

Multi-Agent Framework for Semantic Code Divergence Benchmark

Final Report

Team - UG_FS_2

Pranava Sai Maganti
Meghasyam Peddireddy
Kaining Gao
Farhan Ahmed

Client

Md. Muzakker Hossain
Dr. Ali Jannesari

Course Staff

Dr. Simanta Mitra
Fatema Siddika (TA)

COM S 4020C - Spring 2026

Contents

1 Requirements 1

1.1 Project Overview 1

1.2 Problem Statement 2

1.3 Project Goals and Objectives 2

1.4 Target Users and Research Community 3

1.5 Functional Requirements 3

1.6 Non-Functional Requirements 3

1.7 Constraints & Assumptions 3

2 Design 4

2.1 System Architecture 5

2.2 Multi-Agent Framework Design 5

2.3 Dataset Generation Pipeline 5

3 Work Done 6

4 Results Achieved 7

5 Conclusion 8

Appendix A Demo Presentation Slides 9

Appendix B Additional Figures and Diagrams 10

Appendix C API Documentation 11

Appendix D Sample MLIR Programs 12

1

Requirements

1.1 Project Overview

1.1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to define the functional and non-functional requirements of **MASCoD-MLIRBench**, an execution-grounded multi-agent benchmarking framework designed for evaluating the semantic reasoning capabilities of Large Language Models (LLMs) over MLIR compiler intermediate representations.

This document establishes the expected behavior of the framework, describes the interactions between the system components, and outlines the requirements necessary for the system to be considered complete and operational for research-driven compiler semantic benchmarking.

1.1.2 Scope

MASCoD-MLIRBench is a research-oriented benchmarking framework that focuses on evaluating semantic understanding at the compiler intermediate representation level instead of traditional source-code reasoning.

The framework supports controlled mutation generation, deterministic execution validation, semantic comparison, benchmark dataset generation, and downstream LLM evaluation workflows. The current implementation supports the `torch` and `linalg` MLIR dialects and utilizes a modular multi-agent architecture for scalability and extensibility.

The system additionally supports downstream tasks including:

- Semantic reasoning evaluation
- Automatic correction of compiler-level failures
- Execution-grounded dataset generation
- Controlled mutation analysis
- Benchmark reproducibility workflows

1.1.3 Definitions, Acronyms, and Abbreviations

- **MLIR:** Multi-Level Intermediate Representation
- **LLM:** Large Language Model
- **IREE:** Intermediate Representation Execution Environment
- **SSA:** Static Single Assignment
- **IR:** Intermediate Representation
- **JSON:** JavaScript Object Notation
- **HPC:** High Performance Computing
- **EQUIV:** Semantically Equivalent Execution Result
- **DIVERGE:** Semantically Divergent Execution Result
- **ERROR:** Execution or Compilation Failure
- **SFT:** Supervised Fine-Tuning

1.1.4 References

- MLIR Official Documentation
- LLVM Infrastructure Documentation
- IREE Compiler Documentation
- Torch-MLIR Documentation
- Iowa State University Computer Science Curriculum

1.2 Overall Description

1.2.1 Framework

MASCoD-MLIRBench is designed as an execution-grounded semantic benchmarking framework for compiler intermediate representations.

Unlike traditional AI code benchmarks that focus primarily on source-code genera-

tion and completion tasks, MASCoD-MLIRBench evaluates whether LLMs can reason about compiler-level semantics under controlled mutations and execution-based validation.

The framework introduces a modular multi-agent orchestration pipeline consisting of:

- Root Agent
- Mutation Agent
- Execution Sandbox Agent
- LLM Reasoning Agent
- Judge Agent

Each agent performs a specialized task and communicates through structured JSON-based interfaces with restricted context-sharing policies.

1.2.2 Concept of Operations

The system operates as an execution-grounded semantic benchmarking pipeline.

The Root Agent initializes the benchmarking workflow and coordinates communication between agents.

The Mutation Agent accepts valid MLIR programs and generates controlled semantic mutations across five mutation levels:

- Level 1: Syntactic Mutations
- Level 2: Parameter Mutations
- Level 3: Operator Mutations
- Level 4: Structural Mutations
- Level 5: Failure-Inducing Mutations

The Execution Sandbox Agent validates, compiles, and executes both original and mutated MLIR programs using MLIR verifier pipelines and IREE execution workflows.

The LLM Reasoning Agent performs semantic reasoning and predicts relationships between original and mutated programs without execution access.

The Judge Agent compares execution-derived ground truth against LLM predictions and generates deterministic benchmark labels.

1.2.3 Curriculum Mapping

The implementation of MASCoD-MLIRBench integrates concepts from multiple Computer Science courses completed throughout the undergraduate curriculum:

- **COM S 3420 - Principles of Programming Languages:** Compiler semantics, type systems, program transformations, and MLIR reasoning workflows.
- **COM S 3090 - Software Development Practices:** Collaborative software engineering, backend APIs, and modular system implementation.
- **COM S 3190 - Construction of User Interfaces:** Interactive benchmark visualization dashboards and frontend-based experiment presentation interfaces.
- **COM S 4740 / COM S 5740 - Machine Learning:** LLM evaluation methodologies, supervised fine-tuning workflows, and benchmark analysis.
- **COM S 3210 - Computer Architecture and Machine-Level Programming:** Deterministic execution validation and hardware-aware execution pipelines.
- **COM S 3630 - Database Management Systems:** Structured dataset serialization, JSON storage, and benchmark metadata organization.
- **COM S 4170 - Software Testing:** Validation pipelines, benchmark correctness testing, reproducibility analysis, and deterministic evaluation workflows.

1.2.4 Major User Interfaces

The framework provides the following major interfaces:

- Benchmark generation dashboard
- Benchmark execution monitoring interface
- Mutation visualization interface
- Benchmark dataset viewer
- Semantic reasoning evaluation interface
- API-based benchmark execution endpoints
- Benchmark statistics and visualization panels

1.2.5 Hardware Interfaces

The system requires:

- Hardware supporting modern web technologies
- Access to HPC infrastructure such as the ISU Nova cluster
- Multi-core CPU environments for parallel benchmark execution

- Sufficient storage for benchmark datasets and execution outputs

1.3 Framework Functionalities

The system provides the following core functionalities:

1.3.1 FF-1: Controlled Mutation Generation

The Mutation Agent generates controlled semantic mutations over MLIR programs using dialect-aware rules and compiler legality constraints.

1.3.2 FF-2: Compiler Validation

The framework validates generated MLIR programs using MLIR verifier pipelines and Torch-MLIR lowering workflows.

1.3.3 FF-3: Execution-Grounded Labeling

The Execution Sandbox Agent compiles and executes MLIR programs through IREE and compares outputs deterministically.

1.3.4 FF-4: Semantic Classification

The Judge Agent generates benchmark labels including:

- `EQUIV`
- `DIVERGE`
- `ERROR`

1.3.5 FF-5: Dataset Serialization

The framework stores benchmark outputs and metadata in structured JSON format for reproducibility.

1.3.6 FF-6: Downstream Fine-Tuning Tasks

The framework supports downstream tasks including:

- Semantic reasoning evaluation
- Automatic semantic error correction

- Execution-aware supervised fine-tuning

1.3.7 FF-7: Benchmark Visualization

The frontend dashboard visualizes benchmark statistics, mutation distributions, and semantic evaluation outputs.

1.4 User Characteristics

The primary users of the system include:

- Compiler researchers
- Machine learning researchers
- AI benchmarking researchers
- Systems and programming languages researchers

Users are expected to possess familiarity with:

- Compiler infrastructures
- MLIR and LLVM ecosystems
- Python-based execution workflows
- Machine learning concepts
- JSON-based data structures

1.5 Constraints

- The current implementation is optimized specifically for the `torch` and `linalg` MLIR dialects.
- Benchmark generation depends on the availability of MLIR, Torch-MLIR, LLVM, and IREE infrastructures.
- Large-scale dataset generation requires access to HPC systems.
- Mutation legality is dependent on compiler verification pipelines.
- Execution correctness assumes deterministic runtime behavior.

1.6 Assumptions and Dependencies

1.6.1 Assumptions

- Input MLIR programs are syntactically valid before mutation generation begins.
- The execution environment supports all required compiler toolchains.
- Generated mutations follow dialect-aware structural constraints.

1.6.2 Dependencies

- LLVM
- MLIR
- Torch-MLIR
- IREE
- Python runtime
- HPC infrastructure
- FastAPI backend services
- JSON serialization libraries

1.7 Specific Requirements

1.7.1 FEATURE-1: Multi-Agent Orchestration

The system shall support modular communication between independent agents using JSON-based interfaces.

1.7.2 FEATURE-2: Semantic Mutation Generation

The Mutation Agent shall generate controlled mutations across five mutation levels while preserving dialect constraints.

1.7.3 FEATURE-3: Compiler Verification

The framework shall validate generated MLIR programs using MLIR verifier pipelines.

1.7.4 FEATURE-4: Deterministic Execution

The Execution Sandbox Agent shall compile and execute MLIR programs deterministically through IREE.

1.7.5 FEATURE-5: Benchmark Label Generation

The Judge Agent shall produce deterministic semantic labels based on execution comparisons.

1.7.6 FEATURE-6: Dataset Generation

The framework shall serialize benchmark samples and metadata into reproducible JSON-based datasets.

1.7.7 FEATURE-7: Visualization Dashboard

The frontend system shall provide benchmark visualization and execution monitoring capabilities.

1.8 Performance Requirements

- The framework shall support generation of thousands of benchmark samples.
- The system shall support parallel execution workflows on HPC systems.
- Dataset serialization shall remain deterministic across benchmark runs.
- Mutation validation shall complete within acceptable compiler verification time limits.

1.9 Design Constraints

- The architecture must preserve modular agent separation.
- Agent communication must occur only through structured interfaces.
- Context-sharing between agents must remain restricted to task-specific information.
- Compiler legality must be preserved for all non-failure-inducing mutation levels.

1.10 Software System Attributes

1.10.1 Reliability

Execution-derived labels must remain deterministic and reproducible across repeated benchmark executions.

1.10.2 Availability

The system shall remain operational within supported HPC and server environments.

1.10.3 Security

The framework shall enforce controlled inter-agent communication with restricted context-sharing policies.

1.10.4 Maintainability

The modular architecture shall support future dialect additions and benchmark extensions.

1.10.5 Portability

The framework shall remain deployable across Linux-based research computing environments.

1.11 Other Requirements

- The framework should support future integration of additional MLIR dialects.
- The benchmark dataset should remain reproducible for external research validation.
- The system should support future AI-agent benchmarking extensions.
- Benchmark outputs should support downstream fine-tuning workflows.

2

Design

2.1 System Architecture

MASCoD-MLIRBench is designed as a modular multi-agent benchmark pipeline for generating, executing, evaluating, and scoring MLIR program pairs. The system follows a staged architecture in which each agent is responsible for a specific part of the benchmark workflow. This separation makes the framework easier to debug, extend, and scale across different MLIR dialects, mutation types, compiler backends, and LLM evaluation settings.

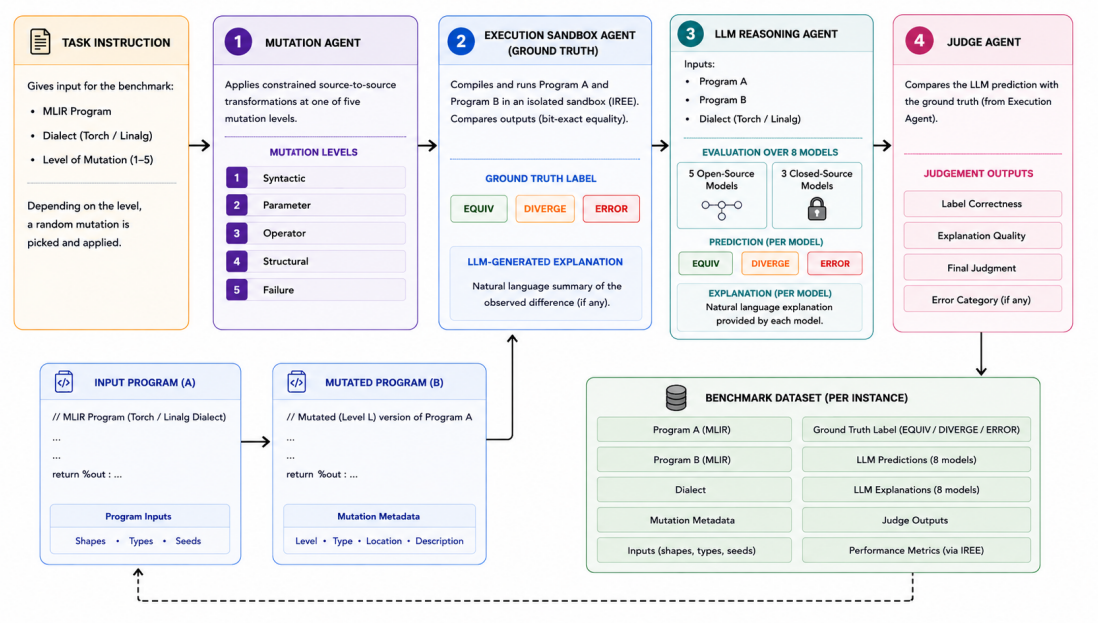


Figure 2.1: Overall system architecture of the MASCoD-MLIRBench multi-agent benchmark framework.

As shown in Figure 2.2, the pipeline begins with a valid MLIR input program and user-selected configuration options such as dialect and mutation level. The Root

Agent validates the request and forwards the normalized payload to the Mutation Agent. The Mutation Agent then applies dialect-aware mutation rules to generate a paired benchmark sample consisting of Program A, the original MLIR program, and Program B, the mutated MLIR program.

The generated program pair is passed to the Execution Sandbox Agent, which provides the execution-grounded foundation of the benchmark. This agent compiles and executes both programs through the configured MLIR, Torch-MLIR, LLVM, and IREE toolchains. It records compilation status, runtime status, program outputs, benchmark timing results, and whether the two programs produce divergent outputs. These execution results become the ground truth labels used by the rest of the framework.

After execution validation, the LLM Reasoning Agent evaluates the same program pair without being given the execution result or ground-truth label. It asks multiple language models to predict the behavior of Program A and Program B, classify the pair as `EQUIV`, `DIVERGE`, or `ERROR`, and provide an explanation grounded in MLIR semantics. This stage measures whether models can reason about compiler IR behavior rather than simply observe execution outputs.

Finally, the Judge Agent compares each model prediction against the execution-derived ground truth. It computes hard label accuracy, evaluates predicted output correctness, scores explanation quality, and produces a final verdict for each model response. The final benchmark payload stores mutation metadata, execution results, model reasoning outputs, judge scores, and logs in a structured JSON format.

This architecture provides several advantages. First, each agent can be developed and tested independently. Second, execution-derived labels ensure that benchmark answers are grounded in real compiler behavior. Third, the modular payload structure allows the system to support additional dialects, mutation rules, models, judge rubrics, and storage backends in future extensions. Overall, the architecture transforms MLIR mutation generation into a complete semantic reasoning benchmark pipeline.

