Compiling Agentic AI Programs for Dataflow Execution

An MLIR Approach

Miguel Cárdenas, **Rafael A Herrera Guaitero**,
Isaac Bermudez, Jose M. Monsalve Diaz

LLVM Colombia

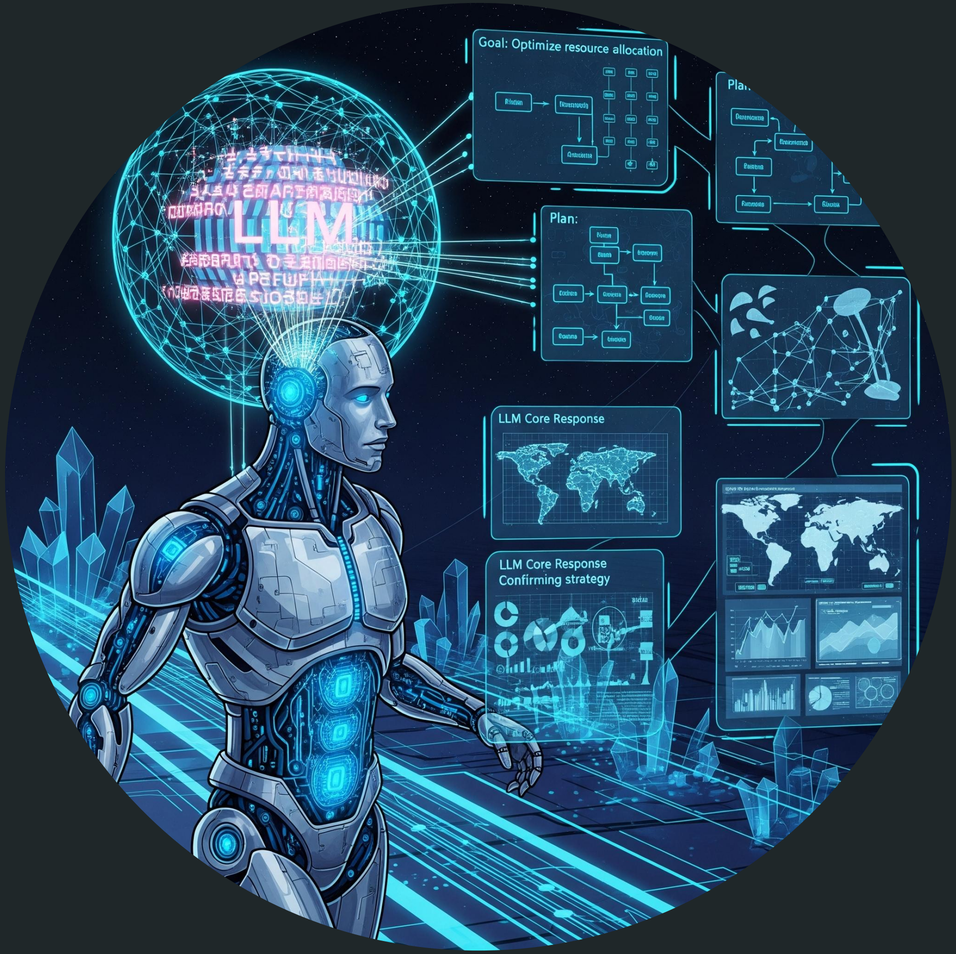
Tenth LLVM Performance Workshop at CGO 2026

