Agentic Architect: An Agentic AI Framework for Architecture Co-Design

Alexander Blasberg

April 2026

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

COMMITTEE

Dimitrios Skarlatos (Advisor)

Submitted in partial fulfillment of the requirements
for the Senior Honors Thesis

Keywords: computer architecture, microarchitecture, CPU design optimization, large language models, LLM, agentic AI, evolutionary search, co-design, cache replacement, data prefetching, branch prediction, ChampSim

Abstract

Rapid advances in Large Language Models (LLMs) are creating new opportunities for research communities by enabling efficient exploration of broad and combinatorial design spaces. This is particularly valuable in Computer Architecture, where performance depends on microarchitectural policies that operate in vast design spaces traditionally explored through manual human effort. In this thesis, I introduce Agentic Architect, an end-to-end framework for agentic co-design of CPU microarchitectural policies that couples LLM-driven code evolution with cycle-accurate simulation. In this approach, the role of the architect remains central: the human specifies the seed policy, scoring function, training workloads, and prompt strategy, while the LLM explores candidate implementations within these constraints through hundreds of mutate-evaluate-select cycles. I instantiate the framework on three foundational microarchitectural domains that span the full spectrum of optimization headroom: cache replacement, hardware prefetching, and branch prediction. Across all three, the evolved policies match or exceed published state-of-the-art designs. The best evolved cache replacement policy achieves a 1.062× geomean IPC speedup over LRU across 11 SPEC CPU traces, an additional 0.6% over Mockingjay (1.056×). The evolved prefetcher achieves a 1.76× geomean IPC speedup over no prefetching, an additional 17% over its VA/AMPM Lite seed (1.59×) and an additional 21% over the strongest evaluated published baseline (SMS, 1.55×). In the most constrained domain, the evolved branch predictor achieves a 1.100× geomean IPC speedup over Bimodal, an additional 1.5% over its Hashed Perceptron seed (1.085×), driven by a 39% reduction in mispredictions on the most prediction-sensitive trace.

These gains do not imply that LLMs can design microarchitecture autonomously. Each of the four architect inputs (seed, scoring function, trace selection, and prompt strategy) materially shapes the quality of the evolved policy, and the choice of evolutionary framework or underlying LLM matters less than any of them. Across all three domains the evolved policies also share a structural regularity I call the *learned ensemble pattern*, characterized in Section 5, which explains both why the framework reliably produces improvements and why those improvements are bounded by the strength of the seed. The greatest current value of LLMs in computer architecture therefore lies in agentic co-design, where the architect defines the search space and the LLM accelerates exploration within it. Agentic Architect is the first end-to-end framework for that form of co-design.

Acknowledgements

This thesis would not exist without the people who supported me through it.

To my advisor, Dimitrios: thank you for your guidance, your feedback, and your willingness to take a chance on me. Your mentorship shaped both the thesis and how I will forever think about research.

To Vasilis: thank you for being a constant collaborator and for always finding some way that my plots could be better.

To Sena: thank you for your patience and steady encouragement through every late night and every "just one more experiment". I could not have finished this without you.

And to my Mom and Dad: thank you for raising me to keep going when things got hard, and to keep getting better when they didn't. None of this would have happened without you.

Contents

1 Introduction

With the slowdown of Moore's Law and the rising cost of larger on-chip structures, microarchitectural policy design has emerged as a key bottleneck in processor development. Microarchitectural policies, the embedded mechanisms that decide which cache lines to evict, which addresses to prefetch, which branches to predict, and many similar choices, operate in vast combinatorial design spaces. The traditional workflow, an architect designs a policy, implements it, simulates it, inspects the results, and iterates, is too slow to cover such a space manually; substantial performance is routinely left on the table simply because no human has the bandwidth to find it.

Recent advances in agentic AI systems, in which Large Language Models (LLMs) are paired with evolutionary search and automated evaluation [25, 28, 32, 7, 3, 20, 21, 38], have demonstrated AI-driven discovery of novel algorithms across scientific and engineering domains. Microarchitecture is a particularly suitable target: policies are naturally represented as modular code, cycle-accurate simulators provide consistent quantitative evaluation, and the design space is large but well-structured. The central question of this thesis is whether LLMs, embedded in an evolutionary loop and grounded in cycle-accurate simulation, can generate microarchitectural policies competitive with the state of the art, and what role the human architect plays in the process.

The thesis answers the first part in the affirmative and uses the sensitivity of the answer to the architect's choices to resolve the second. I instantiate Agentic Architect on three microarchitectural domains chosen to span the full spectrum of search-space breadth: cache replacement (moderately constrained headroom), data prefetching (a broad design space in which multiple mechanisms can be combined), and branch prediction (a highly constrained domain where existing policies already capture nearly all available behavior). The chapters that follow report the per-domain results (Section 4), characterize what the evolution actually produces (Section 5), and interpret what the findings imply for the division of labor between the human architect and the LLM (Section 6).

Contributions

- Agentic Architect, the first end-to-end framework for agentic CPU microarchitectural design optimization, built on a modular architecture that supports interchangeable evolutionary frameworks, LLMs, simulators, and microarchitectural targets.
- Evolved policies that match or exceed state-of-the-art designs across three structurally different microarchitectural domains, demonstrating discovery in both broad and highly constrained search spaces.
- A systematic analysis of the framework's sensitivity to its design factors (seed, scoring, traces, prompt), showing that the architect's inputs, not the choice of LLM or evolutionary framework, dominate the quality of the evolved policy.
- A characterization of what LLM-driven evolution actually produces, which I call the *learned ensemble pattern*. The pattern and its structural stages are developed in Section 5.

2 Background and Prior Work

Agentic Architect sits at the intersection of two bodies of work: the microarchitectural design literature, which provides the policies the framework evolves, and the LLM-driven algorithm-discovery literature, which provides the search machinery.

2.1 Microarchitectural Policies

A modern out-of-order processor relies on many small embedded *microarchitectural policies*: rules for evicting cache lines, prefetching memory, predicting branches, and dozens of similar choices. Each operates under tight latency, area, and power constraints, and each contributes to overall performance. This section reviews the three policies this thesis targets: cache replacement, hardware prefetching, and branch prediction.

Cache Replacement

When a cache fills and a new line must be inserted, the replacement policy chooses which existing line to evict; a bad eviction causes a future miss costing dozens to hundreds of cycles. This thesis focuses on the Last-Level Cache (LLC), where misses are the most expensive. The classical baseline is Least Recently Used (LRU), but recency alone is a weak signal at the LLC: many lines are never reused between insertion and eviction, so a policy that can identify those lines early outperforms LRU substantially. A long line of research has attacked this problem: SHiP [39] and SHiP++ [40] use the program counter that filled a line to predict reuse; Hawkeye [13] approximates Belady's optimal algorithm by simulating it over past accesses; Glider [33] uses an attention-based neural network. Mockingjay [31] (HPCA 2022), the seed for replacement evolution in this thesis, predicts each line's reuse distance using temporal difference learning [36] trained on a sampled shadow cache, evicting the line with the largest Estimated Time to Reuse (ETR).

Hardware Prefetching

Hardware prefetchers issue speculative loads ahead of the demand stream to hide memory latency. The design problem is delicate: an overly aggressive prefetcher consumes bandwidth, evicts useful data, and saturates the Miss Status Holding Registers (MSHRs); a conservative one leaves coverage on the table. Simple approaches include next-line prefetching and IP Stride [9]; spatial prefetchers like SMS [34] record access footprints within memory regions; SPP [17] extends this with a signature path predictor. More recent work includes IPCP [27] (DPC-3 winner), Pythia [5] (MICRO 2021), and Berti [24] (MICRO 2022). VA/AMPM Lite, a variant of the Access Map Pattern Matching prefetcher [12] used as the prefetching seed in this thesis, maintains a 128-entry per-page region table ($\sim$3 KB of state) in which each entry tracks which cache lines within that page have been accessed, detecting both stride and irregular patterns while remaining competitive with substantially more complex designs.

