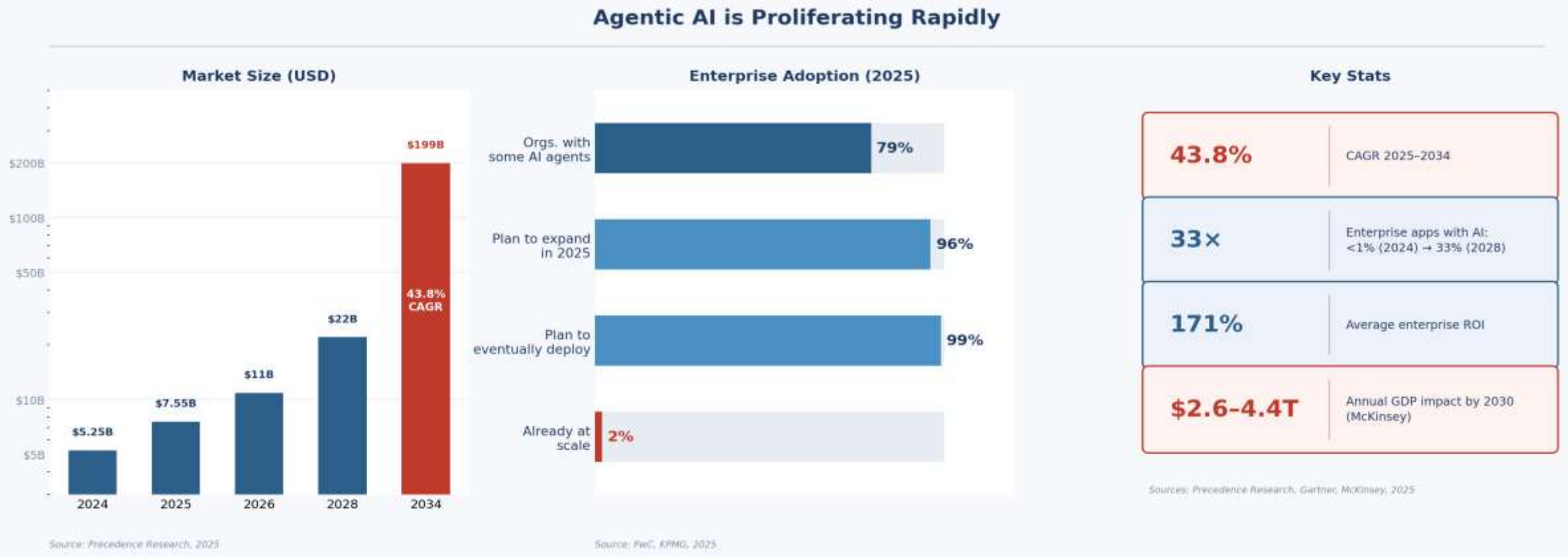
ApproxMLIR: An Accuracy-Aware Compiler for Compound AI Systems

