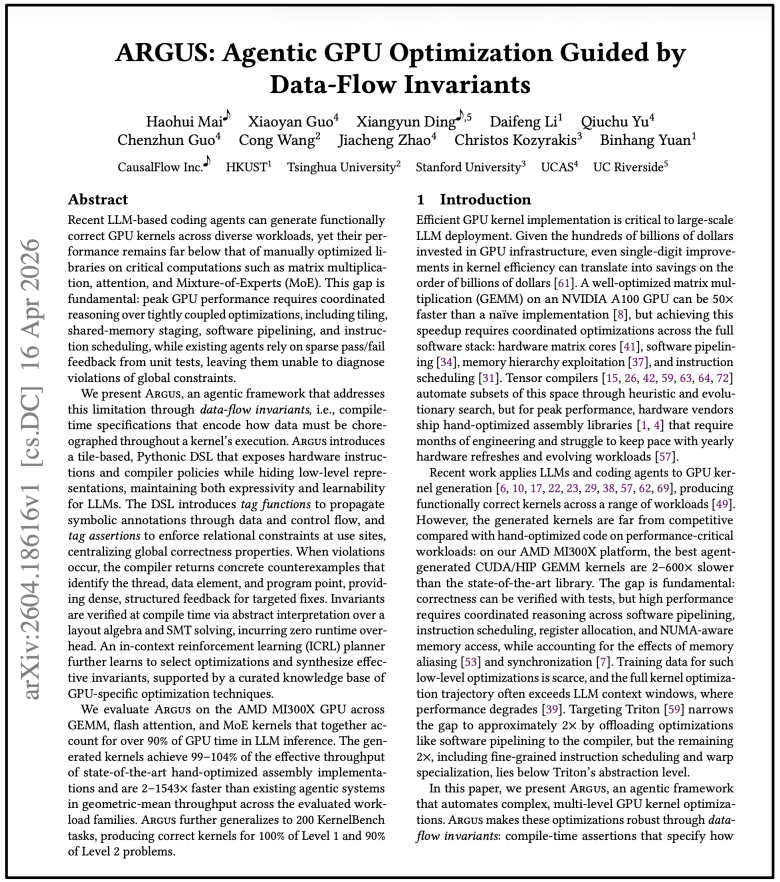
1 ARGUS

Offline kernel synthesis

Agent-generated GPU kernels that reach hand-tuned assembly performance.

Data-flow invariants · compile-time counterexamples · ICRL planner

GEMM · flash attention · MoE, AMD MI300X

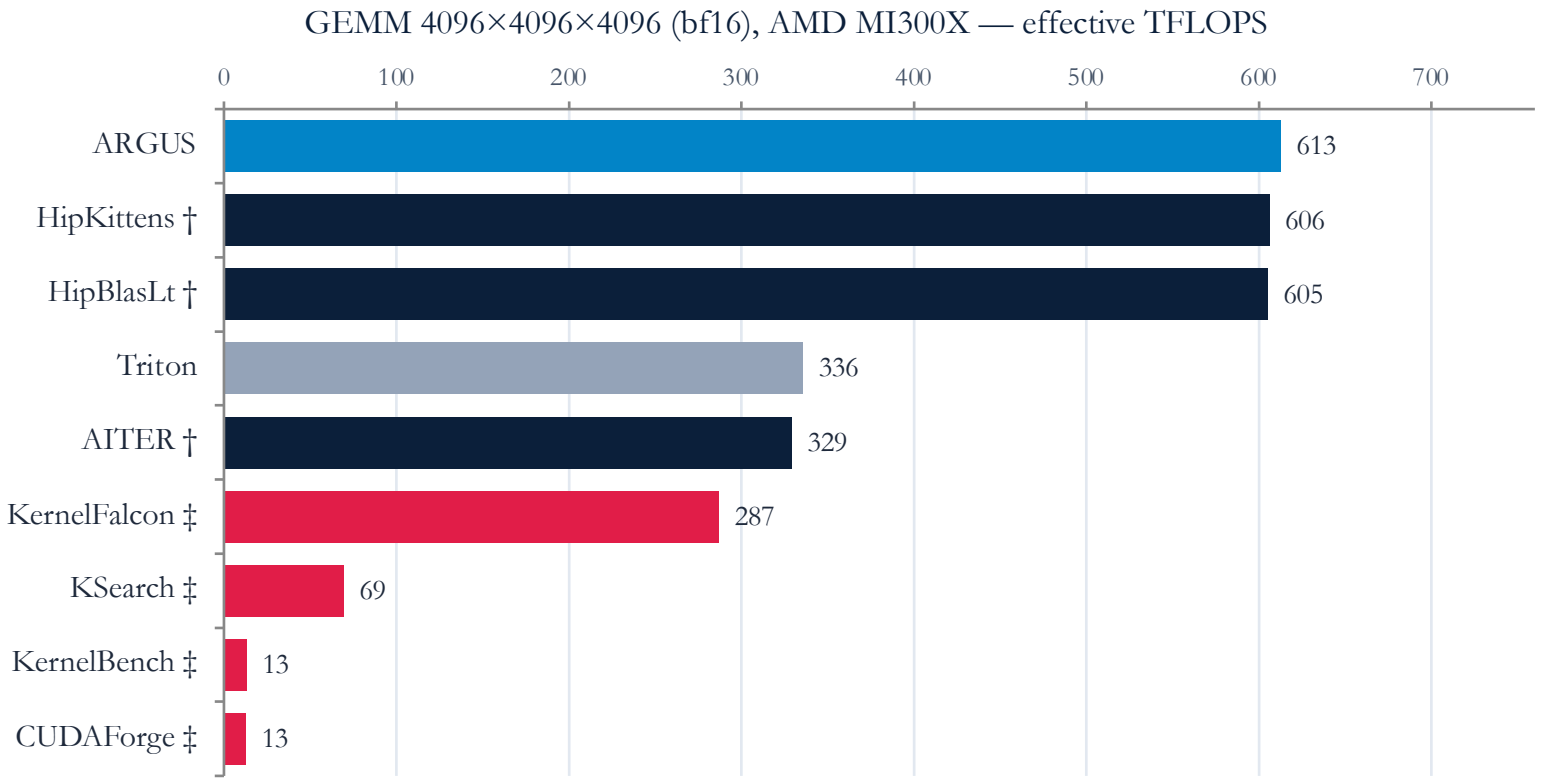


ARGUS: Agentic GPU Optimization Guided by Data-Flow Invariants

arXiv: 2604.18616 (to appear in **SOSP 2026**)

Correct Is Not Fast

> ***The gap.*** ***LLM agents already generate functionally correct kernels. On the workloads that dominate inference, they remain one to three orders of magnitude below hand-tuned libraries.***



† *hand-optimised library* ‡ *agentic kernel-generation framework*

2–600×

Best agent-generated GEMM vs. the state-of-the-art vendor library on MI300X.

