

Magellan: Autonomous Discovery of Novel Compiler Optimization Heuristics with AlphaEvolve

Hongzheng Chen, Alexander Novikov, Ngān Vũ
Mircea Trofin, Amir Yazdanbakhsh

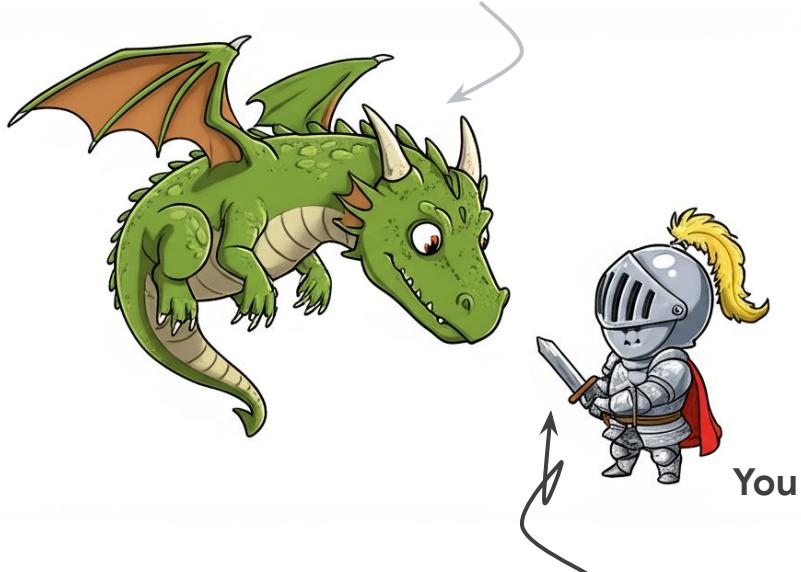


Google DeepMind

LLVM Developers' Meeting

10/28/25

The compiler optimization problems
that haunt your dreams



Your painstakingly
handcrafted heuristics



It's 2025!!!

Let's use **LLMs** to slay the dragon!

ML-Guided _(compiler) Optimizations (aka MLGO)

- Compiler optimization is a big \$ lever for fleet efficiency
- ML can help with some compiler optimization problems
- Back in 2021...
 - Inlining for size, register allocation
 - Successfully deployed and used in Fuchsia OS, mobile Chrome, Android apps, and instrumented FDO data center, etc.

arXiv:2101.04808v1 [cs.PL] 13 Jan 2021

MLGO: a Machine Learning Guided Compiler Optimizations Framework

Mircea Trofin*
Google, Inc.
mtrofin@google.com

Zinan Lin
Carnegie Mellon University
zinanl@andrew.cmu.edu

Yundi Qian*
Google, Inc.
yundi@google.com

Krzysztof Choromanski
Google, Inc.
kchoro@google.com

Eugene Brevdo
Google, Inc.
ebrevdo@google.com

David Li
Google, Inc.
davidxl@google.com

Abstract

Leveraging machine-learning (ML) techniques for compiler optimizations has been widely studied and explored in academia. However, the adoption of ML in general-purpose, industry strength compilers has yet to happen.

We propose MLGO¹, a framework for integrating ML techniques systematically in an industrial compiler – LLVM. As a case study, we present the details and results of replacing the heuristics-based inlining-for-size optimization in LLVM with machine learned models. To the best of our knowledge, this work is the first full integration of ML in a complex compiler pass in a real-world setting. It is available in the main LLVM repository.

We use two different ML algorithms: Policy Gradient and Evolution Strategies, to train the inlining-for-size model, and achieve up to 7% size reduction, when compared to state of the art LLVM - Oz. The same model, trained on one corpus, generalizes well to a diversity of real-world targets, as well as to the same set of targets after months of active development. This property of the trained models is beneficial to deploy ML techniques in real-world settings.

