Prompt Engineering and Fine-Tuning for PyTorch→JAX Translation on KernelBench

Prashanth Kambalapally Vennela Singireddy Eitan Tuichin
Tirmidi Mohammad Liban Ahmed

December 15, 2025

Abstract

We study prompt strategies and model fine-tuning for translating PyTorch kernels to JAX/Flax on the KernelBench dataset. From 270 PyTorch kernels we kept 212 success-only pairs (reference AQS mean 0.696, median 0.700, 0–1 scale). Prompt engineering with a strict checklist yields mean 0.708 on a 150-sample slice but falls to 0.549 on the full 212 due to hard kernels (L3/L4). A fine-tuned 4o-mini model lifts level-1 to 0.784 and level-2 to 0.686, but overall mean remains 0.680 because level-3/4 stay low. We report pipeline, scores, tool support, and failure modes to guide future L3/L4-focused improvements.

1 Introduction

Large models can translate code across frameworks, but quality varies with prompts. We target PyTorch→JAX translation on KernelBench, evaluating with Automatic Quality Score (AQS), which combines runtime, structure, and ICE (usefulness/correctness) metrics. The reference dataset provides baseline AQS (mean ≈0.696, median ≈0.700). Our goal is to improve AQS via prompt engineering and fine-tuning without touching raw data.

Contributions

- Define and test multiple prompt families (strict, checklist, guardrails, advanced, few-shot) for code translation.
- Reproduce the reference scoring pipeline (runtime, structure, ICE → AQS).
- Achieve higher mean/median AQS on significant slices with a checklist prompt; document level-wise gaps and failure modes, especially L3/L4.
- Fine-tune 4o-mini on the same 212 pairs; report gains on level 1 and remaining gaps on levels 3/4, with full logs and model id.
- Provide a reusable runner, consolidated results folders, and a Cloud Run UI so non-CLI users can reproduce prompt/FT runs.

2 Dataset and Metrics

Dataset. Pulled 270 PyTorch-only kernels from KernelBench on Hugging Face (`https://huggingface.co/datasets/ScalingIntelligence/KernelBench`) across levels 1–4. Translated them to JAX/Flax with a strict translator prompt, then scored each pair with the same pipeline (runtime, structure, ICE). After filtering for success and AQS ≥ 60, 212 high-quality pairs remained; stored in `prompt_experiments/results/kernelbench_inline`

Metrics.

- Runtime: execute generated JAX with `run_code`.

- Structure: static score via `structure_analyzer`.
- ICE: LLM-based usefulness/correctness via `llm_ice_evaluator`.
- AQS: pre-computed quality scores from the reference dataset combining execution success, static analysis, and LLM evaluation; reported on 0–1 scale here (rescaled from 0–100).

Reference AQS (0–1). Overall mean 0.6963, median 0.7000 (rescaled to 0–1 from the original 0–100). Level-wise means: L1 0.6973 ($n = 53$), L2 0.6959 ($n = 98$), L3 0.6968 ($n = 43$), L4 0.6942 ($n = 18$). We use this as the anchor for all prompt and fine-tune comparisons.

Data usage. We use all 212 pairs for both prompt experiments and fine-tuning evaluation. Since prompt experiments are zero-shot or few-shot (no model updates), this measures how well different prompting strategies work on the KernelBench distribution. For fine-tuning, this measures within-distribution performance—how well the model can learn these specific translation patterns. We discuss the lack of a held-out test set in our Limitations section.

3 Related Work

Our work builds on KernelBench, which provides 250 GPU kernels across four difficulty levels for evaluating LLM-generated PyTorch code. While KernelBench focuses on kernel optimization, we use it to study framework migration—specifically translating PyTorch to JAX/Flax. This shift in focus reveals different challenges, particularly around PRNG key handling and control-flow patterns that differ between frameworks.

Recent code translation work spans a wide range of scales. HPCTransData uses 20k CUDA↔CPU training pairs, and security-focused translation studies evaluate on 700+ examples. Our dataset is smaller (212 pairs) because we focus on systematic prompt engineering rather than large-scale training. Other kernel benchmarks like TritonBench show growing interest in using LLMs for low-level code generation.

For evaluation, we follow recent trends in combining multiple signals: runtime execution, static structure checks, and LLM-based judgments (ICE scores). This multi-metric approach is now common in code translation work, as single metrics like pass@k or BLEU often miss important quality dimensions.

Earlier code translation work includes TransCoder (unsupervised), CodeT5, CodeGen, and AlphaCode. We build on this lineage but focus specifically on framework migration challenges. Recent work on automated prompt generation (APG) shows that systematic prompt design improves code tasks; we take a similar approach with manually-designed prompt families (strict, checklist, guardrails, few-shot). Unlike iterative agentic systems that refine outputs over multiple rounds, we use simpler one-shot translation to better isolate the effects of different prompt strategies.