> 90%

Share of LLM-inference GPU time spent in GEMM, attention, and Mixture-of-Experts.

≈ 2×

Residual gap even when an agent targets Triton — the rest lies below its abstraction.

Why the Gap Is Fundamental

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
RELAXED SYSTEM LAB

> ***Diagnosis.*** ***Correctness is testable. Peak performance requires coordinated reasoning over tightly coupled optimizations that a pass/fail test cannot reveal.***

What peak performance actually requires

Global intrusive changes

`Software pipelining · Split-K · MFMA matmul · Stagger-K · asynchronous memcpy`

Restructure the whole kernel — hardest for an LLM to produce correctly.

Local source changes

`Bank-conflict mitigation · vectorised loads · loop unrolling · workgroup swizzling`

Small localised patches; the compiler does much of the lifting.

ISA-specific optimisations

`Hardware OOB-guarded loads · AGPR register class · instruction scheduling`

Inline assembly or intrinsics; no agentic baseline implements these.

Why agents cannot get there

- Tests verify correctness but give no signal about global data-flow constraints.
- Training data for low-level GPU optimisation is scarce.
- Full trajectories exceed the context window, where performance degrades — "lost in the middle".
- Aliasing and synchronisation make effects non-local and hard to attribute.
- Triton closes the gap to ≈ 2×; the residual lies below its abstraction.

The consequence. *The agent cannot diagnose a violation of a global constraint, so it patches locally by trial and error — and converges to a local optimum well short of peak.*

Data-Flow Invariants in a Learnable DSL

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

> ***Idea.*** ***Data-flow invariants are compile-time specifications of how data must be choreographed throughout a kernel's execution — a technique adapted from information-flow security.***

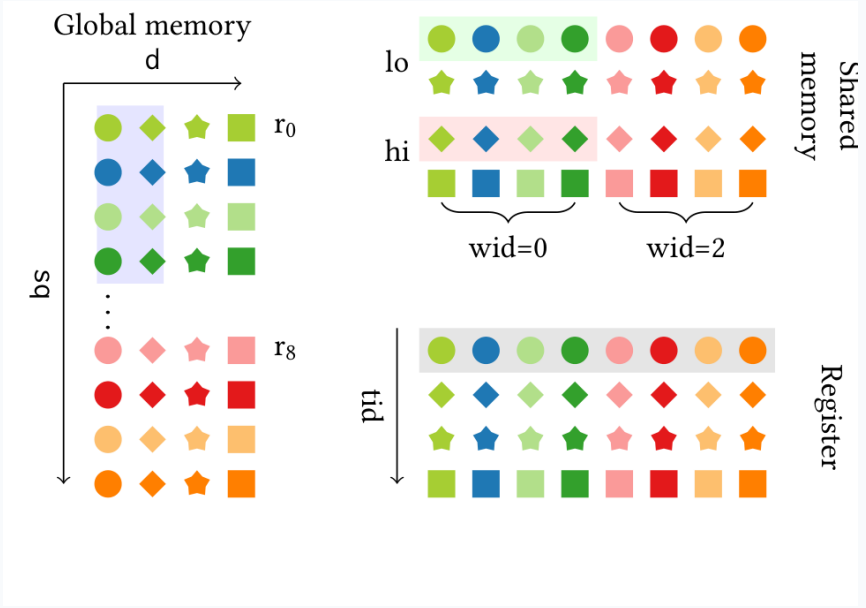
① **Tag functions** — attach symbolic tags (logical coordinates) to tensor elements; tags propagate through data and control flow.

```
# Assign tags to QKV
Γ_Q, Γ_K = Q[sq,h_q,d]→(sq%32,h_q/gqa,d), K[sq,h_kv,d]→(sq%32,h_kv,d)
Γ_V = V[sq,h_kv,d]→(sq,h_kv,d%32)
```

② **Tag assertions** — conformity: paired operands must match. Non-conformity: concurrent producers must not collide.

```
for j in range(16):
  tK,tQ = sK[j/8,wtid%32,j%8,wtid/32], rQ[j%8]
  tK,tQ = tK.view((8,), bf16), tQ.view((8,), bf16)
  assert tag(tK[...]) == tag(tQ[...]) # tQ and tK should match
```

The DSL. *Tile-based and Pythonic, syntactically close to CuTe, TileLang, and Triton, so the model can generalise from its training data. It exposes hardware instructions and compiler policy while hiding mechanism and intermediate representations — expressive enough for peak performance, concise enough to stay learnable.*



Tags for V propagating across memory levels: global → shared → register. Each colour–shape pair is one tag; the assertion at the MFMA use site checks the pairing survived.

[ARGUS, Figure 1]

From Verification to Teaching Signal

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
RELAXED SYSTEM LAB

> ***Key claim.*** ***Verification is not the product. The counterexample is: every failed assertion becomes a dense, structured reward signal that tells the agent exactly what to fix.***

How the analysis stays tractable

- At control-flow joins, tags merge over the lattice $\bot < t < \top$. Constants carry $\bot$; two writes with different tags to one shared-memory location yield $\top$.
- Shared-memory tags can be reset to $\bot$, so pipeline stages may safely reuse a buffer without losing tracking.
- Tracking granularity is one byte — sufficient, since mxfp4/nvfp4 and sub-byte matrix instructions are byte-aligned.
- Path-insensitive merging avoids enumerating branch conditions: validation, not proof — precision traded for a loop that closes automatically.

Zero runtime overhead

Tags are purely compile-time constructs. Nothing is materialised, so a fully optimised kernel is not perturbed at all.

Why not full verification?

Proof engineering multiplies effort 2–6× and does not generalise across optimisations. Lightweight analysis keeps the agentic loop automatic.