1 Introduction

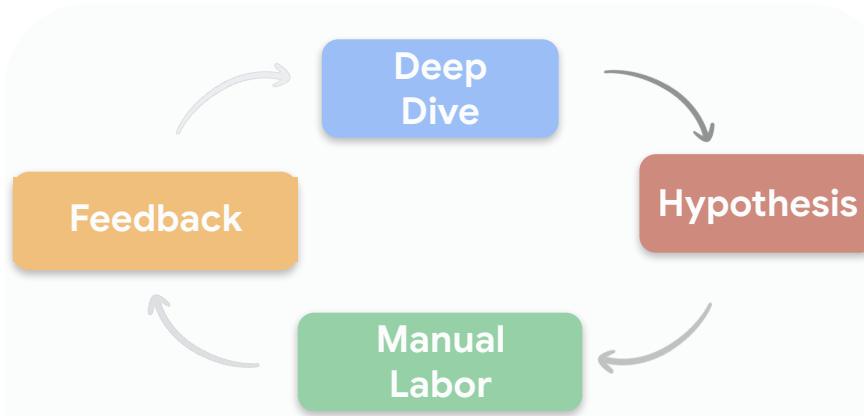
Previous work [13, 25] has shown promise in replacing compiler optimization heuristics with machine-learned policies. Heuristics are algorithms that, empirically, produce reasonably optimal results for hard problems, within pragmatic constraints (e.g. "reasonably fast"). In the compiler case, heuris-

Our focus is ahead-of-time (AOT) compilers, specifically, C/C++. In a real-world setting, we expect two main benefits from machine learning techniques: first, heuristics are human-trained based on a human-manageable set of benchmarks and regression cases. Machine learning easily scales to large corpora of training examples - which we expect to increase the likelihood of obtaining policies that generalize well. This is important because, as we will explore in detail, we do not want to retrain policies too frequently (it is an adoption blocker), nor do we want to train 'online', while the compiler is running in production (it would affect determinism). Second, heuristics are human-written code that needs to be maintained. This places a downward pressure on the number of program properties ("features") and the combinations between them that can be practically leveraged. We believe using more features and feature combinations would result in better optimization decisions. ML scales well with the addition of features, and can discover profitable feature combinations. While ML techniques may be able to address these two points, a trade-off is that maintaining and evolving them requires practices and approaches different from those used for heuristics.

As pointed out, applying ML to compiler optimizations has been explored by academia, but it has not been adopted in production environments. To explore why, we chose a pilot optimization problem and approached it with the intention to deploy in production. The goal of the pilot is to inform problem framing and design choices. Other than performing better than the tip-of-tree production compiler, we did not

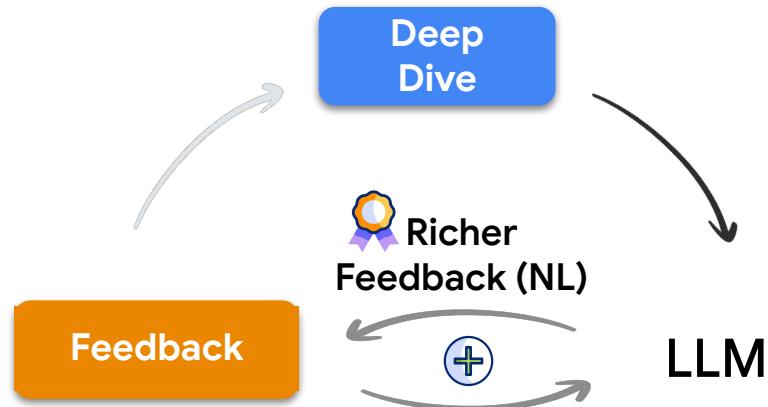
How LLM accelerates heuristic optimization discovery?