Branch Prediction

Modern out-of-order processors depend on accurate branch prediction to keep their long pipelines fed; a misprediction flushes the pipeline and wastes many cycles. State-of-the-art predictors achieve 97–99% accuracy, but even small reductions in misprediction rate translate to measurable IPC gains on branch-sensitive workloads. The simplest considered predictor, Bimodal, uses a saturating counter per branch. Perceptron predictors [16] compute a weighted sum of features derived from branch history. The Hashed Perceptron (HP) [14], this thesis's branch prediction seed, hashes branch history at multiple geometric lengths (3 to 232 bits across 16 tables) and indexes weight tables with these hashes. The Multiperspective Perceptron [15] extends HP with many additional feature tables, and TAGE [30] is a separate family of tagged, geometric-history predictors.

2.2 LLM-Driven Code Evolution

The other half of Agentic Architect's machinery comes from a recent line of work using LLMs as mutation operators inside an evolutionary search loop. An evolutionary algorithm maintains a population of programs, mutates them, evaluates each candidate against a fitness function, and selects the best to seed the next generation. Replacing the traditionally hand-crafted mutation operator with a call to an LLM provides three attractive properties: the LLM's training on a vast code corpus gives a strong mutation prior; the LLM can introduce structurally non-trivial changes such as new data structures; and because it is conditioned on natural-language prompts, the human can steer the search by writing prompts rather than re-implementing the operator.

FunSearch [28] demonstrated that LLMs paired with evolutionary search can discover novel algorithms, finding new constructions for the cap-set problem and new solutions to bin-packing. AlphaEvolve [25] generalized this paradigm to general code optimization across mathematics and hardware design, inspiring a family of open-source frameworks. This thesis uses two: OpenEvolve [32], a MAP-Elites [23] island-based reimplementation of the AlphaEvolve paradigm, and AdaEvolve [7], which adds UCB [4] bandit island selection, AdaGrad-inspired search-intensity adaptation, and a k-nearest neighbor novelty archive.

Microarchitecture is a particularly suitable target for this paradigm: policies are discrete modular logic blocks with well-defined interfaces, cycle-accurate simulators provide a ground-truth fitness function queryable many times per iteration, and the design space, while large, is well-bounded.

2.3 Related Work

LLM-driven algorithm discovery. Following FunSearch and AlphaEvolve, a growing family of open-source frameworks now evolve programs using LLMs [32, 7, 3, 20, 21, 38], each exploring different strategies for sample efficiency, mutation diversity, and search adaptation. None of these has targeted CPU microarchitecture as an application domain.

Machine learning for systems. Machine learning and automated search are increasingly applied across systems research: learned indexes [19] replace B-trees with neural networks,

reinforcement learning has been used for query optimization [22], ensemble search has been applied to compiler autotuning [1], Cheng et al. [8] describe AI-driven research for systems using LLM-based evolutionary search, and Sankaralingam [29] proposes an automated "Idea Factory" for architecture research. This thesis provides concrete evidence for that vision in the microarchitecture policy domain, with the additional constraint that evolved policies must be realizable in conventional hardware.

Machine learning embedded in microarchitecture. A separate line of work embeds ML models directly into microarchitectural policies. Pythia [5] formulates prefetching as reinforcement learning; Glider [33] uses an attention-based neural network for replacement; LeCaR [37] applies RL to cache admission. These approaches require ML inference on the critical path at runtime, introducing latency, area, and power overhead. By contrast, the policies Agentic Architect evolves are standard C++ implementations with fixed logic and lookup tables, requiring no runtime inference and no special hardware support. The LLM is a *design-time* tool, not a runtime component.

3 Methodology

This section describes the Agentic Architect framework component by component, followed by the per-domain instantiation and the shared experimental infrastructure.

3.1 Framework Overview

Figure 1 illustrates the end-to-end workflow. The architect provides four inputs: a system prompt (⓪), a seed policy, a scoring function, and a trace database split into training and evaluation sets (③). At each iteration, the evolutionary agent (①) queries an LLM to mutate the current best policy; the candidate is compiled and run by the simulator (②) on each training trace; the evaluator (④) aggregates per-trace metrics into a scalar fitness score that the solution database uses to maintain a diverse high-fitness population. The loop repeats for a fixed number of iterations.

3.2 The Evolutionary Agent

Agentic Architect's evolutionary agent extends the generic LLM-driven evolutionary loop (Section 2) into a domain-aware agent through an integration layer connecting code evolution to microarchitecture. The agent is built on three principles.

Compilation-gated evolution. Each candidate is compiled and linked against the simulator before any simulation is attempted. Compilation failures are caught, the candidate receives a sentinel fitness score, and the specific compiler error is fed back to the LLM in subsequent iterations, substantially reducing repeated failures.

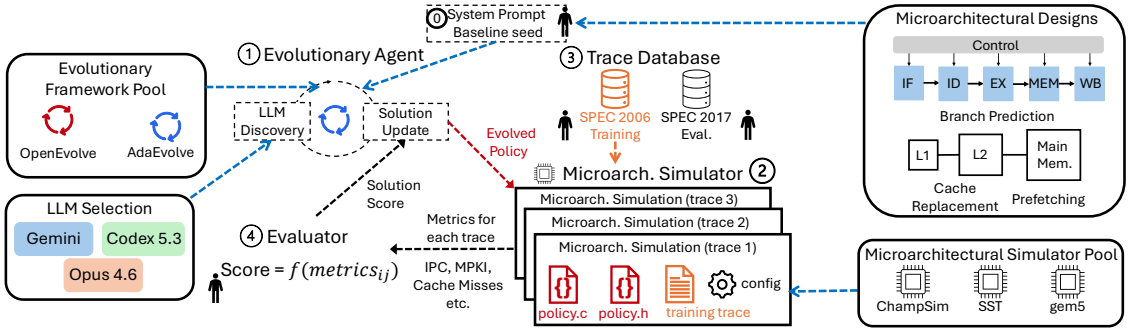


Figure 1: Overview of the Agentic Architect framework and the evolution loop. See Section 3 for a walkthrough.

Simulation timeout enforcement. LLM-generated policies can contain algorithmically expensive logic that causes simulation to exceed practical time budgets. The agent enforces a per-candidate time limit; policies exceeding this limit are terminated and assigned the sentinel score. Without this guardrail, a single slow candidate can stall the entire loop. As Section 4 shows, this mechanism has a major impact on search productivity.

Domain-agnostic interface. The agent is agnostic to the specific microarchitectural domain or seed policy. A human-provided seed plus a domain-specific system prompt is sufficient to define a new optimization target, allowing the same agent to optimize a broad range of policies.

3.3 Framework and Model Selection

Agentic Architect treats both the evolutionary framework and the underlying LLM as interchangeable components. OpenEvolve [32] and AdaEvolve [7] use different island selection, mutation intensity, and diversity mechanisms, but both plug into the same compilation, simulation, and evaluation pipeline. Similarly, the LLM is accessed through a model-agnostic API (Claude Opus [2], Kimi [18], Gemini [10], Codex [26]) without modifying the framework. Section 4 treats both as experimental variables.

3.4 Integration with Microarchitectural Simulators

The simulator serves as an automated evaluation backend queried at every iteration. It also defines the interface each candidate must implement: each domain imposes a small fixed set of hooks (e.g., `find_victim` for replacement, `prefetcher_cache_operate` for prefetching, `predict_branch` for branch prediction). The system prompt communicates this interface to the LLM, which writes arbitrary logic inside the hooks but cannot alter the interface. If a simulator can evaluate a policy, Agentic Architect can optimize it.

3.5 The Evaluator and Scoring Function

The evaluator combines per-trace simulator metrics into a scalar fitness score, balancing a primary objective (typically IPC) against secondary penalties that guard against local optima. A pure IPC objective is insufficient because a design might achieve high IPC on traces where the target component is not the bottleneck; optimizing a single secondary metric (e.g., cache misses) can over-reward improvements that do not translate to performance. The evaluator therefore uses a composite score, aggregated across the training set with equal per-trace weight. It is the primary mechanism through which the architect encodes domain knowledge into the search.

3.6 Trace Database

The trace database is split into a training set used during the evolutionary loop and an evaluation set used only after evolution completes to measure generalization. The split exposes a tradeoff: a training set that is too small risks overfitting; one that is too large increases per-iteration cost without necessarily improving generalization. Section 4 quantifies these effects empirically.