Jan 30 2026

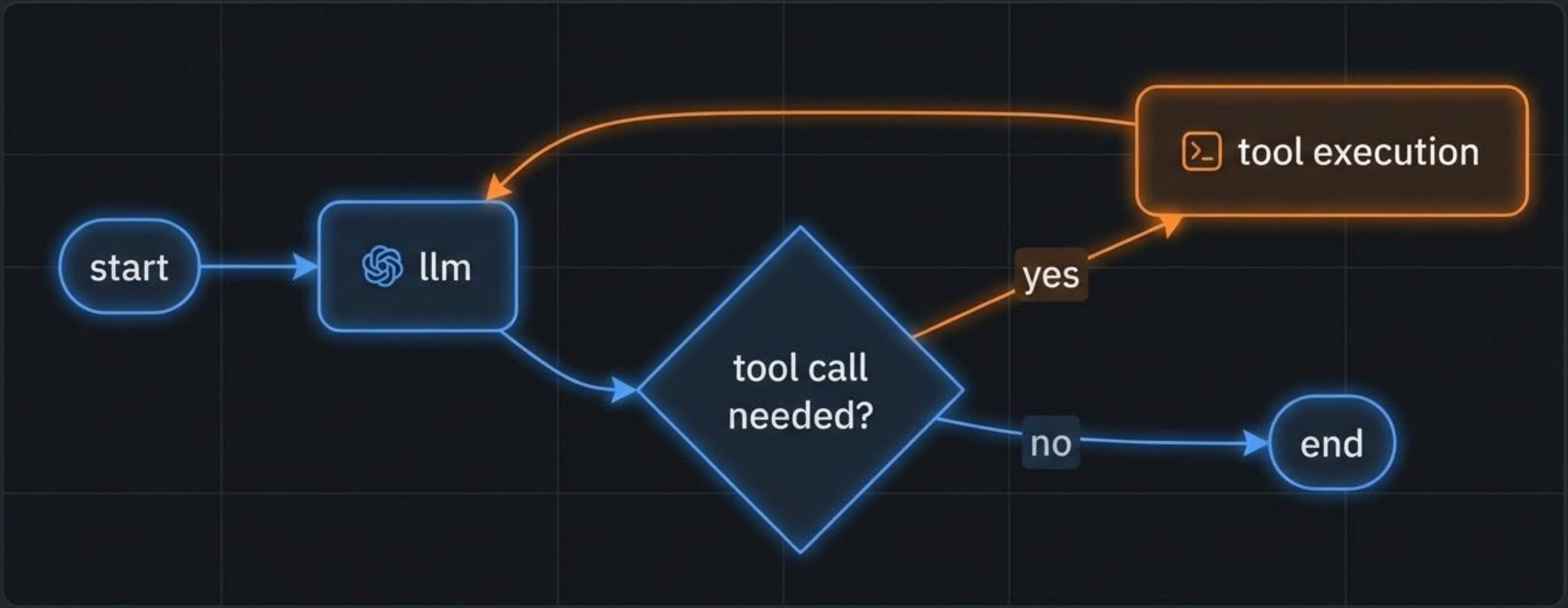
What Is an Agent?

An agent is an autonomous, goal-oriented program that executes multi-step workflows by interleaving Large Language Model (LLM) calls, tool I/O, and memory operations.