The payoff. *The agent reasons about correctness at the level of invariants rather than patching implementations by trial and error.*

The Agentic Optimisation Harness

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

Whole > parts. ***Tag functions specify; static analysis verifies; counterexamples reward. Alone, none suffices — together they close the loop.***

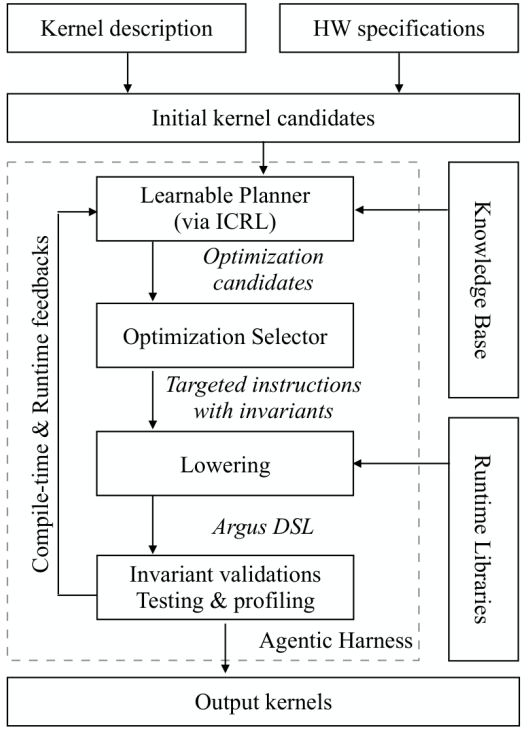
1. **Persistent knowledge base** Expert-curated. Each entry records a DSL transformation pattern and the invariants that must hold after the rewrite.

2. **Learnable planner (ICRL)** Binds abstract optimisations to the concrete kernel; emits ⟨optimization, context, score⟩ and specialises invariant templates into concrete tags and assertions.

3. **Optimization selector** Samples from the proposal distribution rather than taking the top rank — preserving exploration in a coupled space.

4. **Lowering agent** Implements the plan in the DSL at the level of tiles, layouts, and thread control; inserts the required tag assertions.

5. **Validator agent** Invariants → unit tests → runtime profile. Only candidates passing both static and dynamic checks are profiled.



[ARGUS, Figure 1 — agentic harness]

ICRL. ***The planner prompt is the mutable policy θ, updated by text gradients. Reward = runtime performance + process rewards from invariant violations. The knowledge base stays fixed and auditable; θ learns to bind it.***

ARGUS: Results

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

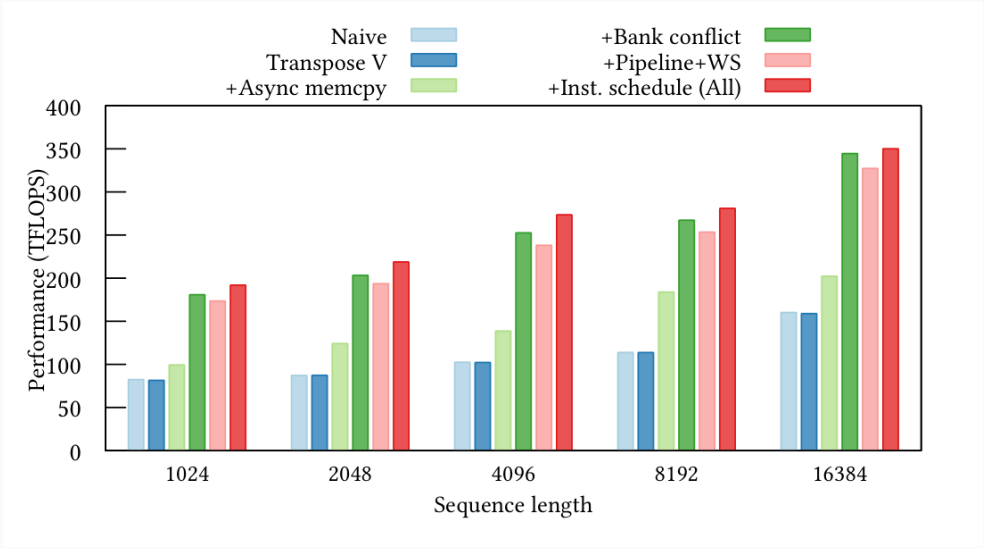
99–104%
of hand-tuned assembly throughput on GEMM, flash attention, and MoE.

2–1543×
geometric-mean gain over agentic baselines across workload families.

2.4×
over KernelFalcon, the strongest agentic baseline, on flash attention.

100 / 90%
of KernelBench Level-1 / Level-2 tasks correct — it generalises.

Ablation: cumulative optimisations on flash attention



[ARGUS, Figure 2]

Do the invariants earn their keep?

60 KernelBench GEMM / convolution problems; the same planner with and without invariants.

	Pass@1	Avg. tokens
Level 1	**60% → 75%**	18.1M → 15.0M
Level 2	**40% → 57%**	14.2M → 13.5M

More effective and cheaper at once: invariants raise the success rate while cutting token spend.

Argus is an agentic framework to implement every optimization class required to match expert-tuned production kernels.

Takeaway. Structured compile-time feedback is what closes the distance between an LLM agent and a human expert.

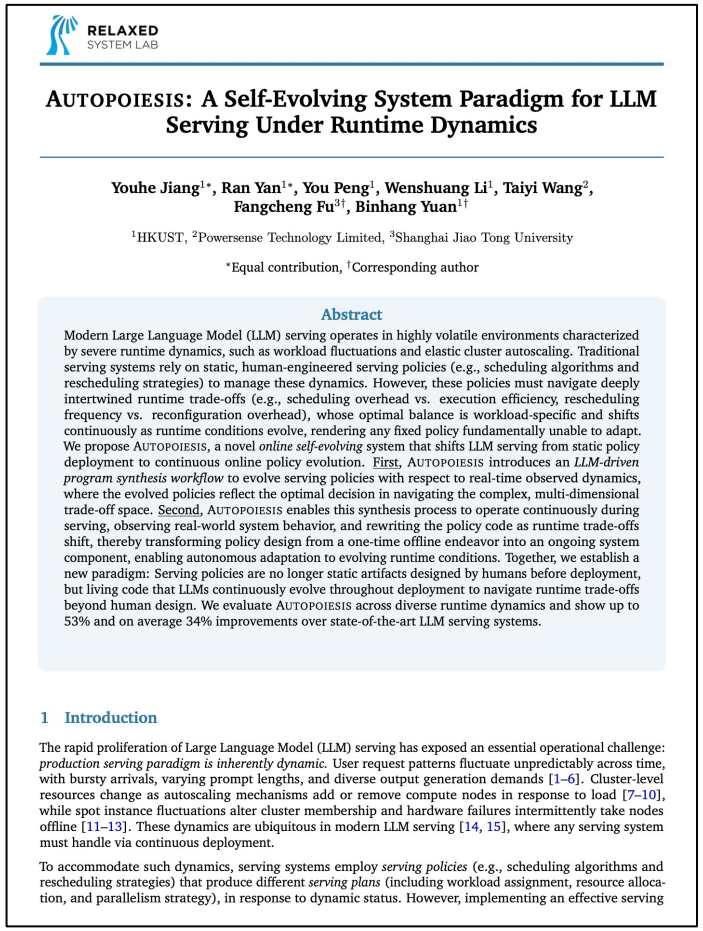
2 Autopoiesis

Online inference adaptation

Serving policies that an agent rewrites while the system is live.

