The Architect and the Agent

In pursuit of the AlphaZero of System Design

Amir Yazdanbakhsh

Research Scientist

Google DeepMind

ayazdan.com

ASAP
Sep. 3, 2026

*** Many results reflect joint work with academic and industry partners**

The Legacy of AlphaGo / AlphaZero



March 9, 2016

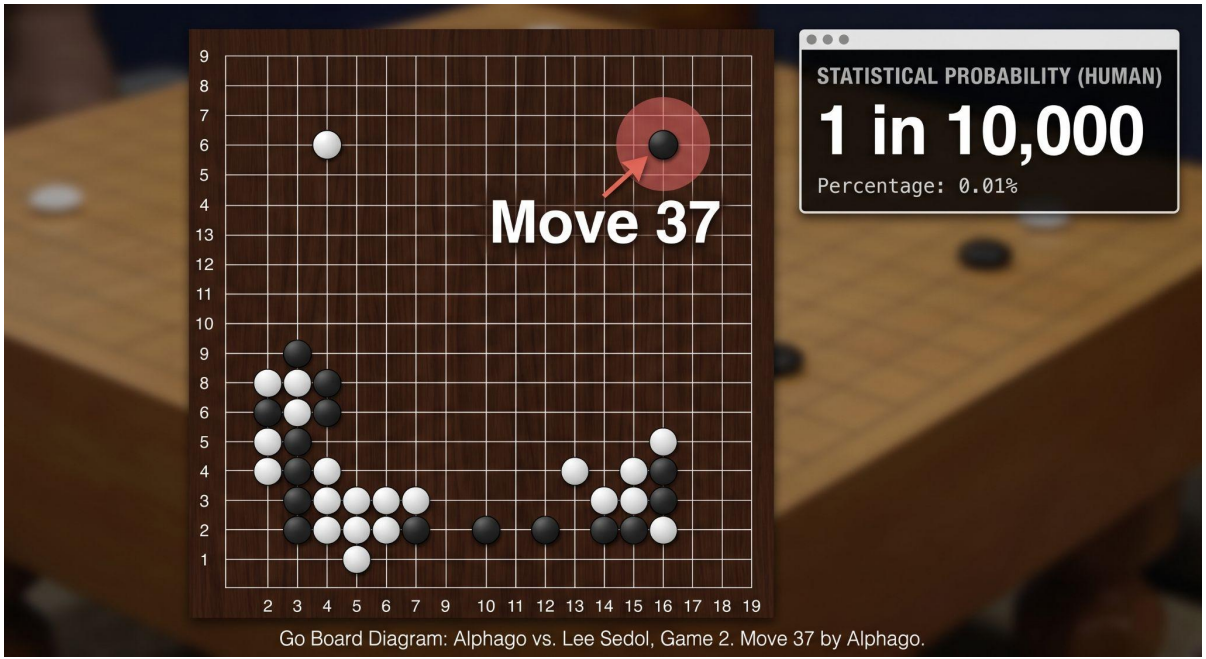
The Legacy of AlphaGo / AlphaZero



March 9, 2016

The first program to beat a professional human Go player without any handicap.

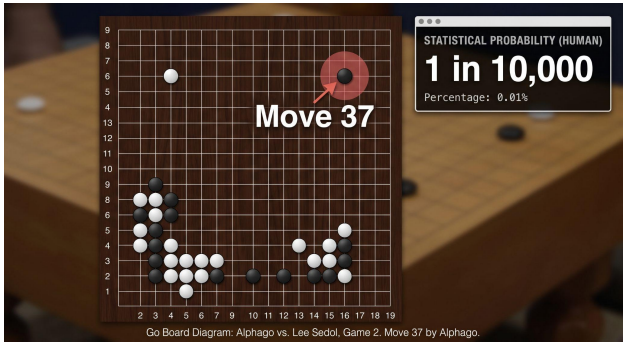
Move 37!

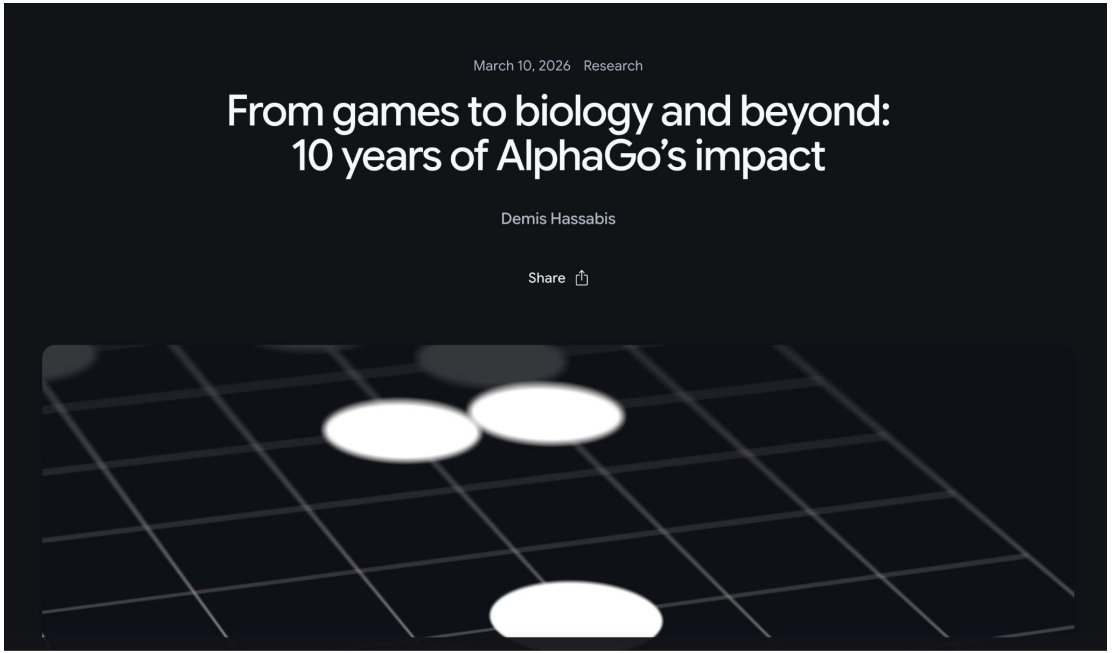


Go Board Diagram: Alphago vs. Lee Sedol, Game 2. Move 37 by Alphago.

"With a single creative play, the famous 'Move 37,' AlphaGo demonstrated the potential of AI and signaled that we now had the techniques to begin tackling real-world scientific problems." - Demis Hassabis

Move 37!Move 37: From Games to Real-World Science





"With a single creative play, the famous 'Move 37,' AlphaGo demonstrated the potential of AI and signaled that we now had the techniques to begin tackling real-world scientific problems." - Demis Hassabis

The Evolution of System Optimization



Auto-tuning hardware parameters / DSE Sweeps (Vizier, OpenTuner)



Copilots generating Verilog / C++ kernels (local code assistance and optimization engine)

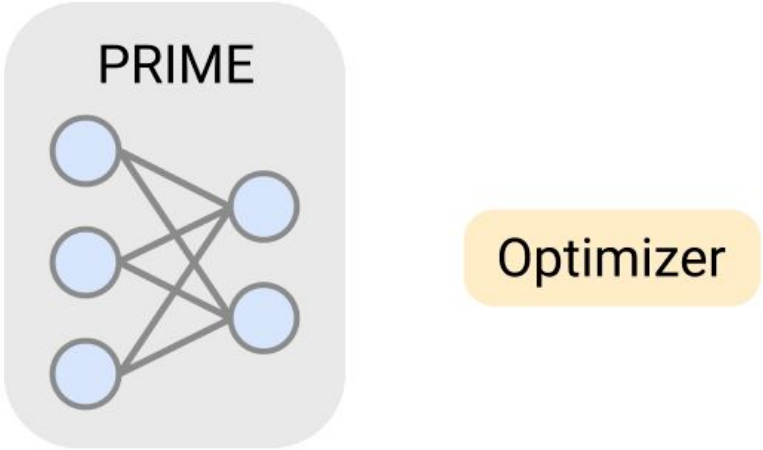


End-to-end autonomous agents discovering novel compiler passes, custom hardware-software interfaces, etc.

Past

Auto-tuning hardware parameters / DSE Sweeps (Vizier, OpenTuner)





Offline Optimization for Architecting Hardware Accelerators

Recent

Copilots generating Verilog / C++ kernels (local code assistance and optimization engine)



Moore's Law

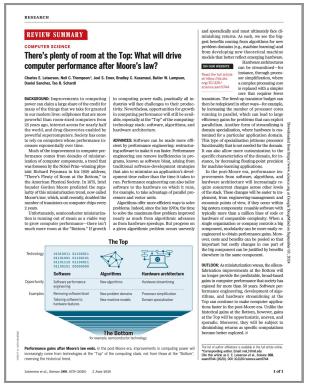
Diminishing returns

Code optimization

Soaring profits (xB$)



There is plenty of room at the top!



RESEARCH

REVIEW SUMMARY

COMPUTER SCIENCE

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson, Neil C. Thompson*, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl

BACKGROUND: Improvements in computing power can claim a large share of the credit for many of the things that we take for granted in our modern lives: cellphones that are more powerful than supercomputers of just a decade ago, internet access for nearly half the world, and drug discoveries enabled by powerful supercomputers. Society has come to rely on computers whose performance improves exponentially over time.

Much of the improvement in computer performance comes from decades of miniaturization of integrated circuits—a trend that was foreseen by the Nobel Prize–winning physicist Richard Feynman in his 1959 address, There's Plenty of Room at the Bottom. At the American Physical Society in 1975, Intel founder Gordon Moore predicted the regular, exponential miniaturization trend, which Moore's Law holds will eventually double the number of transistors on computer chips every 2 years.

Unfortunately, semiconductor miniaturization is running out of steam as a viable way to grow computer performance—there isn't much more room at the "Bottom." If growth in computing power stalls, practically all industries will face challenges to their productivity. Nevertheless, opportunities for growth in computing performance will still be available "at the Top," in the form of algorithms, technology stacks, software, algorithms, and hardware architecture.

ADVANCES: Software can be made more efficient by performance engineering: restructuring software to make it run faster. Performance engineering can remove inefficiencies in programs, which sometimes arise as collateral damage from layered software development strategies that aim to minimize an application's development time by composing it from existing components. Performance engineering can also tailor software to the hardware on which it runs, for example, to exploit the parallel resources of multicore and vector units.

Algorithms offer more efficient ways to solve problems. Indeed, since the late 1970s, the time to solve the maximum-flow problem improved nearly as much as computer performance did, as Tardos hardware speedups. But progress on a given algorithmic problem occurs unevenly and sporadically and may ultimately face diminishing returns. New problems and new applications often arise, however, as the nature of computing changes. Advances in machine learning have been enabled by algorithms for high-dimensional gradient descent, which was discovered in the mid-Cauchy and has been rediscovered several times since.

ADVANCES: Hardware architecture can be streamlined—for instance, through processor simplification, where a complex processing core is replaced with a simpler core that requires fewer transistors. The freed-up transistor budget can then be redeployed in other ways—for example, by increasing the number of processor cores running in parallel, which can lead to large gains for parallel code while hardly affecting serial performance. Another form of streamlining is domain specialization, where hardware is customized to a particular domain, such as graphics. This type of specialization jettisons processor functionality that is not needed for the domain. It can also allow more customization to the specific characteristics of the domain, for example, by decreasing floating-point precision for machine-learning applications.

In the post-Moore era, performance improvements from software and hardware architecture will increasingly vary across different problem domains, depending on the characteristics of each domain. From an engineering-management and economic points of view, if they occur within big system components (consider software with typically more than a million lines of code or hardware with billions of transistors), then a single organization or company controls a big component, workability can be more easily ensured, modularity can be preserved, and costs and benefits can be pooled so that important but costly changes in one part of the big component can be justified by benefits elsewhere in the same component.