Key components: Core LLM, Planning, Memory, tools



The Problem - Agentic AI Programs

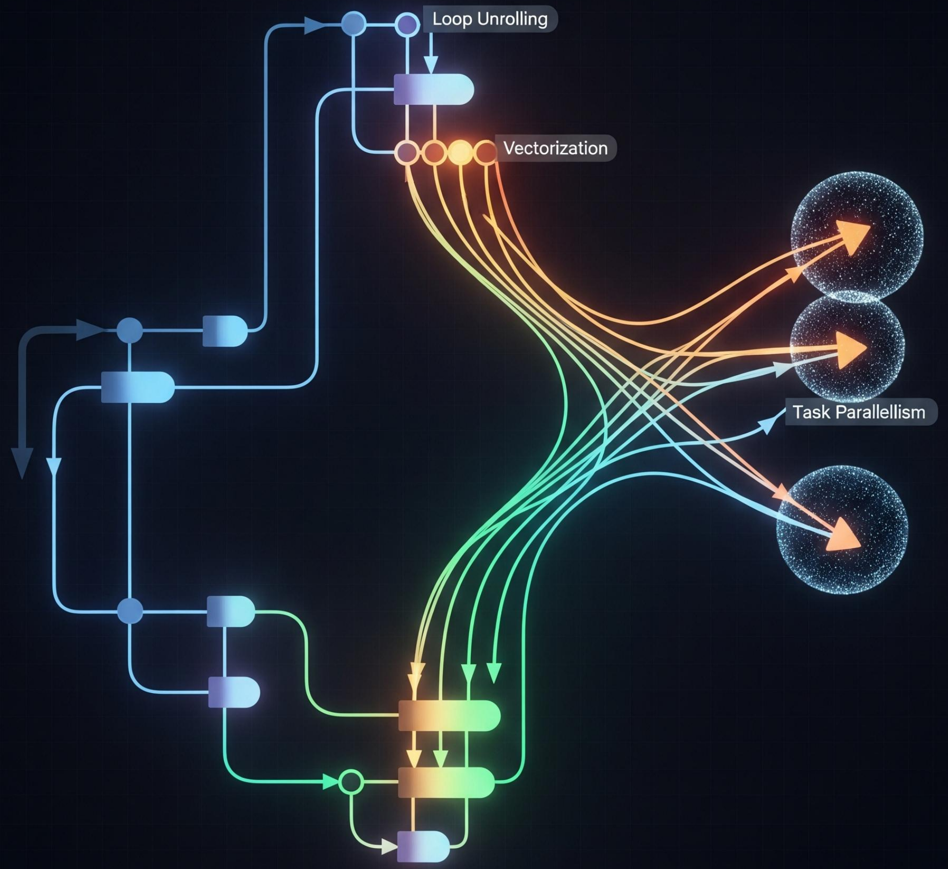


source

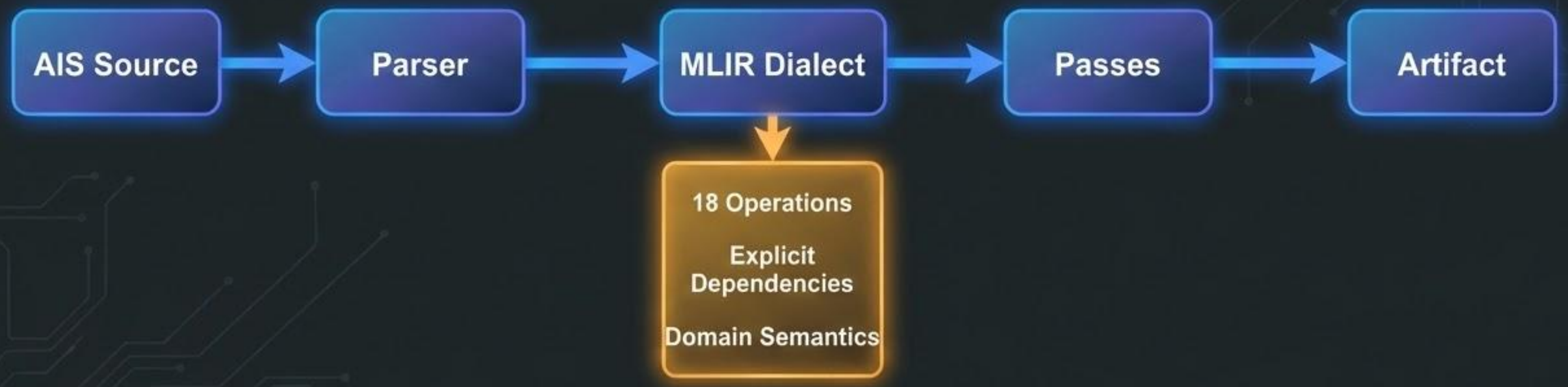
What are compilers good at?