The Conventional Workflow



Slow, iterative, and labor-intensive

The LLM-in-the-loop Workflow

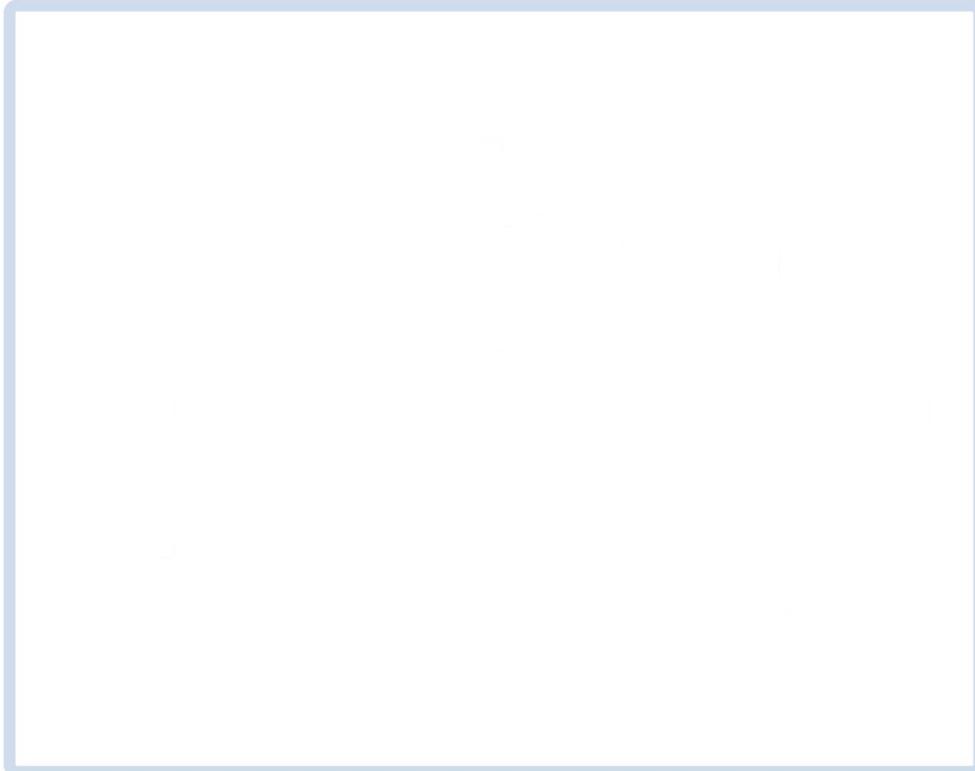


Faster discovery & better sample efficiency

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

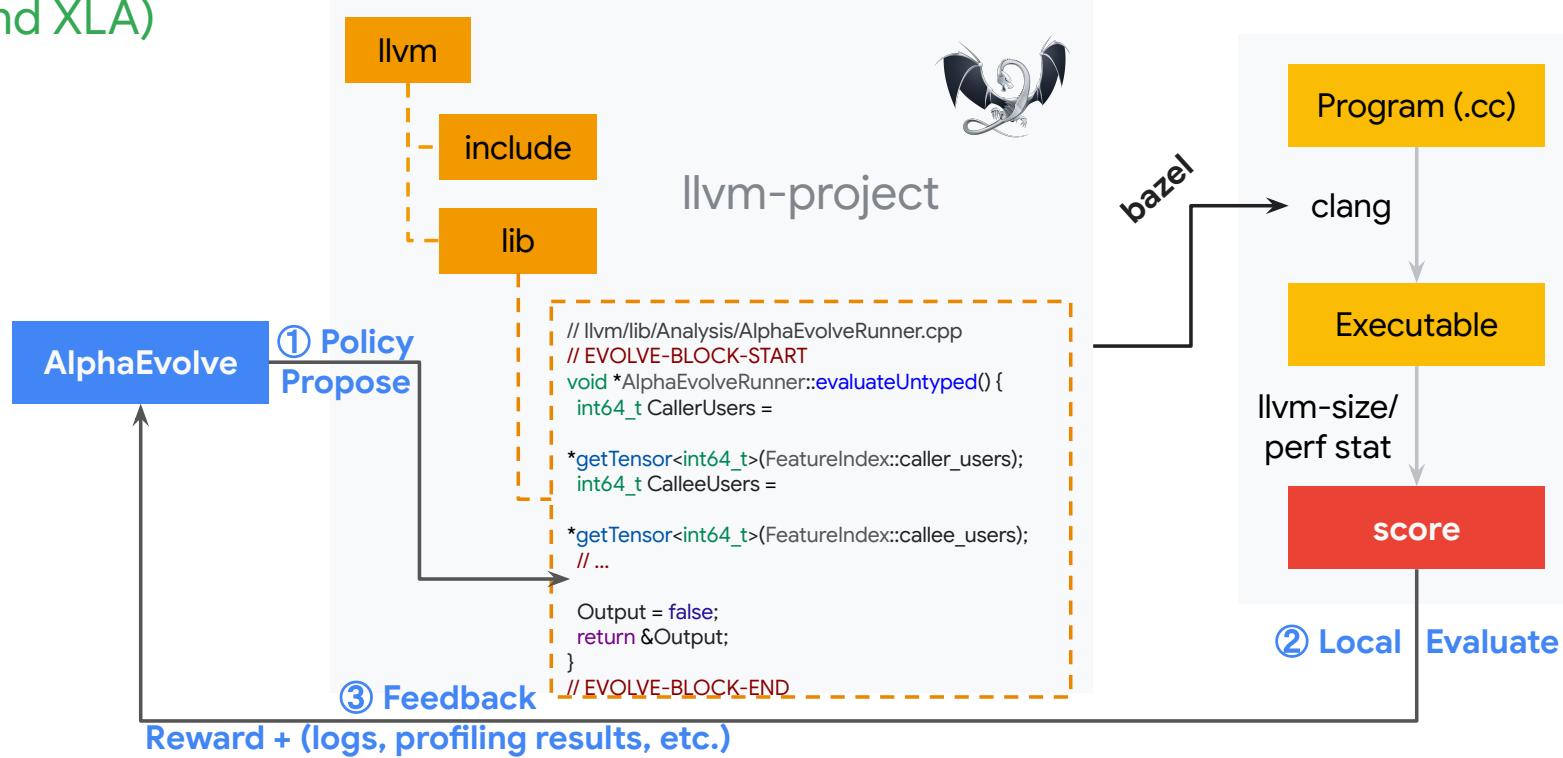
AlphaEvolve in a Nutshell

Specification



**Evolutionary
Search to select
the best program**

System Architecture for Heuristic Discovery in LLVM (and XLA)