OUTLOOK: As miniaturization wanes, the effective fabrication improvements at the Bottom will be fewer and farther between, but performance gains in computer performance due to algorithms, software, and hardware architecture will continue to be available at the Top for many years. Software performance engineering, development of algorithms, and hardware streamlining at the Top can all be practiced hand in hand with improvements at the Bottom. The extent of performance improvements at the Top, however, is likely to be much smaller than the historical gains at the Bottom, however, gains at the Top will be more frequent across application domains. Moreover, they will be subject to diminishing returns as specific computations become better explored.

The Top

Technology

Opportunity

Examples

Software

Algorithms

Hardware architecture

Software performance engineering

New algorithms

Hardware streamlining

Removing software bloat

Taking software to hardware limits

New problem domains

New machine models

Processor simplification

Domain specialization

The Bottom

Performance gains after Moore's law ends. At the past Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom," reversing the historical trend.

Leiserson et al., Science 368, 1079 (2020) 5 June 2020

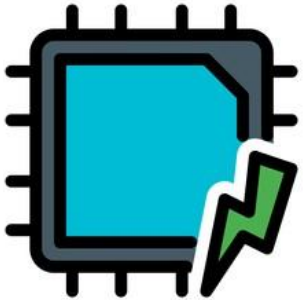
1 of 1

Why Code Optimization for Hyperscale Data Centers?

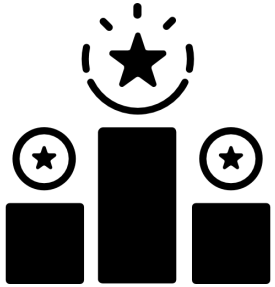






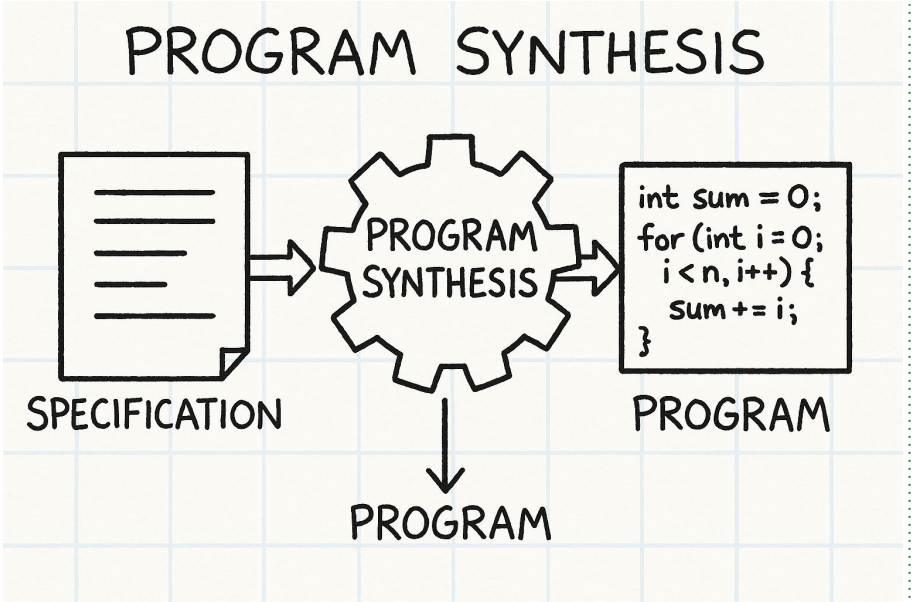




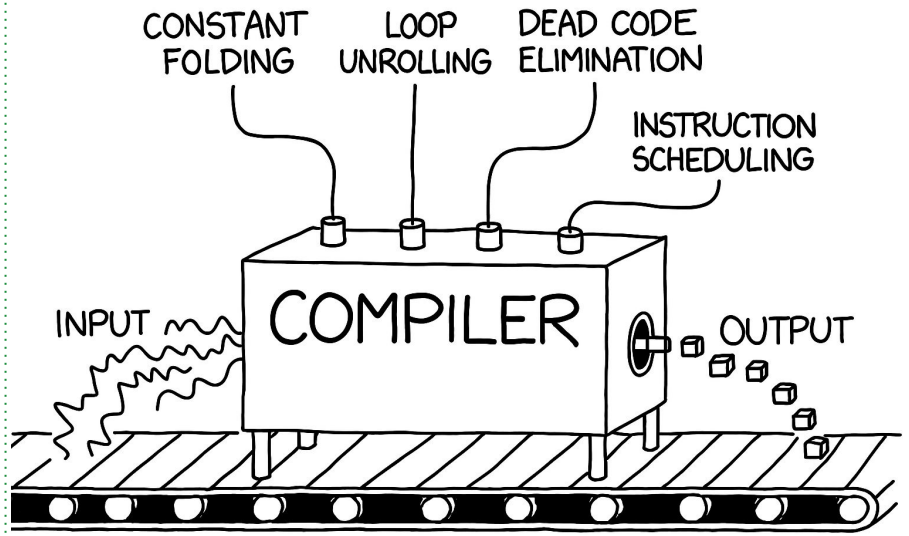


Conventional Optimization ToolBox

Program Synthesis? Compiler?



Scalability and Domain Specific Insights



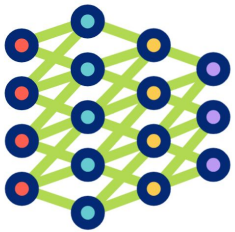
Predefined Rules and Heuristics

Can GenAI be effective at program optimization?

hello-world-snail.cpp

```cpp
int main(int argc, char* argv[]) {
    int n = atoi(argv[1]);
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    printf("Total: %s\n", sum);
    return 0;
}
```





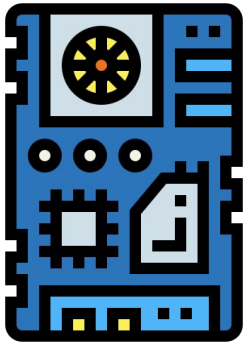
hello-world-rocket.cpp

```cpp
int main(int argc, char* argv[]) {
    int n = atoi(argv[1]);
    int sum = n * (n + 1) / 2;
    printf("Total: %d\n", sum);
    return 0;
}
```



Building a Deterministic Evaluation Environment



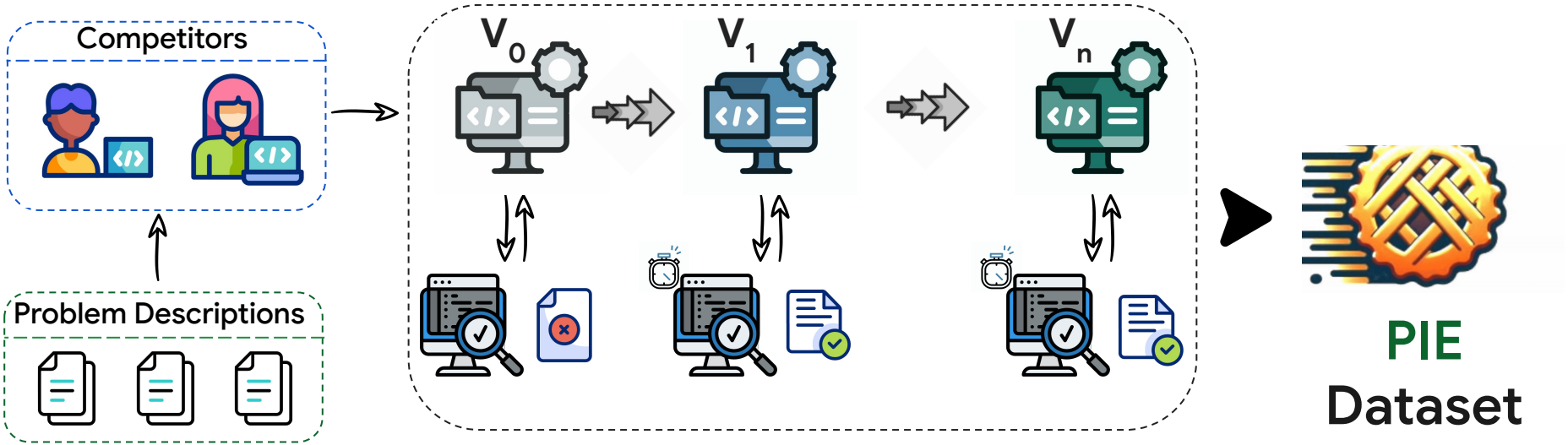


- Capable of full system simulator
- Modeling detailed microarchitecture
- Extensible to different microarchitecture and system details
- Docker based and accessible API for reproducibility



In this work, we use Skylake CPU configuration

PIE: Dataset for Performance Optimization Edits



~77K train and ~1K unique tests

>42 Million CPU simulations

Evaluation Metrics

Generated Performance Optimization Edits

%Opt

The fraction of programs improved by a certain method. A program must be at least 10% faster and correct to contribute.

%Correct

The proportion of programs in the test set that are at least functionally equivalent to the original program.

Speedup

The absolute improvement in running time. Speedup(c1, c2) = Runtime(c1)/Runtime(c2). A program must be correct to contribute.

Are Pre-Trained LLMs Effective at Program Optimization?



CodeLlama 34B

Speedup: **1.30 ×**

GPT 3.5 and GPT 4

Speedup: **1.60 ×**



Human Reference

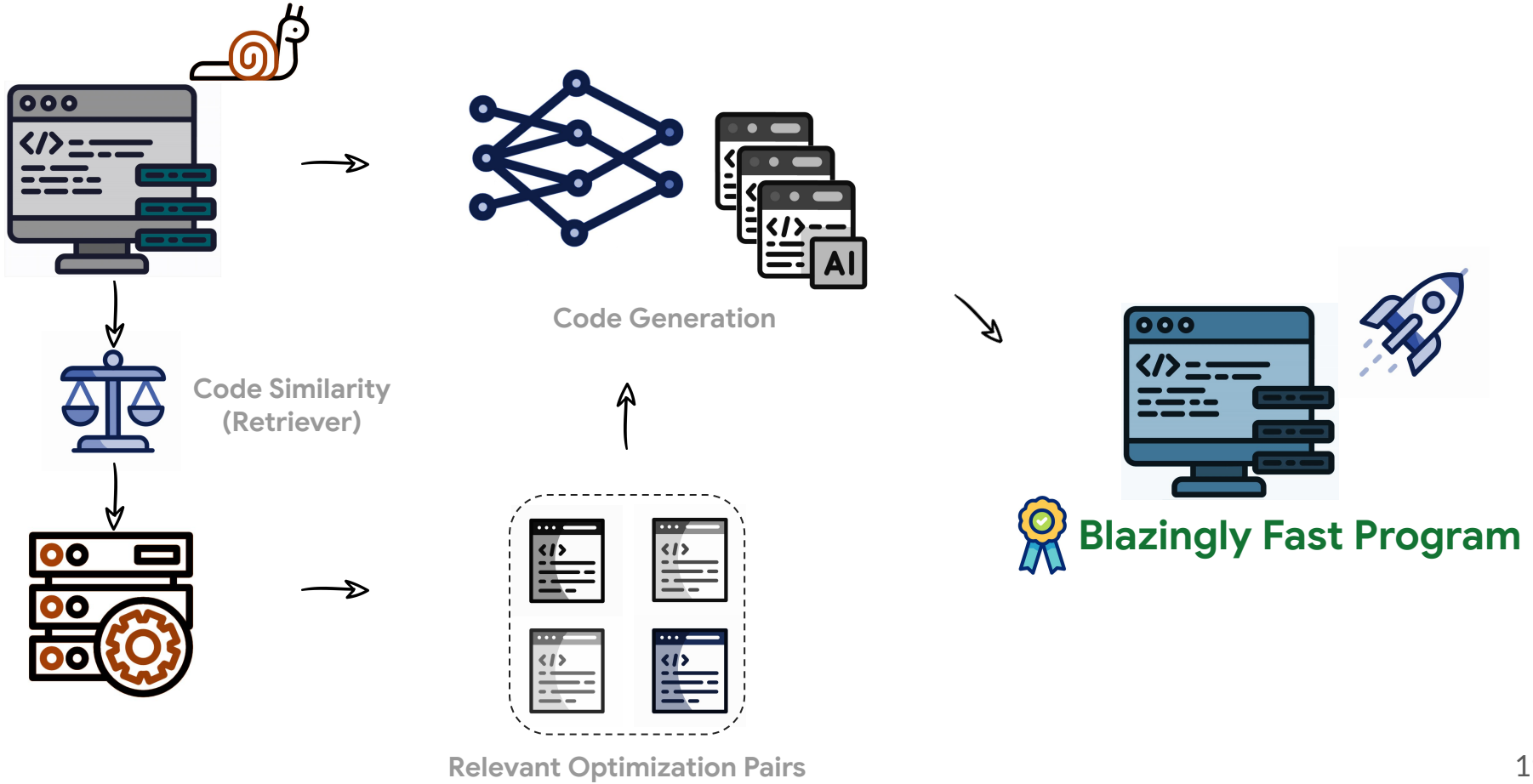
Speedup: **3.66 ×**



Significant Gap

Significant Improvements with Retrieval

Retrieval-Based Program Optimization



Retrieval-Based Program Optimization

GPT4 Results (Best@8)



	Dynamic Retrieval K = 2	Dynamic Retrieval K = 4
%OPT	74.74%	76.07%
%CORRECT	97.85%	95.71%
SPEEDUP	3.95x	3.93x

Does using Similar Code in Prompts Improve Performance?