Hao Ren, Yi Mu, Sasa Misailovic

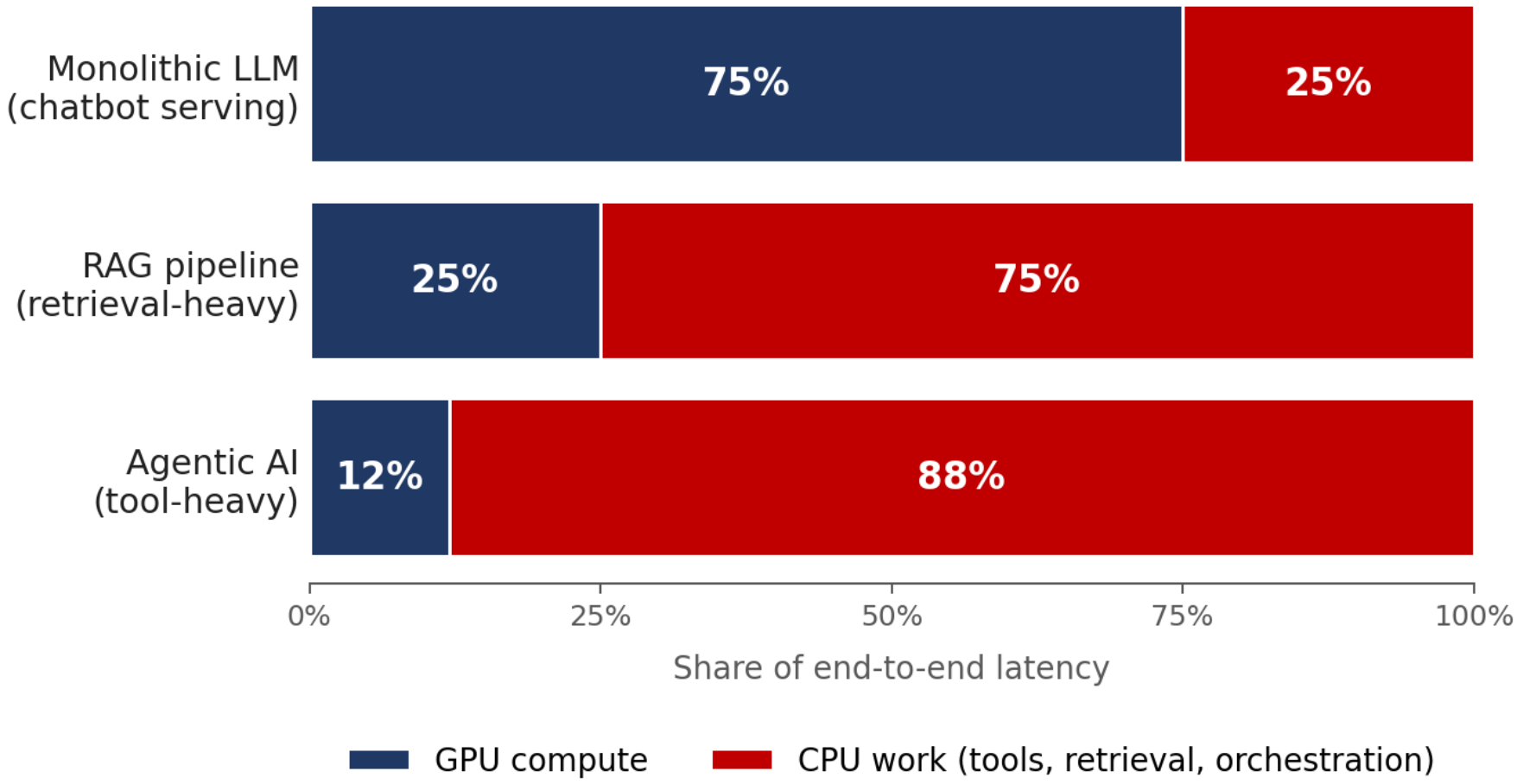
University of Illinois Urbana-Champaign

Agentic AI is Proliferating

Agentic AI is moving to production, orchestrating **both ML and non-ML kernels** autonomously.



Agentic AI Inverts the CPU/GPU Time Budget



Sources: Georgia Tech & Intel, arXiv:2511.00739, Nov 2025 · vLLM Llama-3-8B / H100 trace · AgentCgroup, arXiv:2602.09345

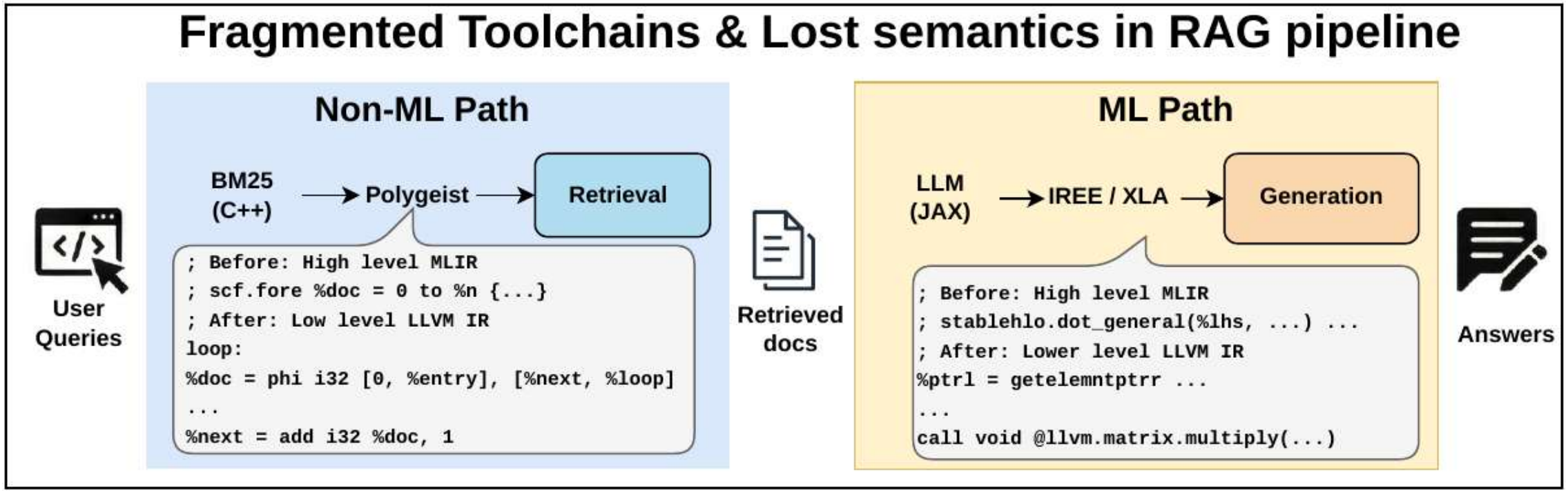
ML and non-ML **both matter**.