Compiler Advantages for Agents

- Compilers enable whole-workflow optimization and analysis.
- They expose data dependencies for automatic parallelism.
- Compilers provide static checks, catching errors before execution.
- They allow IR-level transforms like operation fusion.



Solution Overview - MLIR DSL And Dialect for Agentic AI



AIS Dialect Architecture

Metadata (1 ops)	`agent`
Memory (2 ops)	`qmem, umem`
Control Flow (7 ops)	`jump, branch_on_value, loop_start, loop_end, return, switch, flow_call`
Error Handling (2 ops)	`try_catch, error`
Inference (6 ops)	`ask, think, reason, plan, reflect, verify`
Tools (2 ops)	`inv, exc`
Synchronization (3 ops)	`merge, fence, wait_all`
Communication (1 ops)	`communicate`
Internal (2 ops)	`const_str, yield`

Operation Example - AIS MLIR Syntax

```
agent Coordinator {
  @entry flow main(topic: str) → str {
    // Parallel: no data dependencies
    Researcher.research(topic) → res
    Critic.prepare(topic) → prep
    Analyst.analyze(topic) → analysis

    // Barrier: synchronize results
    wait_all(res, prep, analysis)

    // Synthesize final output
    ask("Synthesize...") → report
    return report
  }
}
```

Operation Example - AIS MLIR Syntax

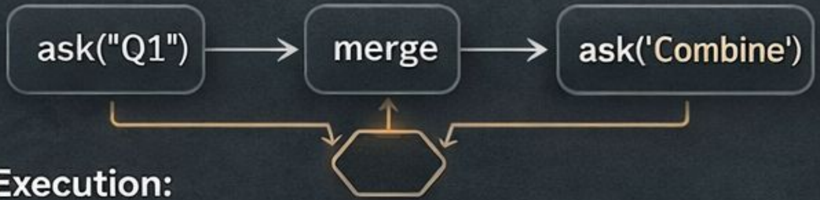
```
  module attributes {ais.fused_pairs = #ais.fused_pairs<0>, ais.
graph_normalized = #ais.graph_normalized<0>, ais.scheduling_annotations =
#ais.scheduling_annotations<1>} {
    ais.agent "Coordinator" {beliefs = [], capabilities = [], goals = [],
memories = []}
    func.func @Coordinator.main(%arg0: !ais.token<i64> {ais.param_name =
"topic", ais.param_type = "str"}) -> !ais.token<i64> attributes {ais.entry}
{
      %0 = ais.flow_call "Researcher" "research"(%arg0 : !ais.token<i64>) :
!ais.token<i64>
      %1 = ais.flow_call "Critic" "prepare"(%arg0 : !ais.token<i64>) : !ais.
token<i64>
      %2 = ais.flow_call "Analyst" "analyze"(%arg0 : !ais.token<i64>) : !ais.
token<i64>
      %3 = ais.ask "Synthesize..." {ais.estimated_cost = #ais.
estimated_cost<2>, ais.intent = #ais.intent<reasoning>, ais.latency = "low",
ais.parallel_safe = #ais.parallel_safe, ais.tier = #ais.tier<reasoning>} :
!ais.token<i64>
      return %3 : !ais.token<i64>
    }
  }
```

Dataflow Example:

MLIR Code:

```
%a = ais.ask "Q1"
%b = ais.ask "Q2"
%c = ais.merge %a, b%
%d = ais.ask "Combine: {0}" [%c]
```

Dataflow Graph:



Execution:

Parallel starts for ask("Q1") and ask("Q2"), waits at merge, then executes ask('Combine)

Scheduling

CLASSIFICATION by operation type:

io tier: web_search, fetch, http_call
↓ Estimated cost: base + (10 × context_tokens)

compute tier: math_solve, solve_equation, calc
↓ Estimated cost: base + (1 × context_tokens)

reasoning tier: ais.think, ais.reason
↓ Estimated cost: base + (5 × context_tokens)

memory tier: qmem, umem
↓ Estimated cost: base + (2 × context_tokens)

ANNOTATE each operation:

- **ais.tier** = {io, compute, reasoning, memory}
- **ais.estimated_cost** = integer
- **ais.parallel_safe** = **true** (if speculation−safe)

→ Runtime scheduler uses annotations for parallelism

Why ask, think, reason instead of one llm op?