CodeLlama 34B

Speedup: 2.43 ×

GPT 4

Speedup: 3.93 ×



Human Reference

Speedup: 3.66 ×



Often Faster!

Fine-Tuning with Data Augmentation

Performance-Conditioned and Self-Play



CodeLlama 13B

Speedup: **5.65 X**

Performance-Conditioned

GPT 3.5

Speedup: **6.86 X**

Self-Play



Human Reference

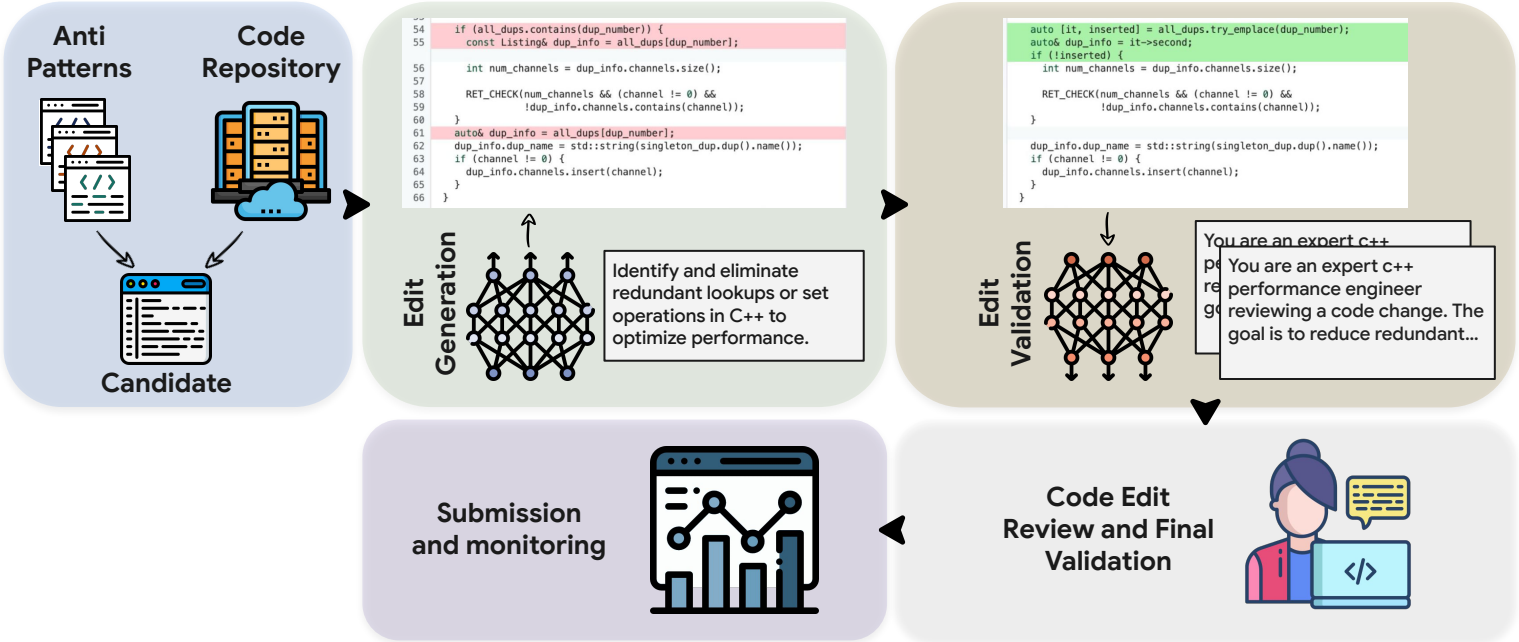
Speedup: **3.66 X**

Significant Improvements!

Deploying Performance Optimization System at Scale



Building Performance Optimization System at Scale



Anti-Pattern Categories for Performance Edits

- **Alloc** — Unnecessary allocations
- **Args** — Unnecessary copied args
- **Copy** — Unnecessary copies
- **Map** — Missing std::moves
- **Sort** — Unnecessary sorts
- **Vector** — Missing vector reserves

Copy

```
const shopping::MerchantServices merchant_services = merchant_view.merchant_services();
for (const auto& merchant_service : merchant_services.services()) {
  if (merchant_service.id() == global_merchant_id) {
    return true;
  }
}
```

Map

```
if (activity->type() ==
  activity::ActivityType::FEED_CARD_CLICK) {
  languages[*lang].total_activity++;
} else if (activity->type() ==
  activity::ActivityType::FEED_CARD_VIEW) {
  languages[*lang].views++;
  if (duration_in_seconds >= kLongViewDurationSeconds) {
    languages[*lang].total_activity++;
  }
}
```

Vector

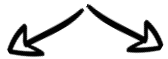
```
std::vector<std::string> output_columns;
for (const Key& key : input.keys()) {
  output_columns.push_back(FormatKey(key));
}
for (const Value& value : input.values()) {
  output_columns.push_back(FormatValue(value));
}
```

Microbenchmark Code Optimization Samples

Original code snippet with an unnecessary copy anti-pattern.

original.cpp

```
std:string CreateSimpleNames(
      absl:flat_hash_map<std::string, Loc> map, std::string key) {
    Loc value = map.at(key);
    if (value.locations.empty()) {
        return “simple_name”;
    }
    return SimpleStrName(value.locations[0]);
}
```



correct_edit.diff

```
@@
std:string CreateSimpleNames(
       absl:flat_hash_map<std::string, loc> map, std::string key) {
-    Loc value = map.at(key);
+    const Loc& value = map.at(key);
     if (value.locations.empty()) {
           return “simple_name”;
       }
}
```

Sample of a successful model generated edit

wrong_edit.diff

```
@@
std:string CreateSimpleNames(
-      absl:flat_hash_map<std::string, loc> map, std::string key) {
+      const absl::flat_hash_map<absl::string_view, Loc>& map,
+      absl::string_view key) {
     Loc& value = map.at(key);
     if (value.locations.empty()) {
           return “simple_name”;
```

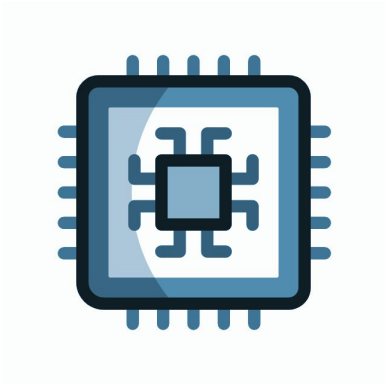
The proposed type change requires updating the file includes but this was not added by the LLM

Feedback from Human and Automated Code Review

		Copy	Map	Vector
Submitted	Submitted	39.97%	4.86%	40.89%
	Submitted User Dis.	6.55%	10.49%	15.99%
Rejected	Reject Regression	0.16%	0.14%	0.12%
	Reject Test	16.14%	37.39%	20.07%
	Reject User	35.81%	46.55%	22.87%
	Reject Failure	0.00%	0.07%	0.00%
	Reject Other	1.37%	0.49%	0.05%
Savings	Total Commit	4,959	1,421	4,035
	Normalized Core Savings	331k	125k	618k

Performance Optimization Impact at Scale

Impact Evaluation



500k Normalized Cores
per Quarter



6,400 Submitted Changes
25k Lines of Code Edits
Production Code



>99.5%
Success Rate

Present

In Pursuit of AlphaZero of System Design

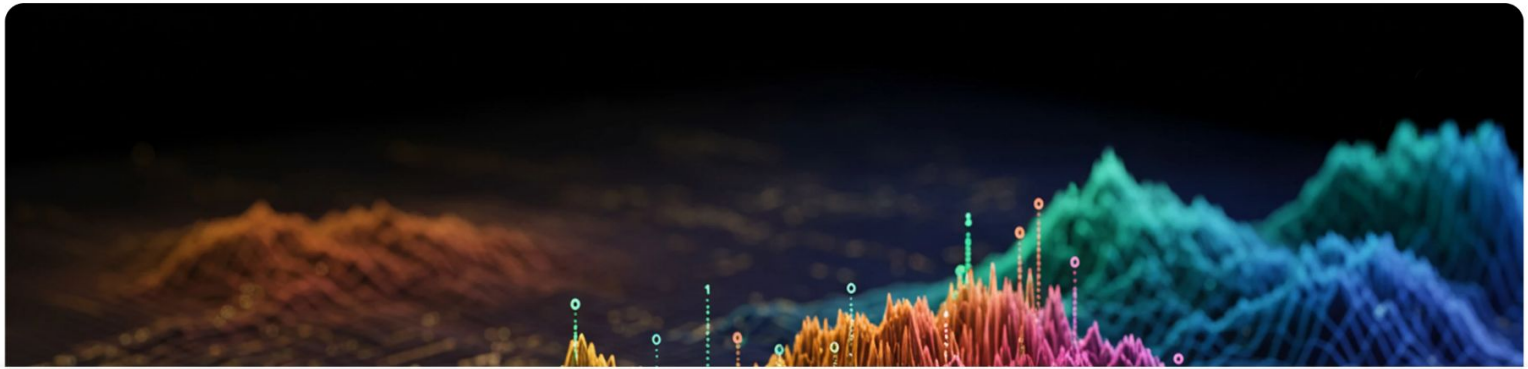


May 14, 2025 Science

AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms

AlphaEvolve team

Share

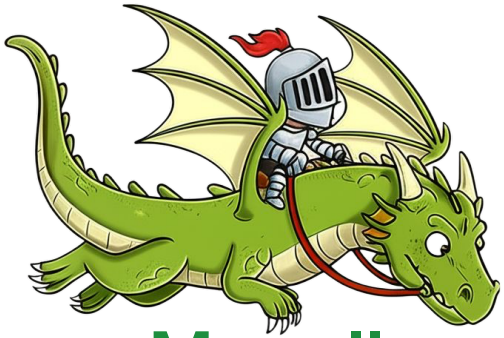


AlphaEvolve in a Nutshell

Specification



Evolutionary Search to select the best program



Magellan: Autonomous Discovery of Novel Compiler Optimization Heuristics with AlphaEvolve

Hongzheng Chen[1,3], Alexander Novikov[2], Ngân (NV) Vũ[2], Hanna Alam[1], Zhiru Zhang[3], Aiden Grossman[1], Mircea Trofin[1], Amir Yazdanbakhsh[2]

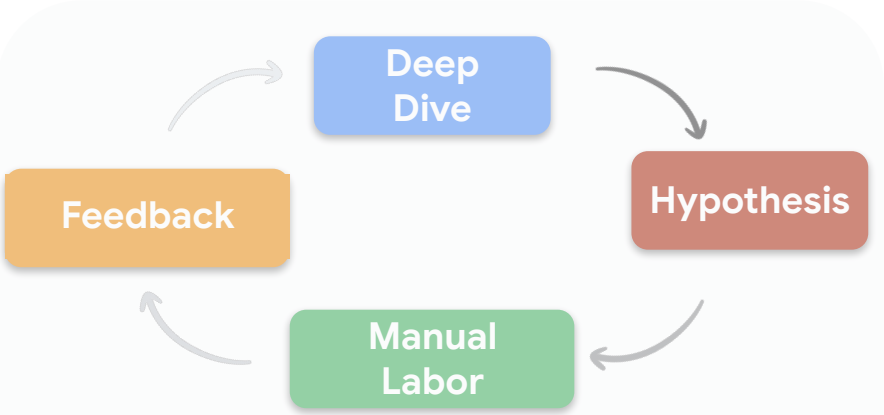
[1] Google, [2] Google DeepMind, [3] Cornell University

ML-Guided (compiler) Optimizations (aka MLGO)

- **Compiler optimization is a big $ lever for fleet efficiency**
- **ML can help with some compiler optimization problems**
- Last ~5 years, ML-Guided Opt in LLVM:
  - Inlining for size
  - Register Allocation (perf)
  - Successfully deployed and used in Fuchsia OS, mobile Chrome, Android apps, and instrumented FDO data center, etc.