Running Example: BM25 RAG

- Calculate score of text corpus.
- Select top-K filtered entries.
- Compile the prompt from the selected entries
- Generate answer with retrieved data.

```
1: function BM25_RAG(query, corpus)
2:     scores ← BM25SCORE(query, corpus)
3:     topdoc ← FILTERTOPKDOCS(scores)
4:     context ← CREATEPROMPT(topdoc, query)
5:     return GENERATE(LLM, context)
6: end function
```

Running Example: BM25 RAG



One application, **multiple ecosystems**.

GCC / Clang → LLVM / … for non-ML · JAX / … → Triton / … → PTX / … for ML

Where Can We Approximate?

1. Corpus Subsetting — Non-ML

Probabilistically skip documents during preprocessing.

2. Term Scoring — Non-ML

Skip per-term scoring with 10–20% probability.

3. Context Selection — Non-ML

Truncate prompt by retrieval similarity.

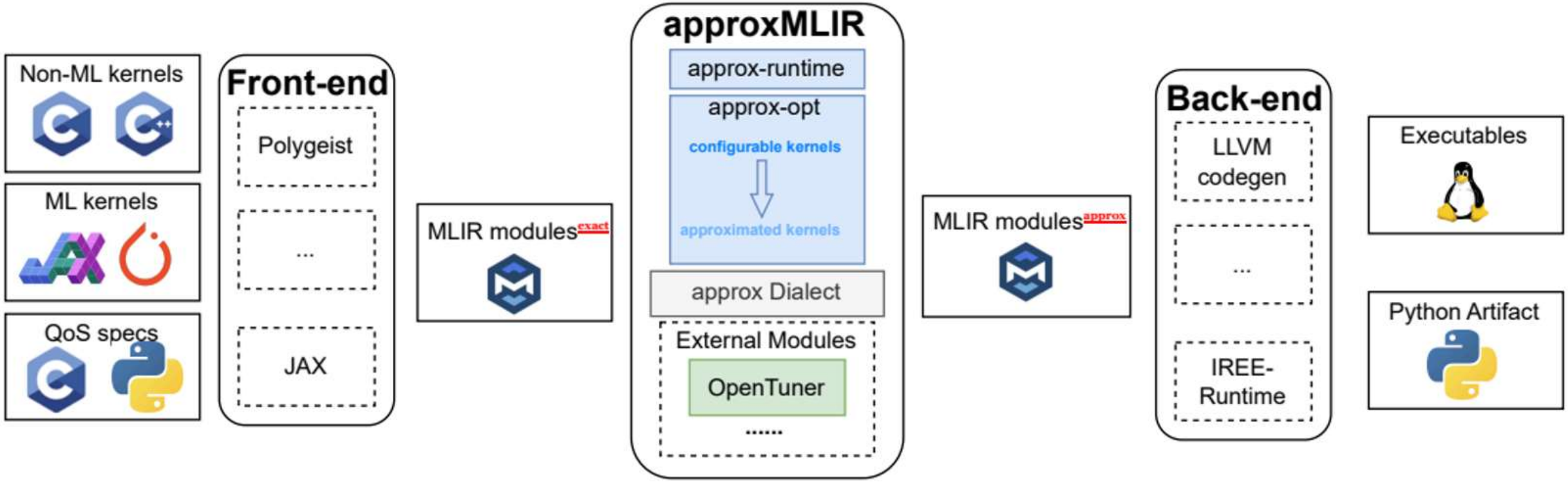
4. LLM Substitution — ML

Swap in a smaller or quantized LLM.

But **no shared compiler infrastructure to coordinate them**.

The approx Dialect, in One Picture

A reusable accuracy-aware compilation toolchain organized around one **MLIR** dialect.



approx-opt rewrites approximation, approx-runtime selects at runtime, the **approx Dialect** ties them together.