Two-plane architecture · LLM-driven program synthesis · hot-swap

Homogeneous · heterogeneous · elastic cloud clusters



RELAXED SYSTEM LAB

AUTOPOIESIS: A Self-Evolving System Paradigm for LLM Serving Under Runtime Dynamics

Youhe Jiang[1]*, Ran Yan[1]*, You Peng[1], Wenshuang Li[1], Taiyi Wang[2], Fangcheng Fu[3]†, Binhang Yuan[1]†

[1]HKUST, [2]Powersense Technology Limited, [3]Shanghai Jiao Tong University

*Equal contribution, †Corresponding author

Abstract

Modern Large Language Model (LLM) serving operates in highly volatile environments characterized by severe runtime dynamics, such as workload fluctuations and elastic cluster autoscaling. Traditional serving systems rely on static, human-engineered serving policies (e.g., scheduling algorithms and rescheduling strategies) to manage these dynamics. However, these policies must navigate deeply intertwined runtime trade-offs (e.g., scheduling overhead vs. execution efficiency, rescheduling frequency vs. reconfiguration overhead), whose optimal balance is workload-specific and shifts continuously as runtime conditions evolve, rendering any fixed policy fundamentally unable to adapt. We propose AUTOPOIESIS, a novel *online self-evolving* system that shifts LLM serving from static policy deployment to continuous online policy evolution. First, AUTOPOIESIS introduces an *LLM-driven program synthesis workflow* to evolve serving policies with respect to real-time observed dynamics, where the evolved policies reflect the optimal decision in navigating the complex, multi-dimensional trade-off space. Second, AUTOPOIESIS enables this synthesis process to operate continuously during serving, observing real-world system behavior, and rewriting the policy code as runtime trade-offs shift, thereby transforming policy design from a one-time offline endeavor into an ongoing system component, enabling autonomous adaptation to evolving runtime conditions. Together, we establish a new paradigm: Serving policies are no longer static artifacts designed by humans before deployment, but living code that LLMs continuously evolve throughout deployment to navigate runtime trade-offs beyond human design. We evaluate AUTOPOIESIS across diverse runtime dynamics and show up to 53% and on average 34% improvements over state-of-the-art LLM serving systems.

1 Introduction

The rapid proliferation of Large Language Model (LLM) serving has exposed an essential operational challenge: *production serving paradigm is inherently dynamic.* User request patterns fluctuate unpredictably across time, with bursty arrivals, varying prompt lengths, and diverse output generation demands [1–6]. Cluster-level resources change as autoscaling mechanisms add or remove compute nodes in response to load [7–10], while spot instance fluctuations alter cluster membership and hardware failures intermittently take nodes offline [11–13]. These dynamics are ubiquitous in modern LLM serving [14, 15], where any serving system must handle via continuous deployment.

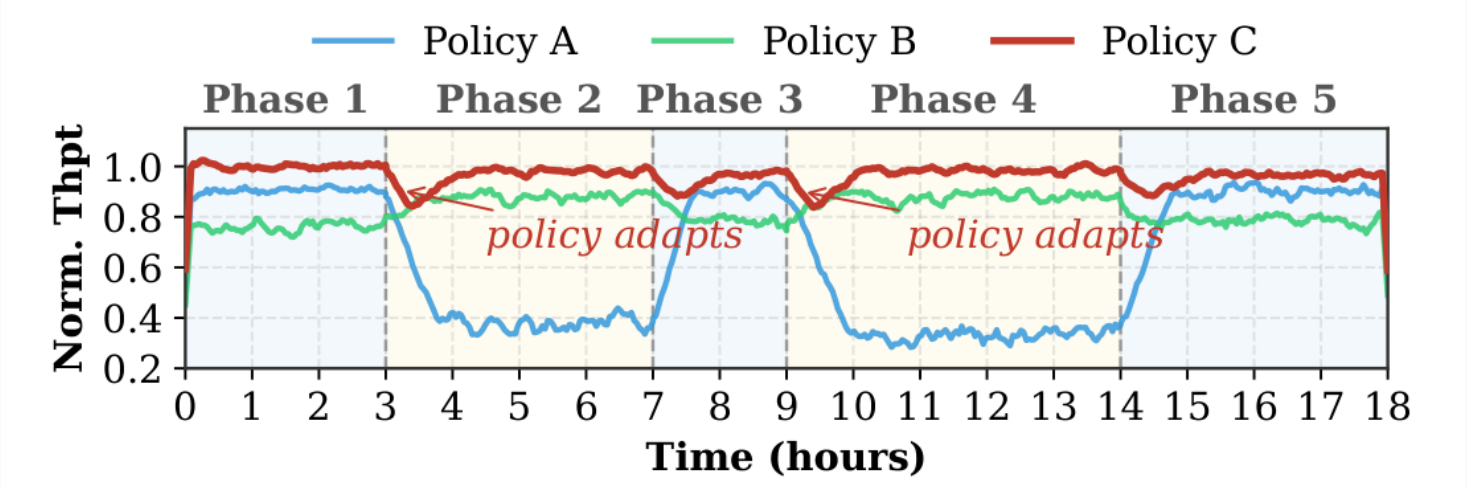
To accommodate such dynamics, serving systems employ *serving policies* (e.g., scheduling algorithms and rescheduling strategies) that produce different *serving plans* (including workload assignment, resource allocation, and parallelism strategy), in response to dynamic status. However, implementing an effective serving

Autopoiesis: A Self-Evolving System Paradigm for LLM Serving Under Runtime Dynamics

arXiv: 2604.07144,

When Static Policies Break

The problem. ***Serving systems react to runtime dynamics by swapping plans — but the policy that produces those plans is static and human-engineered, so it cannot follow the trade-offs as they shift.***



Five phases: steady (1, 3, 5) alternating with bursty (2, 4). [Autopoiesis, Figure 1]

Policy A

Thorough scheduling, aggressive reconfiguration. Excels under stability; collapses under bursts.