2.2 Multi-Agent Framework Design

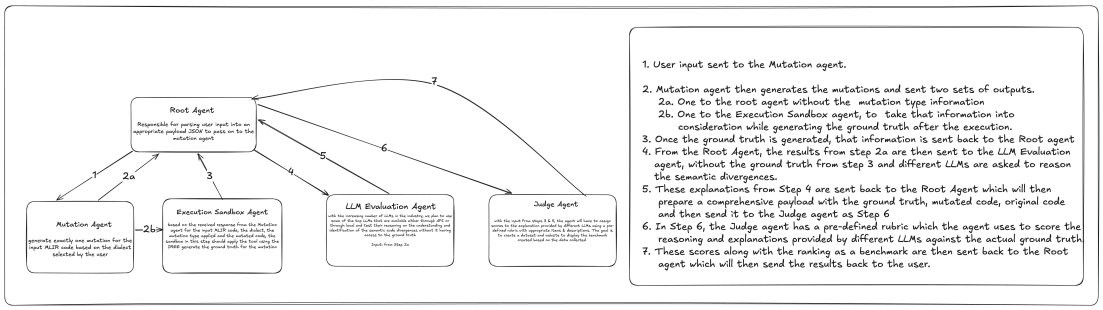


Figure 2.2: End-to-end multi-agent orchestration pipeline of MASCoD-MLIRBench. The Root Agent coordinates communication between the Mutation, Execution Sandbox, LLM Evaluation, and Judge agents using structured JSON-based payloads with strict need-to-know context isolation. The framework generates controlled MLIR mutations, derives execution-grounded semantic truth through sandbox validation, evaluates reasoning produced by multiple LLMs, and aggregates benchmark scores and rankings through the Judge Agent before returning the final evaluation results.

2.3 Database and Dataset Storage Design

MASCoD-MLIRBench stores benchmark data using structured JSON-based records rather than a traditional relational database. This design was chosen because each benchmark sample contains heterogeneous information, including MLIR source programs, mutation metadata, execution outputs, model predictions, judge scores, benchmark timing results, and debugging logs. JSON storage makes the dataset easy to serialize, inspect, replay, and share with downstream research workflows.

The primary API entry point accepts a benchmark request containing the original MLIR program, selected dialect, and mutation level:

```
{
  "input_mlir": "...",
  "dialect": "linalg | torch",
  "level": 1
}
```

During execution, the system progressively expands this request into an agent-based payload. The Mutation Agent stores mutation context and the blind program pair, the Execution Sandbox stores execution-grounded outputs and labels, the LLM Reasoning Agent stores model predictions, and the Judge Agent stores final evaluation scores.

The final benchmark payload is organized around the `Agents` object:

```
{
  "Agents": {
    "Mutation_Agent": {
      "context_results": {
        "requirement": {...},
        "metadata": {...}
      },
      "blind_results": {
        "input_mlir": "...",
        "mutated_mlir": "..."
      },
      "toExecutionAgent": {
        "input_mlir": "...",
        "mutated_mlir": "...",
        "dialect": "linalg | torch"
      }
    },

    "Execution_Agent": {
      "execution_results": {
        "programA_output": "...",
        "programB_output": "...",
        "compilation_status_A": "success | error",
        "compilation_status_B": "success | error",
        "outputs_diverge": true,
        "divergence_details": "...",
        "semantic_explanation": "...",
        "status": "success | compile_error | runtime_error"
      },
      "benchmark_results": {
        "benchmark_result_program_a": {...},
        "benchmark_result_program_b": {...}
      },
      "embed_inputs": {
        "embed_inputs_program_a": {...},
        "embed_inputs_program_b": {...}
      },
```

```
  "toLLMReasoning": {
    "programA": "...",
    "programB": "...",
    "dialect": "linalg | torch",
    "embed_inputs": {...}
  }
},

"LLM_Reasoning_Agent": {
  "results": [
    {
      "provider": "...",
      "llm_name": "...",
      "model_key": "...",
      "predicted_output_A": "...",
      "predicted_output_B": "...",
      "predicted_label": "EQUIV | DIVERGE | ERROR",
      "confidence": 0.0,
      "key_differences": [],
      "witness": null,
      "explanation": "...",
      "status": "SUCCESS"
    }
  ]
},

"Judge_Agent": {
  "evaluations": [
    {
      "model_key": "...",
      "llm_name": "...",
      "provider": "...",
      "judge_payload": {
        "program_id": "...",
        "pair_id": "...",
        "evaluated_model": "...",
        "ground_truth_label": "EQUIV | DIVERGE | ERROR",
        "predicted_label": "EQUIV | DIVERGE | ERROR",
        "subscores": {
```

```
              "hard_label_accuracy": 1.0,
              "result_accuracy": 10.0,
              "explanation_rating": 4.0
            },
            "weighted_scores": {...},
            "total_score": 100.0,
            "verdict": "CORRECT"
          }
        }
      ],
      "evaluated_count": 8
    }
  },

  "logs": [...]
}
```

The dataset design separates storage into several logical layers. The mutation layer preserves the original and mutated MLIR programs together with mutation requirements and metadata. The execution layer stores compiler status, runtime outputs, divergence labels, benchmark timing information, and semantic explanations. The reasoning layer stores one prediction record per evaluated model. The judge layer stores normalized ground-truth labels, model labels, subscores, weighted scores, final verdicts, and evidence used for scoring.

For local dataset generation, intermediate mutation records are written as timestamped JSON files under `data/runs`. Each record stores the mutation metadata and blind program pair:

```
{
  "metadata_output": {...},
  "blind_output": {
    "input_mlir": "...",
    "mutated_mlir": "..."
  }
}
```

For larger offline benchmark runs, the same schema is stored as JSONL files, where each line represents one benchmark sample or one model evaluation result. This format supports streaming writes, resume behavior, partial reruns, and scalable processing across many program pairs. Unique identifiers such as `program_id`, `mutation_id`,

`dialect`, and `level` are used to cross-reference mutation records, execution results, timing results, model predictions, and judge outputs.

This storage design keeps the benchmark reproducible and extensible. A single final payload contains enough information to reconstruct the full benchmark path from mutation generation through execution validation, model reasoning, and final judging. At the same time, the modular JSON structure allows future work to add new dialects, new compiler backends, additional reasoning models, database indexing, or dashboard visualization without changing the core benchmark record format.

2.4 Technology Stack

MASCoD-MLIRBench was developed using a combination of compiler infrastructures, LLM orchestration frameworks, distributed execution systems, and modern backend technologies to support scalable semantic benchmarking over MLIR compiler intermediate representations.

The framework was primarily implemented in **Python**, which was used for agent orchestration, mutation generation, execution coordination, dataset construction, benchmark evaluation, and API integration workflows.

The benchmark is built on top of the **MLIR** and **LLVM** compiler ecosystems. The current implementation focuses on the `torch` and `linalg` MLIR dialects while integrating compiler tooling including:

- `mlir-opt`
- `torch-mlir-opt`
- **Torch-MLIR**
- **LLVM**

These tools were used for legality verification, dialect lowering, compiler validation, and semantic mutation analysis throughout the benchmark pipeline.

Execution-grounded semantic validation was performed using **IREE**, which compiled and executed both original and mutated MLIR programs inside isolated execution environments to derive deterministic semantic labels.

The multi-agent orchestration pipeline was implemented using **LangGraph** and **LangChain**, enabling modular communication between the Root Agent, Mutation Agent, Execution Sandbox Agent, LLM Reasoning Agent, and Judge Agent through structured JSON-based payload workflows.

Large-scale benchmark generation and execution workflows were deployed on **High Performance Computing (HPC)** infrastructure provided by the Iowa State Uni-

versity Nova cluster. The infrastructure utilized high-memory CPU nodes together with **GPU-enabled systems** for scalable mutation generation, compiler validation, and LLM evaluation workloads.

Additional supporting technologies included:

- **FastAPI** for backend API services
- **Uvicorn** for asynchronous API hosting
- **NumPy** for deterministic execution comparison
- **JSON** for inter-agent communication and dataset serialization
- **GitLab** for version control
- **Hugging Face Datasets** for benchmark dataset hosting
- **React** for frontend visualization interfaces

Together, these technologies enabled the development of a scalable, execution-grounded, and modular benchmarking framework for evaluating semantic reasoning capabilities of large language models over compiler intermediate representations.

3

Work Done

3.1 Team Contributions

The MASCoD-MLIRBench project evolved beyond the scope of a traditional senior design project and developed into a research-oriented benchmark framework targeting semantic reasoning evaluation for compiler intermediate representations.

Throughout the project lifecycle, the team collaboratively designed and implemented a scalable multi-agent benchmarking infrastructure integrating mutation generation, execution-grounded validation, semantic reasoning evaluation, and large-scale dataset generation workflows.

The project additionally resulted in:

- the construction of a publishable benchmark dataset for MLIR semantic reasoning evaluation,
- the development of execution-grounded mutation validation pipelines,
- downstream experimentation involving semantic reasoning fine-tuning and autocorrection tasks, and
- the preparation and submission of a research paper to an A* conference venue.

The benchmark framework and associated datasets were designed to support future research in compiler infrastructures, semantic reasoning systems, AI-for-code evaluation, and MLIR-aware language model analysis.

Team members contributed across multiple areas including mutation generation, execution validation, infrastructure deployment, dataset engineering, frontend systems, API development, benchmark evaluation, experimentation, and research paper preparation.

3.2 Pranava Sai Maganti

My primary contribution to MASCoD-MLIRBench focused on the design and implementation of the Mutation Agent together with the overall semantic mutation generation infrastructure used throughout the benchmark pipeline.

I designed and implemented dialect-aware mutation workflows for both the `torch` and `linalg` MLIR dialects. This included developing semantic mutation rules across all five mutation levels ranging from verifier-aware mutations to structural and failure-inducing semantic transformations.

A major portion of my work involved building inspection workflows capable of analyzing:

- SSA dependencies,
- tensor shapes,
- affine indexing maps,
- iterator structures,
- operation hierarchies, and
- verifier-aware legality constraints.

I additionally integrated GPT-based mutation generation workflows and designed the metadata generation pipelines used for structured benchmark dataset construction.

Beyond the mutation infrastructure, I also contributed to:

- JSON-based dataset serialization,
- API orchestration between the COM S public server and the Nova HPC cluster,
- SSH reverse tunneling infrastructure,
- Torch-MLIR, LLVM, and IREE integration workflows,
- benchmark generation deployment on Nova,
- execution validation coordination,
- downstream experimentation involving semantic reasoning fine-tuning and automatic correction tasks, and
- research paper preparation and evaluation workflows.

A significant portion of my work focused on ensuring that generated mutations remained semantically meaningful while still preserving execution-grounded validation correctness across compiler execution pipelines.

I also worked on debugging infrastructure issues related to compiler tooling, lowering workflows, HPC deployment environments, and API communication between the public server and Nova-based validation services.

3.3 Meghasyam Peddireddy

My primary contribution was the design and implementation of the LLM Reasoning Agent, which is Phase 3 of the MASCoD-MLIRBench pipeline. This agent evaluates whether LLMs can reason about the behavior of MLIR program pairs without being given execution results, compiler traces, lowered IR, or ground-truth labels.

The Reasoning Agent receives Program A, Program B, the MLIR dialect, and concrete input values from the previous pipeline stages. It then asks each model to predict the behavior of both programs and classify the pair as `EQUIV`, `DIVERGE`, or `ERROR`. The purpose of this component is to test real semantic reasoning instead of allowing the model to rely on output lookup or surface-level syntax matching.

A major design choice was the predict-then-compare protocol. Instead of asking the model to immediately choose a label, the prompt first requires it to analyze each program independently. The model must inspect the function signature, trace SSA values, reason about tensor shapes and element types, interpret `linalg.generic` regions, indexing maps, iterator types, and arithmetic operations, and then describe the predicted output of each program. Only after this step does the model compare the two behaviors and assign a final label.

I also implemented a structured JSON output contract so the Judge Agent could consume results consistently. Each model response includes `predicted_output_A`, `predicted_output_B`, `label`, `confidence`, `key_differences`, `witness`, and `explanation`. This lets the system evaluate not only whether the model selected the correct label, but also whether it predicted the correct behavior and gave a grounded explanation.

The Reasoning Agent supports multiple LLMs through a model registry. In the final benchmark, it was configured to run eight models, including open-weight and closed-source systems such as Qwen, Llama, GLM, Kimi, GPT-OSS, GPT-4.1 Mini, Claude Sonnet, and Claude Haiku. All models were queried using the same task format so their reasoning ability could be compared fairly.

A significant part of my work involved making the agent robust to inconsistent model outputs. Different models sometimes returned markdown code fences, prose around JSON, malformed JSON, missing fields, invalid labels, empty responses, or confidence values outside the expected range. I implemented parsing and validation logic that strips code fences, extracts valid JSON objects, checks schema fields, validates labels, clamps confidence values, normalizes predicted outputs, records raw response snippets, and preserves error status for debugging.

The agent also includes retry behavior. If a model fails to return a valid response, the system retries once with a stronger forced-verdict instruction. If the model still fails, the row is marked with an error status while the rest of the models continue

running. This prevents one model failure from stopping the entire benchmark.

Through this work, I learned how to connect compiler IR analysis with LLM evaluation. I gained experience with MLIR semantics, SSA tracing, tensor-shape reasoning, dialect-level program behavior, prompt engineering, structured JSON generation, LangChain-style model routing, OpenAI-compatible APIs, Anthropic APIs, and reproducible multi-model benchmarking.

Overall, my work on the Reasoning Agent helped the project measure whether LLMs truly understand MLIR program behavior or only recognize surface-level code patterns.

3.4 Kaining Gao

My primary contribution to MASCoD-MLIRBench focused on the design and implementation of the Judge Agent, which is Phase 4 of the benchmark pipeline. The Judge Agent evaluates how well each LLM Reasoning Agent response matches the execution-grounded behavior produced by the Execution Sandbox.

The Judge Agent receives the original MLIR program, the mutated MLIR program, mutation metadata, execution results, ground-truth labels, and each model's predicted output, label, confidence, and explanation. It then scores the model response using both deterministic label comparison and rubric-based semantic evaluation.

A major part of my work was making the Judge Agent robust to the different payload formats produced by upstream agents. I implemented normalization logic for extracting program pairs, execution ground truth, mutation context, and model predictions from the LangGraph state. This allowed the Judge Agent to operate reliably even when fields appeared under different names such as `programA`, `program_a`, `original_mlir`, `predicted_label`, `label`, or nested `evaluation_result` objects.

I also implemented the objective scoring layer. The Judge Agent derives the ground-truth label directly from execution status and output comparison results. If either program fails compilation or execution, the pair is labeled `ERROR`. Otherwise, the `outputs_diverge` field is normalized into either `EQUIV` or `DIVERGE`. The predicted label is then compared against this execution-derived ground truth to compute hard label accuracy.