MLIR has nowhere to put approximation

Naive approach: attach approximation as an attribute on an existing op.

Before lowering

```
scf.for %i = 0 to %n {
  approx.transform =
"skip"
  // loop body
}
```

–

Tiling pass runs

rewrites loop into outer + inner

attribute is dropped silently

–

After lowering

```
scf.for %i_outer ... {
  scf.for %i_inner ... {
    // loop body
  }
}
// approximation lost
```

We need a **first-class dialect**, not attributes that get dropped.

Stand-alone Key Operations Instead of Attributes

`approx.knob`

Scope and interface

Wraps the region of code to approximate.

Exposes id and params for the autotuner.

Every approximation site begins here.

`approx.<management>`

When and where

Dispatch based on a runtime function and thresholds.

Guard approximate results with a recovery contract.

`approx.transform`

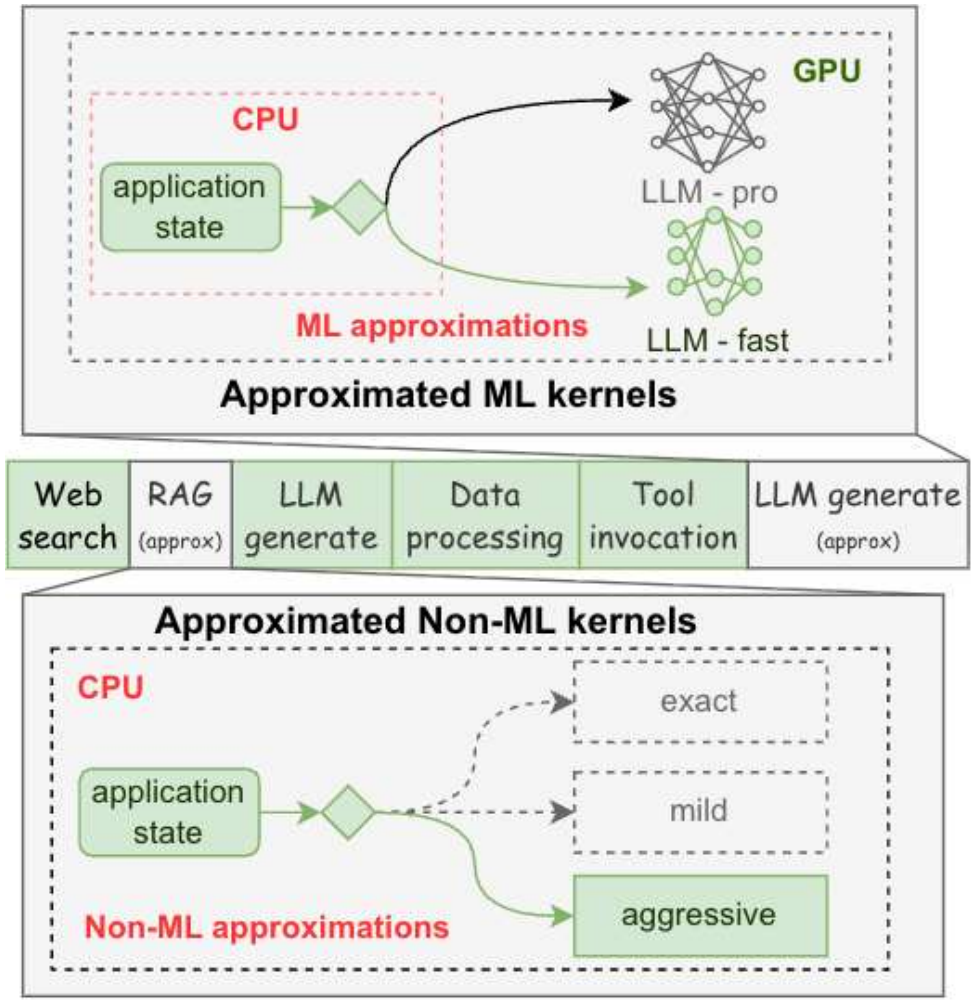
How

Define the concrete rewrite: loop_perforate, func_substitute, task_skipping.

Lowered by approx-opt rewrite rules.

knob and management answer **WHEN and WHERE**. transform answers **HOW**.

Exploring System States



Application can have **system states**

- Some approximations are more sensitive on certain states
- Some approximations are mitigated when state transitions
- We can exploit those properties w.r.t. approximations during runtime

Step 0. Input ML and Non-ML kernels

```
// runtime = "get_state",
// thresholds_range = [...],
// decision_values = [...],
// transform_type = ...
int kernel(int n, ...) {
  ...
  for (int i = 1; i < n; i++) {
    ...
  }
  ...
}
```

- Source-level annotation.
- Nothing emitted yet.