3.7 Prompt Design

The prompt is the primary mechanism through which the architect steers the search. It consists of a static system message specifying the function interface, hardware constraints, and domain context (Figure 1, Step ⓪); and a dynamic message, constructed automatically each iteration, containing the current best design's source code, fitness score, and a history of prior attempts.

The architect may optionally include technique suggestions from the literature, creating a spectrum between a *prescriptive prompt* that names specific techniques and a *minimal prompt* that provides only the problem specification. Both share a key principle: they specify *what* the policy should optimize, not *how* to optimize it. Figure 2 shows abbreviated examples for cache replacement.

3.8 Use Cases: Three Microarchitectural Domains

I instantiate Agentic Architect on three core mechanisms, each with its own seed and scoring function. Replacement is seeded from Mockingjay [31] (with an additional SHiP [39] seed for the sensitivity study in Section 4.4) and evaluated against LRU, SHiP++ [40], Hawkeye [13], and Glider [33]. Prefetching is seeded from VA/AMPM Lite [12] and evaluated against a no-prefetch baseline, IP Stride [9], Next Line, SMS [34], IPCP [27], Pythia [5], and Berti [24]. Branch prediction is seeded from the Hashed Perceptron [16, 14] and evaluated against Bimodal. Because replacement and prefetching target the same memory-system bottleneck, they share a composite score that rewards IPC and penalizes LLC misses; branch prediction

(a) Full Prompt (excerpt)

```
You are an expert architect inventing
novel LLC replacement policies.

# GOAL
Maximize ‘combined_score‘ across
MULTIPLE workloads.

# REPLACEMENT TECHNIQUES
- RRPV: per-line reuse-distance counter
- Per-PC prediction: hash IP -> SHCT
- Learning from Belady’s optimal
- Reuse-distance prediction

# TASK
Synthesize a novel policy that combines,
extends, or reimagines these strategies.
```

(b) Minimal Prompt (excerpt)

```
You are a researcher trying to discover
a novel CPU cache replacement policy
that has never been published.

# GOAL
Maximize ‘combined_score‘ across
MULTIPLE workloads.

# PROBLEM
On insertion, choose a victim. You see:
addr, ip, type, hit, set.

# THINGS WORTH EXPLORING
- IP is the most underused signal.
- Access type matters.
- Can you detect phase changes?
```

Figure 2: Abbreviated system prompt excerpts. Both specify the interface, hardware context, and scoring function. The full prompt (a) names published techniques; the minimal prompt (b) poses open questions without prescribing solutions.

uses a different secondary metric (MPKI):

$$\text{score}_{\text{repl./pref.}} = \text{IPC} \times 10000 - \frac{\text{LLC\_misses}}{1000}, \tag{1}$$

$$\text{score}_{\text{branch}} = \text{IPC} \times 10000 - \text{MPKI} \times 200. \tag{2}$$

3.9 Experimental Setup

Simulator. I use the ChampSim cycle-accurate simulator [11] to model a modern out-of-order processor. Replacement and prefetching experiments use a 4 GHz, 4-wide issue core with a 128-entry reorder buffer and a 1-cycle mispredict penalty. Branch-prediction experiments use a wider 4 GHz, 6-wide core with a 352-entry reorder buffer and a 20-cycle mispredict penalty, since a narrow core with a negligible mispredict penalty would mask branch-prediction impact. Both configurations share the same cache hierarchy and main memory: 32 KB L1 instruction and data caches, a 256 KB L2 cache, and a 2 MB 16-way set-associative LLC, backed by DDR5-3200 main memory.

Benchmark traces. I evaluate policies on 11 traces drawn from SPEC CPU 2006 [35] and 2017 [6]: `gcc`, `mcf`, `bwaves`, `cactuBSSN`, `lbm`, `omnetpp`, `wrf`, `xalancbmk`, `fotonik3d`, `roms`, and `gobmk`. Each simulation uses 25 million warmup instructions followed by 50 million simulation instructions. The trace set spans the full range of memory and control behavior: streaming (`lbm`), pointer-chasing (`mcf`), irregular access (`omnetpp`, `xalancbmk`), structured scientific codes (`bwaves`, `fotonik3d`, `wrf`, `cactuBSSN`, `roms`), branch-heavy game-tree search (`gobmk`), and compiler workloads (`gcc`).

Evolutionary frameworks, models, and prompts. I experiment with two evolutionary backends: OpenEvolve [32] and AdaEvolve [7], each run for 100 iterations. I use Claude Opus

4.6 [2] as the primary LLM, with controlled comparisons against Kimi K2.5 [18], Gemini 2.5 Pro [10], and GPT-5.3 Codex [26] under matched settings. I evaluate two prompt strategies per domain: a *full prompt* that includes named techniques from prior work, and a *minimal prompt* that provides only the interface, hardware context, scoring function, and problem description.

4 Results

I evaluate Agentic Architect from two perspectives: the quality of the evolved policies, and the sensitivity of the framework to key design decisions. Figure 3 previews the headline result: across all three domains, the evolved policies match or exceed the strongest published baselines.

4.1 Evolved Microarchitectural Policies

For each domain I start from a state-of-the-art seed and report the best result across both frameworks (OpenEvolve and AdaEvolve). Figure 4 shows per-trace breakdowns for the cache replacement and prefetching domains; Figure 5 covers branch prediction.

Cache Replacement

Figure 3(a) summarizes the cache replacement results. The best evolved policy, seeded from Mockingjay, achieves a 1.062× geomean IPC speedup over LRU across all 11 SPEC traces, exceeding Mockingjay's 1.056×. Figure 4a shows the per-trace breakdown. Gains concentrate on memory-intensive traces: 1.12× on `mcf` and 1.49× on `xalancbmk`, where high LLC miss rates make eviction decisions critical. On compute-bound traces (`cactuBSSN`, `bwaves`), the evolved policy matches Mockingjay with no regression.

The evolved policy preserves Mockingjay's core structure while augmenting it with five independent components: (1) a 3-way ensemble predictor combining the inherited Reuse Distance Predictor with a PC-transition predictor and a set-context predictor, with accuracy tracking that dynamically arbitrates between them; (2) eviction regret tracking that penalizes PC signatures responsible for premature evictions; (3) per-PC demand/prefetch dead-block prediction with asymmetric training; (4) a stride detector that bypasses streaming PCs; and (5) per-set adaptive aging and thrashing detection. Where Mockingjay relies on a single RDP, the evolved policy combines multiple independent signals and adapts at runtime.

Prefetching

Prefetching is the domain where LLM-driven evolution achieves its strongest result. As shown in Figure 3(b), the evolved prefetcher achieves a 1.76× geomean IPC speedup over no prefetching, exceeding both its VA/AMPM Lite seed (1.59×) and the strongest non-seed published baselines (SMS at 1.55×, Berti and Pythia at 1.54×) by wide margins.

Figure 4b shows the evolved prefetcher dominates across the entire trace suite: large gains on streaming workloads (`lbm`, `mcf`), irregular access patterns (`omnetpp`, `xalancbmk`), and structured scientific codes (`bwaves`, `fotonik3d`).

