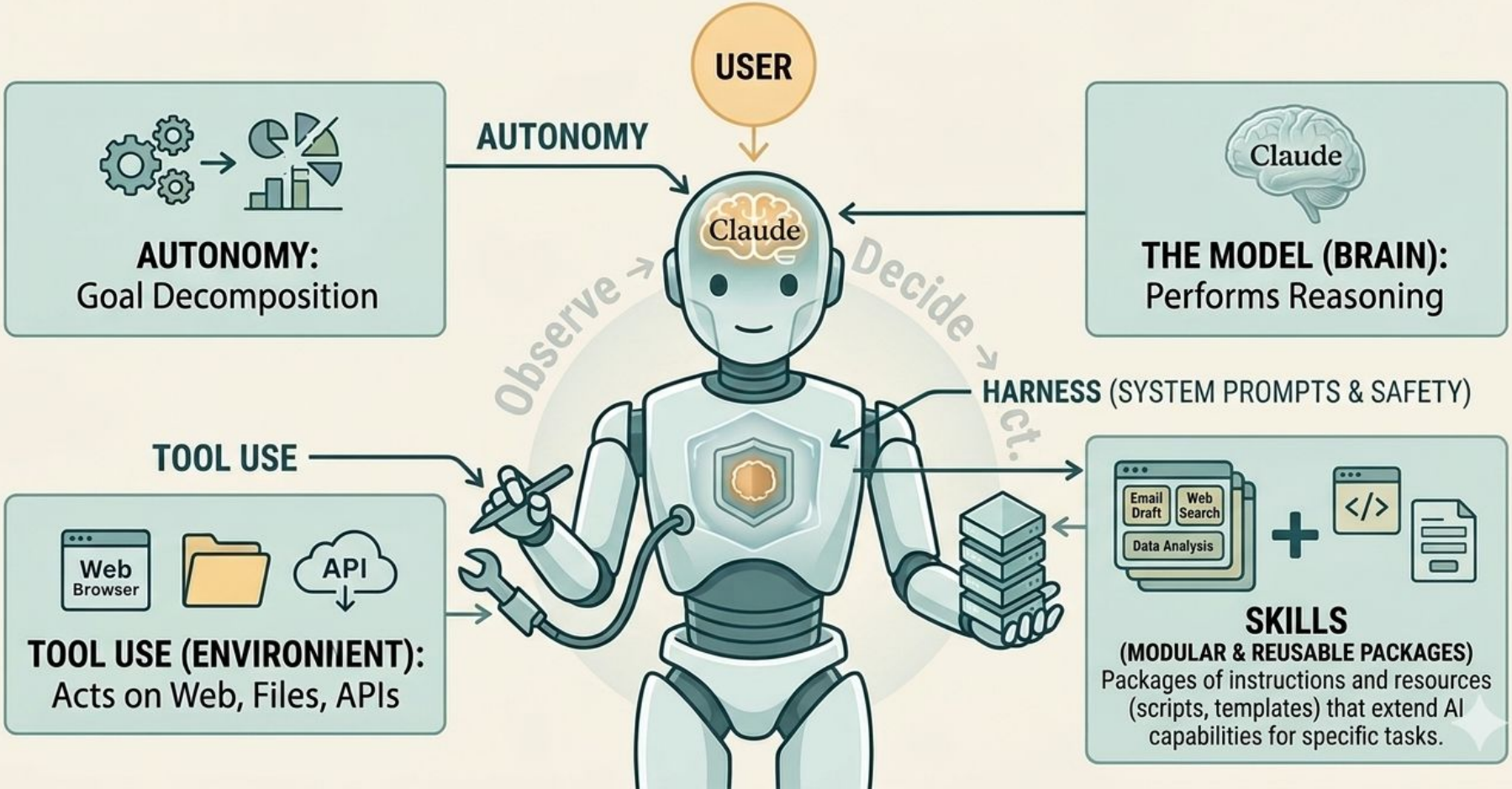
A-PXM: Multi-Agent Workflows Analysis and Optimizations via MLIR

Miguel Cárdenas, Rafael Herrera, Jose M. Monsalve, Isaac Bermudez

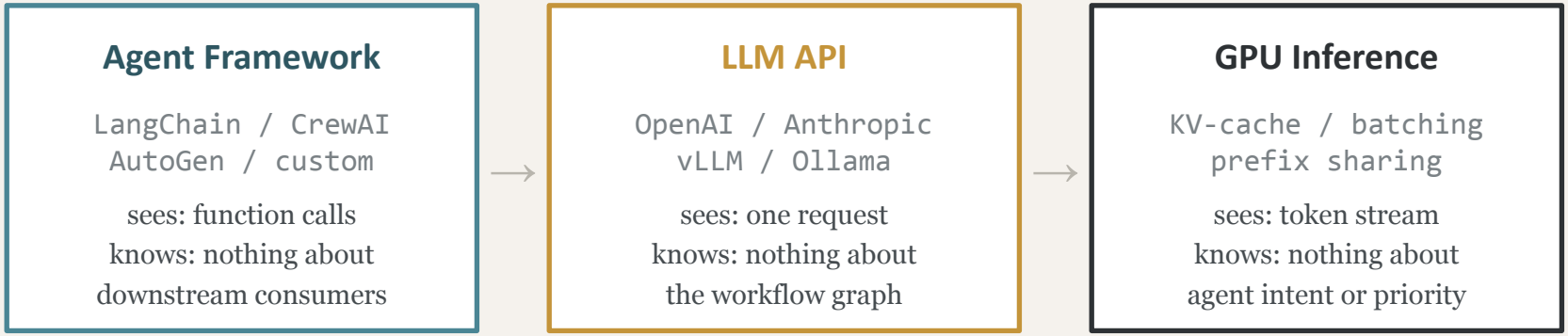
Universidad de Medellin · AMD · Universidad de los Andes