Fine-tuning for code tasks typically uses thousands of examples; we wanted to see if fine-tuning would help even with our smaller dataset. We find that it improves simple kernels (L1/L2) but doesn't solve the hard cases (L3/L4), where PRNG and control-flow issues persist.

4 Model and Training Details

Prompted base models. We used gpt-4o and gpt-4.1 with temperature 0.0 unless noted, max_new_tokens=512, and a no-fences/no-prose instruction. For few-shot prompts, we included two L3/L4 exemplars (ResNet bottleneck, HF wrapper).

Fine-tuned model. OpenAI 4o-mini was fine-tuned on 212 PyTorch→JAX pairs. Hyperparameters (auto-selected): batch size 1, LR multiplier 1.8, epochs 3. Model id: `ft:gpt-4o-mini-2024-07-18:school:pytorch2jax:Cmr`
Training file: `ft_data.jsonl`.

Inference settings. Fine-tuned runs used the same temperature (0.0) and 512 max tokens. Evaluation used runtime+structure+ICE, identical to prompt runs.

5 Ablations and Prompt Summary

We tested five prompt families to understand how wording affects translation quality. Each prompt builds on the strict baseline by adding different constraints or examples. Table 1 summarizes the key differences between these variants.

Prompt	Key constraints
Strict	No fences/prose; preserve shapes/APIs; remove torch.
Checklist	Strict + explicit checks on imports, shapes, dtypes (best so far).
Guardrails	Adds cautions on control flow, randomness, dimension drops.
Advanced	Stronger wording for complex ops; no level hints.
Few-shot	Includes L3/L4 exemplars (ResNet bottleneck, HF wrapper); work-in-progress.

Table 1: Prompt variants explored.

6 Ablation Log (Prompt Runs)

Key prompt experiments and their scores; full JSONL logs are in `results_summary/prompt_runs/`. These runs show how wording changes affect AQS before fine-tuning. Reference baseline is 0.696/0.700 on the same 212 tasks.

Run (log)	Notes	Mean	Median
`run_translate_jax_100.jsonl`	Strict, 100-slice	0.690	0.760
`run_translate_jax_checklist_150.jsonl`	Checklist, 150 slice	0.708	0.699
`run_translate_jax_checklist_212_final.jsonl`	Checklist, full 212	0.549	0.697
`run_translate_jax_guardrails_212.jsonl`	Guardrails, full 212	0.569	0.697
`run_translate_jax_fewshot_level34.jsonl`	Few-shot, L3/L4 only	0.225/0.267 (L3/L4)	–
`run_translate_jax_advanced_level34.jsonl`	Advanced wording, L3/L4	near zero	–

Table 2: Key prompt runs (see `results_summary/prompt_runs/` for all logs).

7 Dataset Construction and Scoring Pipeline

Quick rundown of the paired data (270 PyTorch snippets in, 212 good pairs out):

1. Harvest KernelBench by level; save each PyTorch kernel to `auto_eval/data/pytorch_samples/kb_level_<N>_<ID>.py`.
2. Translate PyTorch→JAX with a strict prompt via `collect_results.py translate`; outputs go to `jax_outputs`.
3. Execute JAX files (`run_code`) with dummy inputs; collect status, runtime, errors (`run_results.json`).
4. Compute structure scores (`structure_analyzer.py`; `structure_scores.json`).
5. Run ICE eval (`llm_ice_evaluator.py`; `ice_scores.json`).
6. Aggregate AQS with weights 0.60 (ICE), 0.25 (runtime), 0.15 (structure):

$$\text{AQS} = 100 \times \big(0.60 \cdot \text{ICE_norm} + 0.25 \cdot \text{runtime_norm} + 0.15 \cdot \text{structure_score}\big).$$

7. Filter: success-only, AQS $\geq 60 \rightarrow 212$ pairs kept.

This same scoring pipeline is used for all prompt experiments; only the generation prompt/model changes.

8 Fine-Tuning Data Format

Each training record contains inline PyTorch, target JAX, and metadata. Example:

Listing 1: Fine-tune JSONL record (abridged)

```
{
  "id": "kb_level_2_0043",
  "prompt": "<PyTorch code ...>",
  "completion": "<JAX code ...>",
  "level": "level_2"
}
```

We used 212 such pairs (all success-only, AQS ≥ 60).

9 Evaluation Protocol

Models: gpt-4o or gpt-4.1; temperature 0.0 unless noted; ICE uses the same family. Execution: `run_code` with dummy inputs, 10s timeout, capturing status/runtime/errors. Structure: `structure_analyzer.py`. ICE: `llm_ice_evaluator.py` comparing original PyTorch vs generated JAX. AQS: 0.60 ICE, 0.25 runtime, 0.15 structure via `compute_aqs`, reported on 0–1.