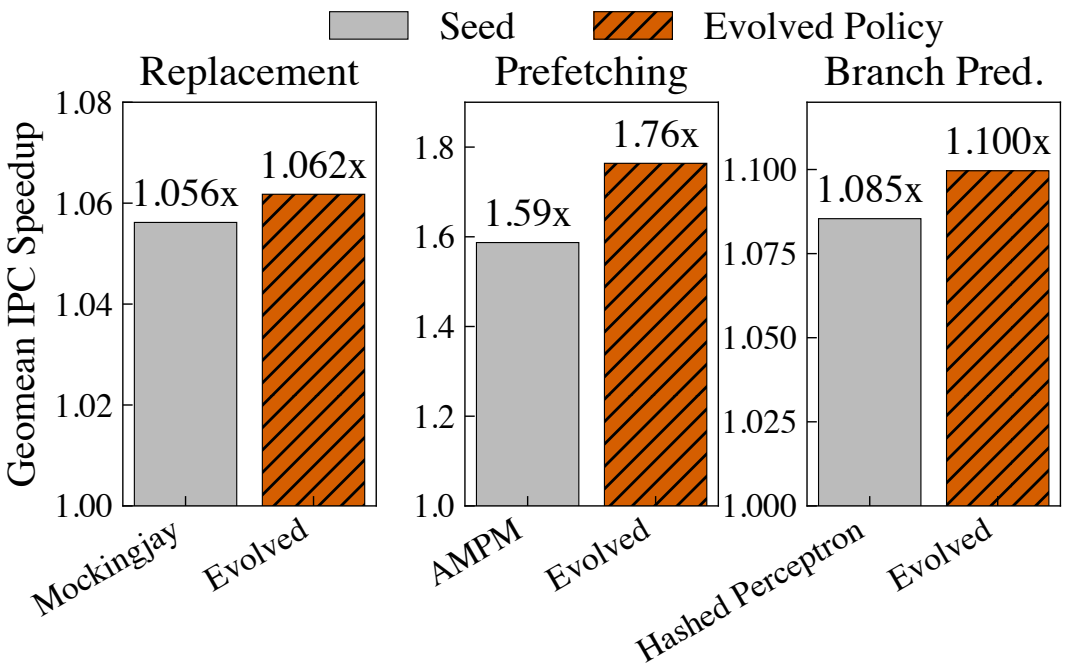
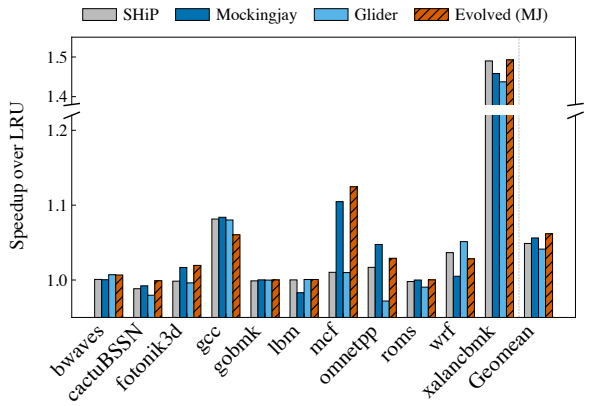
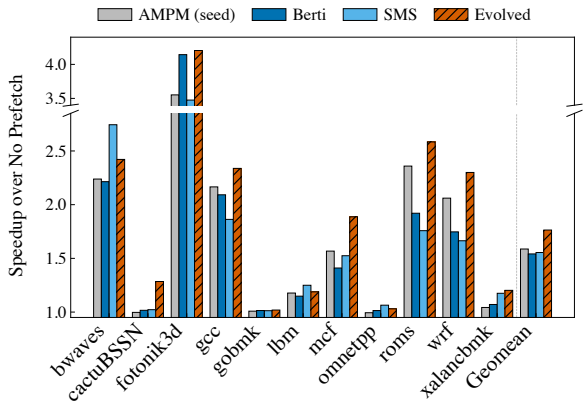


Figure 3: Geomean IPC speedup across all three domains. Left: cache replacement vs. LRU. Center: prefetching vs. no prefetcher. Right: branch prediction vs. Bimodal.



(a) Cache replacement: per-trace IPC speedup over LRU.



(b) Prefetching: per-trace IPC speedup over no prefetching.

Figure 4: Per-trace IPC speedup for cache replacement and prefetching policies, including state-of-the-art baselines and the evolved policies.

Starting from VA/AMPM Lite's single per-page access map, evolution produced a six-engine architecture covering complementary patterns: a page stream detector for sequential and strided accesses within pages, a correlation engine tracking address-to-address transitions, an IP stride engine with adaptive confidence, an IP delta engine for complex multi-stride patterns, a global delta engine for cross-page patterns, and a spatial fallback for low-confidence situations. The architecture's most novel feature is its runtime adaptivity: every 256 accesses,

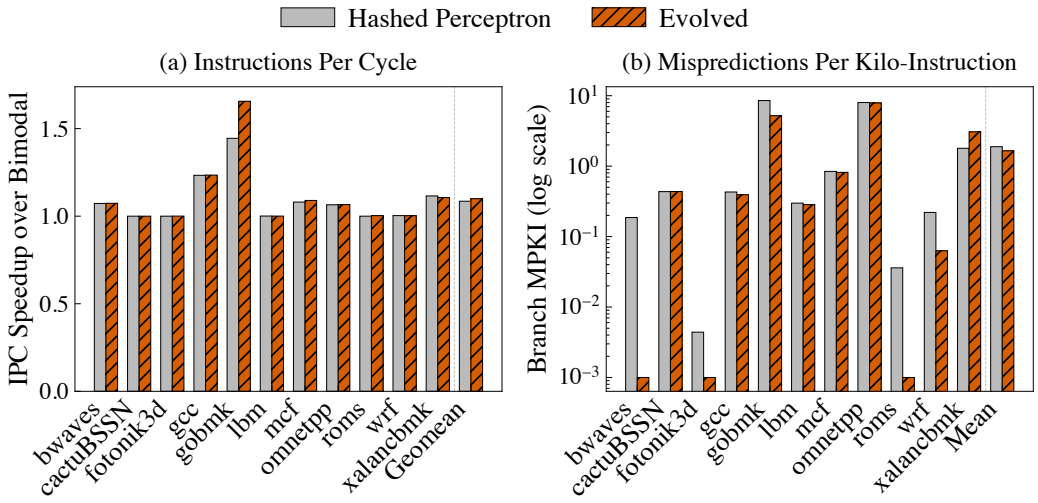


Figure 5: Per-trace IPC over Bimodal and MPKI for branch prediction policies, including the Hashed Perceptron seed and the evolved policy.

the prefetcher re-evaluates each engine's accuracy and temporarily disables underperformers. It also incorporates MSHR awareness, throttling speculative engines as outstanding prefetches approach capacity. Together these mechanisms make the architecture more adaptive than many published prefetchers.

Branch Prediction

Branch prediction is the most constrained domain: baseline accuracy exceeds 97%, leaving little headroom. Nevertheless, Figure 3(c) shows that the best evolved predictor achieves a 1.100× geomean IPC speedup over Bimodal, an additional 1.5% over the Hashed Perceptron seed (1.085×).

Figure 5 shows that the overall gain is driven almost entirely by `gobmk`, where IPC improves from 1.031 to 1.182 (1.147×) and MPKI falls from 8.53 to 5.20, a 39% reduction in mispredictions. On nearly all other traces, the evolved predictor matches Hashed Perceptron, suggesting that the seed already captures most of the available branch behavior on those workloads.

The `gobmk` trace contains complex game-tree patterns with strong local-history correlations that Hashed Perceptron's global-history tables miss. The evolved predictor combines a TAGE-SC-L backbone with a multiperspective-perceptron component, so it brings both tagged-table long-history prediction and a bank of diverse feature tables to bear on exactly this kind of structure; Section 5 discusses the evolved predictor's structural relationship to these two prior architectures and what that relationship reveals about the search.

Figure 6 illustrates that the relationship between MPKI reduction and IPC improvement is highly non-linear. On `wrf`, a 3.5× MPKI reduction (0.22 to 0.06) yields no IPC gain because branch prediction is not the bottleneck on this memory-bound trace. On `gobmk`, a comparable reduction yields 14.6% IPC because the pipeline is prediction-limited. A pure MPKI objective would over-reward improvements where mispredictions do not limit performance.