Beyond simple label matching, I designed a more detailed scoring rubric for evaluating whether the model actually understood the MLIR semantics. The final score combines three dimensions:

- hard label accuracy,

- result accuracy, and
- explanation quality.

Result accuracy evaluates whether the model correctly predicted the output behavior of both programs, including returned values, tensor shapes, ranks, types, and error behavior. Explanation quality evaluates whether the model's reasoning is factually consistent, causally grounded, specific to the MLIR code, and aware of dialect-level semantics.

I implemented mutation-aware dynamic rubrics so that different mutation categories are scored appropriately. For example, syntactic or semantics-preserving mutations receive lower structure weight, while shape-changing, indexing-map, broadcasting, or iteration-space mutations receive higher structure weight. Failure-inducing Level 5 mutations use an error-oriented scoring mode where the model must recognize compilation or runtime failure behavior.

The Judge Agent also includes dialect-specific judging knowledge for both `linalg` and `torch` MLIR. For `linalg`, the prompt emphasizes indexing maps, iterator types, region bodies, yielded values, tensor shapes, and destination-style operation semantics. For `torch`, it emphasizes ATen operator behavior, shape and dtype constraints, broadcasting rules, reshape/view legality, transpose dimensions, and Torch-MLIR type semantics.

Another important part of my work was preventing the judge from rewarding shallow or unsupported explanations. The Judge Agent prompt explicitly instructs the scoring model not to reward verbosity, formal tone, or label-only answers. Instead, it checks whether the explanation cites concrete MLIR operations, attributes, shapes, outputs, diagnostics, or semantic causes that are supported by the code and execution ground truth.

I also implemented structured judge outputs so downstream evaluation could be stored and analyzed consistently. Each judged response includes the evaluated model, predicted label, ground-truth label, label-match result, subscores, weighted scores, final score, verdict, mutation metadata, execution details, and judge evidence. This made it possible to compare model performance across different dialects, mutation levels, and semantic failure categories.

Through this work, I gained experience with MLIR semantic evaluation, LangGraph agent integration, structured benchmark scoring, LLM-as-a-judge workflows, JSON payload normalization, prompt-based rubric design, and execution-grounded model evaluation.

Overall, my work on the Judge Agent helped turn MASCoD-MLIRBench from a dataset generation pipeline into a complete benchmark evaluation framework. The

Judge Agent provides the final scoring layer that determines not only whether an LLM selected the correct label, but whether it actually reasoned correctly about the MLIR programs and their execution behavior.

3.5 Farhan Ahmed

I built the Execution Sandbox, which is Phase 2 of the pipeline. It takes MLIR program pairs from the mutation engine, compiles and runs both through IREE, compares their numerical outputs to determine whether they diverge, and generates a natural-language explanation of what changed. The LLM Evaluation Agent and Judge Agent both rely on the outputs this phase produces.

The sandbox is the terminal node in the LangGraph graph. It receives the program pair, dialect, and mutation metadata from the upstream agent, then compiles, executes, and compares both programs before writing a result payload back to state. The divergence label always comes from actual program outputs, not from any model. GPT-4.1 mini generates a semantic explanation after the label is already set, so it has no influence over the ground truth.

Compilation and execution both run in separate child processes. IREE can kill the host process if it hits a bad input, so isolating each step means a failing pair writes an error record instead of taking down the rest of the run.

For the dataset pipeline, I compare outputs using strict bit-exact equality rather than a numerical tolerance. That only works if the compiler isn't introducing its own floating-point variation between runs, so both programs compile with flags that turn off loop vectorization, SLP vectorization, loop interleaving, and loop unrolling. With those flags set, the same program produces the same output every time, so any difference between program A and program B means the mutation actually changed behavior.

Input handling works differently for each dialect. For linalg, I embed deterministic constants directly into the function body so the program is fully self-contained. For torch, that approach crashes IREE's canonicalizer. There is a type-mismatch bug in the constant-folding pass that fires when a float tensor is compared against an integer scalar, which comes up constantly in the torch seed programs. The workaround was passing inputs externally as runtime flags, which gives the same result since IREE folds the values at compile time anyway. I filed the bug upstream.

Getting the explanations right also took some work. Early on, models would regularly cite SSA values and operations that weren't actually in the source code. I fixed it by computing a static diff of the two programs before the model call, pulling out

concrete facts like renames, constant changes, affine map modifications, and operation substitutions, and putting those into the prompt. The model explains what the facts mean instead of trying to find them, and the hallucinations stopped.

The offline dataset pipeline runs in three scripts. The first handles execution and output comparison. The second adds explanations using the static facts approach. The third benchmarks both programs with `iree-benchmark-module` at 30 repetitions and a 10-second warmup, recording mean, standard deviation, and tail-percentile latencies. All three write output as they go and support resume and retry flags so a crash doesn't set things back too far.

I also wrote three validation scripts: one checks schema completeness across all 1,805 benchmark records, another verifies that the timing and judge payload files line up on program IDs and compilation status, and a third confirms every explanation is present, grounded in the actual source, and consistent with the divergence verdict.

Most of what I used here I hadn't worked with before. That included IREE's compilation and runtime model, the current torch-mlir API and the Docker setup required to run it on macOS, parsing Google Benchmark's output format, and how LangGraph handles state across nodes.

4

Results Achieved

4.1 Mutation Generation Results

The mutation generation pipeline was evaluated using a collection of PyTorch programs that were converted into MLIR representations across both the `torch` and `linalg` dialects.

The initial dataset consisted of 40 unique PyTorch programs. Each program was converted into:

- a `torch` dialect MLIR representation, and
- a `linalg` dialect MLIR representation.

This resulted in a total of 80 base MLIR programs distributed equally across the two supported dialects.

The mutation generation framework then applied approximately 25 mutation rules and semantic transformation requirements for each dialect. These mutation rules were designed to generate controlled semantic modifications across varying levels of complexity, structural divergence, and execution behavior.

The mutation pipeline ultimately generated more than 2,000 mutated MLIR program pairs across both dialects.

Table 4.1: Initial dataset generation statistics.

Dataset Component	Count	Description
PyTorch Programs	40	Original source programs
Torch Dialect MLIR Programs	40	Converted MLIR programs
Linalg Dialect MLIR Programs	40	Converted MLIR programs
Total Base MLIR Programs	80	Combined dialect representations
Mutation Rules per Dialect	25	Semantic mutation rules
Generated Program Pairs	2,000	Total mutated benchmark samples

During validation and inspection, several generated mutations were identified as inconsistent with dialect-specific legality constraints, structural requirements, or execution expectations. These inconsistencies primarily originated from:

- verifier constraint violations,
- invalid tensor shape propagation,
- unsupported lowering behaviors,
- dialect-specific legality mismatches, and
- structural inconsistencies introduced during mutation generation.

As a result, 232 generated benchmark samples were removed from the final benchmark dataset prior to downstream evaluation and execution validation.

Table 4.2: Mutation filtering and validation statistics.

Validation Stage	Count	Percentage
Generated Program Pairs	2000	100.0%
Removed Invalid Samples	232	11.6%
Final Benchmark Samples	1768	88.4%

The remaining benchmark samples were subsequently processed through execution-grounded validation workflows involving IREE, Torch-MLIR, and MLIR execution pipelines to generate semantic equivalence and divergence labels for downstream evaluation tasks.

4.2 Execution Validation Results

Running the execution pipeline across all four seeds and both dialects produced 1,768 program pair records: 925 linalg and 843 torch. Table 4.3 shows how the runs broke down.

Table 4.3: Execution status across all 1,768 program pairs.

Status	Count	%
`success`	1,437	81.3
`expected_invalid`	301	17.0
`runtime_error`	25	1.4
`compile_error`	4	0.2
`benchmark_error`	1	0.1
Total	1,768	100

The 301 `expected_invalid` records are Level 5 mutations, which are designed to fail compilation. All of them did, so the pipeline handles that category correctly. The 25 `runtime_error` records are torch mutations that changed the function's input signature, which broke compatibility with the seed's input specification at runtime. Those are kept in the dataset with a category label rather than dropped.

Three validation scripts confirmed the dataset is clean before delivery. The structural validator ran 13 checks across all 1,768 benchmark records and found no failures. The cross-reference validator confirmed that all 1,437 successful pairs have matching entries in both the timing and judge payload files, with no status mismatches anywhere. The explanation auditor found full coverage across all records, no hallucinated SSA citations, and no cases where an explanation contradicted the divergence verdict.

4.3 Semantic Divergence Analysis Results

For each of the 1,437 successful pairs, the pipeline determined whether the two programs produced the same output under the same inputs. Table 4.4 shows the divergence breakdown by mutation level and dialect.

Level 2 and T2 had the highest divergence rates at 48% and 55%. Those are the semantic-level mutations, so the pattern makes sense. Levels 3 and 4 came in lower even though they represent more complex transformations. It looks like the higher-level mutations tend to add complexity without actually changing what the program computes, at least for programs at this scale.

Level 1 showed 9% divergence in linalg and 2% in torch, which was surprising since

Table 4.4: Behavioral divergence by mutation level. Percentages are within-level. `ExpInv` denotes expected-invalid records (compile failure). Five compile and benchmark errors are included in each row's total but not shown as a separate column.

Dialect	Level	Total	Diverge	Equiv.	ExpInv
Linalg	L1	185	17 (9%)	168 (91%)	0
Linalg	L2	185	87 (48%)	95 (52%)	0
Linalg	L3	185	41 (22%)	143 (78%)	0
Linalg	L4	185	32 (17%)	152 (83%)	0
Linalg	L5	185	0 (0%)	7 (4%)	178
Torch	T1	200	3 (2%)	197 (98%)	0
Torch	T2	160	88 (55%)	72 (45%)	0
Torch	T3	160	10 (6%)	150 (94%)	0
Torch	T4	200	8 (4%)	192 (96%)	0
Torch	T5	123	0 (0%)	0 (0%)	123
Total		**1,768**	**286 (16%)**	**1,181 (67%)**	**301 (17%)**

those are labeled as purely syntactic mutations. Digging into the explanations for those records showed two things happening. Some had operations reordered in a way that changed floating-point summation order, which produces a genuine output difference even if the math is technically the same. A smaller group had constant value changes or attribute modifications that were mislabeled as syntactic by the mutation engine. Neither is a pipeline bug; the first is the strict equality comparison doing exactly what it is supposed to, and the second is an issue with how those mutations got classified upstream.

Seven linalg Level 5 records came back as successful runs with equivalent outputs rather than compile failures. Those are cases where IREE accepted constructs the mutation engine intended to be invalid, likely through implicit type coercion. They are kept in the dataset with their original labels.

On the performance side, the execution time delta between program A and program B across all successful pairs averaged about -0.001 ms with a standard deviation of 0.020 ms. Mutations at Levels 1 through 4 had basically no measurable effect on runtime for programs this size, which suggests that semantic divergence and performance divergence are mostly independent at this scale.

4.3.1 System Performance Results

The seed benchmark pipeline measured execution latency for all 80 seed programs across both dialects. Simple linalg kernels ran around 0.054 ms mean, while torch kernels ranged from 0.1 to 0.4 ms. Each measurement used 30 repetitions with a 10-second warmup.

Across the 1,437 successful pairs, the execution time delta between original and mutated programs averaged −0.001 ms with a standard deviation of 0.020 ms. Mutations at Levels 1 through 4 had no measurable effect on runtime at these kernel scales, meaning semantic divergence and performance divergence are essentially independent in this dataset.

The benchmarking pass took roughly 40–80 seconds per pair due to the warmup period, making the full 1,768-pair run a multi-day process across several sessions on Apple Silicon. The explanation generation pass was much faster, completing all 1,768 records in about 30 minutes at 10 concurrent workers.

4.4 LLM Reasoning Agent Results

The LLM Reasoning Agent evaluated eight language models on MLIR program pairs under a strict information boundary. Each model received only Program A, Program B, the dialect tag, and concrete input values. The models were not given execution outputs, lowered IR, compiler traces, or ground-truth labels.

The evaluated model set included open-weight and closed-source models, including Qwen, Llama, GLM, Kimi, GPT-OSS, GPT-4.1 Mini, Claude Sonnet, and Claude Haiku. All models were queried using the same prompt structure and JSON output schema so that their reasoning behavior could be compared fairly.

Table 4.5: LLM models evaluated by the Reasoning Agent.

Model	**Provider / Family**	**Type**
Qwen3-Next-80B-A3B-Instruct	Alibaba	Open-weight
Llama-4-Maverick	Meta	Open-weight
GLM-5.1	Zhipu AI	Open-weight
Kimi-K2.6	Moonshot AI	Open-weight
GPT-OSS-120B	OpenAI OSS	Open-weight
GPT-4.1-Mini	OpenAI	Closed-source
Claude Sonnet 4.5	Anthropic	Closed-source
Claude Haiku 4.5	Anthropic	Closed-source

Table 4.6: Information boundary enforced during LLM reasoning.

Visible to LLM	Hidden from LLM
Program A MLIR source	Program A executed output
Program B MLIR source	Program B executed output
MLIR dialect tag	Ground-truth label
Concrete input values	Lowered IR / compiler trace
Prompt instructions	Judge score

Table 4.7: Key Reasoning Agent evaluation results.

Metric	Result
Models evaluated	8
Best aggregate model	Claude Sonnet 4.5
Best aggregate score	62.96 / 100
Best model hard-label accuracy	77.4%
Best model result-behavior score	41% of maximum
Average hard-label accuracy across models	72.3%
Average result-behavior accuracy across models	34.5%

The benchmark showed that models were often better at predicting the broad label than at accurately predicting the actual program behavior. In several cases, a model selected the correct `EQUIV`, `DIVERGE`, or `ERROR` label but gave incomplete or incorrect predicted outputs. This indicates that label accuracy alone does not fully measure semantic understanding.

Table 4.7 shows a clear gap between label prediction and actual semantic understanding. Models were relatively better at selecting the correct high-level label, but much weaker at predicting the concrete behavior of the MLIR programs.

The strongest aggregate model was Claude Sonnet 4.5, but even the best model did not achieve robust MLIR semantic reasoning. The final results showed a clear gap between hard-label accuracy and result-behavior accuracy. This suggests that current LLMs can often recognize mutation patterns while still struggling to trace SSA values, tensor shapes, affine indexing maps, iterator semantics, and arithmetic behavior.

The Reasoning Agent also exposed hallucination behavior. Some models produced confident explanations that sounded plausible but were not grounded in the actual MLIR code. This supported the need for the Judge Agent to evaluate explanation quality and result behavior in addition to final label correctness.

4.5 Judge Evaluation Results

After execution-grounded labels were generated, the benchmark was used to evaluate eight LLMs on MLIR semantic reasoning. The evaluated models were Claude-Sonnet-4.5, GPT-OSS-120B, Kimi-K2.6, Claude-Haiku-4.5, GLM-5.1, GPT-4.1-mini, Qwen3-Coder-Next, and Llama-4. Each model was asked to predict the behavior of the original and mutated MLIR programs without access to execution outputs or ground-truth labels. The Judge Agent then scored each response using three normalized dimensions: hard-label accuracy, result accuracy, and explanation quality.