Step 1. Emit knob op and interface with tuner

```
func.func @kernel(%arg0: i32, ...)
{
  ....
  ... = approx.knob(...) <{
    id = 0 : i32,
    params = ....
  }> ({
    // loop, stride = 1 (omitted)
  }) : (....) -> ...
  ...
}
```

- approx.knob wraps the loop.
- params is the tuner's hook.
- Knob is opaque downstream.

Step 2. Emit decision_tree op

```
func.func @kernel(%arg0: i32, ...) {
  ....
  ... = approx.knob(...) <{id = 0 :
i32,
                          params =
....}> ({
    approx.decision_tree(%arg0, ...)
<{
      transform_type =
"loop_perforate",
      runtime = "get_state",
      thresholds = ... }> : i32
    // loop, stride = 1 (omitted)
  }) : (....) -> ...
  ...
}
```

- Dispatch captured; rewrite still deferred.
- Survives standard passes.

Step 3. Lower decision_tree op

```
.... // knob op and func omitted here
%0 = call @get_state(%arg0, ...) : ...
%1 = arith.cmpi sge, %0, <threshold> :
i32
%2 = arith.select %1, <one>, <zero> :
i32
... // count thresholds < %0, assign
to %cnt
scf.index_switch (%cnt)
case ... {
  approx.transform <{
    transform_type = "loop_perforate",
    knob_val = .... }>
  // loop, stride = 1 (omitted)
}
....
```

- Dispatch becomes std MLIR.
- Each branch holds a transform.

Step 4. Bind runtime

```
func.func @get_state(%arg0: i32) ->
i32 {
  ...
}

// knob op and func omitted here
%0 = call @get_state(%arg0, ...) : ...
....
scf.index_switch (...)
case ... {
  approx.transform <{
    transform_type = "loop_perforate",
    knob_val = .... }>
  // loop, stride = 1 (omitted)
}
```

- **Link the user-provided auxiliary function.**
- **Code-gen the call and consumer of the runtime value.**

Step 5. Apply transform op

```
%c2 = arith.constant 2 : index
scf.index_switch (...)
....
case .... { // approximate at runtime
  %4 = arith.index_cast %arg0 : i32 to
index
  %5 = scf.for %arg2 = %c1 to %4 step
%c2
      -> (...) {
    ... // loop body
  } // loop, stride = 2
}
...
```

- **Nothing approx remains.**
- **Extensible by rewrite rule.**

Example: LLM Decode Activation Quantization

Inside the kernel

```
%out = approx.knob(%a, %b) <{
  id = …
}> ({
  approx.transform <{
    transform_type = "decode_quantize",
    transform_value = 1
  }>
 %acc = tt.dot %a, %b, %acc_init
  approx.yield %acc
})
```