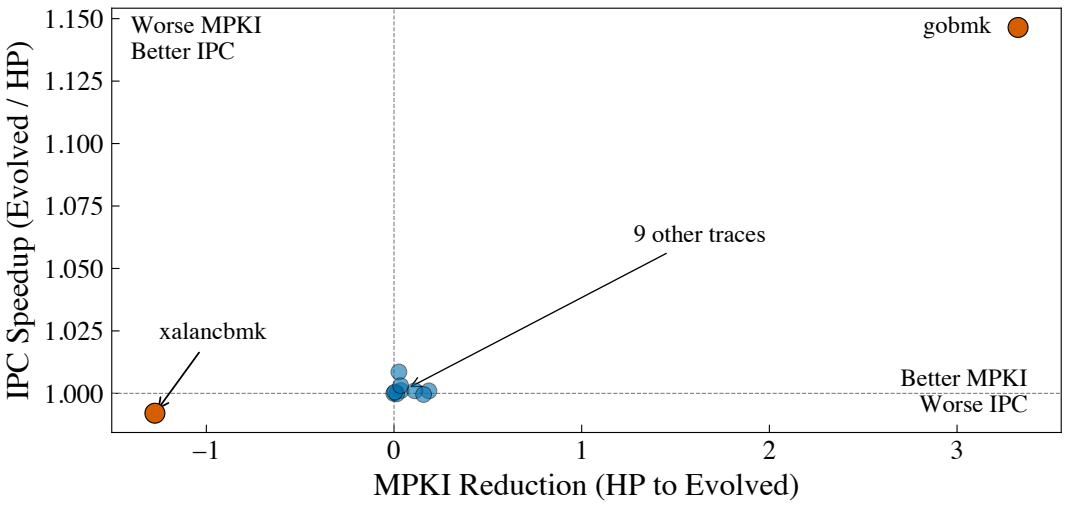


Figure 6: MPKI improvement vs. IPC improvement across traces. The relationship is highly non-linear: large MPKI reductions on memory-bound traces produce no IPC gain, while comparable reductions on prediction-bound traces produce large gains.

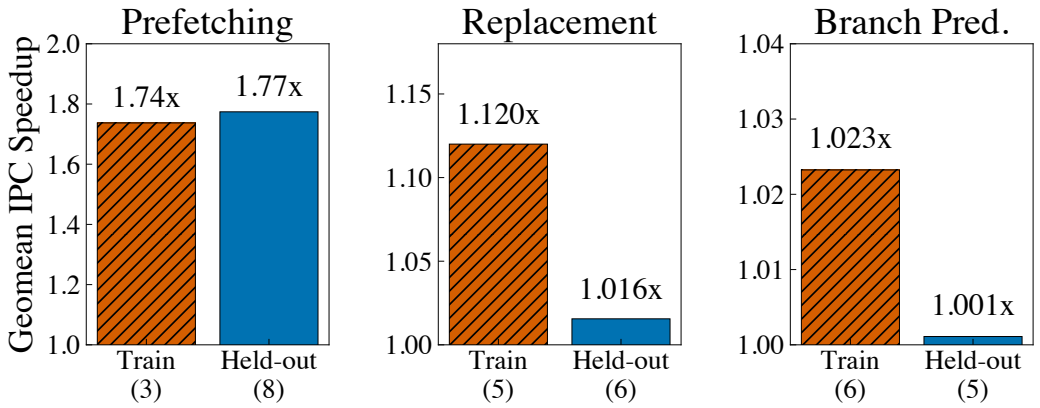


Figure 7: Training vs. held-out trace geomean IPC speedup across domains. Prefetching shows remarkable generalization (held-out exceeds training); replacement and branch prediction concentrate gains on training traces.

4.2 Generalization and Trace Selection

A critical question for any learned policy is whether it generalizes beyond its training traces. I compare each domain's best policy on the training trace subset against the held-out traces from the full 11-trace benchmark (Figure 7).

Prefetching: This evolved policy shows remarkable generalization: trained on only three traces (`gcc`, `mcf`, `lbm`) spanning stride, pointer-chasing, and streaming patterns, its held-out geomean of 1.77× *exceeds* its training geomean of 1.74×. The six-engine architecture with runtime adaptation naturally extends to held-out traces' combinations of these patterns.

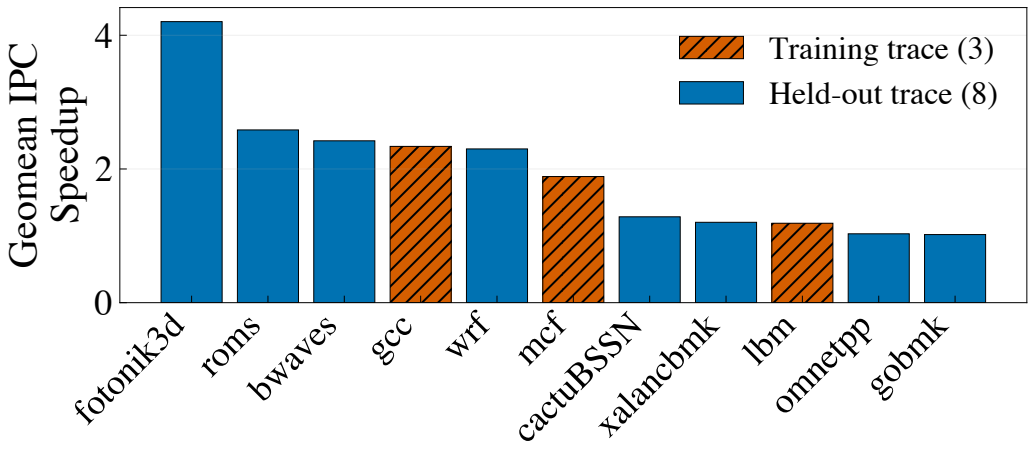


Figure 8: Per-trace IPC speedup over no prefetch for the evolved prefetcher, with training traces highlighted. Held-out traces (unmarked) show comparable or larger gains than training traces, demonstrating robust generalization.

Table 1: Storage cost of evolved policies versus the best published policy. All values in KB.

Domain	Best Published	KB	Evolved KB
Replacement	Mockingjay	168	783 (Mockingjay seed)
Prefetch	Berti	103	87 (VA/AMPM Lite seed)
Branch	Hashed Perceptron	64	128 (HP seed)

Cache replacement and branch prediction: These policies show the opposite pattern, concentrating gains on the training set. Replacement reaches 1.13× on training but only 1.004× on held-out, with gains almost entirely on `mcf` and `xalancbmk`. Branch prediction reaches 1.023× on training vs. 1.001× held-out, driven almost entirely by `gobmk`. Both domains overfit to the workloads where the target component matters most.

A natural experiment in the replacement domain illustrates how critical trace selection is. Replacing `gcc` with `xalancbmk` in the training set *hurts* benchmark performance by 4%. `xalancbmk` has an extreme IPC sensitivity to replacement policy (54% spread across all evaluated policies), and its outsized fitness contribution causes evolution to overspecialize at the expense of broader performance. Trace selection should prioritize *diversity of access patterns* over *coverage of workloads*.

4.3 Storage Overhead

Table 1 compares each evolved policy against the strongest published policy in its domain.

The evolved prefetcher (87 KB) is *smaller* than Berti (103 KB) while delivering substantially higher performance, making it Pareto-optimal in storage–performance terms. The evolved branch predictor (128 KB) is 2× the Hashed Perceptron seed but remains within typical predictor budgets. The evolved replacement policy (783 KB) is the most storage-intensive, exceeding Mockingjay by 4.7×, driven by the auxiliary structures described above: the

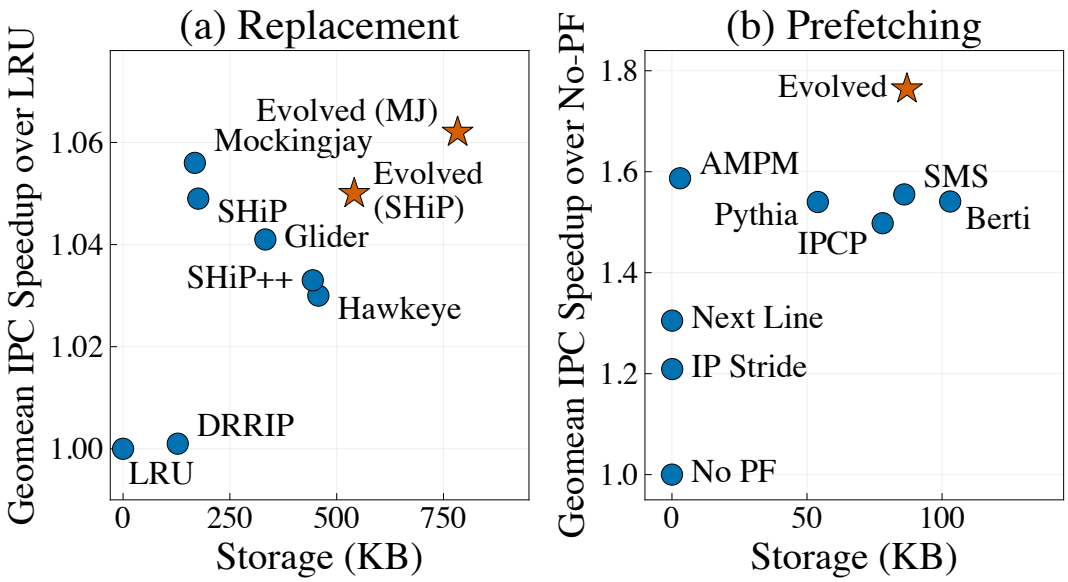


Figure 9: Storage–performance Pareto frontier across evaluated policies. X-axis: storage cost. Y-axis: geomean IPC speedup.

eviction-regret tracker, the PC-transition and set-context predictors, the per-PC dead-block predictor tables, and the per-set adaptive aging counters. Whether this overhead is acceptable depends on the target silicon budget.