Policy B

Reactive scheduling, lightweight reconfiguration. Thrives during bursts; underperforms in steady state.

Policy C

Continuously adapts to shifting conditions. Holds high throughput across every phase.

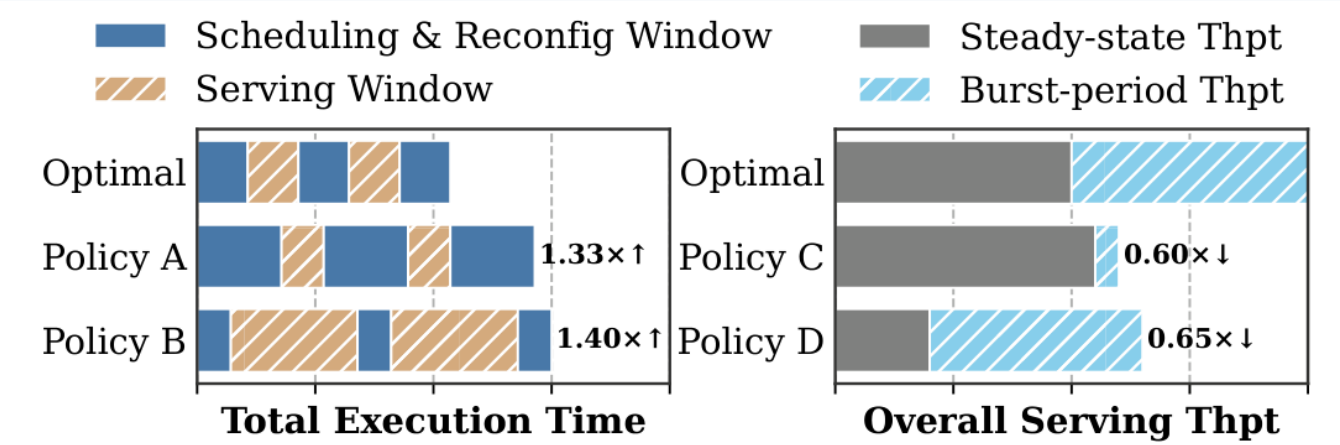
No fixed policy wins across regimes — the optimum is workload-specific and moves.

- **Human intuition cannot locate balances in a space this intertwined.**
- Offline precomputation cannot rescue it either — the space is not enumerable in advance.

An Intertwined Trade-Off Space

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
RELAXED SYSTEM LAB

> ***Root cause.*** ***Serving never pauses. Under this continuous-execution constraint every scheduling decision and every reconfiguration action has a direct, concurrent cost — which couples the trade-offs together.***



Left: neither extreme is competitive (1.33× / 1.40× worse than the oracle). Right: a policy tuned for one regime loses 35–40% in the other. [Figure 2]

Two requirements for an effective serving paradigm

- Automated trade-off exploration — discover policies tailored to the current workload and cluster, rather than balancing the axes by intuition.
- Continuous online evolution — observe real system behaviour and rewrite the policy code as the trade-offs shift, throughout deployment.

Rescheduling frequency vs. per-interval overhead

Frequent rescheduling (large N) keeps plans fresh but accumulates cost; infrequent rescheduling serves under stale plans.

Scheduling thoroughness vs. stale serving cost

Thorough scheduling lowers t_serve but extends the stale-plan window t_stale; lightweight scheduling inverts the pair.

Reconfiguration aggressiveness vs. transition overhead

Aggressive reconfiguration reaches the optimal plan but pays prolonged degradation proportional to the change.

Trade-offs (i)–(iii) are tightly coupled: the optimum is nonlinear, deployment-specific, and moves continuously.

Serving Policy as Living Code

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
RELAXED SYSTEM LAB

> ***Reframing.*** *A serving policy is not a static artefact designed before deployment. It is living code that an LLM continuously evolves throughout deployment.*

The policy is a pair of pure functions

`should_reschedule(ctx) → {0, 1}`

have conditions shifted enough to justify rescheduling? ⟶ controls N

`schedule(ctx) → plan`

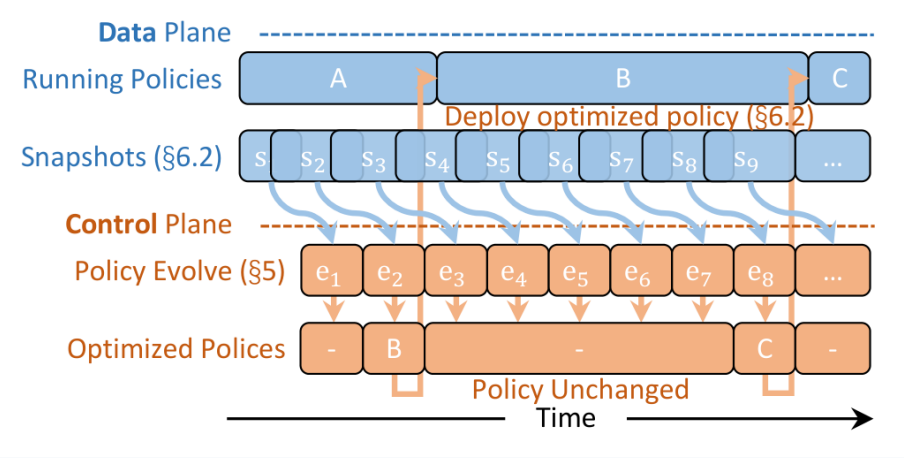
produce the new serving plan ⟶ controls scheduling cost and reconfiguration magnitude

The execution model these functions drive

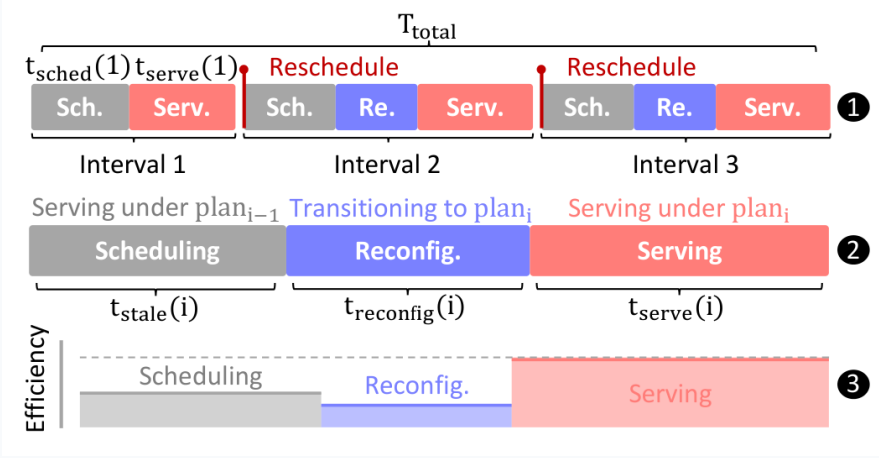
`T_total = t_sched(1) + t_serve(1) + Σ i=2..N [ t_stale(i) + t_reconfig(i) + t_serve(i) ]`

N is not a constant — it is whatever should_reschedule decides. Each cost term reflects the behaviour of schedule at that interval. The synthesis objective is to evolve both implementations to jointly minimise T_total.