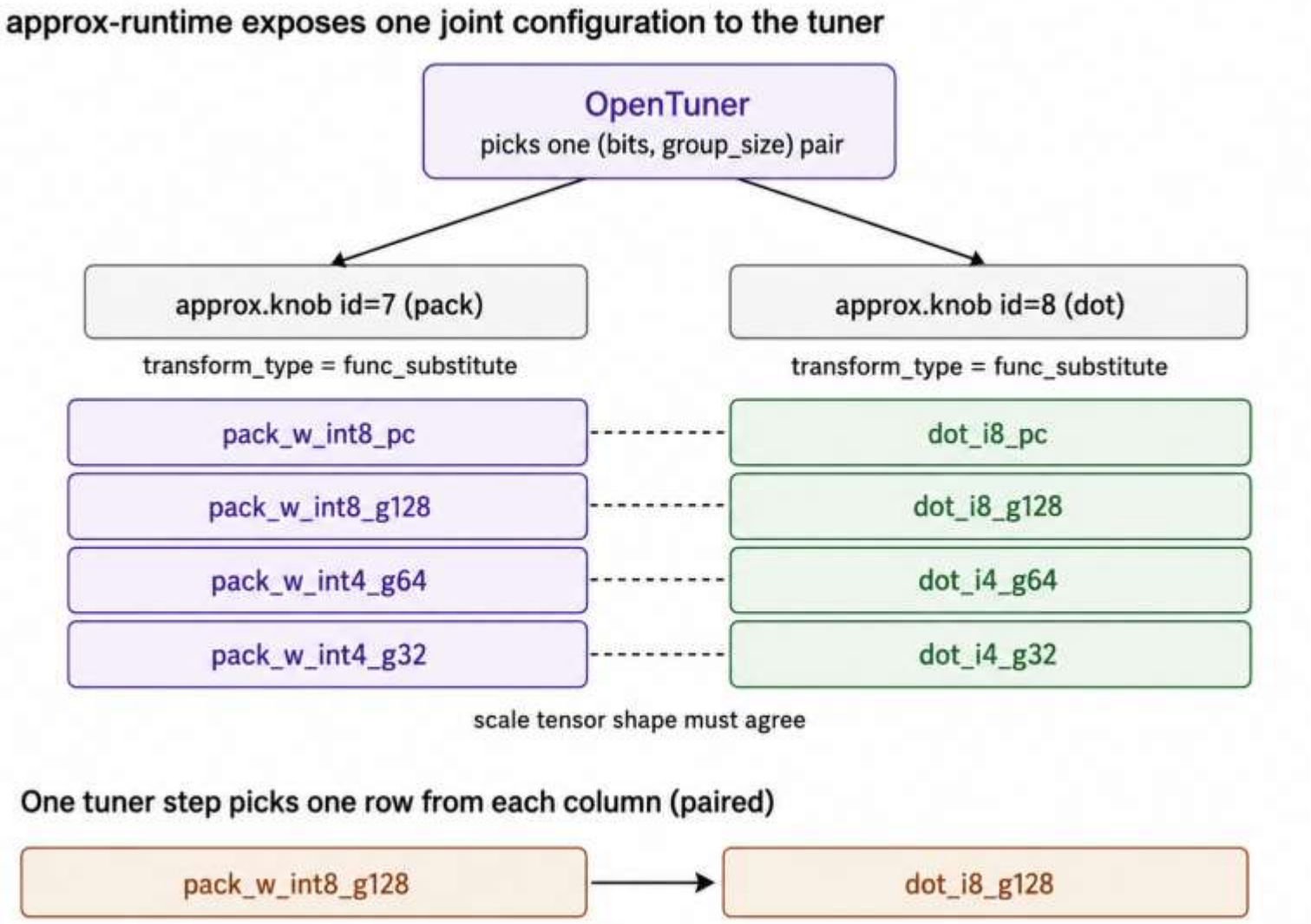
After lowering

```
// rewritten kernel
%aq, %a_scale =
  quantize_activation(%a_bf16)
%acc_i32 = tt.dot %aq, %bq, %zero_i32
  : tensor<1xKxi8> * tensor<KxNxi8>
  -> tensor<1xNxi32>
%out = ... dequant(%acc_i32,
                   %a_scale, %b_scale)
```

What about **Weight Quantization** where **intra-kernel approximation won't work**

approx-runtime to solve inter-kernel coordination

1. Integration with externa modules such as auto-tuners
2. Coordinate tuning acros correlated knobs
3. Link user-defined auxiliary functions to the compiled module