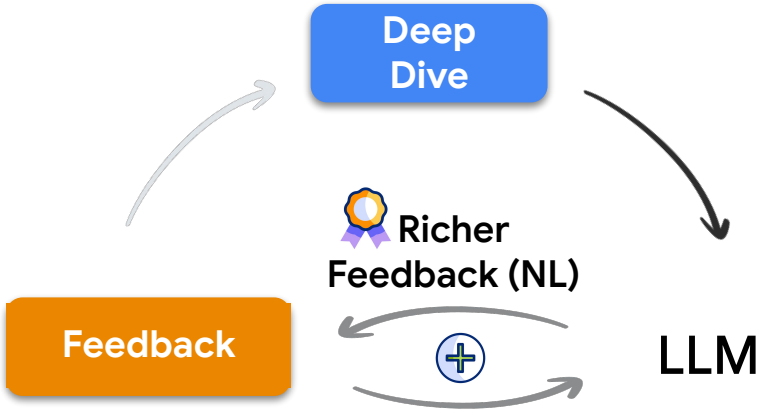
How LLM accelerates heuristic optimization discovery?

The Conventional Workflow



Slow, iterative, and labor-intensive

The LLM-in-the-loop Workflow

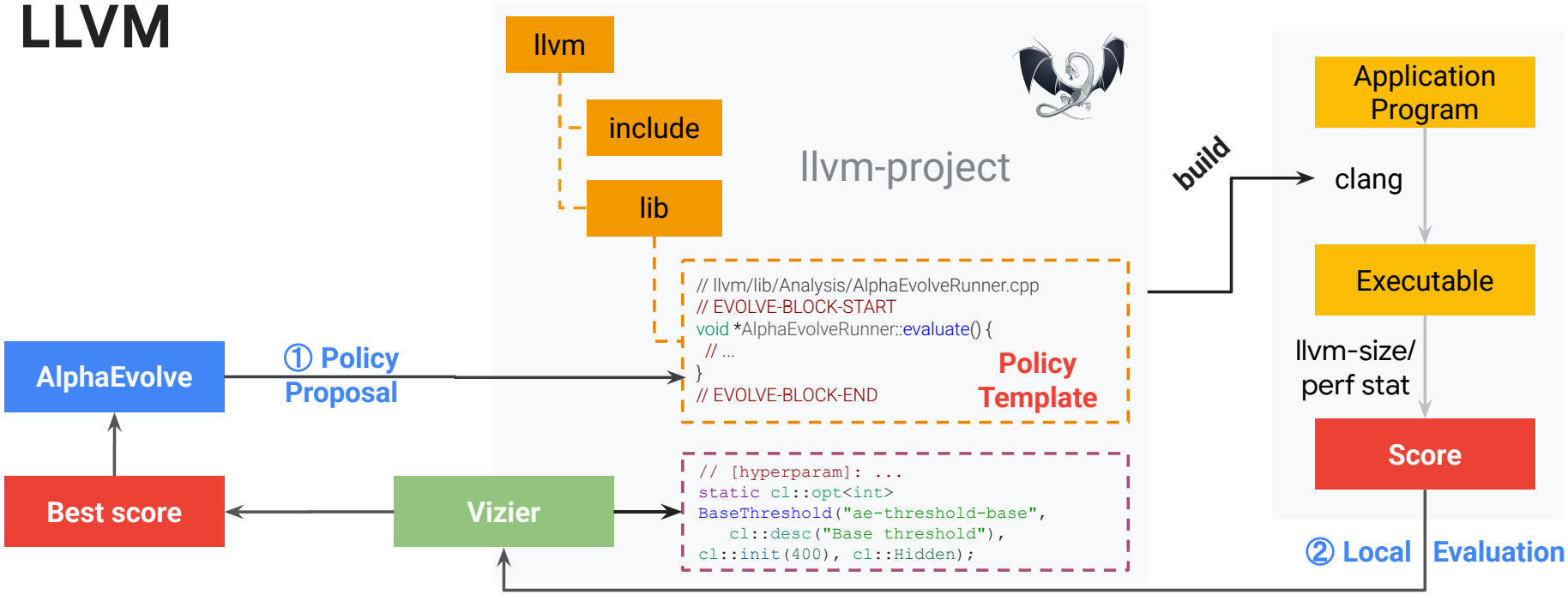


Faster discovery & better sample efficiency

LLM acts as a force multiplier, improving the compiler engineers productivity and facilitating the novel optimization heuristics discovery.

③ Hyperparam ④ Feedback
Reward (logs, profiling results, etc.)

System Architecture for Heuristic Discovery in LLVM



The system works in an autonomous loop: A two-stage pipeline that decouples high-level policy design (LLM-driven) from low-level parameter optimization (executed via an autotuner).

Key Compiler Optimization Problems under LLVM & XLA

- **LLVM: Inlining for binary size reduction**
  - This is the pilot - unambiguous reward signal used in MLGO
  - Currently in production, used by Chrome on mobile, Fuchsia, Android Google Search App
  - **Good candidate for 1st milestone with AlphaEvolve**
- **LLVM: Inlining for performance**
  - Using NNs is difficult b/c modeling performance reward is challenging for datacenter
- **LLVM: Register allocation**
  - Priority function that determines the order in which live ranges should be processed
- **XLA: Graph rewrite**
  - Determine the optimal tensor sharding strategy for a given deep learning model inside a distributed TPU environment
- **XLA: Auto-sharding**
  - Determine the optimal rewrite rules for a given computation graph to achieve high performance

Case Study I: Function Inlining for Size (w/o autotuner)

1. Partial heuristic based on predefined feature sets
   a. Same input features as NN
   b. Output boolean inlining decision
   c. Implement evaluateUntyped inherited from MLInlineAdvisor

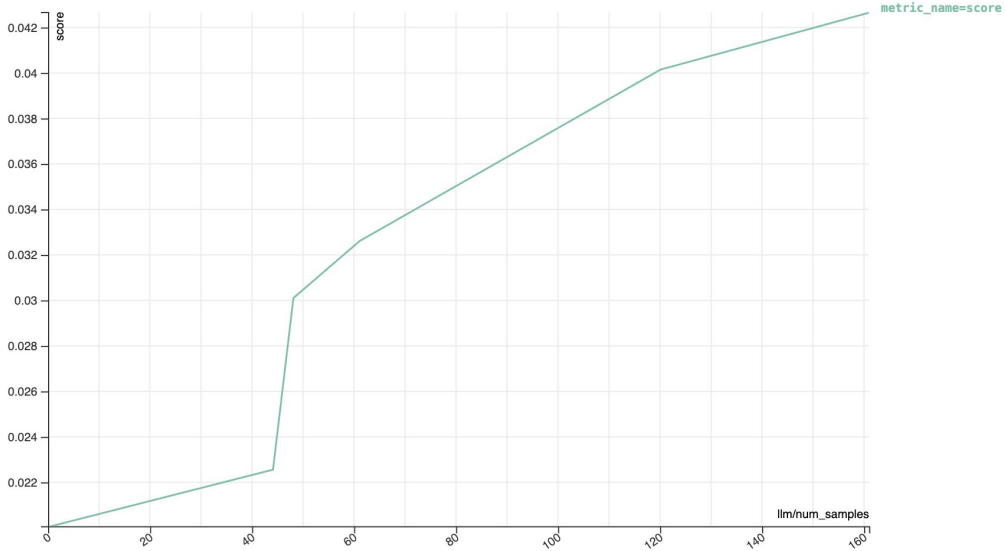
```
// llvm/lib/Analysis/AlphaEvolveRunner.cpp
// EVOLVE-BLOCK-START
void *AERunner::evaluateUntyped() {
  int64_t CallerUsers =
     *getTensor<int64_t>(FeatureIndex::caller_users);
  int64_t CalleeUsers =
     *getTensor<int64_t>(FeatureIndex::callee_users);
  // ...
}
// EVOLVE-BLOCK-END
```

2. Full heuristic based on arbitrary LLVM API calls
   a. Input the CallBase CB
   b. Implement getAdviceImpl inherited from InlineAdvisor
   c. Evolved code after correctness checks

```
// llvm/lib/Analysis/AlphaEvolveRunner.cpp
// EVOLVE-BLOCK-START
std::unique_ptr<InlineAdvice>
AEInlineAdvisor::getAdviceImpl(CallBase &CB) {
   bool IsInliningRecommended = false;
   Function *Callee = CB.getCalledFunction();
   Function *Caller = CB.getCaller();
   // ...
}
// EVOLVE-BLOCK-END
```

Case Study I: Function Inlining for Size (w/o autotuner)

- Automatically composed predefined feature sets to form a heuristic
  - Evaluated on the internal search engine and started from a naïve policy
  - Iterated for two days, score consistently improved

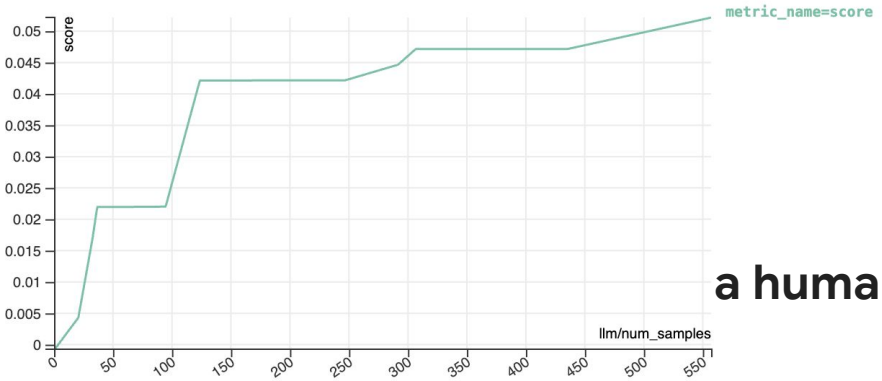



```
std::unique_ptr<InlineAdvice> AEInlineAdvisor::getAdviceImpl
(CallBase& CB) {
  bool IsInliningRecommended = false;
  return std::make_unique<InlineAdvice>(
     this, CB,
     FAM.getResult<OptimizationRemarkEmitterAnalysis>
     (*CB.getCaller()), IsInliningRecommended);
}
```

4.27% better than the upstream LLVM heuristic, iterating for 1.5 days!