Two-plane architecture



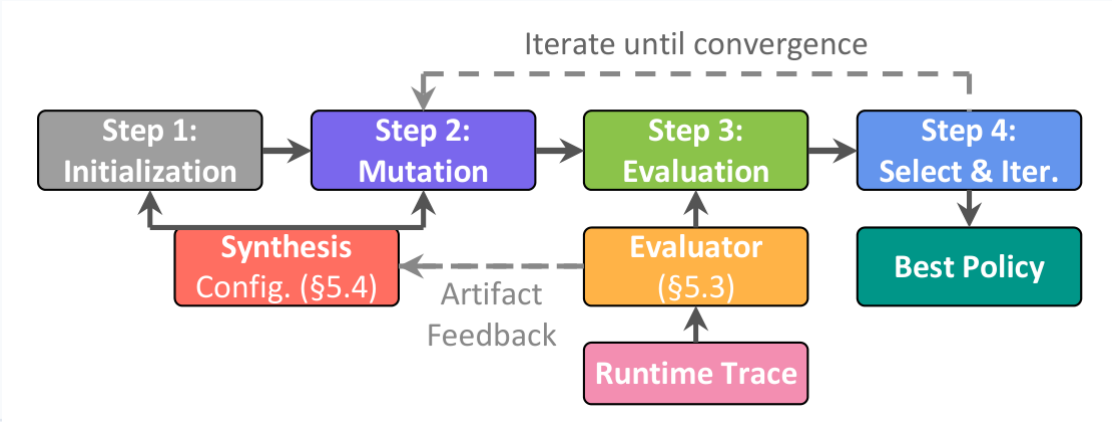
Execution model: three phases per interval



LLM-Driven Program Synthesis

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
RELAXED SYSTEM LAB

> ***The reward signal.*** ***What compile-time counterexamples are to ARGUS, a calibrated simulator and a structured cost breakdown are to Autopoiesis — the same dense-feedback move, one layer up.***



Artifact feedback: the cost breakdown

	N	Σ stale	Σ reconf	Σ serve	T_total
Parent	2	7.6 s	12.0 s	25.2 s	**44.8 s**
Child A	5	5.5 s	4.0 s	24.0 s	**33.5 s**
Child B	1	10.0 s	17.0 s	23.0 s	**50.0 s**

The agent sees which axis its own edit moved: Child A reschedules more often yet pays less — lightweight, frequent rescheduling suits this workload.

The evaluator is the bridge to reality

- Candidates are replayed against the actual runtime trace snapshotted from the data plane — not a synthetic benchmark.
- A roofline-based simulator models per-operator time, continuous batching, parallelism, and reconfiguration overhead.
- Selection uses MAP-Elites with island-based population management, preserving diversity while retaining elites.

Trade-off-aware mutation

- Formal structure — the system prompt encodes the execution model and the trade-off characterisations, so the LLM knows how a move on one axis shifts the others.
- Per-iteration feedback — the parent/child cost breakdown plus population context, steering away from redundant mutations.
- **Result: trade-off-aware modifications, not random variations.**

Making Evolution Truly Online

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
RELAXED SYSTEM LAB

> ***Engineering.*** ***Offline synthesis is a research artefact; online synthesis is a systems problem. Four mechanisms make the control plane fast enough and safe enough to run beside live traffic.***

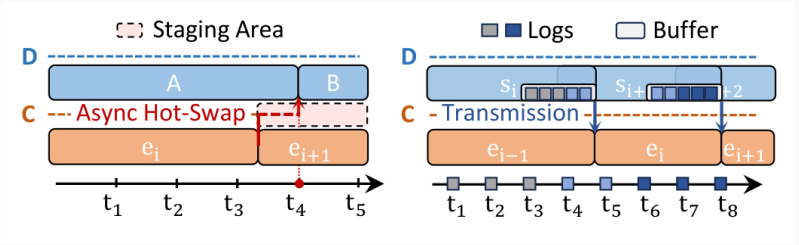
1. **Warm-start re-evolution** Consecutive snapshots reflect gradual shifts, so cycle e·i is seeded with the elites of e·i−1 and their mutations — targeted refinement instead of re-exploring from scratch.

2. **Multi-level timeout hierarchy** An evolution-level timeout bounds each cycle so the control plane delivers on a predictable latency; a candidate-level timeout kills any schedule function that diverges.

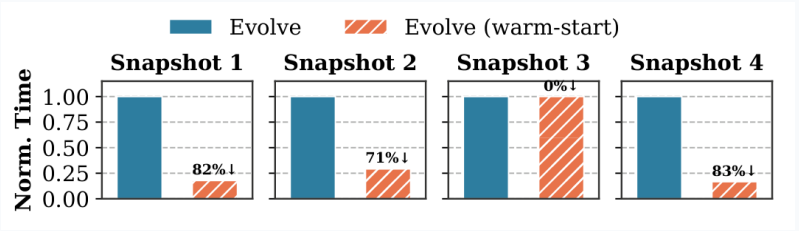
3. **Asynchronous policy hot-swap** Because the policy is two pure functions, deployment is a code replacement. The control plane writes to a staging area; the data plane loads it at the next monitoring step. No in-flight plan is disrupted.

4. **Sliding-window snapshotting** Conditions are logged to a fixed-size circular buffer; a background thread transmits a contiguous window on request, decoupled from live request processing. Windows may overlap, exposing trends across cycles.

Convergence in under 10 minutes (≈ 80 iterations); evaluation is embarrassingly parallel — up to 8× with 10 threads.