Evaluation Setup

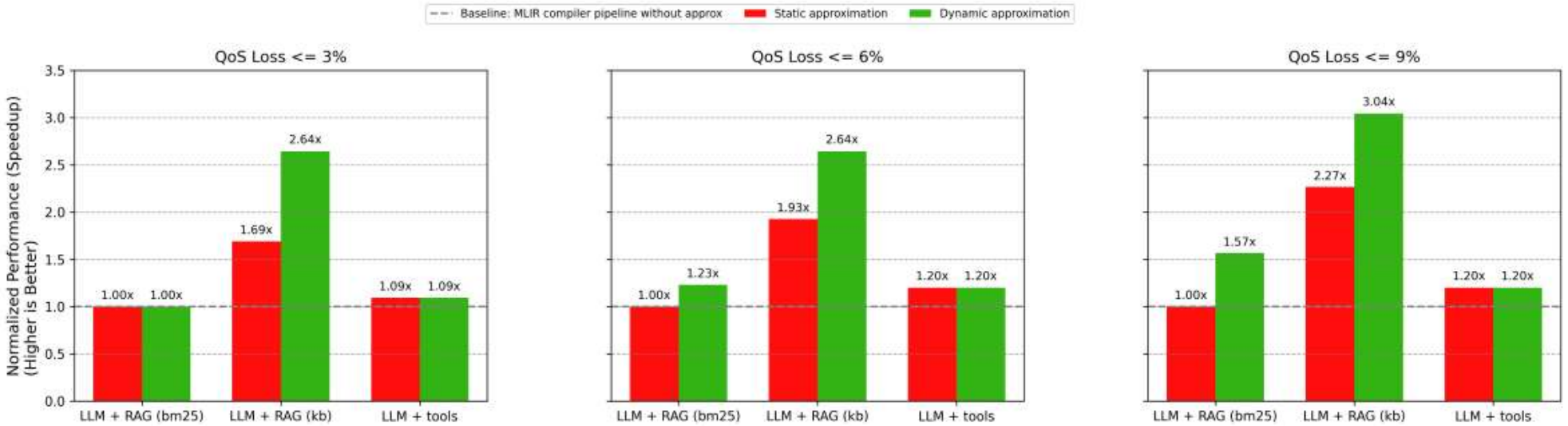
Five non-ML kernels

- lavaMD: particle simulation
- knowledge base (kb): vector similarity
- bm25: keyword retrieval
- pagerank: graph ranking
- kmeans: clustering

Three compound AI systems

- LLM + RAG (bm25) 4 knobs ~1.8×10^{7} configs
- LLM + RAG (kb) 4 knobs ~9.6×10^{8} configs
- LLM + tools 12 knobs ~1.7×10^{25} configs

