Interpretable AI with MLIR (IAM): A Unified Framework for Neural Network Transparency

PhD Research Statement || Shakthi Bachala

Department of Computer Science, University of Nebraska–Lincoln

October 2025

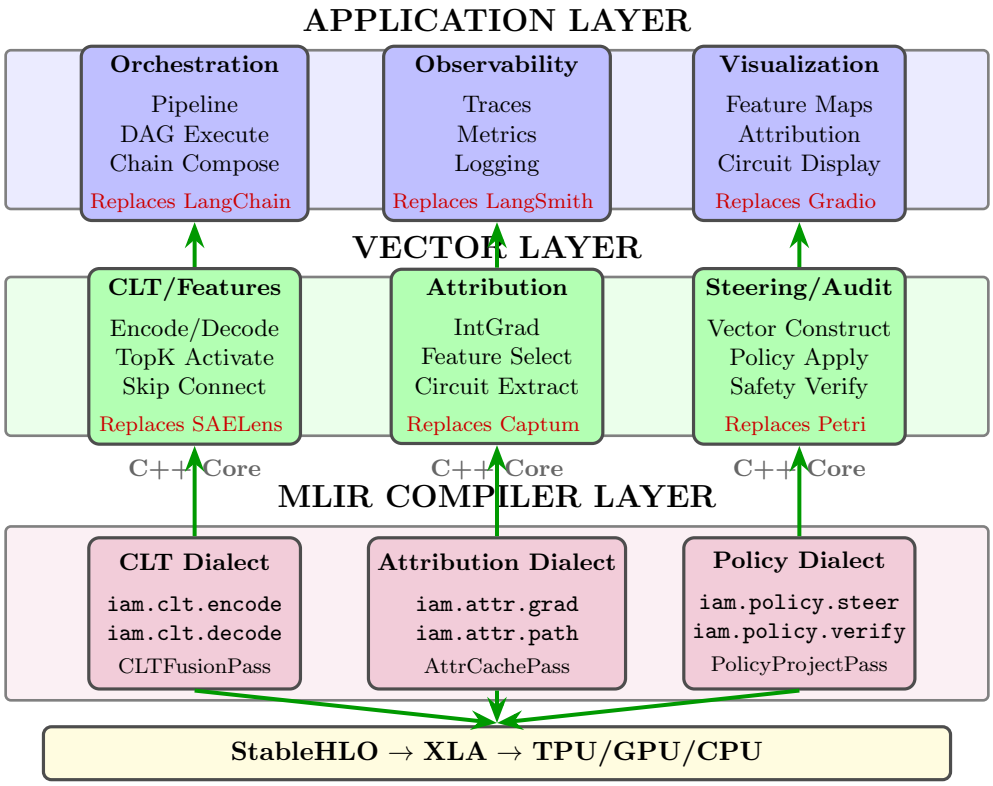
1 The Fragmentation Crisis in AI Interpretability

Modern AI systems have become powerful yet opaque. Understanding, aligning and auditing their behavior requires multiple specialized tools, creating severe practical challenges.

1.1 The Current Landscape

Today's interpretability ecosystem consists of isolated tools operating independently:

2.1 Architectural Layers



Note: This represents a sample vision. The actual industry interpretability landscape is significantly more fragmented with dozens of specialized tools at each layer.

Application Layer provides the user-facing API for orchestration, observability, and visualization, replacing LangChain, LangSmith, and Gradio with unified interfaces.
Vector Layer implements core interpretability primitives in high-performance C++, replacing SAELens, Captum, and Petri with integrated CLT operations, attribution analysis, and policy steering.
MLIR Compiler Layer defines custom dialects representing interpretability as first-class compiler constructs, enabling optimization passes that fuse operations and cache results.
Current fragmented tools require multiple forward passes and operate independently. IAM's vertical integration enables the compiler to reason about the full computation graph, fusing interpretability with model execution in a single optimized pipeline.