Case Study I: Function Inlining for Size (w/o autotuner)

- Automatically discovered the entire heuristic based on arbitrary LLVM API calls
  - Evaluated on internal search application and started from a naïve heuristic
  - Require more trials and errors compared to giving pre-defined features
  - **Production Ready** ⟶ The generated heuristic is clean C++ that is ready to be directly deployed upstream with minimal manual intervention

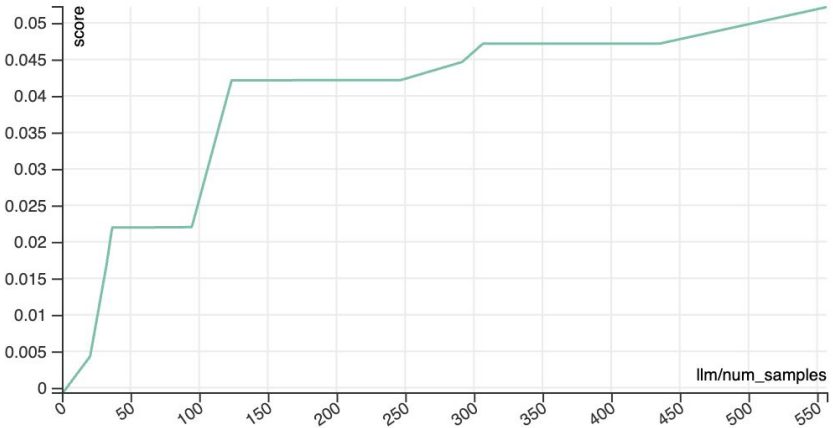


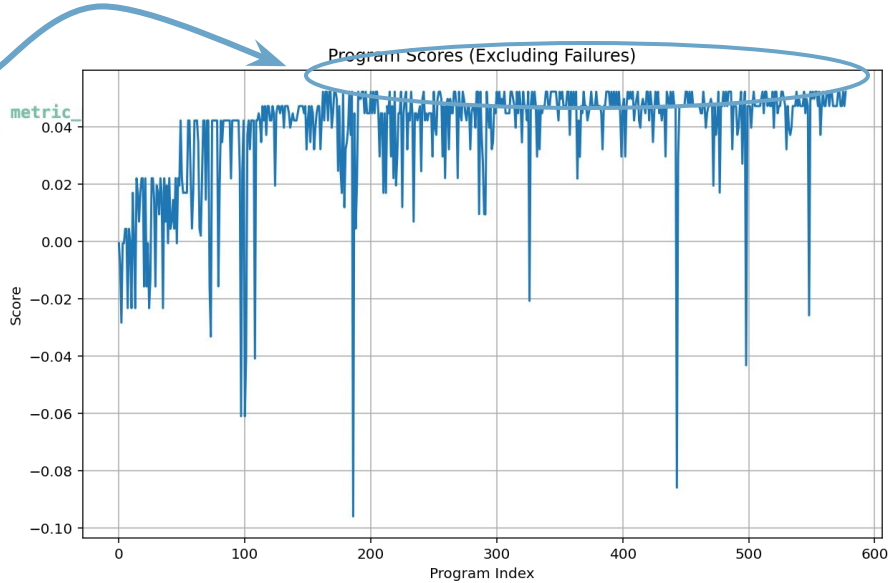
5.23% better than

a human-developed heuristic after 1.5 days!

Case Study I: Function Inlining for Size (w/o autotuner)

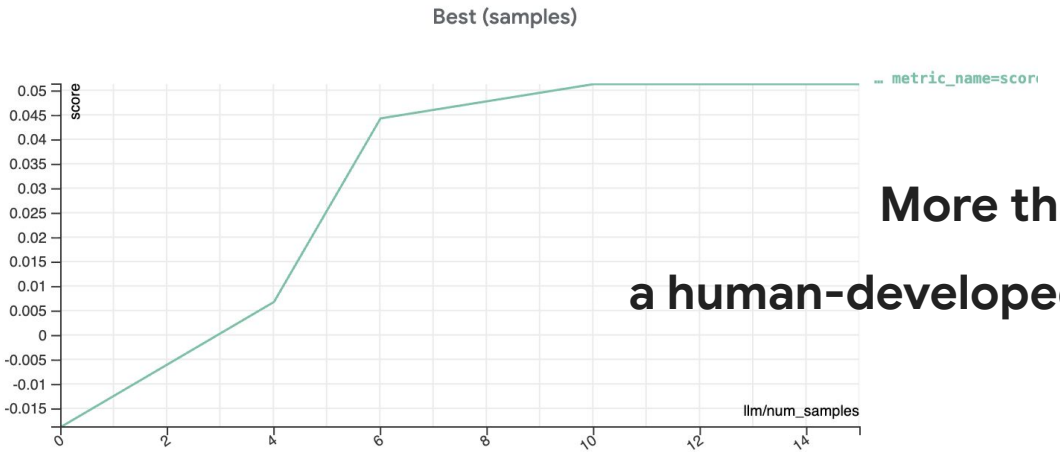
- Further analysis on the workflow w/o autotuner
  - Total: 1616 samples; Failed compilation: 1036 samples
  - **More than <u>65%</u>** of programs are invalid!
  - After ~180 iterations, the score got stuck





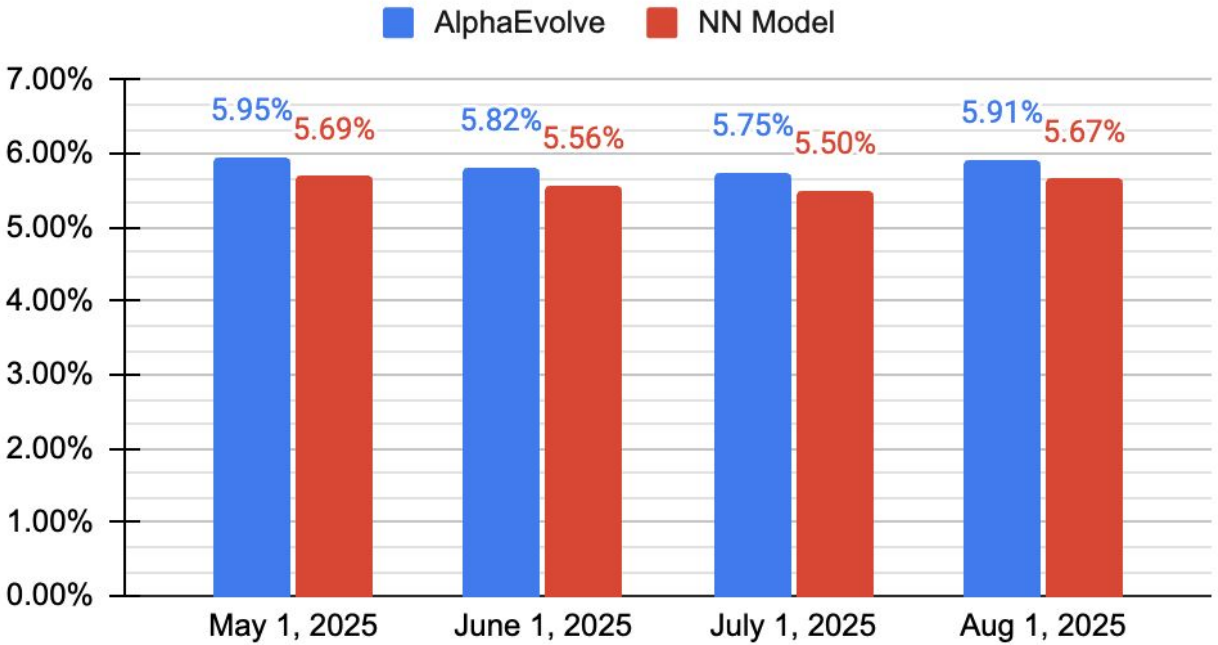
Case Study I: Function Inlining for Size (w/ Autotuner)

- Use the workflow combining AlphaEvolve+Vizier
  - 15 iterations * 10 Vizier samples / iter
  - Significant boost on program **sampling efficiency**!
  - **Only 2/15 ≈ 13%** of programs are invalid! **5x** reduction compared w/ previous one
  - Only **5 hours** to achieve the same level of size reduction! **7x** time reduction



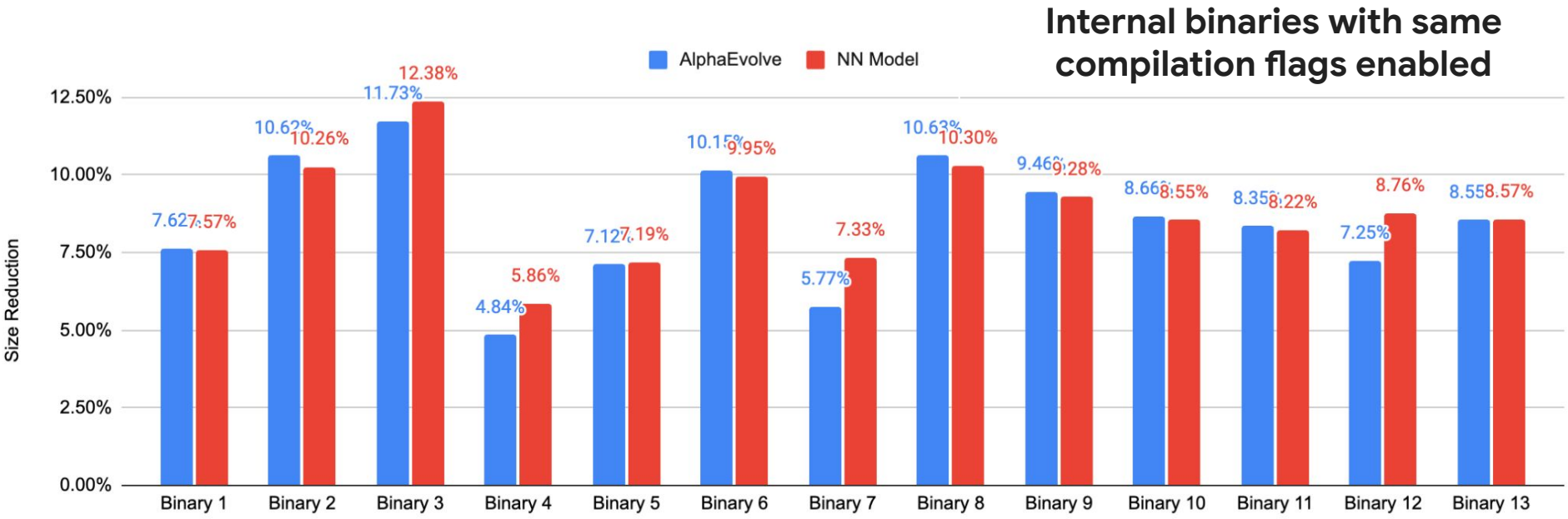
More than +5% better than a human-developed heuristic with only 5 hours!

Case Study I: Function Inlining for Size
Temporal Generalization



- Evaluated on four different timestamps with one-month interval
- Compared w/ a two-year old TensorFlow NN model (we didn't retrain it!)

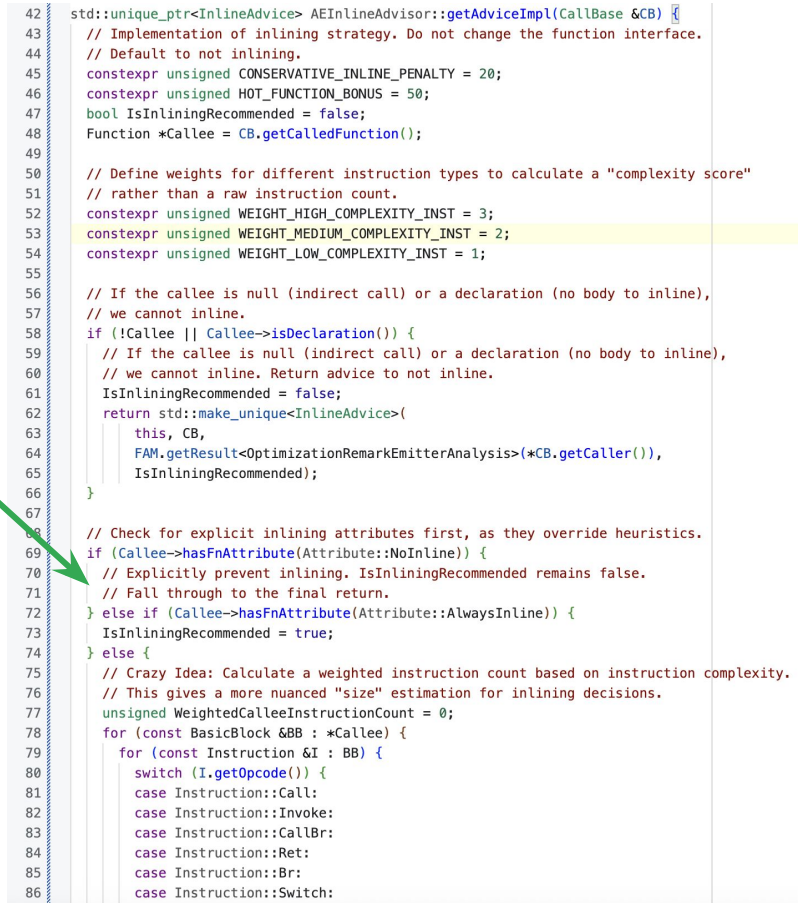
Case Study I: Function Inlining for Size
Domain Generalization



On average (10+ production applications), AlphaEvolve achieves a 8.79% size reduction on average, on par with the NN model's 8.52%.