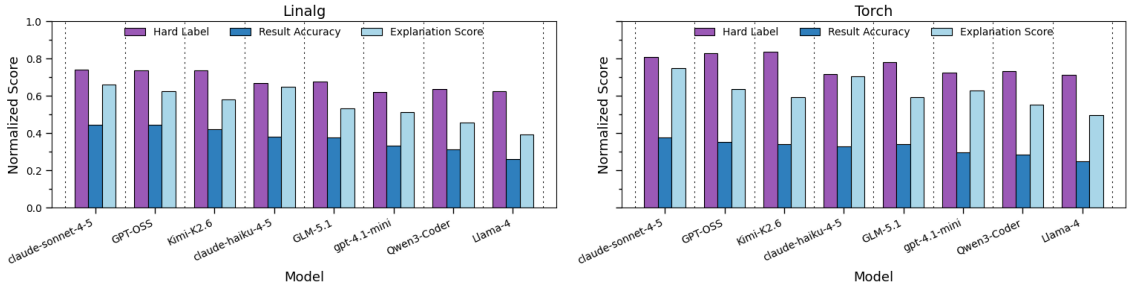


Figure 4.1: Eight-model benchmark subscore breakdown across the `linalg` and `torch` dialects.

Figure 4.1 shows that hard-label accuracy was consistently higher than result accuracy for every model and both dialects. In other words, models were often able to classify a pair as `EQUIV`, `DIVERGE`, or `ERROR`, but were much less reliable at predicting the concrete output behavior that justified the label. This is an important result because MASCoD-MLIRBench is intended to measure semantic execution reasoning, not only classification.

For the `linalg` dialect, Claude-Sonnet-4.5, GPT-OSS-120B, and Kimi-K2.6 had the strongest overall profiles. Their hard-label scores were around the low-to-mid 0.70 range, while their result accuracy scores were only around 0.42–0.45. This gap shows that even strong models struggled to trace lower-level MLIR semantics involving affine indexing maps, iterator types, destination tensors, and `linalg.yield` behavior. Explanation scores were generally between hard-label and result accuracy, meaning models often produced partially correct explanations but did not fully compute the exact program behavior.

For the `torch` dialect, hard-label scores were slightly higher overall. GPT-OSS-120B and Kimi-K2.6 reached the highest hard-label performance, both above 0.80, while Claude-Sonnet-4.5 also performed strongly. However, result accuracy again remained much lower, mostly between 0.25 and 0.38. This suggests that Torch MLIR is more recognizable to LLMs because it exposes ATen-style operators, but models still strug-

gle with exact tensor values, broadcasting behavior, reshaping, dtype effects, and operation-level semantic changes.

Model	Overall	L1	L2	L3	L4	L5	T1	T2	T3	T4	T5
Claude-Sonnet-4.5	62.96	62.84 *n=185*	58.77 *n=149*	68.08 *n=185*	62.77 *n=185*	54.42 *n=185*	64.79 *n=200*	62.66 *n=160*	62.68 *n=160*	67.48 *n=200*	63.80 *n=160*
GPT-OSS-120B	60.39	63.65 *n=185*	62.88 *n=149*	65.40 *n=185*	60.90 *n=185*	48.57 *n=185*	58.15 *n=200*	59.28 *n=160*	64.64 *n=160*	64.05 *n=200*	56.61 *n=160*
Kimi-K2.6	58.33	58.43 *n=185*	56.54 *n=149*	64.83 *n=185*	62.17 *n=185*	47.03 *n=185*	56.53 *n=200*	56.50 *n=160*	62.41 *n=160*	62.04 *n=200*	56.30 *n=160*
Claude-Haiku-4.5	57.42	58.39 *n=185*	55.71 *n=149*	59.61 *n=185*	58.13 *n=185*	50.88 *n=185*	59.89 *n=200*	57.69 *n=160*	52.97 *n=160*	62.94 *n=200*	56.25 *n=160*
GLM-5.1	54.94	58.60 *n=185*	56.70 *n=149*	62.85 *n=185*	57.76 *n=185*	29.09 *n=185*	60.07 *n=200*	60.09 *n=160*	60.14 *n=160*	65.61 *n=200*	36.46 *n=160*
GPT-4.1-mini	51.78	59.10 *n=185*	53.17 *n=149*	55.33 *n=185*	56.44 *n=185*	20.42 *n=185*	63.36 *n=200*	58.63 *n=160*	51.59 *n=160*	63.19 *n=200*	33.33 *n=160*
Qwen3-Coder-Next	49.51	55.25 *n=185*	51.73 *n=149*	54.13 *n=185*	54.01 *n=185*	19.95 *n=185*	57.98 *n=200*	52.68 *n=160*	62.33 *n=160*	61.88 *n=200*	22.40 *n=160*
Llama-4	45.48	50.36 *n=185*	47.01 *n=149*	51.20 *n=185*	50.03 *n=185*	14.52 *n=185*	54.48 *n=200*	51.71 *n=160*	53.87 *n=160*	57.22 *n=200*	21.77 *n=160*

Figure 4.2: Eight-model aggregate score by mutation level across `linalg` and `torch`. Each cell reports the Judge Agent aggregate score for that model and mutation family.

Figure 4.2 gives a more detailed view of model performance by mutation level. Claude-Sonnet-4.5 achieved the best overall score at 62.96, followed by GPT-OSS-120B at 60.39 and Kimi-K2.6 at 58.33. Claude-Haiku-4.5 followed with 57.42, while GLM-5.1, GPT-4.1-mini, Qwen3-Coder-Next, and Llama-4 scored 54.94, 51.78, 49.51, and 45.48 respectively.

The heatmap shows that Level 3 and Level 4 mutations were often easier than expected for the strongest models. Claude-Sonnet-4.5 reached 68.08 on L3 and 67.48 on T4, while GPT-OSS-120B reached 65.40 on L3 and 64.64 on T3. These levels often involve structural transformations that may preserve high-level behavior, so models could sometimes identify equivalence even when the MLIR structure changed.

Level 2 and T2 mutations were more challenging because they directly modify semantic parameters such as constants, tensor dimensions, indexing maps, iteration spaces, and operator attributes. Claude-Sonnet-4.5 scored 58.77 on L2 and 62.66 on T2, while GPT-OSS-120B scored 62.88 on L2 and 59.28 on T2. These results show that semantic-level mutations remain difficult even for the best-performing models, especially when local changes must be propagated through SSA dataflow.

The clearest weakness appeared in Level 5 and T5 failure-inducing mutations. These mutations are designed to introduce invalid MLIR behavior such as type mismatches, broken dependencies, invalid operator constraints, undefined SSA values, or structural violations. Claude-Sonnet-4.5 remained the most stable model, scoring 54.42 on L5 and 63.80 on T5. GPT-OSS-120B and Kimi-K2.6 also handled T5 reasonably well, scoring 56.61 and 56.30 respectively. However, several other models dropped sharply.

GLM-5.1 scored only 29.09 on L5 and 36.46 on T5, GPT-4.1-mini scored 20.42 on L5 and 33.33 on T5, Qwen3-Coder-Next scored 19.95 on L5 and 22.40 on T5, and Llama-4 scored 14.52 on L5 and 21.77 on T5.

These results show that compiler-error reasoning is one of the hardest parts of the benchmark. Many models can recognize normal semantic equivalence or divergence, but they struggle to identify verifier-level failures and explain why a mutated MLIR program is invalid. This supports the central finding of MASCoD-MLIRBench: current LLMs retain coarse semantic understanding over MLIR, but still lack reliable compiler-internal reasoning over exact execution behavior, tensor-level semantics, and dialect-specific legality constraints.

4.6 Research Contributions

- Introduced **MASCoD-MLIRBench**, one of the first execution-grounded benchmarking frameworks designed specifically for evaluating semantic reasoning capabilities of LLMs over MLIR compiler intermediate representations.
- Developed a deterministic **multi-agent orchestration pipeline** consisting of Mutation, Execution Sandbox, LLM Reasoning, and Judge Agents for reproducible compiler-aware evaluation workflows.
- Designed a structured **five-level semantic mutation framework** capable of generating controlled semantic divergence scenarios ranging from syntactic transformations to compiler-breaking structural mutations.
- Constructed a large-scale **MLIR-centric benchmark dataset** containing 1768 program pairs across multiple dialects including `torch` and `linalg`, enabling compiler-level semantic evaluation.
- Proposed **execution-grounded semantic validation** using IREE-based runtime compilation and bit-exact behavioral comparison to establish objective ground-truth labels.
- Introduced a custom evaluation rubric inspired by existing reasoning assessment frameworks such as ICE to evaluate causal reasoning quality, semantic grounding, and explanation specificity beyond simple label prediction accuracy.
- Performed comprehensive evaluation of frontier LLMs, demonstrating significant gaps between coarse semantic prediction and precise compiler-level reasoning capabilities.
- Demonstrated downstream applications including **compiler-aware auto-correction** and **semantic reasoning fine-tuning** for improving IR-level understanding in language models.

4.7 Challenges Faced

4.7.1 Mutation Agent

- One of the primary challenges in developing the Mutation Agent was generating semantically meaningful MLIR mutations while preserving compiler legality and structural correctness. Unlike high-level programming languages, MLIR operates at a significantly lower compiler IR abstraction level where even minor modifications can easily violate SSA dependencies, tensor compatibility constraints, affine indexing relationships, iterator semantics, or operation validity requirements.
- Another major challenge involved designing controlled semantic divergence mutations across multiple mutation difficulty levels without producing random, unrealistic, or trivial transformations. The mutation generation process required carefully constrained dialect-aware transformations capable of simulating realistic compiler optimization and semantic reasoning scenarios while still remaining valid under compiler verification pipelines.
- The project also encountered significant difficulties while handling multiple MLIR dialects such as `torch` and `linalg`. Certain operations and transformations valid within one dialect frequently failed lowering pipelines, legality verification, or backend conversion stages in another dialect. Addressing this issue required integrating compiler verification workflows using `mlir-opt` and `torch-mlir-opt`, along with dialect-specific legality checks and validation passes.
- Additionally, early experiments using open-source language models resulted in inconsistent mutation quality, hallucinated operations, invalid tensor transformations, malformed affine maps, and structurally broken IR programs. Several generated mutations appeared syntactically plausible but failed compiler verification or execution stages entirely. To address this issue, the framework incorporated constrained mutation workflows, structured metadata extraction, compiler-guided validation pipelines, and progressively stricter execution-grounded filtering mechanisms to ensure dataset reliability, reproducibility, and semantic correctness.
- Additional infrastructure and deployment challenges included:
  - Limited storage availability and restricted software installation permissions on publicly accessible COM S department servers, making it difficult to host large LLVM, MLIR, and `torch-mlir` environments required for mutation validation and execution workflows.
  - Migration of the execution and validation pipeline to the Iowa State Nova

HPC cluster to support compiler builds, large-scale dataset generation, parallel execution workloads, and memory-intensive validation tasks.

- Configuration and maintenance of SSH reverse tunneling workflows to securely expose benchmarking APIs and execution services running on Nova through externally accessible COM S infrastructure.
- Managing compiler dependency compatibility between LLVM, MLIR, IREE, and `torch-mlir` versions, where small version mismatches frequently caused lowering failures, backend incompatibilities, or runtime execution errors.
- Handling large-scale execution validation workloads while ensuring deterministic runtime behavior and reproducible semantic comparison results across generated mutation samples.
- Designing efficient metadata extraction pipelines capable of capturing SSA relationships, tensor shapes, affine indexing maps, operation dependencies, and dialect-specific attributes necessary for controlled mutation generation.

4.7.2 Execution Sandbox Agent

The Execution Sandbox Agent had to produce deterministic ground truth across a large set of MLIR programs without any human review of individual results.

- **Dialect-specific compilation flags:** Linalg programs compiled reliably with a single `auto` flag, but Torch dialect programs frequently required trying multiple input type flags (`auto`, `torch`, `tm_tensor`) in sequence. A wrong flag produced a silent failure rather than a descriptive error, making it hard to distinguish a bad flag choice from a genuinely broken program.
- **Dynamic output shapes:** Programs using `torch.nonzero()` or boolean indexing produce outputs whose shapes depend on runtime values. `iree-benchmark-module` pre-allocates output buffers before execution and crashes with exit code 245 when it cannot. The failure appeared intermittent at first because it was parity-dependent on repetition count. Fixing it required rewriting the affected seeds to use static tensor operations and regenerating MLIR through Docker since torch-mlir has no macOS wheels.
- **Process isolation:** IREE can crash the host process on malformed or unsupported input rather than returning a clean error. Running compilation and execution inline would have killed the pipeline mid-run. Both steps were moved into child subprocesses with hard timeouts to keep the pipeline stable across 80 seeds and 925 mutations.
- **Benchmark tool behavior:** `iree-benchmark-module` writes all output to stderr instead of stdout, and per-repetition lines needed for p50/p90/p99 are

suppressed by default. Two flags had to be discovered through trial and error: `--function=main` (omitting it causes the tool to silently exit with no results) and `--benchmark_report_aggregates_only=false` (required to expose individual repetition lines for percentile computation).

- **Preserving evaluation integrity:** Mutation JSON files from Phase 1 contain two sections: `context_results`, which includes mutation type labels, and `blind_results`, which contains only MLIR source. The benchmarking script had to read exclusively from `blind_results` to prevent mutation category information from reaching the LLM Evaluation Agent.

Overall, most of the work in this phase turned out to be tooling rather than logic. The actual compile-compare pipeline is not complicated, but getting it to run cleanly across 80 seeds and nearly 1800 mutations meant dealing with silent tool failures, dynamic shape crashes, and methodology gaps that only surfaced once the pipeline ran at scale. More time went into understanding how `iree-benchmark-module` behaved than into writing the agent itself.

4.7.3 LLM Reasoning Agent

The LLM Reasoning Agent faced several challenges because it evaluated models under a strict execution-hidden setting. The agent had to ask models to reason about MLIR program pairs without giving them executed outputs, lowered IR, compiler traces, or ground-truth labels. This made the task difficult because the model had to infer behavior directly from MLIR source, SSA dataflow, tensor shapes, indexing maps, iterator types, and operation semantics.

A major challenge was inconsistent model output formatting. Different models returned responses in different forms, including valid JSON, markdown-fenced JSON, prose before or after JSON, missing fields, invalid labels, empty responses, or malformed JSON. Since the Judge Agent required structured predictions, the LLM Reasoning Agent needed robust parsing and validation before outputs could be used downstream.

Another challenge was hallucinated reasoning. Some models produced confident explanations that sounded reasonable but were not grounded in the actual MLIR code. In particular, models sometimes referred to SSA values, operations, tensor shapes, or semantic effects that were not present in the program. To reduce this issue, the prompt required models to cite specific MLIR evidence and return structured fields such as `key_differences`, `witness`, and `explanation`.

The LLM Reasoning Agent also had to handle model uncertainty. Some LLMs attempted to avoid making a final decision or returned vague answers instead of choosing

`EQUIV`, `DIVERGE`, or `ERROR`. To address this, the agent used a forced-verdict retry strategy. If the first response failed validation, the model was queried again with stronger instructions requiring one of the three valid labels.