1.2 The IAM Vision

IAM reimagines interpretability as a first-class compilation primitive. By treating interpretability operations as native compiler concerns, IAM enables systematic optimization while providing unprecedented transparency.
The framework unifies attribution, visualization, tracing, and orchestration into a single pipeline built on JAX's vertically integrated architecture. This eliminates redundant computation and enables compiler optimization through XLA's fusion and memory planning. JAX's functional paradigm ensures transformations compose naturally.

2 IAM Framework Architecture

The architectural layers diagram (Section 1.1) shows IAM's three-tier design. This section details the technical implementation of each layer and how they integrate.

2.1 Foundation: JAX Vertical Integration

IAM builds on JAX's vertically integrated architecture:
JAX Primitives: JAX represents computation through composable primitives transformed by grad, jit, and vmap. IAM defines custom primitives for interpretability that integrate seamlessly. Attribution becomes a natural transformation alongside autodiff.
XLA Compilation: JAX lowers to XLA's HLO enabling aggressive optimization. IAM's primitives lower directly to HLO, allowing the compiler to fuse interpretability operations with model computation.
Memory Optimization: JAX's buffer donation enables sophisticated sharing. Activation buffers for attribution reuse memory across operations, with XLA automatically identifying sharing opportunities.

2.2 MLIR Dialect for Interpretability

IAM extends StableHLO with custom operations representing interpretability primitives:
Sparse Autoencoder Operations: Decompose activations as StableHLO operations that JAX's compiler fuses with model inference and lowers to efficient device code.
Attribution Operations: Computing causal relationships as StableHLO transformations with algebraic simplification and batched computation through vmap.
Circuit Discovery & Policy Operations: Path identification formalized as HLO graph analysis, with behavioral control as StableHLO primitives integrating with JAX's transformation system.

2.3 Progressive Lowering and Optimization

IAM leverages JAX's vertical integration through XLA and MLIR. JAX traces Python to Jaxpr, lowering to StableHLO with IAM's custom operations. StableHLO lowers through MHLO and Linalg to LLVM IR or GPU representations. Specialized passes fuse operations, identify buffer reuse, and exploit sparsity, with XLA generating efficient device code.

3 Unified Interpretability Capabilities

3.1 Attribution Through Integrated Gradients

Attribution identifies which inputs influenced outputs. IAM computes attribution through JAX's transformation system. Custom transformations extend grad to simultaneously compute gradients and attributions, with the compiler enabling algebraic simplification.
For integrated gradients, IAM's JAX primitives enable vectorization through vmap. Rather than sequential steps, the compiler generates vectorized code computing all steps in parallel with optimized memory access.

3.2 Visualization and Tracing

IAM's visualization integrates with sparse autoencoder operations as JAX primitives. Features extracted during jit-compiled inference are immediately available. XLA's memory planning reuses activation buffers, eliminating overhead. Heatmap generation, attention visualization, and feature displays operate on unified representations.

Tracing integrates with XLA's profiling infrastructure. JAX's jit compilation produces HLO graphs that XLA executes with built-in instrumentation. Custom primitives register with XLA's profiler, capturing operations at the compilation level without Python overhead.

3.3 Orchestration

IAM provides orchestration through JAX's functional composition. Interpretability operations compose as JAX transformations, maintaining compiler visibility for global optimization. Conditional attribution leverages JAX's cond primitive, with XLA generating specialized code paths. The jit compiler eliminates conditional overhead when patterns are statically determinable.

4 Policy Steering for Behavioral Control

Beyond understanding behavior, controlling it is essential for safety-critical AI. Policy steering provides precise control guiding models toward desired outcomes.

4.1 Skip-Transcoder Architecture

Skip-transcoders extend sparse autoencoders with residual connections preserving information flow while maintaining interpretability. Skip connections provide direct paths not requiring decomposition, reserving sparse pathways for patterns benefiting from explicit representation. IAM implements skip-transcoders as JAX primitives lowering to StableHLO. XLA fuses transcoder operations with model layers.

4.2 Steering Vector Construction and Application

Policy steering constructs vectors in activation space representing desired changes, adding these to activations during inference. IAM's policy framework provides declarative specifications with automatic feature discovery through contrastive analysis. Statistical analysis identifies differentiating features; causal validation verifies effectiveness.