AlphaEvolve-Generated Code Deep Dive

- Human-readable C++ code
- Comments with rationale



```cpp
std::unique_ptr<InlineAdvice> AEInlineAdvisor::getAdviceImpl(CallBase &CB) {
  // Implementation of inlining strategy. Do not change the function interface.
  // Default to not inlining.
  constexpr unsigned CONSERVATIVE_INLINE_PENALTY = 20;
  constexpr unsigned HOT_FUNCTION_BONUS = 50;
  bool IsInliningRecommended = false;
  Function *Callee = CB.getCalledFunction();

  // Define weights for different instruction types to calculate a "complexity score"
  // rather than a raw instruction count.
  constexpr unsigned WEIGHT_HIGH_COMPLEXITY_INST = 3;
  constexpr unsigned WEIGHT_MEDIUM_COMPLEXITY_INST = 2;
  constexpr unsigned WEIGHT_LOW_COMPLEXITY_INST = 1;

  // If the callee is null (indirect call) or a declaration (no body to inline),
  // we cannot inline.
  if (!Callee || Callee->isDeclaration()) {
    // If the callee is null (indirect call) or a declaration (no body to inline),
    // we cannot inline. Return advice to not inline.
    IsInliningRecommended = false;
    return std::make_unique<InlineAdvice>(
        this, CB,
        FAM.getResult<OptimizationRemarkEmitterAnalysis>(*CB.getCaller()),
        IsInliningRecommended);
  }

  // Check for explicit inlining attributes first, as they override heuristics.
  if (Callee->hasFnAttribute(Attribute::NoInline)) {
    // Explicitly prevent inlining. IsInliningRecommended remains false.
    // Fall through to the final return.
  } else if (Callee->hasFnAttribute(Attribute::AlwaysInline)) {
    IsInliningRecommended = true;
  } else {
    // Crazy Idea: Calculate a weighted instruction count based on instruction complexity.
    // This gives a more nuanced "size" estimation for inlining decisions.
    unsigned WeightedCalleeInstructionCount = 0;
    for (const BasicBlock &BB : *Callee) {
      for (const Instruction &I : BB) {
        switch (I.getOpcode()) {
        case Instruction::Call:
        case Instruction::Invoke:
        case Instruction::CallBr:
        case Instruction::Ret:
        case Instruction::Br:
        case Instruction::Switch:
```

AlphaEvolve-Generated Code Deep Dive

- ~15x shorter implementation but better code size reduction
  - Only counts the actual implementation lines – comments and empty lines are removed

	Manual Heuristic	**AlphaEvolve**
LoC of the policy	2115	143

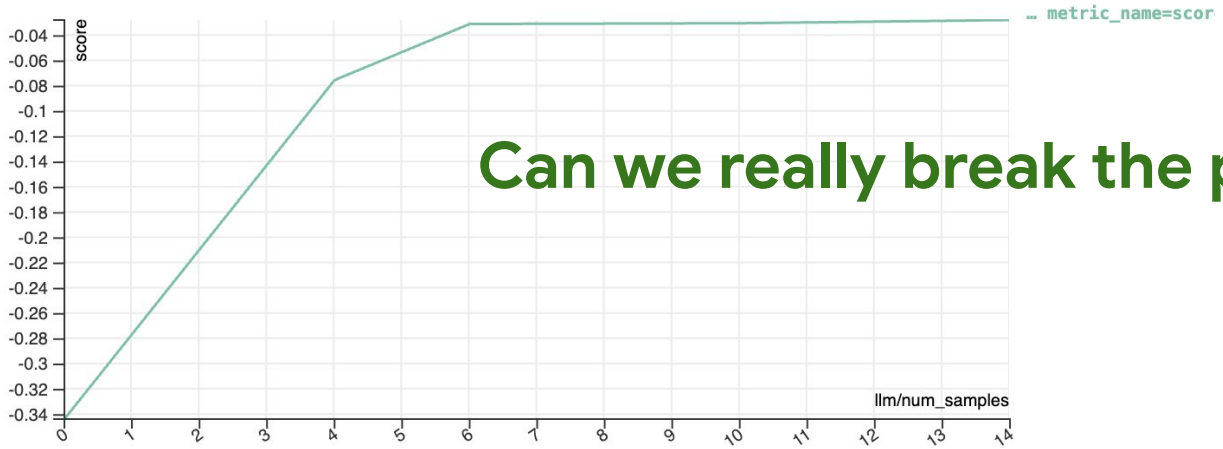
- Key insights discovered by AlphaEvolve
  - Inlining the right functions instead of simply avoiding code growth (in InlineCost.cpp)
    - Aggressively inline single-use functions
    - Prioritize readonly functions that enable substantial dead code elimination after inlining
  - Weighted complexity model that penalizes control-flow instructions more heavily

At this point, we thought...

AlphaEvolve appears to have a much higher sampling complexity than NN techniques.
We (hope to) leapfrog the reward-shaping problem in ML for compiler optimizations.

Case Study II: Function Inlining for Performance

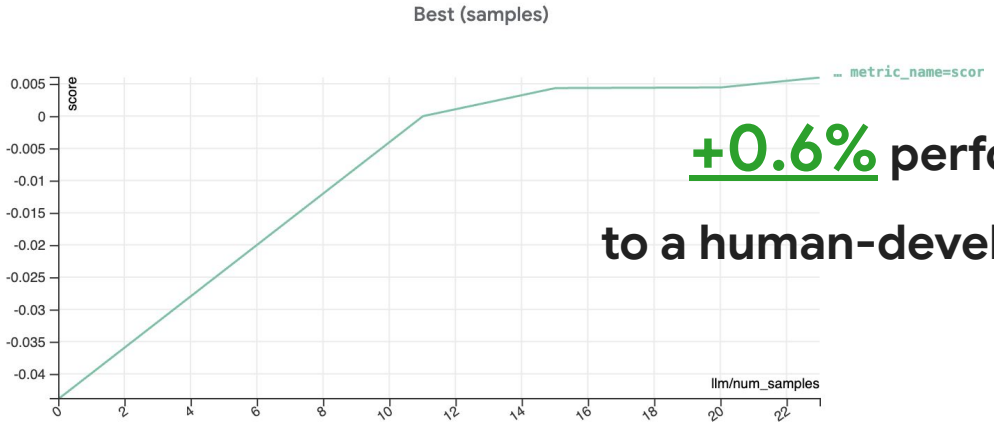
- Used AlphaEvolve+Vizier to evaluate the performance problem on clang *(-O3, ThinLTO, instrumented FDO. Also, **very** overfitted: clang compiles 1 large module)*
  - Tried more than 2 months w/ different prompting techniques w/o luck
  - Started from a naive policy directly returning False
  - -34% regression to -2.7% regression but still cannot get positive results



Can we really break the performance ceiling?

Case Study II: Function Inlining for Performance

- Finally **broke the performance ceiling**!
  - **First time** achieving performance improvement using ML for this problem *at this problem complexity and from a "best practices" baseline*
  - ①Used the clang compiler as the benchmark
  - ②Started from a policy with -4.7% regression obtained by Gemini-2.5-Pro
  - 23 iterations * **40** Vizier samples / iter (3 days)
  - 7 hours per iteration, 4 parallel samples, avg 30 min per sample



+0.6% performance improvement compared to a human-developed heuristic w/ Gemini-3-Pro! ③

Trade Offs

✅ Improved compiler engineers productivity (at least for new heuristics)

- No need to manually train a new model
- Use macro benchmarks!
- Sparse and time-consuming reward & slow evaluation is ok (to some extent)
- Only need hundreds of samples (even for end-to-end metrics) to approximate SOTA
- Land the generated code as if manually-written

❌ We haven't done much study on the convergence / performance ceiling aspect

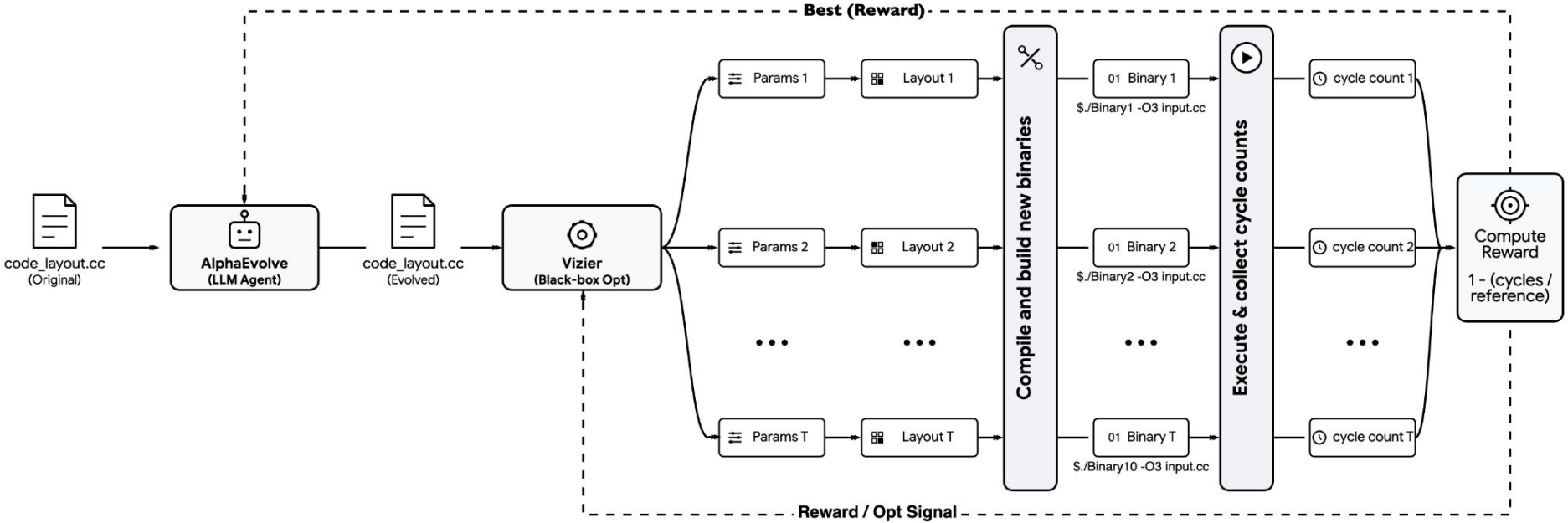
- The performance ceiling of AlphaEvolve is an open research question, despite it provided similar performance improvements as NNs.