The system works in an automated loop: AlphaEvolve (1) proposes a heuristic as C++ code, (2) the toolchain evaluates its performance, and (3) the resulting score provides the feedback (reward, logs, etc.).

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

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
```

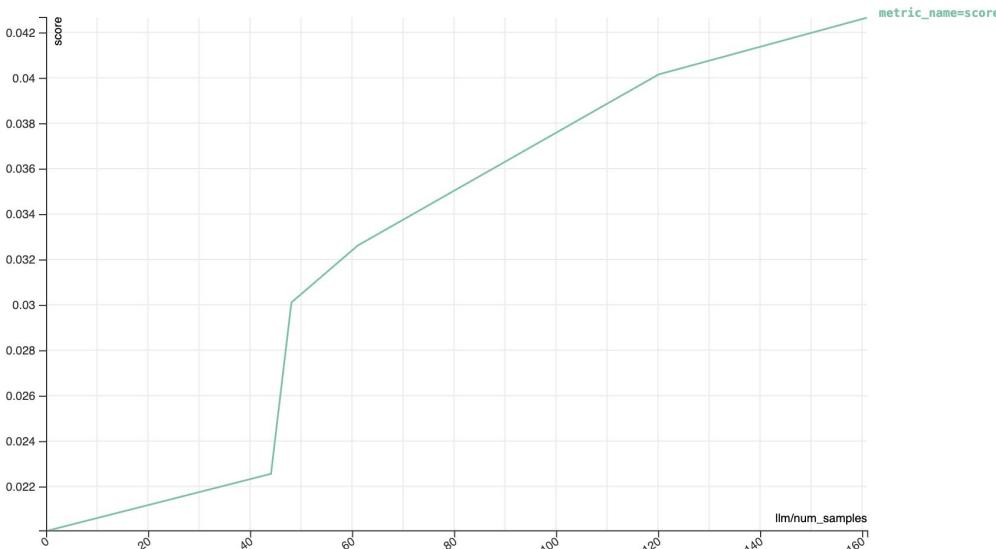
2. Entire heuristic based on arbitrary LLVM API calls

- 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

- 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

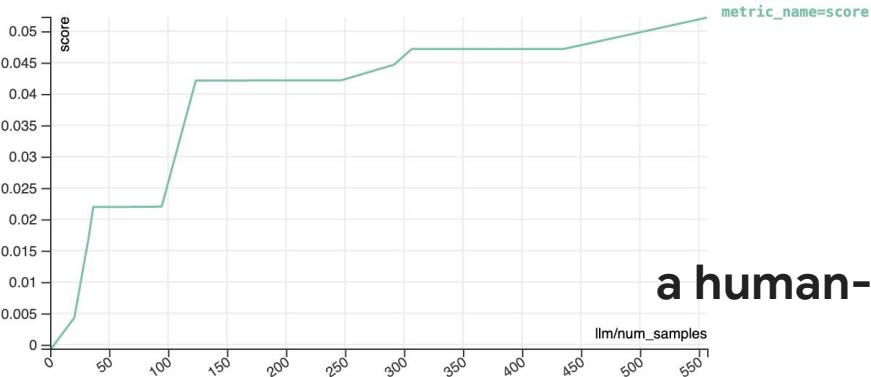