Steering application must be efficient. IAM represents steering as JAX primitives that jit compiles with model execution. XLA's fusion passes combine steering with model layers. JAX's functional nature ensures clean composition.

4.3 Multi-Policy Composition

Real-world applications require simultaneous policy enforcement. IAM supports compositional specifications through JAX's functional composition. Multiple vectors combine through tree operations and custom transformations. The compiler reasons about interactions through symbolic execution, resolving conflicts via optimization passes. XLA fuses composed operations. Vectorization through vmap enables parallel policy application.

4.4 MLIR Integration

Policy steering operations integrate with JAX's compilation pipeline as primitives lowering to custom StableHLO operations. This enables aggressive optimization through XLA's fusion passes, memory optimization via buffer sharing, and sparse operation optimizations.

The lowering pipeline: JAX primitives to Jaxpr, Jaxpr to StableHLO, StableHLO through simplification, then MHLO for hardware optimization, finally to LLVM IR or NVVM IR. JAX's functional semantics ensure correctness preservation throughout.

5 Conclusion and Broader Impact

IAM demonstrates that unified interpretability for AI systems is achievable through vertical integration with compilation infrastructure. By treating interpretability as a first-class compilation primitive within JAX's ecosystem, the framework eliminates the fragmentation plaguing current approaches while enabling unprecedented performance through compiler-level optimization.

The key insight: interpretability operations—attribution, visualization, tracing, and orchestration—naturally compose as JAX transformations. This functional paradigm ensures clean semantics while XLA's optimization pipeline automatically fuses, schedules, and lowers interpretability primitives alongside model computation. The

result is a single unified compilation path from high-level Python specifications through optimized device execution.

For policy steering, this integration proves equally powerful. Skip-transcoders implemented as JAX primitives enable behavioral control through steering vectors that compose cleanly with model inference. Multi-policy scenarios benefit from automatic fusion and conflict resolution at the compiler level. Real-time steering becomes practical for production deployment through XLA's aggressive optimization.

The broader impact extends beyond technical performance. Unified interpretability infrastructure enables developers to understand and control AI behavior as naturally as they debug traditional software. Policy-based steering provides precise control for safety-critical applications. The vertical integration ensures these capabilities remain performant at production scale.

This work establishes the foundation for trustworthy AI systems where interpretability is not an afterthought but a fundamental design principle, engineered through rigorous compiler abstractions and optimized through decades of compilation infrastructure development.

Dissertation Scope

This dissertation focuses on the Google stack (JAX, XLA, StableHLO) for the compilation pipeline. The research targets code generation models exclusively; no NLP, audio, video, or other modalities. Work is limited to the compute plane (inference and training optimization); data plane concerns are out of scope. Future extensions will explore Chain of Thought monitoring, Model Context Protocol (MCP) integration, and multi-agent systems, but these remain beyond the current dissertation boundaries.

Publications

1. **IAM: Interpretable AI with MLIR - A Compiler-Integrated Framework for Trustworthy Code Generation**
   Shakthi Bachala, and Witawas Srisa-An
   (In preparation for submission)

2. **Compiler-Native Policy Steering: MLIR Primitives for Efficient Behavioral Control in Neural Networks**
   Shakthi Bachala, and Witawas Srisa-An
   (In preparation for submission)

3. **ReHAna: An Efficient Program Analysis Framework to Uncover Reflective Code in Android**
   Shakthi Bachala, Yutaka Tsutano, Witawas Srisa-An, Gregg Rothermel, and Jackson Dinh
   EAI MobiQuitous 2021

4. **JitAna: A Modern Hybrid Program Analysis Framework for Android Platforms**
   Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-An, Gregg Rothermel, and Jackson Dinh
   Journal of Computer Languages, Volume 52, 2019

5. **GranDroid: Graph-based Detection of Malicious Network Behaviors in Android Applications**
   Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an, and *Shakthi Bachala*
   SecureComm 2018

6. **An Efficient, Robust, and Scalable Approach for Analyzing Interacting Android Apps**
   Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-An, Gregg Rothermel, and Jackson Dinh
   ICSE 2017