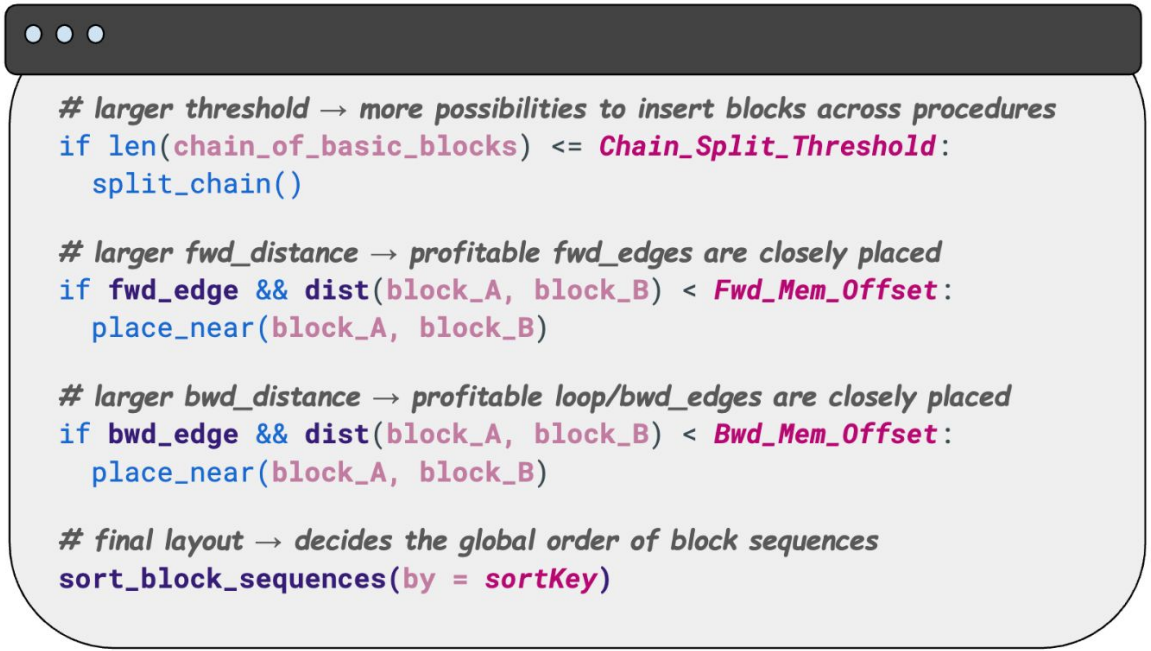


AI-PROPELLER: Warehouse-Scale Interprocedural Code Layout Optimization with AlphaEvolve

Chaitanya Mamatha Ananda, Rajiv Gupta, Mircea Trofin, Aiden Grossman, Sriraman Tallam, Xinliang Li, Amir Yazdanbakhsh

Google, UC Riverside, Google DeepMind


Best (Reward)
code_layout.cc
(Original)
AlphaEvolve
(LLM Agent)
code_layout.cc
(Evolved)
Vizier
(Black-box Opt)
Params 1
Layout 1
Params 2
Layout 2
Params T
Layout T
Compile and build new binaries
01 Binary 1
$./Binary1 -O3 input.cc
01 Binary 2
$./Binary2 -O3 input.cc
01 Binary T
$./Binary10 -O3 input.cc
Execute & collect cycle counts
cycle count 1
cycle count 2
cycle count T
Compute Reward
1 - (cycles / reference)
Reward / Opt Signal



```
# larger threshold → more possibilities to insert blocks across procedures
if len(chain_of_basic_blocks) <= Chain_Split_Threshold:
  split_chain()

# larger fwd_distance → profitable fwd_edges are closely placed
if fwd_edge && dist(block_A, block_B) < Fwd_Mem_Offset:
  place_near(block_A, block_B)

# larger bwd_distance → profitable loop/bwd_edges are closely placed
if bwd_edge && dist(block_A, block_B) < Bwd_Mem_Offset:
  place_near(block_A, block_B)

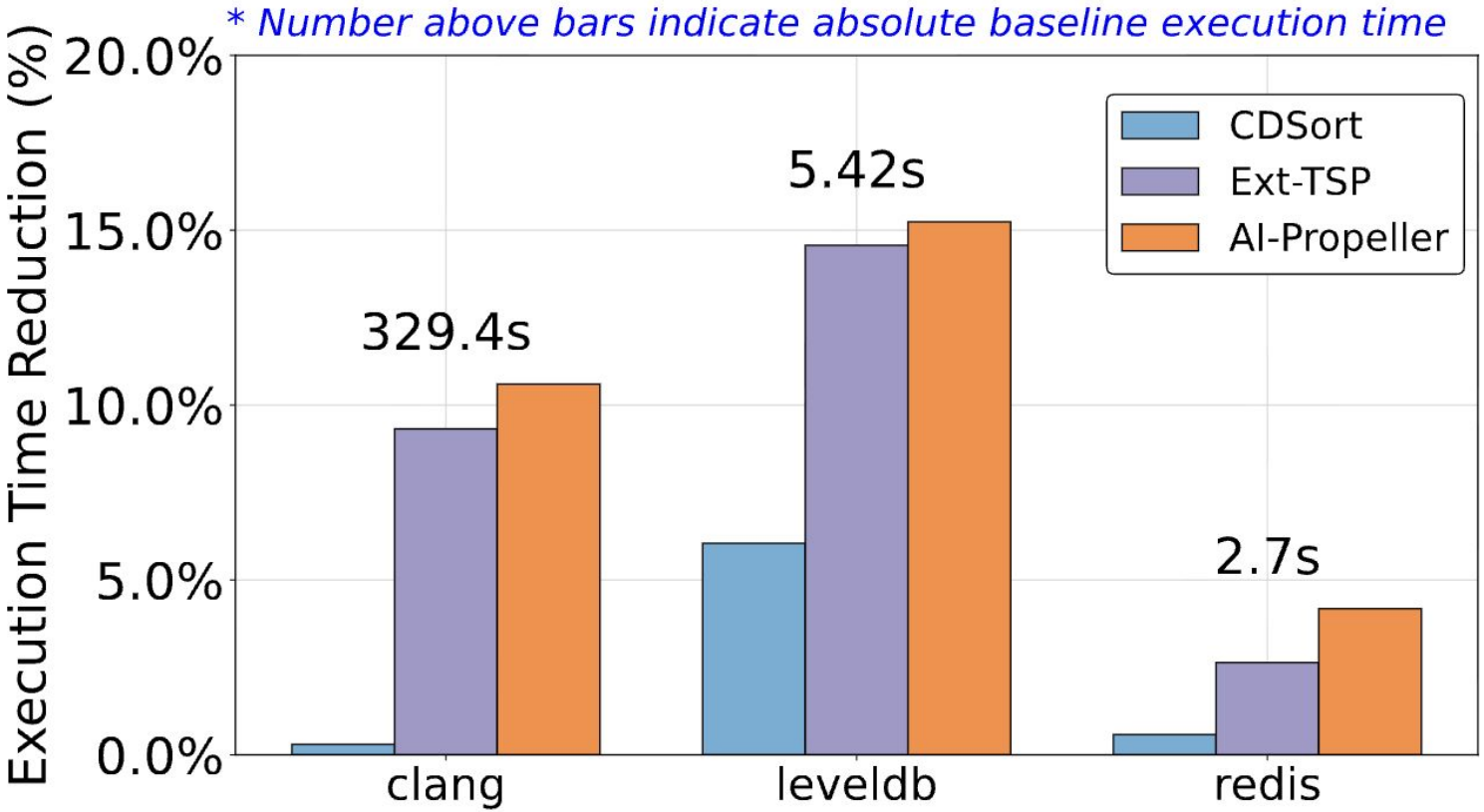
# final layout → decides the global order of block sequences
sort_block_sequences(by = sortKey)
```

Before

$Chain\_Split\_Threshold \leftarrow 0$
$Fwd\_Mem\_Offset \leftarrow 1024$
$Bwd\_Mem\_Offset \leftarrow 680$
$sortKey \leftarrow firstBlockID$

Our Approach

$Chain\_Split\_Threshold \leftarrow 213$
$Fwd\_Mem\_Offset \leftarrow 3059$
$Bwd\_Mem\_Offset \leftarrow 1159$
$sortKey \leftarrow globalSequenceID$


* Number above bars indicate absolute baseline execution time
Execution Time Reduction (%)
20.0%
15.0%
10.0%
5.0%
0.0%
329.4s
5.42s
2.7s
CDSort
Ext-TSP
AI-Propeller
clang
leveldb
redis

Takeaways & Future Work

- **Automated Discovery & Productivity Gain ⟶** AlphaEvolve boosts productivity by automating the labor-intensive process of heuristic design.
- **Human-Competitive Results ⟶** The system consistently generates heuristics that perform on par with, and can sometimes surpass, those created by human experts.
- **Faster iteration pipeline ⟶** The incorporation of autotuner boosts sampling efficiency.



Next steps:

- **Scale up evaluations:** Explore the boundaries of AlphaEvolve and see if it has signals on convergence.
- **Tackle "Green-Field" Problems:** Apply AlphaEvolve (or its variants) to novel compiler domains to evaluate its ability to innovate where little or no prior human expertise exists.
- **Open-source Implementation:** Upstream the skeleton code of our infrastructure to LLVM.

Challenges to find Move 37 for system design!

Can we ever have “Move 37” moment in system design? Maybe!

- Current models excel at interpolating from known architectural patterns but struggle with the structural, first-principles reasoning required to invent entirely novel paradigms.
- Unlike the definitive win/loss rules of Go or the formal proofs of mathematics, system design lacks a perfect "ground truth". In most cases, evaluating a design requires navigating noisy, competing trade-offs across latency, throughput, power, and area.
- AlphaGo mastered a finite grid. System design operates in a near-infinite, continuous, and highly dimensional space that spans hardware microarchitecture, compiler behavior, and application-level code.
- An agent needs millions of rapid iterations to discover a "Move 37." In system design, high-fidelity architectural simulations are notoriously slow and computationally expensive, breaking the rapid feedback loop required for autonomous discovery.
- The rules of board games are static. In system design, the foundational constraints shift constantly, meaning an agent is always aiming at a moving target.
- An agent designing a system rarely has perfect information. Unobservable variables like thermal throttling, network congestion, and OS-level jitter make it difficult for AI to definitively prove that a novel optimization is universally better.

Google Agentic Ecosystem to Accelerate Research

Google’s unified platform for builders to work with agents

Product May 19, 2026

Google Antigravity @ I/O 2026

The Antigravity Team



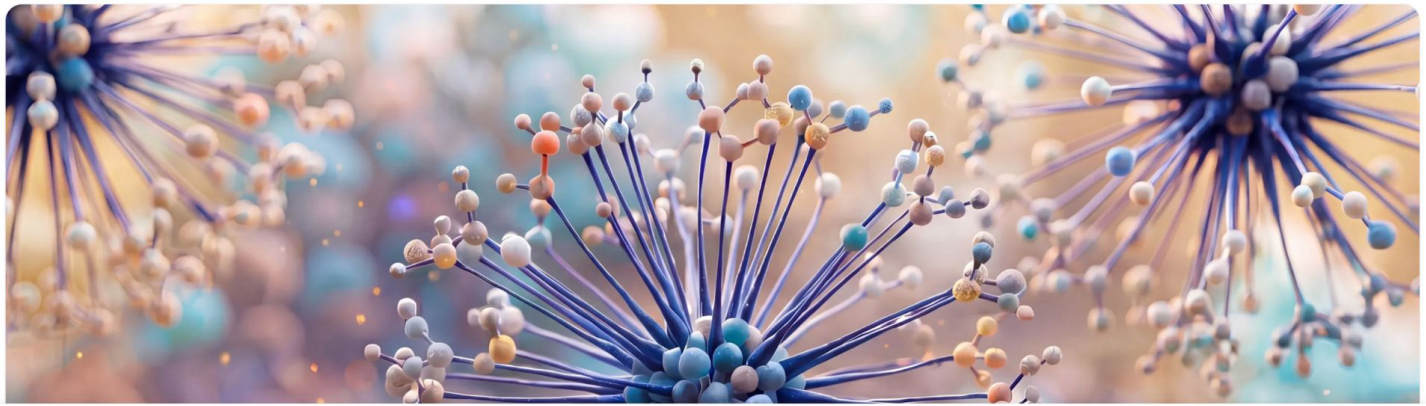
Redefining Research Workflow

May 19, 2026 Science

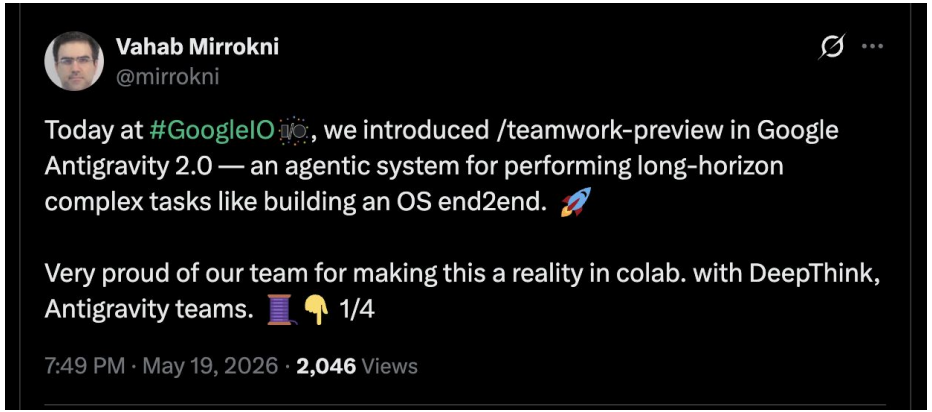
Co-Scientist: A multi-agent AI partner to accelerate research

Co-Scientist team

Share



Teamwork Google Antigravity



The Operating System...