```
std::unique_ptr<InlineAdvice> AEInlineAdvisor::getAdviceImpl  
(CallBase& CB) {  
    bool IsInliningRecommended = false;  
    return std::make_unique<InlineAdvice>(  
        this, CB,  
        FAM.getResult<OptimizationRemarkEmitterAnalysis>  
            (*CB.getCaller()), IsInliningRecommended);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

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

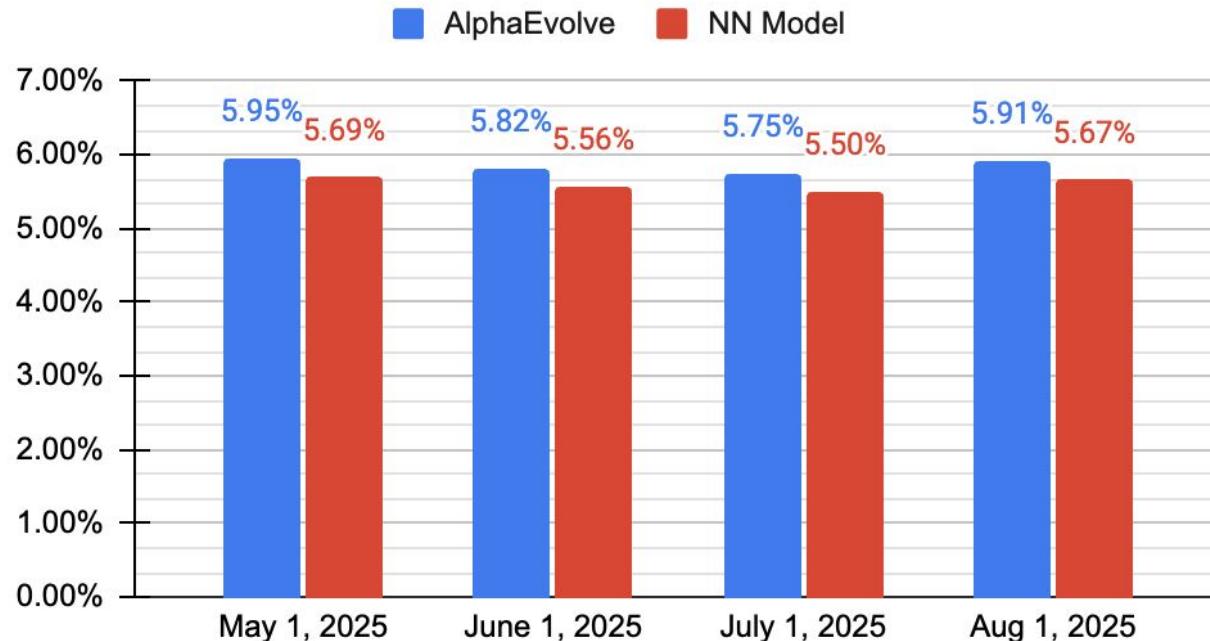
Case Study I: Function Inlining for Size

