Engineering a Hybrid Rust and MLIR Toolchain for AI Agents

A case study in canonical IR design, generated dialect metadata, and explicit compiler-runtime contracts

Miguel Cárdenas
Universidad de Medellin and LLVMCol

EuroLLVM 2026

The Synchronization Problem

Downstream MLIR systems that span multiple frontends, a custom dialect, binary artifacts, and a runtime face a recurring challenge: semantic drift. The same operation semantics appear in multiple places with slightly different assumptions. As the system evolves, invariants that were once aligned begin to diverge silently.

APXM hit this problem across a DSL, a graph IR, a custom MLIR dialect, a binary artifact format, and a Rust runtime. The central engineering challenge was preserving one semantics across every boundary.

The hard problem was not defining a dialect operation, but keeping the whole stack aligned as the system evolves.

Architecture Evolution

The final design was not assumed. It emerged through several iterations, each of which exposed a different failure mode in semantic synchronization.

- `v1` Rust-first stack with melior-heavy ideas. Drifted quickly as MLIR integration deepened.
- `v2` Hybrid MLIR compiler with runtime benchmarks. Duplicated semantics across Rust and C++.
- `v3` Runtime and ecosystem expansion. Documentation and contracts lagged behind code.
- `final` Canonical graph IR before MLIR. Explicit contracts. Consolidated semantic source of truth.

Why not all C++ or all Rust?

`All C++:` strong dialect locality, weak shared semantic layer across runtime and tooling

`All Rust:` strong ergonomics, but weak full-MLIR dialect story once verifiers and passes are needed

`Hybrid:` Rust owns shared semantics; MLIR owns dialect infrastructure and transformations

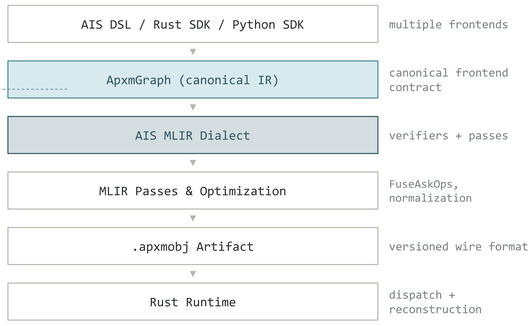
A Canonical IR Stabilized the Compiler Stack

The architecture that survived introduced three structural decisions. First, every frontend normalizes into a canonical IR called ApxmGraph before MLIR. This gave the system one place to validate structure, serialize graphs, and stabilize tooling. Second, operation semantics live in one Rust source of truth, which generates TableGen descriptors and pass metadata. Third, compiler-runtime synchronization is made explicit through wire contracts covering operation indices, encoding, and shared attribute keys.

Why ApxmGraph exists

- **Validation before MLIR:** structural correctness is checked before dialect lowering begins
- **Stable exchange format:** ApxmGraph serialization is versioned independently of MLIR
- **Better tooling:** graph inspection, diffing, and debugging operate on ApxmGraph, not raw MLIR
- **Less frontend/compiler entanglement:** frontends produce ApxmGraph, not dialect ops

Compilation Pipeline



The introduction of ApxmGraph as a canonical IR separated frontend normalization from MLIR lowering, stabilized the toolchain, and enabled verifiers to catch errors before runtime.

Compiler Value: Verification & Passes

Once the IR exposes workflow structure to MLIR, verifiers and passes stop being decorative and start producing real compiler value. Verification catches errors at compile time that would otherwise surface only at runtime, often after expensive execution.

Compile-Time Verification

The AIS MLIR dialect defines verifiers that reject malformed workflows before any runtime execution. Examples of compile-time rejections:

`undefined variable`
referenced variable not defined in any preceding operation

`invalid memory tier`
operation targets a memory tier not supported by its context

`empty capability set`
agent operation has no capabilities, guaranteed to fail at runtime

`malformed op attributes`
required attributes missing or have incompatible types

Domain-Specific Optimization

The IR exposes structure that opaque framework code hides. Once workflows are visible as MLIR operations, domain-specific passes become possible.

`FuseAskOps` pass

Verifier checks reject malformed workflows before runtime. Domain-specific passes collapse redundant operation chains into single fused operations.

Contracts, Tooling & Honest Limitations

The transition from a prototype to a maintainable toolchain required making implicit assumptions into explicit, documented contracts between the compiler and runtime.

Compiler-Runtime Contracts

- **Wire operation indices:** each operation has a stable numeric index shared between compiler and runtime
- **Artifact encoding:** the .apxmobj binary format is versioned and documented as an explicit contract
- **Shared attribute keys:** attribute names and value types are defined once in Rust and propagated to MLIR
- **Synchronization rules:** documented procedures for adding or modifying operations across the boundary

Validation & Tooling

- **CLI validation: apxm validate checks graph and artifact integrity from the command line**
- **MCP contract exposure: runtime contracts are exposed through a model context protocol interface**
- **Interface parity: CLI, SDK, and programmatic interfaces are tested for behavioral consistency**
- **Graph validation tests: automated test suites cover structural invariants of ApxmGraph**

Honest Limitations

- **Some runtime operations exist but are not yet fully wired through the MLIR artifact lowering path**
- **Documentation lags behind the code in some areas, particularly around newer operations**
- **Graph lowering still has unsupported edge cases for complex control-flow patterns**
- **Single-source generation approach may become too clever as operation count grows**

Explicit contracts made artifact evolution and runtime compatibility manageable. Visible limitations build trust.

Reusable Lessons and Open Questions

Takeaways

1. Normalize before MLIR. A canonical IR before the dialect reduced frontend/compiler drift and gave the system one stable validation point.
2. Keep semantics in one source of truth. Rust definitions generate TableGen and pass descriptors, eliminating semantic duplication across languages.
3. Treat compiler-runtime synchronization as a contract. Wire encoding, operation indices, and shared attribute keys are documented contracts, not tribal knowledge.
4. Downstream MLIR value is real. Once semantics are aligned, verifiers catch real errors and domain-specific passes produce real optimizations.

Open Questions

- How far should single-source generation go before it becomes too clever? At what point does generating TableGen from Rust introduce more complexity than it removes?
- When should a canonical IR stay outside MLIR, and when should it become a dialect-native representation? What criteria determine the right boundary?
- How should operation growth be managed without wire-format lag? As new operations are added, how can the system prevent the binary format from falling behind?
- Which heuristics belong in compiler passes versus runtime policy? Some optimization decisions depend on runtime information unavailable at compile time.

APXM is a domain case study, but the pattern is reusable for any downstream MLIR stack spanning frontends, a custom dialect, generated artifacts, and a runtime. The hardest problem is not defining operations. It is keeping one semantics aligned across every boundary.

Happy to discuss dialect design, graph normalization, and compiler-runtime contracts.

Miguel Cárdenas[1], Rafael Herrera[2], Jose M. Monsalve[2], Isaac Bermudez[3]

[1]Univesidad de Medellin, [2]Advanced Micro Devices, [3]Universidad de los Andes

github.com/randreshg/a-pxm