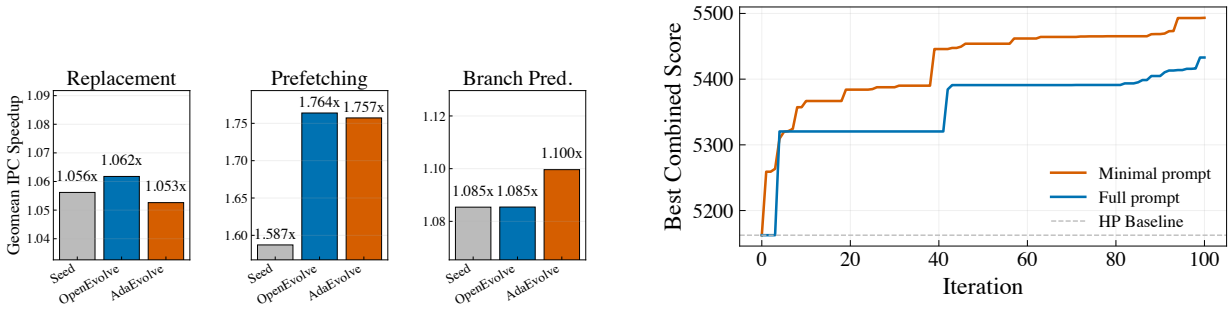
Figure 9 plots the storage–performance Pareto frontier. The evolved policies extend the frontier in each domain but do not dominate at all storage budgets: for replacement, Mockingjay at 168 KB offers a better performance-per-byte ratio than the evolved variant at 783 KB. This motivates incorporating storage as an explicit optimization objective (see Section 7).

4.4 Seed Sensitivity

The choice of seed measurably impacts the final policy. I compare two cache replacement seeds: Mockingjay (1.056× over LRU) and SHiP (1.049×). Using OpenEvolve with 100 iterations, Mockingjay-seeded evolution reaches 1.062×, while SHiP-seeded evolution reaches only 1.050×. The gap between evolved policies (0.012×) closely mirrors the gap between seeds (0.007×). AdaEvolve shows a similar pattern: its best SHiP-seeded run achieves 1.053×, below the unevolved Mockingjay seed. More generally, no evolved weaker-seed policy crossed the unevolved stronger-seed baseline in any of our experiments: evolution amplifies the strengths present in the seed, it does not overcome them.

4.5 Framework and Prompt Sensitivity

I compare two search-design axes under controlled conditions (same model, seed, traces, and 100-iteration budget): the choice of evolutionary framework (OpenEvolve vs. AdaEvolve) and the choice of prompt strategy (full vs. minimal).



(a) Framework comparison across all three domains. Each panel shows geomean IPC speedup for the seed, the best OpenEvolve result, and the best AdaEvolve result.

(b) Minimal vs. full prompt on branch prediction (AdaEvolve, Hashed Perceptron seed, 100 iterations). Y-axis: best combined evaluator score at each iteration.

Figure 10: Search-design sensitivity: framework choice (left) and prompt strategy (right).

Framework. AdaEvolve achieves higher training scores, but this does not always translate to benchmark performance: for replacement, OpenEvolve produces the better 11-trace result (1.062× vs. 1.048×) despite lower training scores; for prefetching, both converge to 1.76×. The gap between frameworks is much smaller than the gap between either and the unmodified seed, suggesting that the evolutionary loop itself, the coupling of LLM-driven mutation with cycle-accurate evaluation, is the primary driver of improvement, and validating Agentic Architect's modularity.

Prompt. Minimal prompts beat full prompts in both cache replacement (1.053× vs. 1.049× over LRU) and branch prediction (1.100× vs. 1.096× over Bimodal). Two factors explain this. First, naming techniques *anchors* the LLM to known architectures, constraining the search to combinations of named ideas. Second, full prompts induce complex implementations: naming techniques like OPTgen or reuse-distance predictors encourages $O(n^2)$ per-access logic, and 61% of full-prompt replacement evaluations timed out. The minimal prompt also yields more consistent evaluation times (1800–1910 s vs. 1400–3600 s for branch prediction), producing 27 new-best discoveries versus 15 over 100 iterations. This reframes the prompt-design paradigm: specify *what you want* through the fitness function, not *how to build it* through technique suggestions.

4.6 Model Comparison and Cost

To isolate the effect of the LLM, I run four models under identical conditions (Mockingjay seed, same prompt and traces, OpenEvolve, 100 iterations each).

Figure 11 shows the results. Opus 4.6 achieves the highest speedup (1.062×) with zero compile failures. Kimi K2.5 is second (1.057×) but wastes 34% of iterations on failures. Codex and Gemini tie at 1.052×, with Codex producing zero failures and Gemini failing on 26%. Per-run API costs for 100 iterations vary widely, from roughly $12 to $50+ depending on the model, with a fair amount of within-model variation as well because prompt length, generated output length, and retry behavior all move the total.

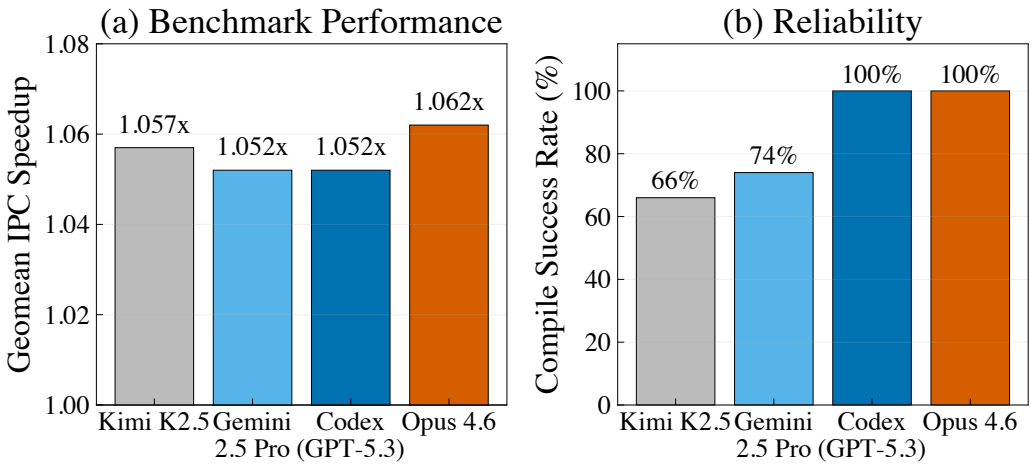


Figure 11: Controlled model comparison on Mockingjay-seeded cache replacement evolution (100 iterations, identical configuration). Left: 11-trace geomean IPC speedup over LRU. Right: compilation success rate.

Two of the four models (Gemini and Codex) produce evolved policies that *underperform* the unevolved Mockingjay seed on the full 11-trace benchmark, despite beating it on the 5-trace training set; the ranking by training score does not match the ranking on held-out performance. This is a single-run observation within one seed and one domain and should not be read as a ranking of these LLMs in general, but it does indicate that per-iteration cost alone is not a reliable proxy for how well the resulting evolved policy will generalize.