Table 4.8: LLM Reasoning Agent challenges and mitigations.

Challenge	**Mitigation**
Malformed JSON responses	Implemented JSON extraction, markdown fence removal, schema validation, and error recording.
Invalid labels or abstentions	Restricted labels to `EQUIV`, `DIVERGE`, and `ERROR`; added forced-verdict retry logic.
Hallucinated explanations	Required explanations to reference concrete MLIR evidence such as SSA values, operations, shapes, indexing maps, and iterator types.
Provider-specific response formats	Added response extraction logic for normal text content, list-based content blocks, and reasoning-model output fields.
Model/API failures	Recorded per-model status and error messages while allowing the remaining models to continue running.
Confidence inconsistencies	Validated confidence values and clamped out-of-range scores to preserve downstream consistency.

Overall, these challenges showed that the LLM Reasoning Agent was not only a prompting component but also a reliability layer for multi-model evaluation. It had to preserve fair reasoning conditions while handling unpredictable model behavior across eight different LLMs.

4.7.4 Judge Agent

- One challenge was handling invalid or incomplete JSON returned by the ChatGPT-based judge model. Even with JSON-mode prompting, some responses occasionally had missing fields or formatting problems, so the Judge Agent needed fallback handling, score clamping, and validation logic to keep the evaluation pipeline from failing.
- Another challenge was processing the full judge workload under the paper dead-

line. The final benchmark required scoring more than 16,000 model-response records in a short amount of time, so we extended the dataset-processing script with multithreading to run judge requests concurrently and finish the evaluation on time.

- A third challenge was preventing the Judge Agent from rewarding polished but unsupported explanations. Some model responses sounded convincing while failing to cite the actual MLIR operations or execution behavior. To address this, I designed the judge prompt to prioritize technical correctness, concrete MLIR evidence, result accuracy, and causal grounding instead of writing style. The final leaderboard remained consistent with and without explanation-based scoring, which helped validate that the rubric was not simply rewarding fancy explanations.

4.8 Future Improvements

Future work on MASCoD-MLIRBench can expand the framework in several directions. First, the benchmark can be extended beyond the current `torch` and `linalg` dialects to include additional MLIR dialects such as `affine`, `scf`, `memref`, `gpu`, and hardware-oriented dialects. This would allow the system to evaluate a broader range of compiler IR reasoning tasks. Second, the mutation engine can be improved with stronger verifier-aware constraints so fewer generated samples need to be filtered after generation. Third, the execution pipeline can support more input sets per program pair, which would reduce the chance that a mutation appears equivalent only because of one fixed deterministic input. Fourth, the Judge Agent can be further calibrated using human-reviewed examples to improve explanation scoring consistency and reduce LLM-judge variability. Finally, future versions can include richer dashboards, leaderboard support, larger benchmark releases, and additional downstream tasks such as automatic MLIR repair, compiler optimization reasoning, and dialect-specific model fine-tuning.

5

Conclusion

5.1 Project Summary

MASCoD-MLIRBench introduced a scalable multi-agent benchmarking framework for evaluating semantic reasoning capabilities of Large Language Models over MLIR-based compiler intermediate representations. The project combined mutation generation, execution-grounded validation, semantic reasoning evaluation, and structured benchmark analysis into a unified benchmarking ecosystem capable of producing reproducible MLIR reasoning datasets.

The framework integrated four major components:

- the Mutation Agent,
- the Execution Sandbox Agent,
- the LLM Reasoning Agent
- the Judge Agent.

Together, these components enabled the generation of controlled semantic mutations together with deterministic execution-derived ground-truth labels for evaluating whether LLMs genuinely understand compiler IR semantics rather than relying on syntactic similarity or memorized source-level code patterns.

The project additionally integrated MLIR, Torch-MLIR, LLVM, and IREE-based execution workflows together with distributed API infrastructure deployed across public servers and HPC systems. Using this infrastructure, the framework generated samples across the `torch` and `linalg` dialects while supporting downstream experimentation involving semantic reasoning evaluation, automatic correction of invalid mutations, and fine-tuning workflows.

Beyond benchmark construction, the project demonstrated how execution-grounded compiler reasoning tasks can support reproducible AI evaluation workflows for compiler systems, program analysis, and semantic reasoning research.

5.2 Research Impact

MASCoD-MLIRBench contributes a new research direction at the intersection of compiler infrastructures and large language model evaluation.

Existing LLM benchmarks primarily focus on high-level programming languages such as Python, Java, or C++, where evaluation tasks are often centered around code generation, repository-level reasoning, debugging, or software engineering workflows. While these benchmarks measure source-level reasoning ability, they do not necessarily evaluate whether models understand lower-level compiler semantics, execution behavior, tensor transformations, or intermediate representation legality constraints.

MASCoD-MLIRBench addresses this gap by introducing one of the first execution-grounded semantic reasoning benchmarks specifically designed for MLIR-based compiler intermediate representations.

The novelty of the project lies in several key areas:

- the use of a modular multi-agent framework for benchmark generation and evaluation,
- execution-grounded semantic labeling using deterministic compiler execution,
- dialect-aware mutation generation for MLIR compiler infrastructures,
- semantic divergence evaluation beyond surface-level syntactic similarity,
- benchmarking reasoning capabilities over compiler IR semantics rather than source-code syntax, and
- enabling downstream tasks such as automatic correction and semantic reasoning fine-tuning.

Unlike traditional static benchmark datasets, MASCoD-MLIRBench introduces a dynamic benchmark generation workflow where semantic mutations, execution validation, reasoning evaluation, and judge-based scoring operate together as coordinated agents inside a unified pipeline.

The project also demonstrates that evaluating LLMs only on high-level languages may provide an incomplete understanding of model reasoning capabilities. Compiler intermediate representations expose significantly deeper semantic structures involving SSA dependencies, affine indexing maps, iterator semantics, tensor propagation, verifier constraints, and lowering behavior. Evaluating reasoning in such environments provides a more rigorous understanding of whether models truly reason about program behavior or primarily rely on statistical code familiarity.

The benchmark therefore opens opportunities for future research involving:

- compiler-aware language models,

- execution-grounded reasoning systems,
- semantic equivalence prediction,
- MLIR-specialized fine-tuning,
- automated compiler IR repair,
- verifier-aware reasoning agents, and
- execution-guided program analysis workflows.

The project additionally contributes reusable infrastructure for future compiler-AI research by providing scalable dataset generation workflows, execution pipelines, distributed validation infrastructure, and modular agent-based orchestration systems.

5.3 Future Scope of MASCoD-MLIRBench

MASCoD-MLIRBench establishes a foundation for future research involving semantic reasoning evaluation over compiler intermediate representations and execution-aware AI systems.

One major future direction involves extending the benchmark beyond the current `torch` and `linalg` dialects to support additional MLIR ecosystems including:

- affine,
- vector,
- tensor,
- sparse tensor,
- GPU,
- SCF, and
- LLVM dialects.

Supporting additional dialects would enable broader evaluation of reasoning capabilities across progressively lower compiler abstraction levels and hardware-aware optimization pipelines.

Future work may additionally involve:

- integrating more advanced semantic mutation strategies,
- incorporating graph-based IR analysis,
- supporting multi-stage lowering evaluation,
- extending execution-grounded benchmarking to distributed compilation workflows,
- evaluating reasoning consistency across compiler passes,

- introducing verifier-aware autonomous repair agents, and
- building reinforcement-learning-based compiler reasoning systems.

Another important future direction involves large-scale fine-tuning of language models directly on execution-grounded MLIR reasoning datasets. Such fine-tuning workflows may help improve:

- compiler-aware reasoning,
- semantic equivalence detection,
- automatic IR correction,
- optimization understanding, and
- execution prediction capabilities.

The framework may also evolve into a fully autonomous compiler reasoning ecosystem where multiple agents collaboratively generate mutations, validate execution behavior, analyze semantic divergence, repair invalid IR, and optimize compiler transformations with minimal human intervention.

In the long term, MASCoD-MLIRBench aims to contribute toward the development of execution-aware AI systems capable of understanding compiler semantics at a deeper level than existing source-code-oriented evaluation benchmarks currently provide.

A

Demo Slides

This appendix contains the presentation slides used throughout the Senior Design project for MASCoD-MLIRBench.

The included presentations are:

- Demo 1
- Demo 2
- Demo 3
- Lightning Talk 1
- Lightning Talk 2
- Lightning Talk 3

The complete presentation slide decks are available at:

`https://drive.google.com/drive/u/1/folders/1A5WC-quJz2U-cYlofPQ21cglk2TWP-8n`

B

Additional Figures and Diagrams

This appendix contains additional experimental figures, evaluation plots, and benchmark visualizations related to the downstream tasks explored using the MASCoD-MLIRBench dataset.

The included figures primarily focus on:

- Semantic Reasoning Fine-Tuning results using GPT-4.1-mini, and
- Auto-Correction downstream task evaluation results.

These figures illustrate training behavior, validation trends, accuracy improvements, semantic divergence analysis, and downstream reasoning performance observed during experimentation with execution-grounded MLIR mutation datasets.

Included Figures

- Semantic Reasoning Fine-Tuning Accuracy Curves
- Semantic Reasoning Training Loss Curves
- Auto-Correction Accuracy Results
- Auto-Correction Training Loss Results
- Downstream Task Evaluation Metrics

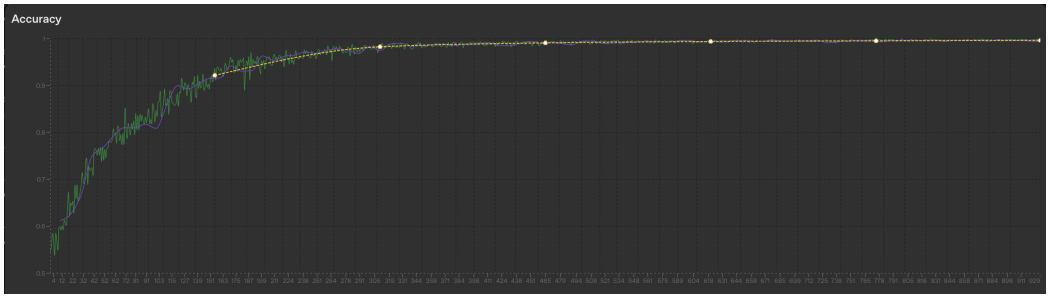


Figure B.1: Fine-tuning accuracy trends observed during the semantic reasoning downstream task using GPT-4.1-mini on MASCoD-MLIRBench datasets.

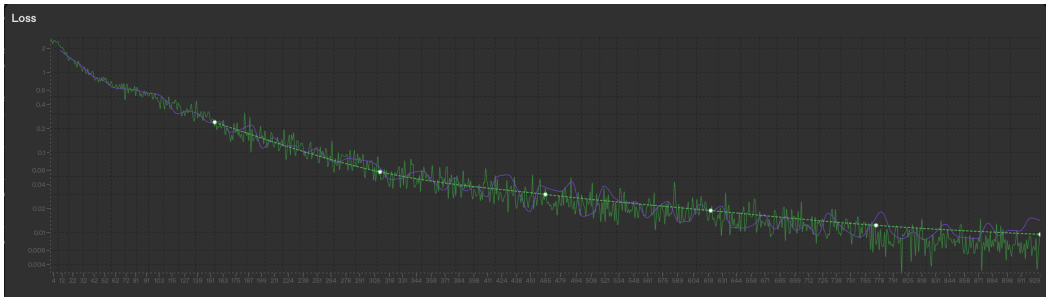


Figure B.2: Training loss curves observed during semantic reasoning fine-tuning using execution-grounded MLIR benchmark samples.

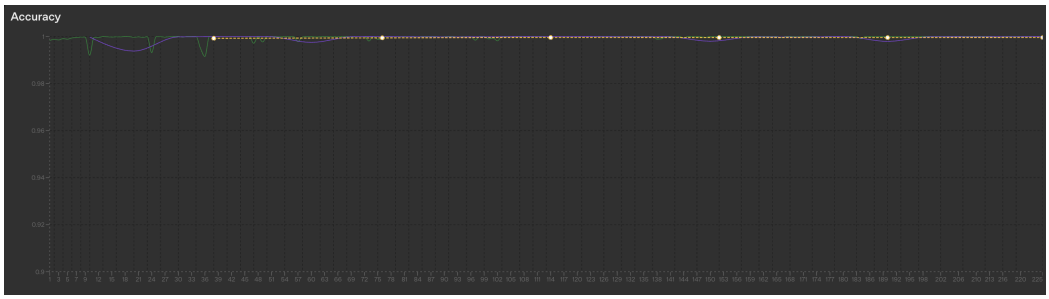


Figure B.3: Accuracy trends for the auto-correction downstream task using GPT-4.1-mini on failure-inducing MLIR mutations.

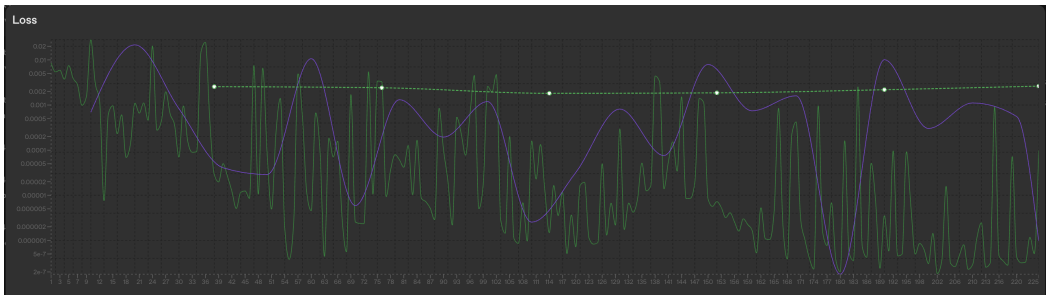


Figure B.4: Training loss curves observed during supervised fine-tuning for automatic correction of failure-inducing MLIR mutations.

C

API Documentation

The MASCoD-MLIRBench framework was deployed using a distributed API and execution infrastructure consisting of two primary systems: the public COM S server infrastructure and the Nova HPC cluster.

C.1 Public API Server

The primary public-facing API server was hosted on:

`old-coms-4020-040.class.las.iastate.edu` or

This server acted as the centralized access point for the MASCoD-MLIRBench framework and exposed APIs for:

- mutation generation requests,
- GPT-based semantic mutation workflows,
- benchmark orchestration,
- request routing,
- execution coordination, and
- communication with the Nova HPC infrastructure.

The COM S server additionally handled incoming API requests from frontend interfaces and external benchmark workflows before forwarding validation-related tasks to the Nova execution environment.

C.2 Nova HPC Validation Infrastructure

Execution validation workflows were deployed on the Nova HPC cluster at Iowa State University. The Nova environment provided access to large-scale compute resources

together with compiler infrastructures required for MLIR execution and validation.

The Nova execution environment integrated:

- LLVM,
- MLIR,
- Torch-MLIR,
- `mlir-opt`,
- `torch-mlir-opt`, and
- IREE execution workflows.