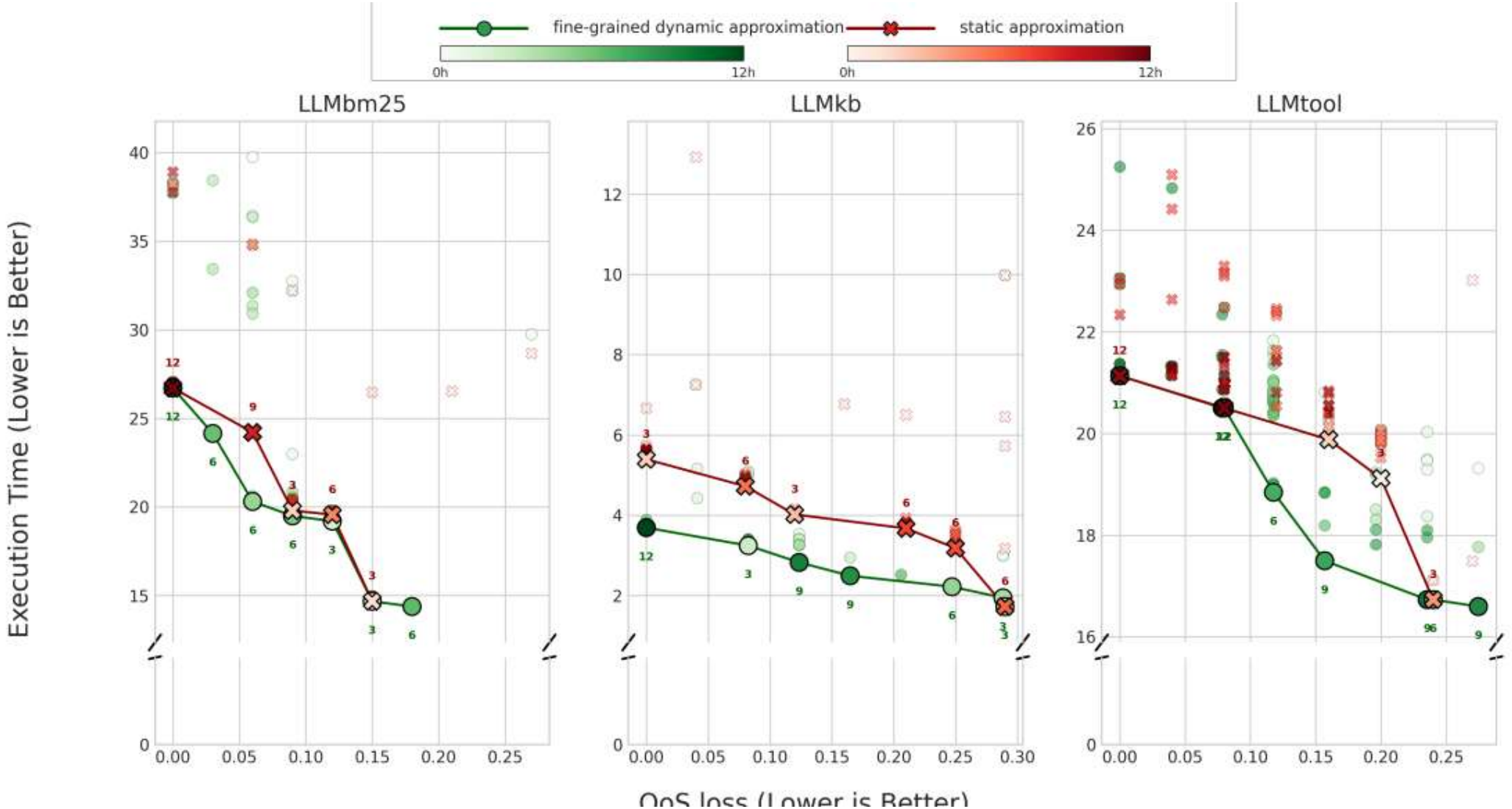
Dynamic vs Static, at Fixed QoS Budget



*Dynamic (green) beats static (red) because it can **see the states & inputs**.*

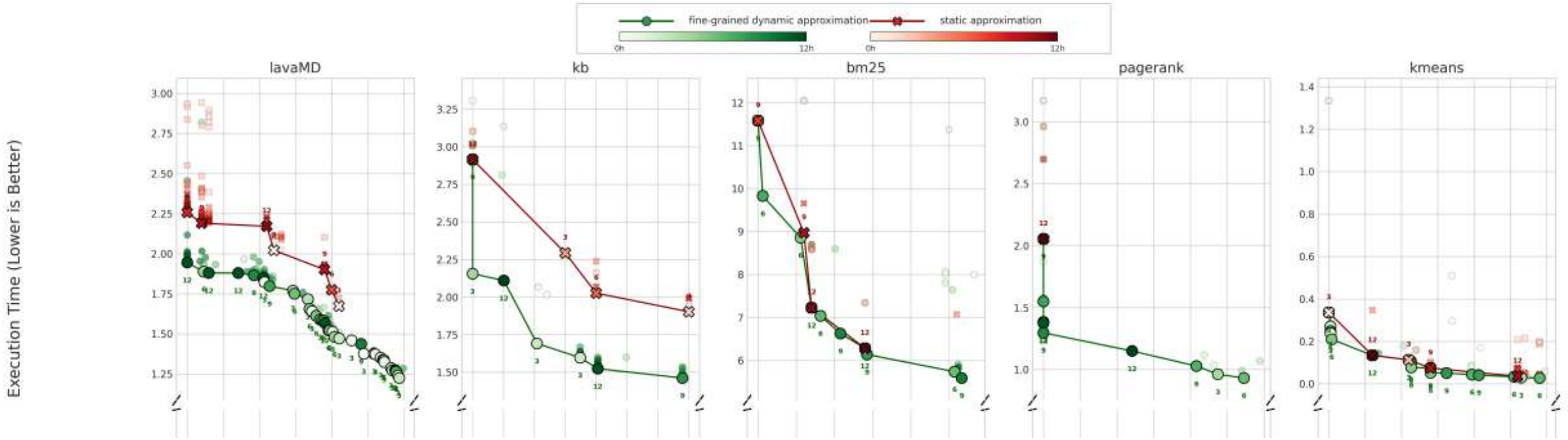
- LLM+RAG(kb): 2.64× / 2.64× / 3.04× (static caps at 1.69 / 1.93 / 2.27×).
- LLM+RAG(bm25): static cannot improve at all, dynamic gets up to 1.57×.
- Tuning generalizes within 18.6 / 19.5 / 3.7% on the held-out eval set.

Pareto Frontiers: Compound AI Systems



Dynamic frontier is **lower** (better tradeoffs) and **denser** (more operating points). Both matter to users.

Pareto Frontiers: Individual Kernels



The same dynamic-beats-static pattern holds at **kernel granularity**, consistent with the compound results.

ApproxMLIR is not mainly about one approximation technique.

It is about giving approximation a proper compiler abstraction.

approx Dialect

First-class operations for approximation scope, policy, and implementation that survive standard MLIR transformations.

approx-opt

Dedicated lowering passes for management and transform ops, staged and composable.

approx-runtime

Runtime branch selection from application state. Pareto frontiers instead of one binary.

Evaluation

Up to 7.38× kernel speedup, 3.04× compound system speedup, consistently better Pareto frontiers.

[Back-up] Prior unification was too low-level

Tools like ApproxHPVM unify approximation at the level of LLVM IR.

High-level — tensors, matmul, loops

↓ *lowering*

Low-level (LLVM IR) — pointer arithmetic, bare loops

We loose: (1) shape and types (2) arithmetic properties (3) structural properties

`tensor ops → pointer math` `matmul → just loop nests` `control flow flattened`

We need a unified IR **that keeps high-level structure** **to improve compilation overhead**