5 What Does Evolution Discover?

Section 4 established that LLM-driven evolution produces measurable gains across all three domains, but did not explain *what* the evolved policies do or *why* they outperform their seeds. This section argues the answer lies in *synthesis*: the evolved policies do not invent fundamentally new algorithms but compose known techniques into tightly integrated architectures that no individual designer assembled. The same structural pattern emerges across all three domains, and that pattern itself is the most important finding of this thesis.

5.1 The Learned Ensemble Pattern

Across all three domains, the evolved policies share a consistent four-stage structure that I call the *learned ensemble pattern*.

Stage 1: Preserve the seed's core. The seed's central mechanism survives evolution intact in every domain: Mockingjay's TD-based reuse-distance predictor, VA/AMPM Lite's per-page access map, and Hashed Perceptron's weighted feature sum each persist unchanged. Any mutation breaking the core rarely outperforms the unmodified seed and is selected against.

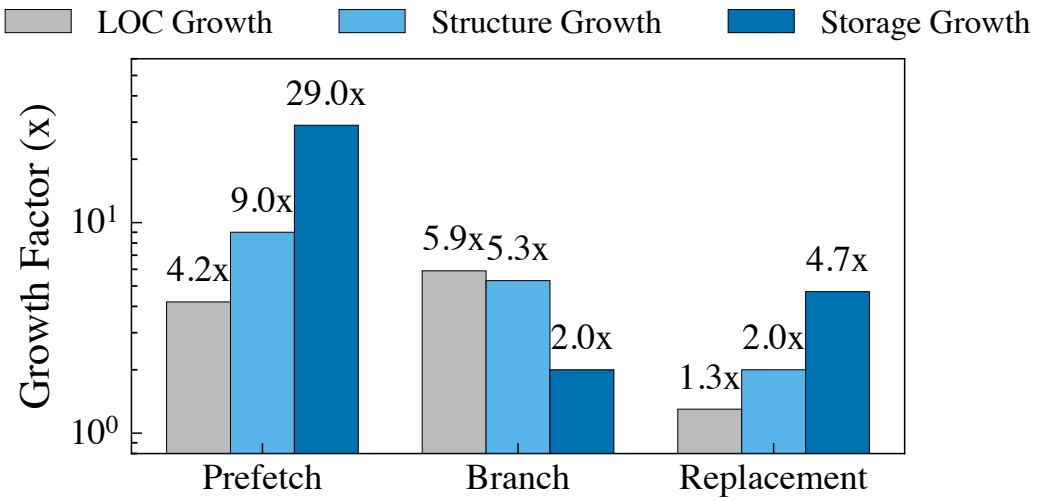


Figure 12: Architecture growth from seed to evolved policy across the three domains. Each axis shows the multiplicative growth in lines of code (LOC) and storage relative to the seed. Growth concentrates in stages 2–4 of the learned ensemble pattern; the seed's core (stage 1) is preserved.

Stage 2: Add orthogonal features. Evolution augments the core with additional predictive signals: PC-transition and set-context predictors for replacement, five additional engines for the prefetcher, local-history tables for branch prediction. Individually these correspond to known published techniques; the LLM reliably recognizes which orthogonal features are useful and how to wire them in.

Stage 3: Integrate through novel coordination. The components are known but the mechanisms combining them are not. In replacement, staggered-lifetime eviction filters arbitrate between predictors at two decay rates and penalize PC signatures responsible for premature evictions. In prefetching, epoch-based engine selection with MSHR-aware throttling decides which engines to activate. In branch prediction, a bias-by-local-direction mechanism weights feature contributions by recent local outcome. None of these has a direct published precedent.

Stage 4: Adapt at runtime. Adaptation layers tune parameters or enable/disable components based on observed behavior: set dueling in replacement, epoch-based engine reconfiguration in prefetching, dynamic feature weighting in branch prediction.

Figure 12 quantifies the resulting growth. The prefetcher expands from 143 to 606 LOC (4.2×) with 29× storage growth (3 KB → 87 KB). The branch predictor expands from 188 to roughly 1,100 LOC (5.9×) but only 2× storage growth (64 KB → 128 KB), since each new feature table is small relative to the weight tables already present in the Hashed Perceptron seed. The replacement policy shows the most conservative LOC evolution at 1.3× but a 4.7× storage expansion (168 KB → 783 KB), reflecting the addition of auxiliary data structures, the eviction-regret tracker, the PC-transition table, and per-set duelers, around an unchanged core algorithm. In all cases, growth concentrates in the latter three stages of the pattern

while the core remains stable.

The consistency of this pattern across three structurally different domains suggests it reflects a property of the search process itself. The evolutionary loop rewards additive complexity and penalizes destructive rewrites; the LLM's training on the architecture literature provides ready knowledge of existing techniques. Together, they produce designs that read like ensembles of published ideas wired together by novel coordination logic.

5.2 Novel Versus Known Techniques

Most individual mutations correspond to known published techniques: PC-based reuse prediction, stride detection, geometric history lengths, set dueling, and epoch-based statistics all appear in the literature. The novelty lies in the *synthesis*: the specific combinations, parameterizations, and runtime adaptation layers binding them together are original. Several components have no direct published precedent:

- **Staggered-lifetime dual eviction filters** in the replacement policy, which track recently evicted addresses at two different decay rates to capture both short-term and medium-term reuse signals.
- **Address-region signatures** that augment PC-based prediction with spatial-locality information from the surrounding memory region, allowing the predictor to distinguish accesses to different parts of the same data structure.
- **Epoch-based engine enable/disable with MSHR awareness** in the prefetcher, which dynamically adjusts the prefetch strategy based on both per-engine accuracy statistics and memory-subsystem backpressure.
- **Bias-by-local-direction** in the branch predictor, which indexes a feature table by both the branch address and the direction of the most recent local branch outcome, capturing a form of conditional correlation that standard feature tables miss.

Each is a compact, constant-overhead mechanism that an LLM can plausibly synthesize but a human architect would be unlikely to add unless specifically thinking about that interaction. The LLM does not need to reason about why a mechanism would help; it only needs to propose it as one of many candidates and let the fitness signal select for it.

The strongest evidence that the evolved policies are not simple recombinations comes from branch prediction. Starting only from a Hashed Perceptron seed and a fitness signal, the LLM produced a predictor that structurally resembles a TAGE-SC-L [30] backbone combined with a Multiperspective Perceptron component [15]: the evolved source contains 18 tagged tables with per-entry useful-bits, a loop predictor, a statistical corrector, and a bank of multiperspective feature tables, together with genuinely novel components beyond either architecture (most notably a bias-by-local-direction mechanism). TAGE-SC-L and MPP both appear in the LLM's training corpus; what the search demonstrates is not first-principles invention but the ability to select and integrate two independently strong architectures into a coherent policy under a fitness signal, without being directed to combine them.

5.3 The Seed Constrains the Ensemble

Because Stage 1 preserves the seed's core, the quality of that core directly bounds what the ensemble can achieve; the empirical support is the seed-sensitivity experiment in Section 4.4. The right way to understand the LLM in this setting is as an *architecture search engine* that proposes structural changes drawn from its prior on the literature, while the evolutionary loop provides the fitness signal that selects survivors. The value of the approach lies in *coverage*: a human architect can in principle propose ensembles of known techniques with novel coordination layers, but cannot propose, evaluate, and select from hundreds of them in the matter of hours. The framework makes that scale of exploration practical.

6 Discussion

6.1 The Architect's Role is Changing

The four sensitivity experiments in Section 4 show that the architect's inputs (seed, scoring function, training traces, and prompt strategy) consistently matter more for final policy quality than the choice of LLM or evolutionary framework. This shifts what the architect does: rather than writing the policy, the architect curates the search that will produce it.

The productive question is therefore not whether LLMs can replace the architect, but how they can accelerate the architect. I describe this division of labor as *agentic co-design*: the architect specifies what should be optimized and under what constraints; the LLM, wrapped in an evolutionary loop, explores the specified space at a scale no individual designer can match. The four sensitivity experiments do not establish that fully autonomous design is impossible, only that the version of it this framework could plausibly support, letting the LLM pick its own seed, objective, or workload set, is not what the experiments tested and is not what produced the results in Section 4.