- 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



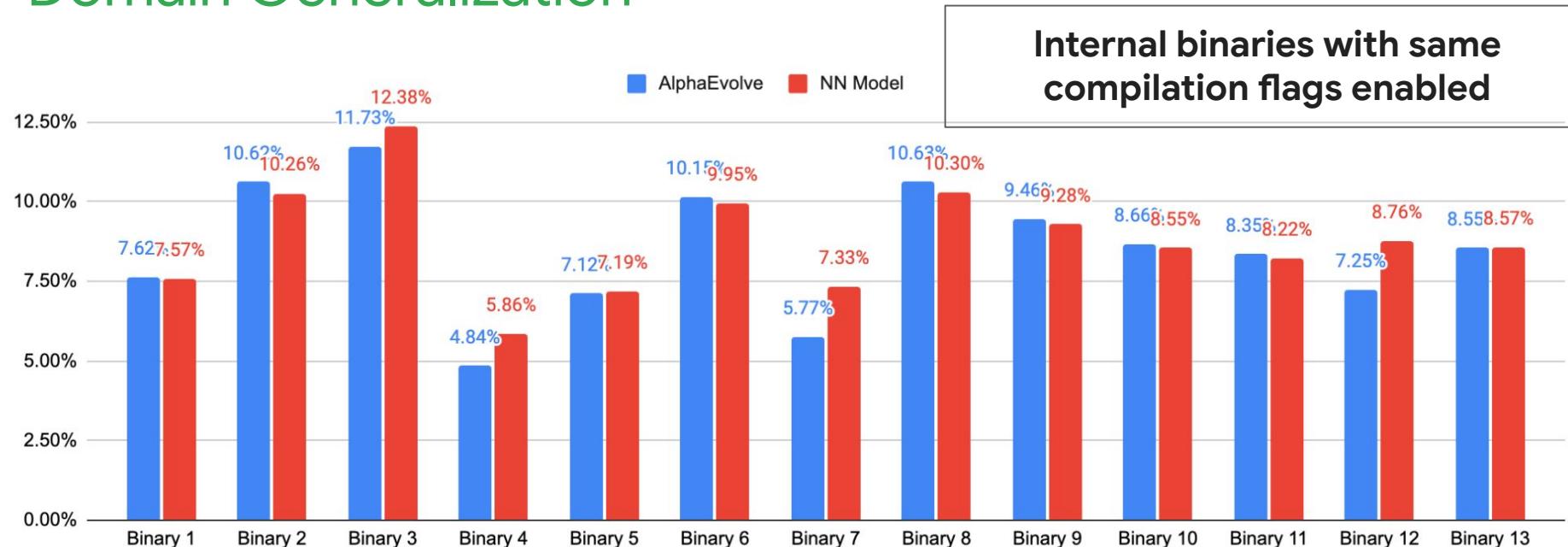
**5.23% better than
a human-developed heuristic after only 1.5 days!**

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!)

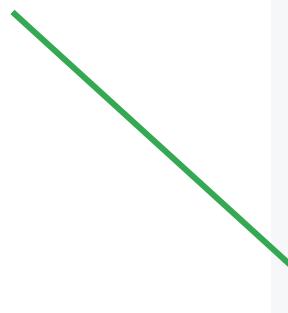
Case Study I: Function Inlining for Size Domain Generalization