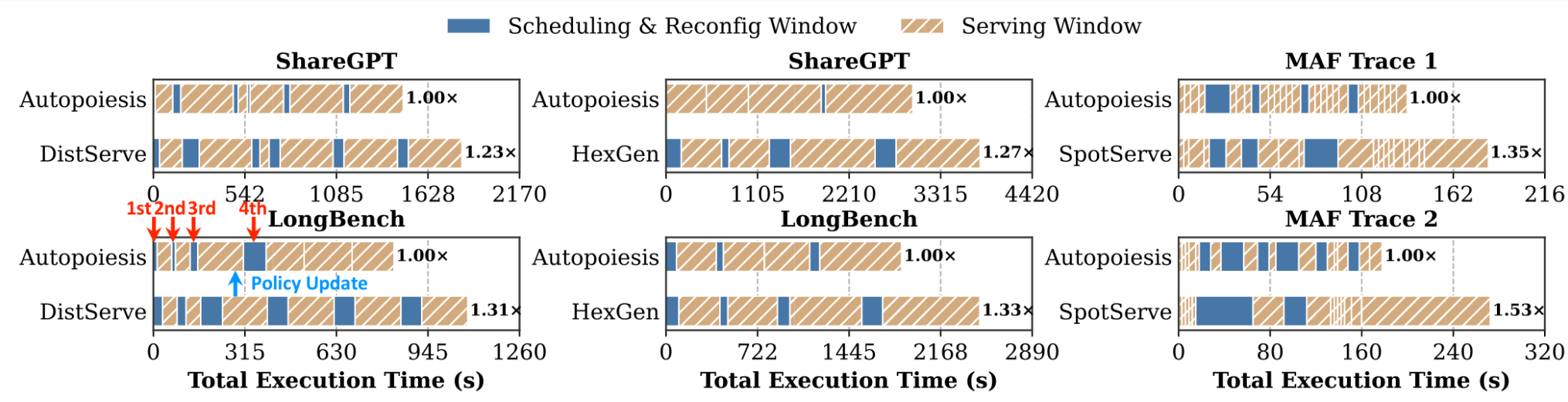


Hot-swap (left) and trace snapshotting (right). [Autopoiesis, Figure 6]



Warm-start cuts evolution time by up to 83% — except when conditions shift so far that no prior structure is reusable (Snapshot 3). [Figure 9]

Autopoiesis: Results



End-to-end execution time vs. DistServe (homogeneous), HexGen (heterogeneous), SpotServe (elastic cloud). [Autopoiesis, Figure 7]

53% / 34%
Peak / average reduction in execution time over state-of-the-art systems.

27%
Average gain over DistServe on a homogeneous cluster (up to 31%).

30%
Average gain over HexGen under hardware heterogeneity (up to 33%).

44%
Average gain over SpotServe on an elastic cloud cluster (up to 53%).

The evolved policies are workload-adaptive

- Under volatility: cuts scheduling and reconfiguration overhead by 65%.
- Under stability: invests 10% more in search — cutting serving latency 22%.
- On heterogeneous clusters, search time drops by up to 92%.

Two insights from the case studies

- Frontier-shifting, not frontier-navigating. Greedy and ILP occupy fixed points on a Pareto frontier; evolved policies improve both axes at once.
- Scheduling algorithm and rescheduling strategy must co-evolve — their cost profiles are coupled.

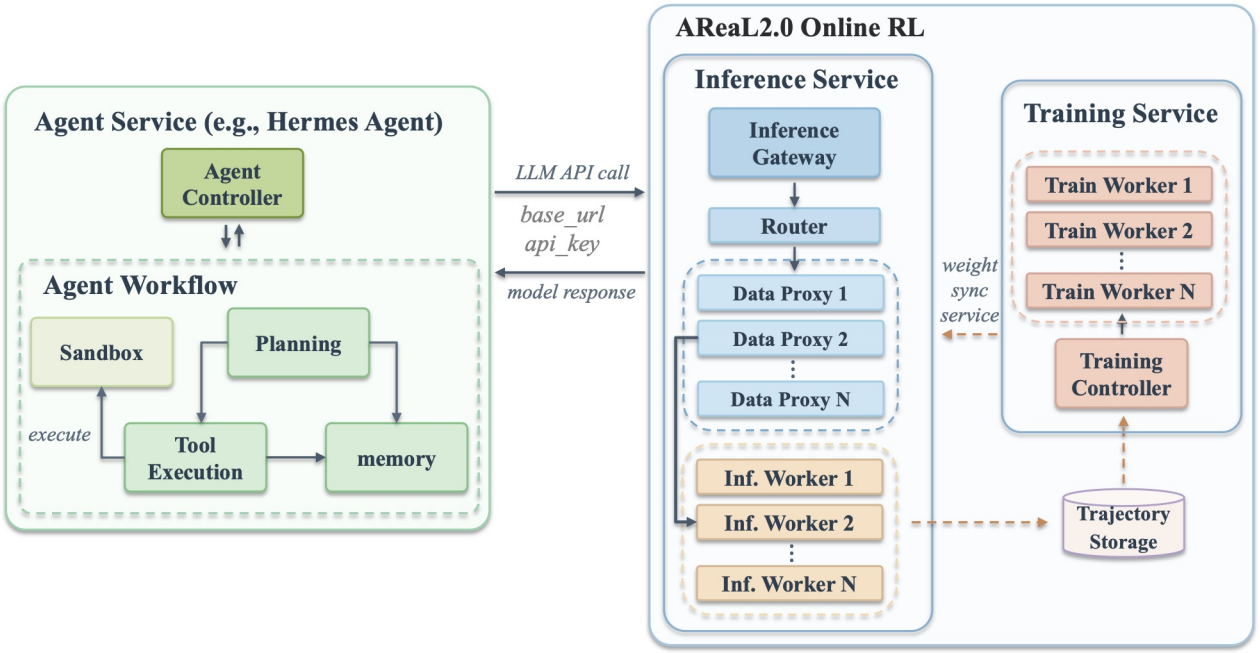
3 AReaL-AutoPilot

Full lifecycle RL tasks (in progress)

An agentic control layer for distributed RL post-training.