OP	Fusible	Semantics
ask	YES	Q&A, low latency
think	MAYBE	Extended reasoning
reason	MAYBE	Structured output

Latency classification enables compile-time optimization. Ask is the only fusible operation. Without this distinction, fusion would merge slow operations incorrectly.

LLM Fusion

Batch sequential LLM calls into single operations

Before: Sequential Calls (2-4 seconds)

LLM Call 1:
What is MLIR?

→

LLM Call 2:
Explain more:
{output from 1}

```
// Before (2 LLM calls = 2-4 seconds):
%a = ais.ask "What is MLIR?" : !ais.token
%b = ais.ask "Explain more: {0}" [%a : !ais.token] : !ais.token

// After (1 LLM call = 1-2 seconds):
%b = ais.ask "What is MLIR?\n---\nExplain more: {0}"
    : !ais.token
```

→

After: Fused Call (1-2 seconds)

Fused LLM Call:
What is MLIR? \n
Explain more: {0}

```
// Before (2 LLM calls = 2-4 seconds):
%a = ais.ask "What is NLIR?" : !ais.token
%b = ais.ask "Explain more: {0}" [%a : !ais.token] : !ais.token

// After (1 LLM call = 1-2 seconds):
%b = ais.ask "What is MLIR?\n---\nExplain more: {0}"
    : !ais.token
```

From MLIR IR to Executable Artifact

MLIR IR example:

```
%ctx = ais.qmem "facts"
%a = ais.ask "Q1" [%ctx]
%b = ais.ask "Q2" [%ctx]
%b = ais.ask "Q2" [%ctx]
%c = ais.merge %a, %b
%d = ais.ask "Smmary: {0} [%c]
```

Lowering Process:

1. Extract SSA dependencies
2. Create DAG nodes per operation
3. Add edges for:
   - Data flow (SSA value uses)
   - Effect flow (memory/resource)
   - Control flow (regions, branches)
4. Serialize to ExecutionDag wire format

ExecutionDag

```
ExecutionDag {
 nodes: [
  Node(id=0, op=qmem, cost-1, tier=memory)
  Node(id=1, op=ask, cost=100, tier=reason
  Node(id=2, op=ask, cost=100, tier=reason
  Node(id=3, op=merge, cost-1, tier=general)
  Node(id=4, op=ask, cost=100, tier=reason
 }
edges: [
   (0->1, data),  # %Ctx to ask1
   (0->2, data),  # %Ctx to ask2
   (1->3, data),  # %a to merge
   (2->3, data),  # %b to merge
   (3->4, data),  # merged to ask3
 ];
}

  entry: node(0)
```

Wire format (binary): ExecutionDag v3

- Serialized to ~15-50 KB per typical program
- Deserialized at runtime by ExecutionEngine
- Multi-DAG support: one DAG per agent flow

Future Work and Directions

- Explore transpilation from orchestration frameworks to AIS.
- Investigate quality-aware optimization for LLM workflows.
- Test it with production datasets

Takeaways for CGO Community

1. **Latency-dominated workloads need different optimizations**
   - Network round-trips >> CPU cycles
   - Fusion > instruction scheduling
2. **Domain-specific dialects enables aggressive optimizations**
   - Semantic knowledge -> better decisions
   - Custom types/effects -> precise analysis
3. **MLIR is powerful for novel compilation targets**
   - Extensible infrastructure
   - SSA + regions natural for dataflow
4. **Compilers for AI orchestration are underexplored**
   - Growing importance as agents become mainstream
   - Opportunities for PL/compiler research