We'll save the suspense. We went straight to building an operating system from scratch. It worked. It took 93 subagents, 15,314 model calls, and over 339M input tokens (with cache reads, output, and thinking, that number goes to over 2.6B tokens). We could play Doom on it. But the code wasn't particularly the exciting part.

It was done using Gemini 3.5 Flash *from a single prompt*.

This (a) goes to show just how intrinsically intelligent the Gemini family of models have gotten, even on Flash (Gemini 3.1 Pro was unable to do this), and (b) suggests that a much more economically feasible productionisation of agent teams is not that far away. At API pricing, building this OS would cost $916.92.

The General Setup

Let's break down our approach to orchestrating these agent teams. We won't pretend that this was just a couple lines of prompts (it ended up being many thousands of lines), but the critically exciting piece to note is that we only ended up reusing core agent primitives that are available today in Antigravity 2.0, such as parallel-running subagents, asynchronous tasks, and hooks. There was no special version of the product used to do this.

Instead of a single agent wearing many hats, we ended up creating a series of subagent types with specialized goals and constraints:

- The Sentinel — the "front-desk manager" of the project that does not write code, analyze logs, or makes technical decisions. It focuses entirely on structuring user intent, spawning the Orchestrator, and supervising overall task completion.
- The Orchestrator — a dispatch-only manager, i.e. never writes code or executes builds itself. Focuses on decomposing requirements into milestones, kicking off other specialized subagents, and synthesizing reports.
- The Explorer — analyzes requirements and previous logs to write formal strategies for the Orchestrator to act on, but never writes code itself.
- The Worker — the actual coder that implements the strategies, builds the code, and runs tests.
- The Reviewer — independently reviews the Worker's changes for design correctness, edge cases, and interface contract compliance.
- The Critic — stress-tests the solution, runs adversarial tests to find gaps in coverage.
- The Auditor — an independent investigator that verifies the authenticity and robustness of the generated solutions.

Teamwork Google Antigravity

Systems

An example in CPU simulation

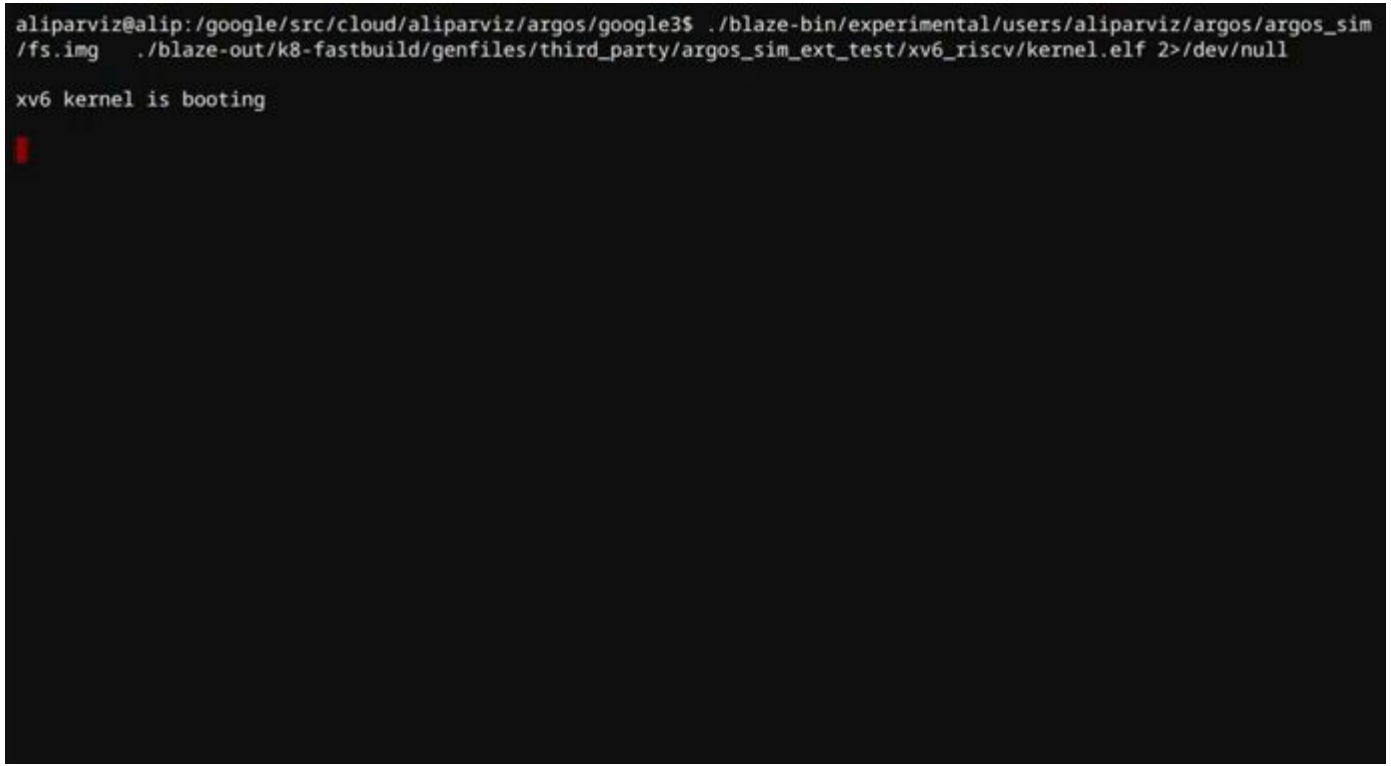
RISC-V simulator that boots an operating system

Using **Gemini 3.7 Flash**, Teamwork built a highly-accurate cycle-level out-of-order (OoO) RISC-V CPU simulator from scratch that successfully boots an operating system to shell (xv6) and successfully simulates 100+ RISC-V standard benchmarks. Teamwork achieves this goal through two separate phases: (1) Microarchitectural Functional Correctness and (2) Cycle-Level Timing Validation. In the first phase, the agents develop the core execution logic, including out-of-order (OoO) pipelines and Reorder Buffers (ROB). This phase focused on ensuring the simulator could boot an OS and maintain correct architectural state when validated against a functional oracle. In the second phase, Teamwork aligns performance characteristics with strict timing oracles.

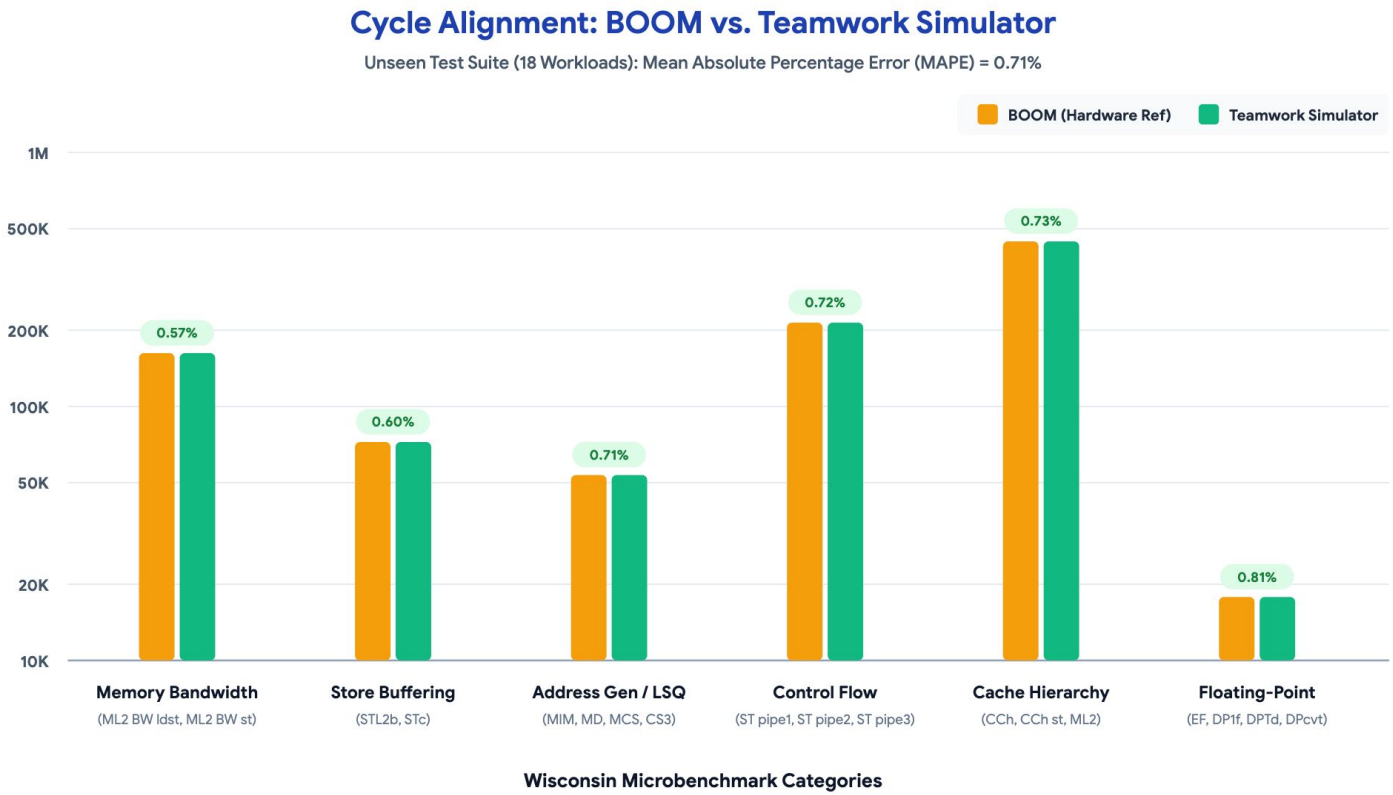
Teamwork autonomously builds and verifies different microarchitectural units (e.g. MSHR, ROB, Cache, etc.) by executing architectural microbenchmarks and reasoning about its own generated architectural traces and comparing the gap between final cycle counts of an air-gapped Boom simulator and Teamwork generated solution. Validated against BOOM hardware execution ground truth, the Teamwork Simulator achieved an average cycle alignment error of 0.71% on unseen test workloads.

We developed a first-of-its-kind, cycle-level RISC-V CPU simulator with minimum human invention (< 1% average error vs. golden reference) and successfully booting an OS to shell.

Teamwork Google Antigravity



Teamwork Google Antigravity



Personal Reflections

Life and Research

1. **Measure What Matters** → Don't be seduced by easy metrics or fads.
2. **Defy the Herd** → Industry often chases the obvious; academia's role is to explore what isn't yet commercializable.
3. **Follow Your Compass**→ isn't thinking about you. Pursue what you're passionate about, not what someone else says is hot.
4. **Embrace the Serendipity** → Luck is real. You can't force it, but you can improve your odds by putting yourself in new environments
5. **Kindness is a Scaling Factor** → Publications, grants, and promotions are transient. The relationships you build last longer. Lift others up; share credit.
6. **Sprinkle Some Noise** → Inject controlled randomness into your projects and your life. The unknown can reveal hidden anomalies.
7. **Forge Your Own Path** → There is no single "best" way to do research. What worked for someone else might be the wrong approach for you.
8. **Mentor as a Force Multiplier** → Help others find their passion. Seek out mentees, share your tools, and create opportunities. The view from the summit is better when shared.
9. **Assume Good Intent** → Almost everyone around you wants to help, even Reviewer #2. Be open to feedback, and on being wrong (TED: On Being Wrong!), ask questions, and engage with your community.
10. **Build Your Own Measurement Toolkit** → Don't rely solely on existing benchmarks or tools. Create or adapt frameworks that reveal your inefficiencies.
11. **Don't work for Awards** → Work for the work. Awards are a nice outcome.
12. ...