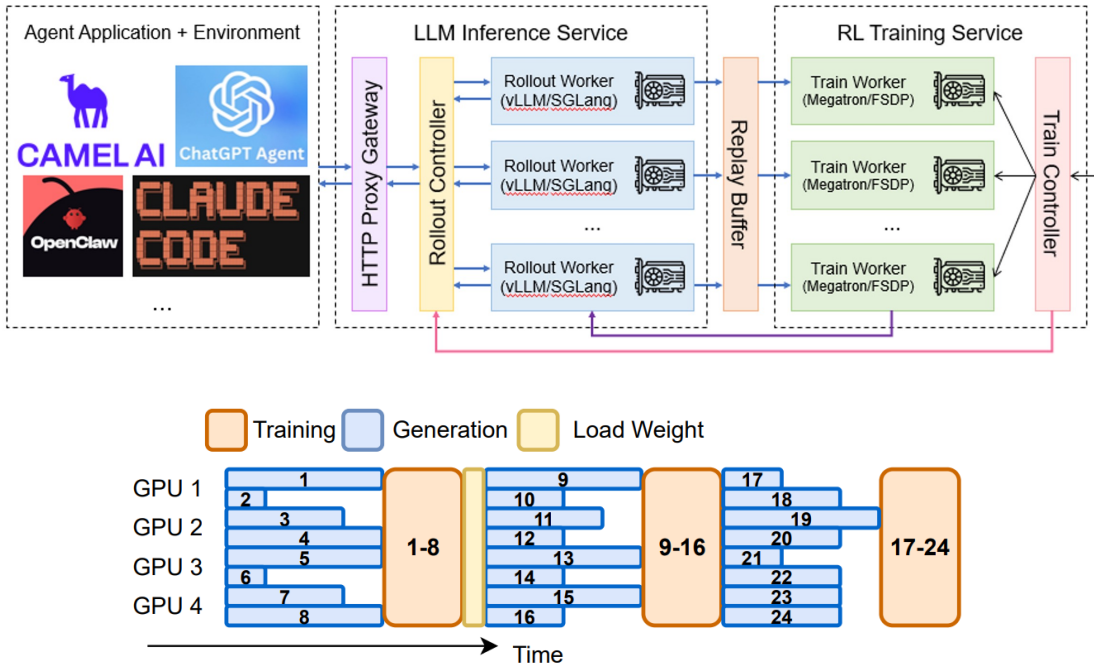
Spec → validated plan → launch → monitor → adapt

Built on AReaL; automatic RL deployment.



AReaL 2.0 online-RL architecture — the execution substrate that AutoPilot configures, launches, and adapts.

AReaL: Introduction



What is AReaL?

A scalable agentic reinforcement-learning (RL) infrastructure bridging foundation-model training with modern agent-based applications.

Jointly open-sourced by Ant Group and Tsinghua University · February 2025

- **Flexibility** — modular architecture with plug-and-play "Agentic RL-as-a-Service" integration.
- **Scalability** — stable, fully asynchronous RL training at large scale.
- **Performance** — state-of-the-art results on math, code, search, and agentic tasks.
- **Research-friendly** — four major RL paradigms and 10+ mainstream RL algorithms.
- **PyTorch Ecosystem project** — accepted May 2026; now applying for Foundation hosting.

AReaL: Community & Governance

5.2K+	500+	100+	1,900+	4 → 14
GitHub stars	Forks	Code contributors	Community users	Maintainers since launch

Vendor-Neutral Governance

- **Maintainer-led model,** publicly documented in GOVERNANCE.md.
- **14 maintainers across 6 organizations** — Tsinghua, Ant Group, Huawei, ByteDance, Xiaomi, and Tencent.
- **Two maintainer approvals per pull request,** including a code owner for modified paths.
- **Consensus decision-making,** with a lead-maintainer tie-break when consensus cannot be reached.
- **Gold OpenSSF Best Practices badge** (bestpractices.dev/projects/12770).
- **Project assets prepared** for transfer to the Linux Foundation.

Contributor & Maintainer Organizations

Maintainers: Tsinghua · Ant Group · Huawei · ByteDance · Tencent · Xiaomi

Contributors incl. Microsoft, Boson AI, PKU, CUHK, HKUST, UIUC

AReaL: Release History & Cadence

AReaL THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY RELAXED SYSTEM LAB

Every release ships three artifacts:

1. **Versioned code release**
   Tagged, installable, and reproducible.
2. **Reproducible training examples**
   End-to-end recipes with configs and scripts.
3. **Research findings & papers**
   Empirical results published with each milestone.

Online RL

- **Jun 2026** — **v2.0** Online-RL modular architecture
- **May 2026** — **PyTorch Ecosystem project** — accepted

Agentic RL

- **Mar 2026** — **v1.0** Agentic architecture — general availability
- **Dec 2025** — **v0.5** Agentic architecture preview

Reasoning RL

- **Aug 2025** — **v0.4** AReaL-lite architecture
- **Jun 2025** — **v0.3** Fully asynchronous RL — 2.7× throughput
- **Mar 2025** — **v0.2** State-of-the-art 7B math model
- **Feb 2025** — **v0.1** Initial release with GRPO

18 months from v0.1 to v2.0 — three architecture generations: Reasoning → Agentic → Online RL.

AReaL-AutoPilot: The Full Lifecycle

> ***Goal.*** ***An agentic control layer that turns an RL workload specification into a validated deployment plan, launches the distributed system, monitors it, and progressively adapts its policies.***

1 Workload spec

Model, algorithm, dataset, resource envelope, objective.

2 Validated plan

Parallelism, placement, batching, rollout/training ratio — checked before launch.

3 Configure & launch

Instantiate the distributed RL system on the cluster.

4 Monitor

Throughput, staleness, memory, stragglers, convergence.

5 Adapt

Rewrite system policies as models, workloads, and hardware change.

progressive adaptation — the plan is never final, only current

Inherits from ARGUS · validate before you act

- A deployment plan is checked against system invariants — memory feasibility, parallelism legality, communication topology — before a single GPU is allocated.
- A rejected plan returns a counterexample, not a crash log: the agent learns which constraint it violated.

Inherits from Autopoiesis · adapt while you run

- RL post-training is itself non-stationary: rollout length drifts as the policy improves, so the optimal system configuration drifts with it.
- The control plane observes execution and rewrites system policies mid-run — the same two-plane separation, applied to training.

Conclusion

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RELAXED SYSTEM LAB

> ***The arc.*** ***From code-generation assistant to autonomous system engineer — offline (ARGUS), online (Autopoiesis), and across the full lifecycle (AReaL-AutoPilot).***

ARGUS

Data-flow invariants turn a compiler into a teaching signal — reaching 99–104% of hand-tuned assembly on GEMM, attention, and MoE.

Autopoiesis

Serving policies become living code, evolved online — 34% average and up to 53% improvement over state-of-the-art systems.

AReaL-AutoPilot

Both halves, applied to distributed RL post-training: validate before launch, adapt while running.

AReaL Repo:
https://github.com/areal-project/AReaL

Personal page:
https://binhangyuan.github.io/site/

Thank you!