EuroLLVM 2026 · Lightning Talk

WHAT IS AN AI AGENT?



The Stack Is Disconnected



No layer sees the full picture. The framework does not know what the GPU could optimize. The GPU does not know what the agent intends. Every boundary is a lost opportunity for analysis.

What If We Represent Skills as a Compilable Graph?

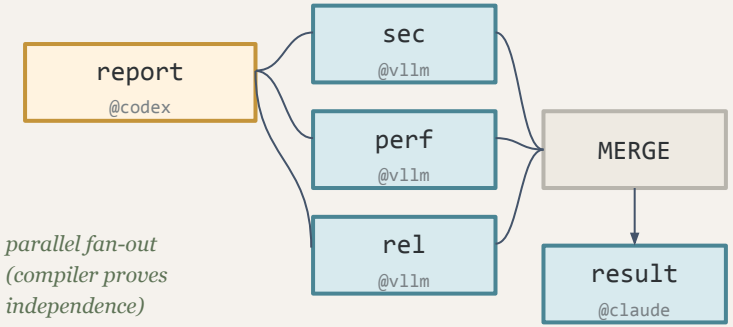
Skill: To analyze this problem spawn a codex agents and ask it to do a deep analysis on Security, Performance and Reliability and after that spawn a Claude agent to execute this plan

@compile() · Python

```
@compile()
def analyze(g):
  codex = g.spawn("analyst", codex_p)
  report = codex.ask("deep-analyze")
  sec = g.ask("security: {report}")
  perf = g.ask("perf: {report}")
  rel = g.ask("reliab: {report}")
  plan = g.merge(sec, perf, rel)
  result = claude.ask("{plan}")
```

compiles to →

Compiled Workflow Graph



Every skill becomes a graph. Every dependency becomes an edge. The graph is the program.

MLIR Can Analyze and Optimize This Graph

AIS MLIR Dialect

```
%codex = apxm.spawn @codex "analyst"
%report = ais.ask @codex "deep-analyze"
%sec = ais.ask @vllm "security: {%report}"
%perf = ais.ask @vllm "perf: {%report}"
%rel = ais.ask @vllm "reliab: {%report}"
%plan = ais.merge %sec, %perf, %rel
%result = ais.ask @claude "{%plan}"
```

The compiler understands future state:

- sec, perf, rel are provably parallel (different effect resources)
- sec, perf, rel share a prefix (same provider, same prompt root)
- result depends on plan → critical path
- report has downstream_nodes = [sec, perf, rel] → warmup candidate

MLIR Passes

fuse-ask-ops
collapse redundant ASK chains into one operation

prompt-canonicalization
normalize prompt structure, deduplicate shared prefixes

assign-priority
mark critical path vs. speculative operations

capability-scheduling
assign cost multipliers for the parallel scheduler

compile-time verification
reject undefined vars, invalid attributes before any LLM call

All components share one IR. Static analysis replaces runtime guessing. The compiler emits a .apxmobj artifact with embedded hints.

Compiler Hints Flow to vLLM Scheduling

.apxmobj → vLLM request

```
POST /v1/chat/completions
{
  "priority": 0,
  "apxm": {
    "reuse_group": "shared_prefix_0",
    "pin_policy": {"mode": "prefix"},
    "downstream_nodes": [4]
  }
}
```

Other backends (OpenAI, Anthropic, Google)
ignore extra_body.apxm at zero cost.

vLLM: a Graph-Aware Backend

Priority scheduling

critical = 0	normal = 5	speculative

KV-cache prefix pinning

4 / 12 blocks pinned · shared_prefix_0

same LLM, same workflow → structural leverage

Optimized Graph

vLLM is the graph-aware backend. It spends the compiler hints. Other backends receive plain requests.

A New Execution Model for Agent Workflows

Agent workflows are graphs. Compilers are good at optimizing graphs.

MLIR gives you the infrastructure. The missing piece was connecting the layers.

1. Represent agent skills as a compilable graph. The DSL is not cosmetic — it is the entry point to a compiler pipeline.
2. Use MLIR to analyze the graph at compile time. Effect resources prove parallelism. Passes optimize before execution.
3. Emit compiler hints into the artifact. Graph-aware backends spend them. Unaware backends ignore them at zero cost.

Questions

Visit the poster session later!

Miguel Cárdenas, Rafael Herrera, Jose M. Monsalve, Isaac Bermudez

Universidad de Medellin · AMD · Universidad de los Andes

github.com/randreshg/a-pxm