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



```
42 std::unique_ptr<InlineAdvice> AEInlineAdvisor::getAdviceImpl(CallBase &CB) {
43     // Implementation of inlining strategy. Do not change the function interface.
44     // Default to not inlining.
45     constexpr unsigned CONSERVATIVE_INLINE_PENALTY = 20;
46     constexpr unsigned HOT_FUNCTION_BONUS = 50;
47     bool IsInliningRecommended = false;
48     Function *Callee = CB.getCalledFunction();
49
50     // Define weights for different instruction types to calculate a "complexity score"
51     // rather than a raw instruction count.
52     constexpr unsigned WEIGHT_HIGH_COMPLEXITY_INST = 3;
53     constexpr unsigned WEIGHT_MEDIUM_COMPLEXITY_INST = 2;
54     constexpr unsigned WEIGHT_LOW_COMPLEXITY_INST = 1;
55
56     // If the callee is null (indirect call) or a declaration (no body to inline),
57     // we cannot inline.
58     if (!Callee || Callee->isDeclaration()) {
59         // If the callee is null (indirect call) or a declaration (no body to inline),
60         // we cannot inline. Return advice to not inline.
61         IsInliningRecommended = false;
62         return std::make_unique<InlineAdvice>(
63             this, CB,
64             FAM.getResult<OptimizationRemarkEmitterAnalysis>(*CB.getCaller()),
65             IsInliningRecommended);
66     }
67
68     // Check for explicit inlining attributes first, as they override heuristics.
69     if (Callee->hasFnAttribute(Attribute::NoInline)) {
70         // Explicitly prevent inlining. IsInliningRecommended remains false.
71         // Fall through to the final return.
72     } else if (Callee->hasFnAttribute(Attribute::AlwaysInline)) {
73         IsInliningRecommended = true;
74     } else {
75         // Crazy Idea: Calculate a weighted instruction count based on instruction complexity.
76         // This gives a more nuanced "size" estimation for inlining decisions.
77         unsigned WeightedCalleeInstructionCount = 0;
78         for (const BasicBlock &BB : *Callee) {
79             for (const Instruction &I : BB) {
80                 switch (I.getOpcode()) {
81                     case Instruction::Call:
82                     case Instruction::Invoke:
83                     case Instruction::CallBr:
84                     case Instruction::Ret:
85                     case Instruction::Br:
86                     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

- Newly discovered (simplified) features
 - Weighted instruction count based on instruction complexity
 - Categorize the instructions into three different types
 - `InlineCost.cpp` uses a more sophisticated cost model `TargetTransformInfo (TTI)`
 - Pointer type casting
 - Penalty for type mismatches
 - `InlineCost.cpp` doesn't give a simple bonus

Case Study II: End-to-End Performance Optimization

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.

Inlining for End-to-End Performance on Clang

- Automatically recovered from an initial -33% regression to finish at -0.2%, on par with manual heuristics.

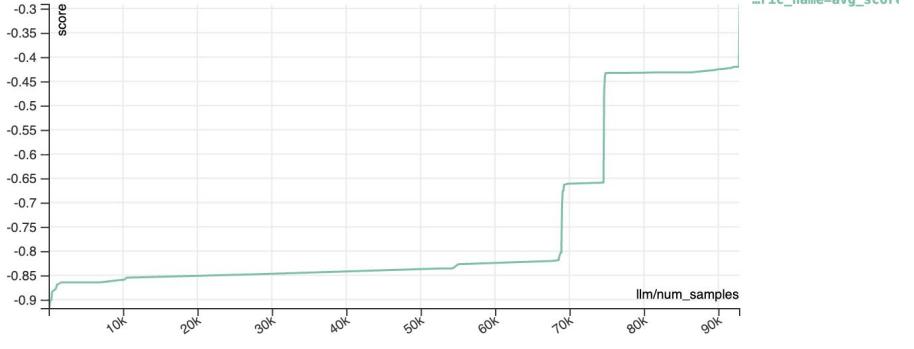
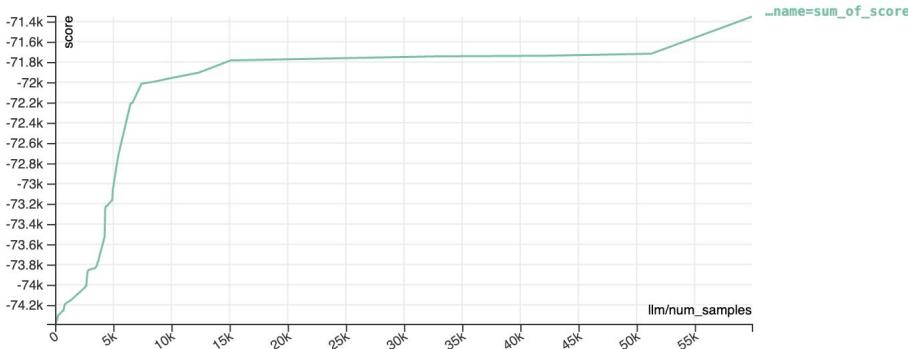
Register Allocation (priority queue) for End-to-End Performance on Search

- Discovered a simple, non-obvious heuristic. AlphaEvolve found a *trivial* on par with complex manual heuristics.
- Improved from an initial -0.55% regression to -0.15%.

[WIP] Preliminary Results on XLA Problems

* Disclaimer: Not yet conducted end-to-end evaluation in the XLA pipeline

- Graph rewrite w/ eqsat
 - Based on Enzyme-JAX [OOPSLA'25]
 - Heuristics for e-graph extraction
 - Rewrite cost based on a cost model
 - 7% better than manual heuristics
- Auto-sharding
 - Based on ASPLOS'25 contest held by Google
 - Transformer, Gemma, and diffusion models in SFT & inference
 - Achieve 4th place out of 20 teams compared to the participants' solutions so far



Trade Offs



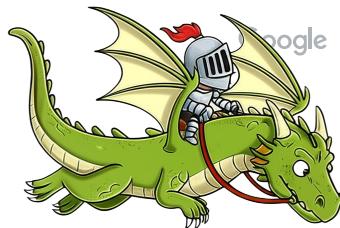
Improved compiler engineers productivity (at least for new heuristics)

- 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.



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.

Next steps:

- **Push Performance Boundaries:** Explore new methods to surpass the current performance ceiling of discovered heuristics (e.g. better prompting, NL rewards, etc.)
- **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:** Based on OpenEvolve and OSS models