These tools were used to validate generated mutations, which were later used for executing MLIR programs, and generate execution-grounded benchmark labels such as `EQUIV`, `DIVERGE`, and `ERROR`.

C.3 SSH Reverse Tunneling Infrastructure

Since the Nova cluster was not directly accessible from external public endpoints, SSH reverse tunneling was used to securely expose validation services through the COM S public server.

The reverse tunneling workflow enabled APIs running on Nova to be forwarded through the public COM S server infrastructure, allowing external benchmark requests to interact with HPC-hosted validation services without directly exposing internal Nova compute nodes.

This architecture provided several advantages:

- centralized public API access,
- secure communication between systems,
- isolation of HPC execution environments,
- scalable validation infrastructure, and
- simplified integration between frontend systems and backend execution pipelines.

The distributed deployment architecture additionally separated lightweight orchestration workflows from compute-intensive compiler validation tasks, allowing the framework to efficiently utilize HPC resources while maintaining a unified API interface for benchmark generation and execution workflows.

The overall API and infrastructure design follows the modular subsystem communication principles discussed in the COM S 309 interface design guidelines, where independent subsystems communicate through clearly defined APIs, execution pipelines, and distributed communication protocols. :contentReference[oaicite:0]index=0

D

Sample MLIR Programs

D.1 Seed 1

D.1.1 LinAlg

```
#map = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> ()>
#map2 = affine_map<(d0, d1) -> (d0)>
#map3 = affine_map<(d0, d1) -> (d1)>
#map4 = affine_map<() -> ()>
module {
  func.func @main(%arg0: tensor<3x4xf32>) -> (tensor<i1>,
     tensor<f32>, tensor<3xi1>, tensor<4xi1>, tensor<f32>,
     tensor<f32>) {
    %cst = arith.constant dense<1.200000e+01> : tensor<f32>
    %cst_0 = arith.constant 0.000000e+00 : f32
    %true = arith.constant true
    %cst_1 = arith.constant 0xFF800000 : f32
    %extracted_slice = tensor.extract_slice %arg0[0, 0] [1,
        4] [1, 1] : tensor<3x4xf32> to tensor<1x4xf32>
    %collapsed = tensor.collapse_shape %extracted_slice
       [[0, 1]] : tensor<1x4xf32> into tensor<4xf32>
    %extracted_slice_2 = tensor.extract_slice %collapsed[0]
        [1] [1] : tensor<4xf32> to tensor<1xf32>
    %collapsed_3 = tensor.collapse_shape %extracted_slice_2
        [] : tensor<1xf32> into tensor<f32>
    %0 = tensor.empty() : tensor<3x4xi1>
    %1 = linalg.generic {indexing_maps = [#map, #map1, #map
       ], iterator_types = ["parallel", "parallel"]} ins(%
       arg0, %collapsed_3 : tensor<3x4xf32>, tensor<f32>)
```

```
    outs(%0 : tensor<3x4xi1>) {
^bb0(%in: f32, %in_4: f32, %out: i1):
  %21 = arith.cmpf oeq, %in, %in_4 : f32
  linalg.yield %21 : i1
} -> tensor<3x4xi1>
%2 = tensor.empty() : tensor<i1>
%3 = linalg.fill ins(%true : i1) outs(%2 : tensor<i1>)
    -> tensor<i1>
%4 = linalg.generic {indexing_maps = [#map, #map1],
    iterator_types = ["reduction", "reduction"]} ins(%1
    : tensor<3x4xi1>) outs(%3 : tensor<i1>) {
^bb0(%in: i1, %out: i1):
  %21 = arith.andi %in, %out : i1
  linalg.yield %21 : i1
} -> tensor<i1>
%5 = tensor.empty() : tensor<3x4xf32>
%6 = linalg.generic {indexing_maps = [#map, #map1, #map
    ], iterator_types = ["parallel", "parallel"]} ins(%
    arg0, %collapsed_3 : tensor<3x4xf32>, tensor<f32>)
    outs(%5 : tensor<3x4xf32>) {
^bb0(%in: f32, %in_4: f32, %out: f32):
  %21 = arith.subf %in, %in_4 : f32
  linalg.yield %21 : f32
} -> tensor<3x4xf32>
%7 = linalg.generic {indexing_maps = [#map, #map],
    iterator_types = ["parallel", "parallel"]} ins(%6 :
    tensor<3x4xf32>) outs(%5 : tensor<3x4xf32>) {
^bb0(%in: f32, %out: f32):
  %21 = math.absf %in : f32
  linalg.yield %21 : f32
} -> tensor<3x4xf32>
%8 = tensor.empty() : tensor<f32>
%9 = linalg.fill ins(%cst_1 : f32) outs(%8 : tensor<f32
    >) -> tensor<f32>
%10 = linalg.generic {indexing_maps = [#map, #map1],
    iterator_types = ["reduction", "reduction"]} ins(%7
    : tensor<3x4xf32>) outs(%9 : tensor<f32>) {
^bb0(%in: f32, %out: f32):
  %21 = arith.maximumf %in, %out : f32
  linalg.yield %21 : f32
} -> tensor<f32>
```

```
%11 = tensor.empty() : tensor<3xi1>
%12 = linalg.fill ins(%true : i1) outs(%11 : tensor<3
   xi1>) -> tensor<3xi1>
%13 = linalg.generic {indexing_maps = [#map, #map2],
   iterator_types = ["parallel", "reduction"]} ins(%1 :
    tensor<3x4xi1>) outs(%12 : tensor<3xi1>) {
^bb0(%in: i1, %out: i1):
  %21 = arith.andi %in, %out : i1
  linalg.yield %21 : i1
} -> tensor<3xi1>
%14 = tensor.empty() : tensor<4xi1>
%15 = linalg.fill ins(%true : i1) outs(%14 : tensor<4
   xi1>) -> tensor<4xi1>
%16 = linalg.generic {indexing_maps = [#map, #map3],
   iterator_types = ["reduction", "parallel"]} ins(%1 :
    tensor<3x4xi1>) outs(%15 : tensor<4xi1>) {
^bb0(%in: i1, %out: i1):
  %21 = arith.andi %in, %out : i1
  linalg.yield %21 : i1
} -> tensor<4xi1>
%17 = linalg.generic {indexing_maps = [#map, #map],
   iterator_types = ["parallel", "parallel"]} ins(%1 :
   tensor<3x4xi1>) outs(%5 : tensor<3x4xf32>) {
^bb0(%in: i1, %out: f32):
  %21 = arith.uitofp %in : i1 to f32
  linalg.yield %21 : f32
} -> tensor<3x4xf32>
%18 = linalg.fill ins(%cst_0 : f32) outs(%8 : tensor<
   f32>) -> tensor<f32>
%19 = linalg.generic {indexing_maps = [#map, #map1],
   iterator_types = ["reduction", "reduction"]} ins(%17
    : tensor<3x4xf32>) outs(%18 : tensor<f32>) {
^bb0(%in: f32, %out: f32):
  %21 = arith.addf %in, %out : f32
  linalg.yield %21 : f32
} -> tensor<f32>
%20 = linalg.generic {indexing_maps = [#map4, #map4, #
   map4], iterator_types = []} ins(%19, %cst : tensor<
   f32>, tensor<f32>) outs(%8 : tensor<f32>) {
^bb0(%in: f32, %in_4: f32, %out: f32):
  %21 = arith.divf %in, %in_4 : f32
```

```
      linalg.yield %21 : f32
    } -> tensor<f32>
    return %4, %10, %13, %16, %19, %20 : tensor<i1>, tensor
       <f32>, tensor<3xi1>, tensor<4xi1>, tensor<f32>,
       tensor<f32>
  }
}
```

D.1.2 Torch

```
module {
  func.func @main(%arg0: !torch.vtensor<[3,4],f32>) -> (!
     torch.vtensor<[],i1>, !torch.vtensor<[],f32>, !torch.
     vtensor<[3],i1>, !torch.vtensor<[4],i1>, !torch.
     vtensor<[],f32>, !torch.vtensor<[],f32>) {
    %0 = torch.vtensor.literal(dense<1.200000e+01> : tensor
       <f32>) : !torch.vtensor<[],f32>
    %int6 = torch.constant.int 6
    %none = torch.constant.none
    %false = torch.constant.bool false
    %int1 = torch.constant.int 1
    %int0 = torch.constant.int 0
    %1 = torch.aten.slice.Tensor %arg0, %int0, %int0, %int1
       , %int1 : !torch.vtensor<[3,4],f32>, !torch.int, !
       torch.int, !torch.int, !torch.int -> !torch.vtensor
       <[1,4],f32>
    %2 = torch.aten.squeeze.dim %1, %int0 : !torch.vtensor
       <[1,4],f32>, !torch.int -> !torch.vtensor<[4],f32>
    %3 = torch.aten.slice.Tensor %2, %int0, %int0, %int1, %
       int1 : !torch.vtensor<[4],f32>, !torch.int, !torch.
       int, !torch.int, !torch.int -> !torch.vtensor<[1],
       f32>
    %4 = torch.aten.squeeze.dim %3, %int0 : !torch.vtensor
       <[1],f32>, !torch.int -> !torch.vtensor<[],f32>
    %5 = torch.aten.eq.Tensor %arg0, %4 : !torch.vtensor
       <[3,4],f32>, !torch.vtensor<[],f32> -> !torch.
       vtensor<[3,4],i1>
    %6 = torch.aten.all %5 : !torch.vtensor<[3,4],i1> -> !
       torch.vtensor<[],i1>
```

```
    %7 = torch.aten.sub.Tensor %arg0, %4, %int1 : !torch.
       vtensor<[3,4],f32>, !torch.vtensor<[],f32>, !torch.
       int -> !torch.vtensor<[3,4],f32>
    %8 = torch.aten.abs %7 : !torch.vtensor<[3,4],f32> -> !
       torch.vtensor<[3,4],f32>
    %9 = torch.aten.max %8 : !torch.vtensor<[3,4],f32> -> !
       torch.vtensor<[],f32>
    %10 = torch.aten.eq.Tensor %arg0, %4 : !torch.vtensor
       <[3,4],f32>, !torch.vtensor<[],f32> -> !torch.
       vtensor<[3,4],i1>
    %11 = torch.aten.all.dim %10, %int1, %false : !torch.
       vtensor<[3,4],i1>, !torch.int, !torch.bool -> !torch
       .vtensor<[3],i1>
    %12 = torch.aten.eq.Tensor %arg0, %4 : !torch.vtensor
       <[3,4],f32>, !torch.vtensor<[],f32> -> !torch.
       vtensor<[3,4],i1>
    %13 = torch.aten.all.dim %12, %int0, %false : !torch.
       vtensor<[3,4],i1>, !torch.int, !torch.bool -> !torch
       .vtensor<[4],i1>
    %14 = torch.aten.eq.Tensor %arg0, %4 : !torch.vtensor
       <[3,4],f32>, !torch.vtensor<[],f32> -> !torch.
       vtensor<[3,4],i1>
    %15 = torch.aten.to.dtype %14, %int6, %false, %false, %
       none : !torch.vtensor<[3,4],i1>, !torch.int, !torch.
       bool, !torch.bool, !torch.none -> !torch.vtensor
       <[3,4],f32>
    %16 = torch.aten.sum %15, %none : !torch.vtensor<[3,4],
       f32>, !torch.none -> !torch.vtensor<[],f32>
    %17 = torch.aten.div.Tensor %16, %0 : !torch.vtensor
       <[],f32>, !torch.vtensor<[],f32> -> !torch.vtensor
       <[],f32>
    return %6, %9, %11, %13, %16, %17 : !torch.vtensor<[],
       i1>, !torch.vtensor<[],f32>, !torch.vtensor<[3],i1>,
        !torch.vtensor<[4],i1>, !torch.vtensor<[],f32>, !
       torch.vtensor<[],f32>
  }
}
```