6.2 Implications for What Can Be Discovered

The learned ensemble pattern characterized in Section 5 tells us not only what this method does produce but also what it cannot. The LLM's mutation prior is shaped by the published literature in its training corpus, and the evolutionary loop rewards incremental extensions over wholesale rewrites. A new algorithmic primitive with no published precedent is therefore not something the framework is positioned to invent; its output will always be a composition of techniques already represented in some form in the prior literature.

This is a boundary, not a dismissal. It implies that practitioners should match problems to the method's strengths: tasks whose productive answer lies in novel combinations of existing techniques, coordinated and adapted under realistic workloads, fit the framework well; tasks whose answer requires a genuinely new primitive do not. It also suggests a concrete research agenda for strengthening the method: changes to the mutation prior (through seed diversity, prompt design, or explicit paradigm injection) are more likely to widen what the framework can discover than changes to the evolutionary loop itself.

6.3 Limitations

The engineering extensions (storage-aware fitness, co-evolution of interacting policies, generalization as an explicit objective) are taken up in Section 7. One non-engineering limitation remains worth naming. The framework inherits the fidelity of its evaluation backend: ChampSim [11] models pipeline, cache, and memory behavior accurately, but abstracts away area, power, and timing at the gate level. An evolved policy that performs well in simulation may be impractical in silicon for reasons the evaluator cannot see. Tightening this loop, by running candidates through an RTL-level or silicon-level backend, would ground the fitness signal in the constraints that ultimately decide whether a design ships.

7 Future Work

The results in this thesis establish LLM-driven evolution as a viable methodology for CPU microarchitectural policy design and expose several directions in which it can be extended.

Physical constraints as explicit fitness objectives. The clearest near-term extension is motivated by the storage results in Section 4: the evolved replacement policy uses 783 KB (4.7× its seed), and the branch predictor uses 2× its seed, because the scoring functions ignore storage entirely. Incorporating storage, area, and power as explicit fitness terms, either as linear penalties or in a Pareto-style multi-objective formulation, would address the most significant practical shortcoming of the policies presented here. The framework already supports this in principle: the per-iteration evaluator is a function the architect controls.

Co-evolution of interacting policies. Policies were evolved one at a time, with all others held at ChampSim defaults. In real processors they interact: an aggressive prefetcher fills the cache with speculative data the replacement policy must manage; an inaccurate branch predictor changes the access patterns the prefetcher and cache see. A more ambitious framework would co-evolve interacting policies against a shared fitness function, potentially reaching performance no single-policy evolution can.

Generalization as an explicit objective. Section 4.2 showed that generalization varies sharply across domains. The framework treats generalization as an emergent property of training-trace selection; making it an explicit objective, by scoring each candidate on a held-out validation set and penalizing candidates whose validation performance trails training, would discourage overfitting directly.

Domain expansion. The framework is simulator-agnostic. Beyond the three domains covered here, candidate targets include memory scheduling, cache coherence protocols, NoC routing, and speculative-execution policies. Each imposes its own interface and scoring requirements, but each would benefit from the same co-design framework.

The shifting boundary between human and automated design. The longest-term question is where the boundary will settle. As models improve, several of the architect's current responsibilities (Section 6) may become tractable to automate. My expectation is that the architect's role will move higher in the design stack rather than disappear: the human will increasingly specify *what* to optimize, while the framework handles *how*. The most exciting consequence is that microarchitectural research becomes accessible to researchers with strong domain insight but less low-level implementation experience.

8 Conclusion

I introduced Agentic Architect, a framework for agentic CPU microarchitectural design optimization that couples LLM-driven code evolution with cycle-accurate simulation. Instantiated across cache replacement, data prefetching, and branch prediction, it produced policies that match or exceed the strongest published baselines in each domain: 1.062× geomean IPC speedup over LRU (surpassing Mockingjay), 1.76× over no prefetching (beating VA/AMPM Lite at 1.59× and the next-best published baselines at 1.54–1.55×), and 1.100× over Bimodal in the most constrained domain, with a 39% misprediction reduction on the most prediction-sensitive trace.

A systematic analysis showed that the architect's choices (seed, scoring function, training traces, prompt) dominate the choice of LLM or evolutionary framework in determining the quality of the evolved policy. The greatest current value of LLMs in computer architecture therefore lies in agentic co-design rather than autonomous discovery: the architect defines the search envelope, and the LLM accelerates exploration within it. Across all three domains, the evolved policies share a consistent four-stage *learned ensemble pattern* (preserve the seed's core, add orthogonal features, integrate through novel coordination, adapt at runtime), which is the most significant structural finding of this work. Agentic Architect is the first comprehensive, end-to-end framework for agentic CPU microarchitectural design optimization, and the results presented suggest that LLM-driven co-design is a practical methodology for producing policies that are both implementable in conventional hardware and competitive with the state of the art.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 303–316, 2014.

[2] Anthropic. System Card: Claude Opus 4 & Claude Sonnet 4. Technical report, 2025.

[3] Henrique Assumpç ao, Diego Ferreira, Leandro Campos, and Fabricio Murai. CodeEvolve: an Open Source Evolutionary Coding Agent for Algorithmic Discovery and Optimization. arXiv preprint arXiv:2510.14150, 2025.

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multi-armed Bandit Problem. *Machine Learning*, 47(2–3):235–256, 2002.

[5] Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1121–1137, 2021.

[6] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[7] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive llm driven zeroth-order optimization, 2026.

[8] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the Gate: How AI is Upending Systems Research. arXiv preprint arXiv:2510.06189, 2025.

[9] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 102–110, 1992.

[10] Gemini Team, Google DeepMind. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. arXiv preprint arXiv:2507.06261, 2025.

[11] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jiménez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv preprint arXiv:2210.14324, 2022.

[12] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access Map Pattern Matching for Data Cache Prefetch. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, pages 499–500, 2009.

[13] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE, 2016.

[14] Daniel A. Jiménez. Strided Sampling Hashed Perceptron Predictor. JILP Special Issue on the 4th Championship Branch Prediction Competition (CBP-4), 2014.

[15] Daniel A. Jiménez. Multiperspective perceptron predictor. Proceedings of the 5th Championship Branch Prediction Competition (CBP-5), 2016.

[16] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[17] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan A. Chishti. Path Confidence based Lookahead Prefetching. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[18] Kimi Team. Kimi K2.5: Visual Agentic Intelligence. Technical report, Moonshot AI, 2026.

[19] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018.

[20] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution. arXiv preprint arXiv:2509.19349, 2025.

[21] Gang Liu, Yihan Zhu, Jie Chen, and Meng Jiang. Scientific Algorithm Discovery by Augmenting AlphaEvolve with Deep Research. arXiv preprint arXiv:2510.06056, 2025.

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. In *Proceedings of the VLDB Endowment*, volume 12, pages 1705–1718, 2019.

[23] Jean-Baptiste Mouret and Jeff Clune. Illuminating Search Spaces by Mapping Elites. arXiv preprint arXiv:1504.04909, 2015.

[24] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibá nez, Víctor Vi nals Yúfera, and Alberto Ros. Berti: an Accurate Local-Delta Data Prefetcher. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[25] Alexander Novikov, Ngân V u, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. arXiv preprint arXiv:2506.13131, 2025.

[26] OpenAI. GPT-5.3-Codex System Card. Technical report, 2026.

[27] Samuel Pakalapati and Biswabandan Panda. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[28] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical Discoveries from Program Search with Large Language Models. *Nature*, 625:468–475, 2024.

[29] Karthikeyan Sankaralingam. Computer Architecture's AlphaZero Moment: Automated Discovery in an Encircled World. arXiv preprint arXiv:2604.03312, 2026.

[30] André Seznec. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 117–127. ACM, 2011.

[31] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective Mimicry of Belady's MIN Policy. In *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[32] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025.

[33] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 413–425, 2019.

[34] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[35] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmark Suite, 2006.

[36] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[37] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[38] Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. ThetaEvolve: Test-time Learning on Open Problems. arXiv preprint arXiv:2511.23473, 2025.

[39] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

[40] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moin Qureshi. Ship + + : Enhancing signature-based hit predictor for improved cache performance. 2017.