D.2 Seed 2

D.2.1 LinAlg

```
#map = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
#map2 = affine_map<(d0) -> (d0)>
#map3 = affine_map<(d0, d1) -> (0, d1)>
#map4 = affine_map<(d0, d1) -> ()>
module {
  func.func @main(%arg0: tensor<4x4xf32>) -> (tensor<4
     x4xf32>, tensor<4xf32>, tensor<4xf32>, tensor<4x4xf32
     >, tensor<4x4xf32>, tensor<f32>) {
    %cst = arith.constant 0.000000e+00 : f32
    %cst_0 = arith.constant 0.000000e+00 : f64
    %cst_1 = arith.constant 0xFF800000 : f32
    %cst_2 = arith.constant 1.000000e-08 : f64
    %cst_3 = arith.constant 4.000000e+00 : f32
    %cst_4 = arith.constant 4.000000e+00 : f64
    %0 = tensor.empty() : tensor<4xf32>
    %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<4xf32
       >) -> tensor<4xf32>
    %2 = linalg.generic {indexing_maps = [#map, #map1],
       iterator_types = ["reduction", "parallel"]} ins(%
       arg0 : tensor<4x4xf32>) outs(%1 : tensor<4xf32>) {
    ^bb0(%in: f32, %out: f32):
      %31 = arith.addf %in, %out : f32
      linalg.yield %31 : f32
    } -> tensor<4xf32>
    %3 = linalg.generic {indexing_maps = [#map2, #map2],
       iterator_types = ["parallel"]} ins(%2 : tensor<4xf32
       >) outs(%0 : tensor<4xf32>) {
    ^bb0(%in: f32, %out: f32):
      %31 = arith.divf %in, %cst_3 : f32
      linalg.yield %31 : f32
    } -> tensor<4xf32>
    %4 = tensor.empty() : tensor<4x4xf32>
    %5 = linalg.generic {indexing_maps = [#map, #map1, #map
       ], iterator_types = ["parallel", "parallel"]} ins(%
       arg0, %3 : tensor<4x4xf32>, tensor<4xf32>) outs(%4 :
        tensor<4x4xf32>) {
```

```
^bb0(%in: f32, %in_5: f32, %out: f32):
  %31 = arith.subf %in, %in_5 : f32
  linalg.yield %31 : f32
} -> tensor<4x4xf32>
%6 = linalg.generic {indexing_maps = [#map, #map1],
   iterator_types = ["reduction", "parallel"]} ins(%5 :
    tensor<4x4xf32>) outs(%1 : tensor<4xf32>) {
^bb0(%in: f32, %out: f32):
  %31 = arith.addf %in, %out : f32
  linalg.yield %31 : f32
} -> tensor<4xf32>
%7 = linalg.generic {indexing_maps = [#map2, #map2],
   iterator_types = ["parallel"]} ins(%6 : tensor<4xf32
   >) outs(%0 : tensor<4xf32>) {
^bb0(%in: f32, %out: f32):
  %31 = arith.divf %in, %cst_3 : f32
  linalg.yield %31 : f32
} -> tensor<4xf32>
%8 = tensor.empty() : tensor<4x4xf64>
%9 = linalg.generic {indexing_maps = [#map, #map],
   iterator_types = ["parallel", "parallel"]} ins(%5 :
   tensor<4x4xf32>) outs(%8 : tensor<4x4xf64>) {
^bb0(%in: f32, %out: f64):
  %31 = arith.extf %in : f32 to f64
  linalg.yield %31 : f64
} -> tensor<4x4xf64>
%10 = tensor.empty() : tensor<1x4xf64>
%11 = linalg.fill ins(%cst_0 : f64) outs(%10 : tensor<1
   x4xf64>) -> tensor<1x4xf64>
%12 = linalg.generic {indexing_maps = [#map, #map3],
   iterator_types = ["reduction", "parallel"]} ins(%9 :
    tensor<4x4xf64>) outs(%11 : tensor<1x4xf64>) {
^bb0(%in: f64, %out: f64):
  %31 = arith.addf %in, %out : f64
  linalg.yield %31 : f64
} -> tensor<1x4xf64>
%13 = linalg.generic {indexing_maps = [#map, #map],
   iterator_types = ["parallel", "parallel"]} ins(%12 :
    tensor<1x4xf64>) outs(%10 : tensor<1x4xf64>) {
^bb0(%in: f64, %out: f64):
  %31 = arith.divf %in, %cst_4 : f64
```

```
    linalg.yield %31 : f64
  } -> tensor<1x4xf64>
  %14 = linalg.generic {indexing_maps = [#map, #map3, #
     map], iterator_types = ["parallel", "parallel"]} ins
     (%9, %13 : tensor<4x4xf64>, tensor<1x4xf64>) outs(%8
      : tensor<4x4xf64>) {
  ^bb0(%in: f64, %in_5: f64, %out: f64):
    %31 = arith.subf %in, %in_5 : f64
    linalg.yield %31 : f64
  } -> tensor<4x4xf64>
  %15 = linalg.generic {indexing_maps = [#map, #map, #map
     ], iterator_types = ["parallel", "parallel"]} ins
     (%14, %14 : tensor<4x4xf64>, tensor<4x4xf64>) outs
     (%8 : tensor<4x4xf64>) {
  ^bb0(%in: f64, %in_5: f64, %out: f64):
    %31 = arith.mulf %in, %in_5 : f64
    linalg.yield %31 : f64
  } -> tensor<4x4xf64>
  %16 = tensor.empty() : tensor<4xf64>
  %17 = linalg.fill ins(%cst_0 : f64) outs(%16 : tensor<4
     xf64>) -> tensor<4xf64>
  %18 = linalg.generic {indexing_maps = [#map, #map1],
     iterator_types = ["reduction", "parallel"]} ins(%15
     : tensor<4x4xf64>) outs(%17 : tensor<4xf64>) {
  ^bb0(%in: f64, %out: f64):
    %31 = arith.addf %in, %out : f64
    linalg.yield %31 : f64
  } -> tensor<4xf64>
  %19 = linalg.generic {indexing_maps = [#map2, #map2],
     iterator_types = ["parallel"]} ins(%18 : tensor<4
     xf64>) outs(%16 : tensor<4xf64>) {
  ^bb0(%in: f64, %out: f64):
    %31 = arith.divf %in, %cst_4 : f64
    linalg.yield %31 : f64
  } -> tensor<4xf64>
  %20 = linalg.generic {indexing_maps = [#map2, #map2],
     iterator_types = ["parallel"]} ins(%19 : tensor<4
     xf64>) outs(%0 : tensor<4xf32>) {
  ^bb0(%in: f64, %out: f32):
    %31 = arith.truncf %in : f64 to f32
    linalg.yield %31 : f32
```

```
} -> tensor<4xf32>
%21 = linalg.generic {indexing_maps = [#map2, #map2],
   iterator_types = ["parallel"]} ins(%20 : tensor<4
   xf32>) outs(%0 : tensor<4xf32>) {
^bb0(%in: f32, %out: f32):
  %31 = arith.truncf %cst_2 : f64 to f32
  %32 = arith.addf %in, %31 : f32
  linalg.yield %32 : f32
} -> tensor<4xf32>
%22 = linalg.generic {indexing_maps = [#map2, #map2],
   iterator_types = ["parallel"]} ins(%21 : tensor<4
   xf32>) outs(%0 : tensor<4xf32>) {
^bb0(%in: f32, %out: f32):
  %31 = math.sqrt %in : f32
  linalg.yield %31 : f32
} -> tensor<4xf32>
%23 = linalg.generic {indexing_maps = [#map, #map1, #
   map], iterator_types = ["parallel", "parallel"]} ins
   (%5, %22 : tensor<4x4xf32>, tensor<4xf32>) outs(%4 :
    tensor<4x4xf32>) {
^bb0(%in: f32, %in_5: f32, %out: f32):
  %31 = arith.divf %in, %in_5 : f32
  linalg.yield %31 : f32
} -> tensor<4x4xf32>
%transposed = linalg.transpose ins(%5 : tensor<4x4xf32
   >) outs(%4 : tensor<4x4xf32>) permutation = [1, 0]
%24 = linalg.fill ins(%cst : f32) outs(%4 : tensor<4
   x4xf32>) -> tensor<4x4xf32>
%25 = linalg.matmul ins(%transposed, %5 : tensor<4
   x4xf32>, tensor<4x4xf32>) outs(%24 : tensor<4x4xf32
   >) -> tensor<4x4xf32>
%26 = linalg.generic {indexing_maps = [#map, #map],
   iterator_types = ["parallel", "parallel"]} ins(%25 :
    tensor<4x4xf32>) outs(%4 : tensor<4x4xf32>) {
^bb0(%in: f32, %out: f32):
  %31 = arith.divf %in, %cst_3 : f32
  linalg.yield %31 : f32
} -> tensor<4x4xf32>
%27 = linalg.generic {indexing_maps = [#map, #map],
   iterator_types = ["parallel", "parallel"]} ins(%26 :
    tensor<4x4xf32>) outs(%4 : tensor<4x4xf32>) {
```

```
    ^bb0(%in: f32, %out: f32):
      %31 = math.absf %in : f32
      linalg.yield %31 : f32
    } -> tensor<4x4xf32>
    %28 = tensor.empty() : tensor<f32>
    %29 = linalg.fill ins(%cst_1 : f32) outs(%28 : tensor<
       f32>) -> tensor<f32>
    %30 = linalg.generic {indexing_maps = [#map, #map4],
       iterator_types = ["reduction", "reduction"]} ins(%27
        : tensor<4x4xf32>) outs(%29 : tensor<f32>) {
    ^bb0(%in: f32, %out: f32):
      %31 = arith.maximumf %in, %out : f32
      linalg.yield %31 : f32
    } -> tensor<f32>
    return %5, %7, %20, %23, %26, %30 : tensor<4x4xf32>,
       tensor<4xf32>, tensor<4xf32>, tensor<4x4xf32>,
       tensor<4x4xf32>, tensor<f32>
  }
}
```

D.2.2 Torch

```
module {
  func.func @main(%arg0: !torch.vtensor<[4,4],f32>) -> (!
     torch.vtensor<[4,4],f32>, !torch.vtensor<[4],f32>, !
     torch.vtensor<[4],f32>, !torch.vtensor<[4,4],f32>, !
     torch.vtensor<[4,4],f32>, !torch.vtensor<[],f32>) {
    %int6 = torch.constant.int 6
    %float1.000000e00 = torch.constant.float 1.000000e+00
    %true = torch.constant.bool true
    %int7 = torch.constant.int 7
    %int4 = torch.constant.int 4
    %float1.000000e-08 = torch.constant.float 1.000000e-08
    %int1 = torch.constant.int 1
    %none = torch.constant.none
    %false = torch.constant.bool false
    %int0 = torch.constant.int 0
    %0 = torch.prim.ListConstruct %int0 : (!torch.int) -> !
       torch.list<int>
```

```
%1 = torch.aten.sum.dim_IntList %arg0, %0, %false, %
   none : !torch.vtensor<[4,4],f32>, !torch.list<int>,
   !torch.bool, !torch.none -> !torch.vtensor<[4],f32>
%2 = torch.aten.div.Scalar %1, %int4 : !torch.vtensor
   <[4],f32>, !torch.int -> !torch.vtensor<[4],f32>
%3 = torch.aten.sub.Tensor %arg0, %2, %int1 : !torch.
   vtensor<[4,4],f32>, !torch.vtensor<[4],f32>, !torch.
   int -> !torch.vtensor<[4,4],f32>
%4 = torch.prim.ListConstruct %int0 : (!torch.int) -> !
   torch.list<int>
%5 = torch.aten.sum.dim_IntList %3, %4, %false, %none :
    !torch.vtensor<[4,4],f32>, !torch.list<int>, !torch
   .bool, !torch.none -> !torch.vtensor<[4],f32>
%6 = torch.aten.div.Scalar %5, %int4 : !torch.vtensor
   <[4],f32>, !torch.int -> !torch.vtensor<[4],f32>
%7 = torch.prim.ListConstruct %int0 : (!torch.int) -> !
   torch.list<int>
%8 = torch.aten.to.dtype %3, %int7, %false, %false, %
   none : !torch.vtensor<[4,4],f32>, !torch.int, !torch
   .bool, !torch.bool, !torch.none -> !torch.vtensor
   <[4,4],f64>
%9 = torch.aten.sum.dim_IntList %8, %7, %true, %none :
   !torch.vtensor<[4,4],f64>, !torch.list<int>, !torch.
   bool, !torch.none -> !torch.vtensor<[1,4],f64>
%10 = torch.aten.div.Scalar %9, %int4 : !torch.vtensor
   <[1,4],f64>, !torch.int -> !torch.vtensor<[1,4],f64>
%11 = torch.aten.sub.Tensor %8, %10, %float1.000000e00
   : !torch.vtensor<[4,4],f64>, !torch.vtensor<[1,4],
   f64>, !torch.float -> !torch.vtensor<[4,4],f64>
%12 = torch.aten.mul.Tensor %11, %11 : !torch.vtensor
   <[4,4],f64>, !torch.vtensor<[4,4],f64> -> !torch.
   vtensor<[4,4],f64>
%13 = torch.aten.sum.dim_IntList %12, %7, %false, %none
    : !torch.vtensor<[4,4],f64>, !torch.list<int>, !
   torch.bool, !torch.none -> !torch.vtensor<[4],f64>
%14 = torch.aten.div.Scalar %13, %int4 : !torch.vtensor
   <[4],f64>, !torch.int -> !torch.vtensor<[4],f64>
%15 = torch.aten.to.dtype %14, %int6, %false, %false, %
   none : !torch.vtensor<[4],f64>, !torch.int, !torch.
   bool, !torch.bool, !torch.none -> !torch.vtensor
   <[4],f32>
```

```
    %16 = torch.aten.add.Scalar %15, %float1.000000e-08, %
       int1 : !torch.vtensor<[4],f32>, !torch.float, !torch
       .int -> !torch.vtensor<[4],f32>
    %17 = torch.aten.sqrt %16 : !torch.vtensor<[4],f32> ->
       !torch.vtensor<[4],f32>
    %18 = torch.aten.div.Tensor %3, %17 : !torch.vtensor
       <[4,4],f32>, !torch.vtensor<[4],f32> -> !torch.
       vtensor<[4,4],f32>
    %19 = torch.prim.ListConstruct %int1, %int0 : (!torch.
       int, !torch.int) -> !torch.list<int>
    %20 = torch.aten.permute %3, %19 : !torch.vtensor
       <[4,4],f32>, !torch.list<int> -> !torch.vtensor
       <[4,4],f32>
    %21 = torch.aten.mm %20, %3 : !torch.vtensor<[4,4],f32
       >, !torch.vtensor<[4,4],f32> -> !torch.vtensor
       <[4,4],f32>
    %22 = torch.aten.div.Scalar %21, %int4 : !torch.vtensor
       <[4,4],f32>, !torch.int -> !torch.vtensor<[4,4],f32>
    %23 = torch.aten.abs %22 : !torch.vtensor<[4,4],f32> ->
        !torch.vtensor<[4,4],f32>
    %24 = torch.aten.max %23 : !torch.vtensor<[4,4],f32> ->
        !torch.vtensor<[],f32>
    return %3, %6, %15, %18, %22, %24 : !torch.vtensor
       <[4,4],f32>, !torch.vtensor<[4],f32>, !torch.vtensor
       <[4],f32>, !torch.vtensor<[4,4],f32>, !torch.vtensor
       <[4,4],f32>, !torch.vtensor<[],f32>
  }
}
```

D.3 Seed 3

D.3.1 LinAlg

```
#map = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
#map2 = affine_map<(d0, d1) -> (d0)>
#map3 = affine_map<(d0, d1) -> ()>
#map4 = affine_map<(d0) -> (d0)>
module {
```

```
func.func @main(%arg0: tensor<3x5xf32>) -> (tensor<3
    x5xf32>, tensor<5xf32>, tensor<3xf32>, tensor<f32>,
    tensor<3xf32>) {
  %cst = arith.constant dense<2.000000e+01> : tensor<3
      x5xf32>
  %cst_0 = arith.constant 2.000000e+01 : f32
  %cst_1 = arith.constant 0.000000e+00 : f32
  %0 = tensor.empty() : tensor<3x5xf32>
  %1 = linalg.fill ins(%cst_1 : f32) outs(%0 : tensor<3
      x5xf32>) -> tensor<3x5xf32>
  %2 = tensor.empty() : tensor<3xf32>
  %3 = linalg.fill ins(%cst_1 : f32) outs(%2 : tensor<3
      xf32>) -> tensor<3xf32>
  %4:2 = tm_tensor.scan dimension(1) inclusive(true) ins
      (%arg0 : tensor<3x5xf32>) outs(%1, %3 : tensor<3
      x5xf32>, tensor<3xf32>) {
  ^bb0(%arg1: f32, %arg2: f32):
    %18 = arith.addf %arg1, %arg2 : f32
    tm_tensor.yield %18 : f32
  } -> tensor<3x5xf32>, tensor<3xf32>
  %5 = tensor.empty() : tensor<5xf32>
  %6 = linalg.fill ins(%cst_1 : f32) outs(%5 : tensor<5
      xf32>) -> tensor<5xf32>
  %7 = linalg.generic {indexing_maps = [#map, #map1],
      iterator_types = ["reduction", "parallel"]} ins(%
      arg0 : tensor<3x5xf32>) outs(%6 : tensor<5xf32>) {
  ^bb0(%in: f32, %out: f32):
    %18 = arith.addf %in, %out : f32
    linalg.yield %18 : f32
  } -> tensor<5xf32>
  %8 = tensor.empty() : tensor<f32>
  %9 = linalg.fill ins(%cst_1 : f32) outs(%8 : tensor<f32
      >) -> tensor<f32>
  %10:2 = tm_tensor.scan dimension(0) inclusive(true) ins
      (%7 : tensor<5xf32>) outs(%6, %9 : tensor<5xf32>,
      tensor<f32>) {
  ^bb0(%arg1: f32, %arg2: f32):
    %18 = arith.addf %arg1, %arg2 : f32
    tm_tensor.yield %18 : f32
  } -> tensor<5xf32>, tensor<f32>
  %11 = tensor.empty() : tensor<3x5xi1>
```

```
%12 = linalg.generic {indexing_maps = [#map, #map, #map
   ], iterator_types = ["parallel", "parallel"]} ins
   (%4#0, %cst : tensor<3x5xf32>, tensor<3x5xf32>) outs
   (%11 : tensor<3x5xi1>) {
^bb0(%in: f32, %in_2: f32, %out: i1):
  %18 = arith.cmpf ogt, %in, %in_2 : f32
  linalg.yield %18 : i1
} -> tensor<3x5xi1>
%13 = linalg.generic {indexing_maps = [#map, #map],
   iterator_types = ["parallel", "parallel"]} ins(%12 :
    tensor<3x5xi1>) outs(%0 : tensor<3x5xf32>) {
^bb0(%in: i1, %out: f32):
  %18 = arith.uitofp %in : i1 to f32
  linalg.yield %18 : f32
} -> tensor<3x5xf32>
%14 = linalg.generic {indexing_maps = [#map, #map2],
   iterator_types = ["parallel", "reduction"]} ins(%13
   : tensor<3x5xf32>) outs(%3 : tensor<3xf32>) {
^bb0(%in: f32, %out: f32):
  %18 = arith.addf %in, %out : f32
  linalg.yield %18 : f32
} -> tensor<3xf32>
%15 = linalg.generic {indexing_maps = [#map, #map3],
   iterator_types = ["reduction", "reduction"]} ins(%
   arg0 : tensor<3x5xf32>) outs(%9 : tensor<f32>) {
^bb0(%in: f32, %out: f32):
  %18 = arith.addf %in, %out : f32
  linalg.yield %18 : f32
} -> tensor<f32>
%16 = linalg.generic {indexing_maps = [#map, #map2],
   iterator_types = ["parallel", "reduction"]} ins(%
   arg0 : tensor<3x5xf32>) outs(%3 : tensor<3xf32>) {
^bb0(%in: f32, %out: f32):
  %18 = arith.addf %in, %out : f32
  linalg.yield %18 : f32
} -> tensor<3xf32>
%17 = linalg.generic {indexing_maps = [#map4, #map4],
   iterator_types = ["parallel"]} ins(%16 : tensor<3
   xf32>) outs(%2 : tensor<3xf32>) {
^bb0(%in: f32, %out: f32):
  %18 = arith.subf %cst_0, %in : f32
```

```
      linalg.yield %18 : f32
    } -> tensor<3xf32>
    return %4#0, %10#0, %14, %15, %17 : tensor<3x5xf32>,
       tensor<5xf32>, tensor<3xf32>, tensor<f32>, tensor<3
       xf32>
  }
}
```

D.3.2 Torch

```
module {
  func.func @main(%arg0: !torch.vtensor<[3,5],f32>) -> (!
     torch.vtensor<[3,5],f32>, !torch.vtensor<[5],f32>, !
     torch.vtensor<[3],f32>, !torch.vtensor<[],f32>, !torch
     .vtensor<[3],f32>) {
    %0 = torch.vtensor.literal(dense<2.000000e+01> : tensor
       <3x5xf32>) : !torch.vtensor<[3,5],f32>
    %int6 = torch.constant.int 6
    %float2.000000e01 = torch.constant.float 2.000000e+01
    %false = torch.constant.bool false
    %int0 = torch.constant.int 0
    %int1 = torch.constant.int 1
    %none = torch.constant.none
    %1 = torch.aten.cumsum %arg0, %int1, %none : !torch.
       vtensor<[3,5],f32>, !torch.int, !torch.none -> !
       torch.vtensor<[3,5],f32>
    %2 = torch.prim.ListConstruct %int0 : (!torch.int) -> !
       torch.list<int>
    %3 = torch.aten.sum.dim_IntList %arg0, %2, %false, %
       none : !torch.vtensor<[3,5],f32>, !torch.list<int>,
       !torch.bool, !torch.none -> !torch.vtensor<[5],f32>
    %4 = torch.aten.cumsum %3, %int0, %none : !torch.
       vtensor<[5],f32>, !torch.int, !torch.none -> !torch.
       vtensor<[5],f32>
    %5 = torch.aten.gt.Tensor %1, %0 : !torch.vtensor
       <[3,5],f32>, !torch.vtensor<[3,5],f32> -> !torch.
       vtensor<[3,5],i1>
    %6 = torch.aten.to.dtype %5, %int6, %false, %false, %
       none : !torch.vtensor<[3,5],i1>, !torch.int, !torch.
       bool, !torch.bool, !torch.none -> !torch.vtensor
```

```
        <[3,5],f32>
    %7 = torch.prim.ListConstruct %int1 : (!torch.int) -> !
        torch.list<int>
    %8 = torch.aten.sum.dim_IntList %6, %7, %false, %none :
         !torch.vtensor<[3,5],f32>, !torch.list<int>, !torch
        .bool, !torch.none -> !torch.vtensor<[3],f32>
    %9 = torch.aten.sum %arg0, %none : !torch.vtensor
        <[3,5],f32>, !torch.none -> !torch.vtensor<[],f32>
    %10 = torch.prim.ListConstruct %int1 : (!torch.int) ->
        !torch.list<int>
    %11 = torch.aten.sum.dim_IntList %arg0, %10, %false, %
        none : !torch.vtensor<[3,5],f32>, !torch.list<int>,
        !torch.bool, !torch.none -> !torch.vtensor<[3],f32>
    %12 = torch.aten.rsub.Scalar %11, %float2.000000e01, %
        int1 : !torch.vtensor<[3],f32>, !torch.float, !torch
        .int -> !torch.vtensor<[3],f32>
    return %1, %4, %8, %9, %12 : !torch.vtensor<[3,5],f32>,
         !torch.vtensor<[5],f32>, !torch.vtensor<[3],f32>, !
        torch.vtensor<[],f32>, !torch.vtensor<[3],f32>
  }
}
```

D.4 Seed 4

D.4.1 LinAlg

```
#map = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> ()>
#map2 = affine_map<() -> ()>
#map3 = affine_map<(d0, d1) -> (d0)>
#map4 = affine_map<(d0) -> (d0)>
#map5 = affine_map<(d0, d1) -> (d1)>
#map6 = affine_map<(d0) -> ()>
module {
  func.func @main(%arg0: tensor<6x4xf32>) -> (tensor<f32>,
     tensor<6xf32>, tensor<4xf32>, tensor<i64>, tensor<i64
     >, tensor<6x4xf32>) {
    %cst = arith.constant 0.000000e+00 : f32
    %c0_i64 = arith.constant 0 : i64
    %cst_0 = arith.constant 0xFF800000 : f32
```

```
    %cst_1 = arith.constant 0x7F800000 : f32
    %cst_2 = arith.constant 2.400000e+01 : f32
    %cst_3 = arith.constant 4.000000e+00 : f32
    %cst_4 = arith.constant 6.000000e+00 : f32
    %0 = tensor.empty() : tensor<f32>
    %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<f32>)
        -> tensor<f32>
    %2 = linalg.generic {indexing_maps = [#map, #map1],
       iterator_types = ["reduction", "reduction"]} ins(%
       arg0 : tensor<6x4xf32>) outs(%1 : tensor<f32>) {
    ^bb0(%in: f32, %out: f32):
      %20 = arith.addf %in, %out : f32
      linalg.yield %20 : f32
    } -> tensor<f32>
    %3 = linalg.generic {indexing_maps = [#map2, #map2],
       iterator_types = []} ins(%2 : tensor<f32>) outs(%0 :
        tensor<f32>) {
    ^bb0(%in: f32, %out: f32):
      %20 = arith.divf %in, %cst_2 : f32
      linalg.yield %20 : f32
    } -> tensor<f32>
    %4 = tensor.empty() : tensor<6xf32>
    %5 = linalg.fill ins(%cst : f32) outs(%4 : tensor<6xf32
       >) -> tensor<6xf32>
    %6 = linalg.generic {indexing_maps = [#map, #map3],
       iterator_types = ["parallel", "reduction"]} ins(%
       arg0 : tensor<6x4xf32>) outs(%5 : tensor<6xf32>) {
    ^bb0(%in: f32, %out: f32):
      %20 = arith.addf %in, %out : f32
      linalg.yield %20 : f32
    } -> tensor<6xf32>
    %7 = linalg.generic {indexing_maps = [#map4, #map4],
       iterator_types = ["parallel"]} ins(%6 : tensor<6xf32
       >) outs(%4 : tensor<6xf32>) {
    ^bb0(%in: f32, %out: f32):
      %20 = arith.divf %in, %cst_3 : f32
      linalg.yield %20 : f32
    } -> tensor<6xf32>
    %8 = tensor.empty() : tensor<4xf32>
    %9 = linalg.fill ins(%cst : f32) outs(%8 : tensor<4xf32
       >) -> tensor<4xf32>
```

```
    %10 = linalg.generic {indexing_maps = [#map, #map5],
       iterator_types = ["reduction", "parallel"]} ins(%
       arg0 : tensor<6x4xf32>) outs(%9 : tensor<4xf32>) {
    ^bb0(%in: f32, %out: f32):
      %20 = arith.addf %in, %out : f32
      linalg.yield %20 : f32
    } -> tensor<4xf32>
    %11 = linalg.generic {indexing_maps = [#map4, #map4],
       iterator_types = ["parallel"]} ins(%10 : tensor<4
       xf32>) outs(%8 : tensor<4xf32>) {
    ^bb0(%in: f32, %out: f32):
      %20 = arith.divf %in, %cst_4 : f32
      linalg.yield %20 : f32
    } -> tensor<4xf32>
    %12 = tensor.empty() : tensor<i64>
    %13 = linalg.fill ins(%c0_i64 : i64) outs(%12 : tensor<
       i64>) -> tensor<i64>
    %14 = linalg.fill ins(%cst_0 : f32) outs(%0 : tensor<
       f32>) -> tensor<f32>
    %15:2 = linalg.generic {indexing_maps = [#map4, #map6,
       #map6], iterator_types = ["reduction"]} ins(%7 :
       tensor<6xf32>) outs(%14, %13 : tensor<f32>, tensor<
       i64>) {
    ^bb0(%in: f32, %out: f32, %out_5: i64):
      %20 = linalg.index 0 : index
      %21 = arith.index_cast %20 : index to i64
      %22 = arith.maximumf %in, %out : f32
      %23 = arith.cmpf ogt, %in, %out : f32
      %24 = arith.select %23, %21, %out_5 : i64
      linalg.yield %22, %24 : f32, i64
    } -> (tensor<f32>, tensor<i64>)
    %16 = linalg.fill ins(%cst_1 : f32) outs(%0 : tensor<
       f32>) -> tensor<f32>
    %17:2 = linalg.generic {indexing_maps = [#map4, #map6,
       #map6], iterator_types = ["reduction"]} ins(%11 :
       tensor<4xf32>) outs(%16, %13 : tensor<f32>, tensor<
       i64>) {
    ^bb0(%in: f32, %out: f32, %out_5: i64):
      %20 = linalg.index 0 : index
      %21 = arith.index_cast %20 : index to i64
      %22 = arith.minimumf %in, %out : f32
```

```
      %23 = arith.cmpf olt, %in, %out : f32
      %24 = arith.select %23, %21, %out_5 : i64
      linalg.yield %22, %24 : f32, i64
    } -> (tensor<f32>, tensor<i64>)
    %18 = tensor.empty() : tensor<6x4xf32>
    %19 = linalg.generic {indexing_maps = [#map, #map1, #
       map], iterator_types = ["parallel", "parallel"]} ins
       (%arg0, %3 : tensor<6x4xf32>, tensor<f32>) outs(%18
       : tensor<6x4xf32>) {
    ^bb0(%in: f32, %in_5: f32, %out: f32):
      %20 = arith.subf %in, %in_5 : f32
      linalg.yield %20 : f32
    } -> tensor<6x4xf32>
    return %3, %7, %11, %15#1, %17#1, %19 : tensor<f32>,
       tensor<6xf32>, tensor<4xf32>, tensor<i64>, tensor<
       i64>, tensor<6x4xf32>
  }
}
```

D.4.2 Torch

```
module {
  func.func @main(%arg0: !torch.vtensor<[6,4],f32>) -> (!
     torch.vtensor<[],f32>, !torch.vtensor<[6],f32>, !torch
     .vtensor<[4],f32>, !torch.vtensor<[],si64>, !torch.
     vtensor<[],si64>, !torch.vtensor<[6,4],f32>) {
    %int24 = torch.constant.int 24
    %int6 = torch.constant.int 6
    %int4 = torch.constant.int 4
    %int0 = torch.constant.int 0
    %false = torch.constant.bool false
    %int1 = torch.constant.int 1
    %none = torch.constant.none
    %0 = torch.aten.sum %arg0, %none : !torch.vtensor
       <[6,4],f32>, !torch.none -> !torch.vtensor<[],f32>
    %1 = torch.aten.div.Scalar %0, %int24 : !torch.vtensor
       <[],f32>, !torch.int -> !torch.vtensor<[],f32>
    %2 = torch.prim.ListConstruct %int1 : (!torch.int) -> !
       torch.list<int>
```

```
    %3 = torch.aten.sum.dim_IntList %arg0, %2, %false, %
       none : !torch.vtensor<[6,4],f32>, !torch.list<int>,
       !torch.bool, !torch.none -> !torch.vtensor<[6],f32>
    %4 = torch.aten.div.Scalar %3, %int4 : !torch.vtensor
       <[6],f32>, !torch.int -> !torch.vtensor<[6],f32>
    %5 = torch.prim.ListConstruct %int0 : (!torch.int) -> !
       torch.list<int>
    %6 = torch.aten.sum.dim_IntList %arg0, %5, %false, %
       none : !torch.vtensor<[6,4],f32>, !torch.list<int>,
       !torch.bool, !torch.none -> !torch.vtensor<[4],f32>
    %7 = torch.aten.div.Scalar %6, %int6 : !torch.vtensor
       <[4],f32>, !torch.int -> !torch.vtensor<[4],f32>
    %values, %indices = torch.aten.max.dim %4, %int0, %
       false : !torch.vtensor<[6],f32>, !torch.int, !torch.
       bool -> !torch.vtensor<[],f32>, !torch.vtensor<[],
       si64>
    %values_0, %indices_1 = torch.aten.min.dim %7, %int0, %
       false : !torch.vtensor<[4],f32>, !torch.int, !torch.
       bool -> !torch.vtensor<[],f32>, !torch.vtensor<[],
       si64>
    %8 = torch.aten.sub.Tensor %arg0, %1, %int1 : !torch.
       vtensor<[6,4],f32>, !torch.vtensor<[],f32>, !torch.
       int -> !torch.vtensor<[6,4],f32>
    return %1, %4, %7, %indices, %indices_1, %8 : !torch.
       vtensor<[],f32>, !torch.vtensor<[6],f32>, !torch.
       vtensor<[4],f32>, !torch.vtensor<[],si64>, !torch.
       vtensor<[],si64>, !torch.vtensor<[6,4],f32>
  }
}
```