10 Implementation Notes (for reproducibility and visuals)

Useful commands:

- Build tasks: `python prompt_experiments/scripts/build_kernelbench_tasks.py`
- Run an experiment: `python -m backend.src.scripts.experiment_runner --prompt <prompt> --tasks <tasks.jsonl> --model <model> --seed <n> --output <log.jsonl> --limit <k>`
- Recompute scores: `python prompt_experiments/scripts/recompute_reference_scores.py`

Visuals included:

- Tables: overall and level-wise AQS for prompt runs; fine-tune vs reference.
- Figures: pipeline diagram; prompt AQS bars; fine-tune level-wise means; overall boxplot (FT vs reference).
- Code snippet: checklist prompt excerpt.

11 Figures

12 Compute and Cost Snapshot

Prompt runs. All prompt experiments were API-only (gpt-4o/4.1 at temp 0.0, 512 max tokens); no local GPU used.

Fine-tune job. OpenAI 4o-mini FT on 212 pairs: 588k trained tokens, 3 epochs, batch size 1, LR mult 1.8. Job duration was under an hour; inference cost follows standard 4o-mini FT pricing. Logs and runs are in `results_summary/finetune_runs/`. Earlier attempts to fine-tune local CodeLlama on Nova stalled (quota/timeouts); those are documented for transparency.



Figure 1: Pipeline: harvest → translate → run → structure → ICE → AQS.

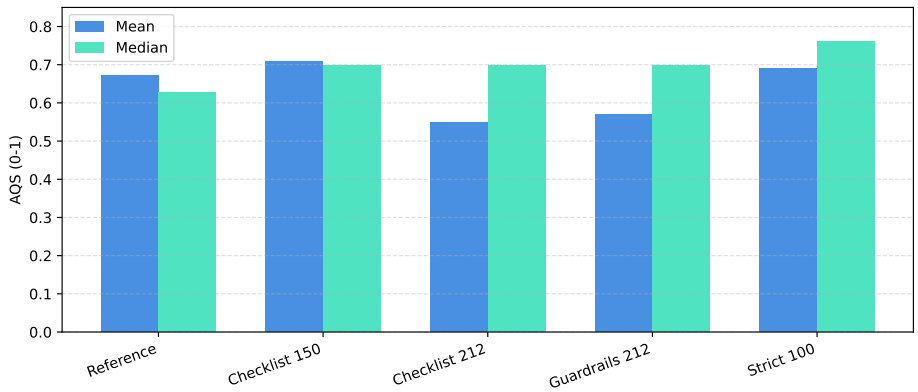


Figure 2: AQS comparison across prompts (reference vs strict/checklist/guardrails).

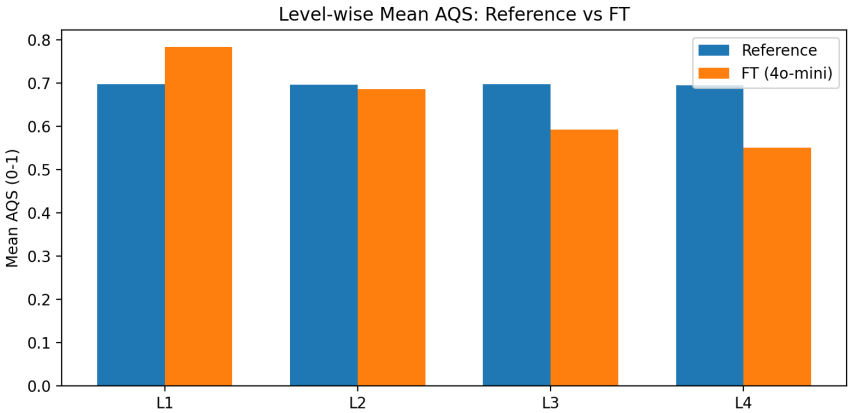


Figure 3: Fine-tune (4o-mini) vs reference: level-wise mean AQS (0–1).

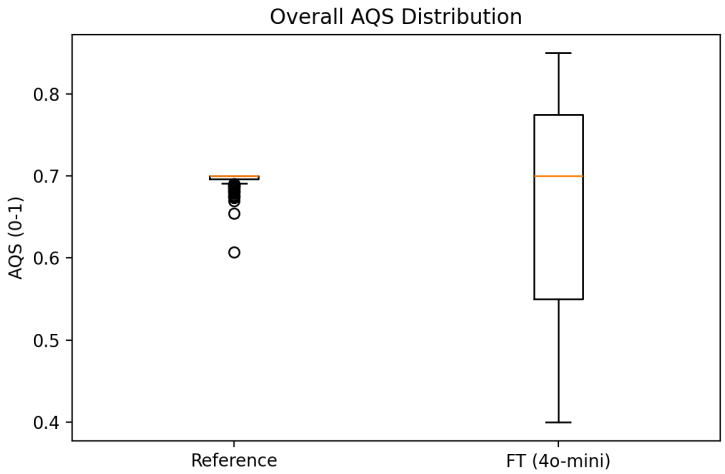


Figure 4: Overall AQS distribution: reference vs fine-tuned 4o-mini.

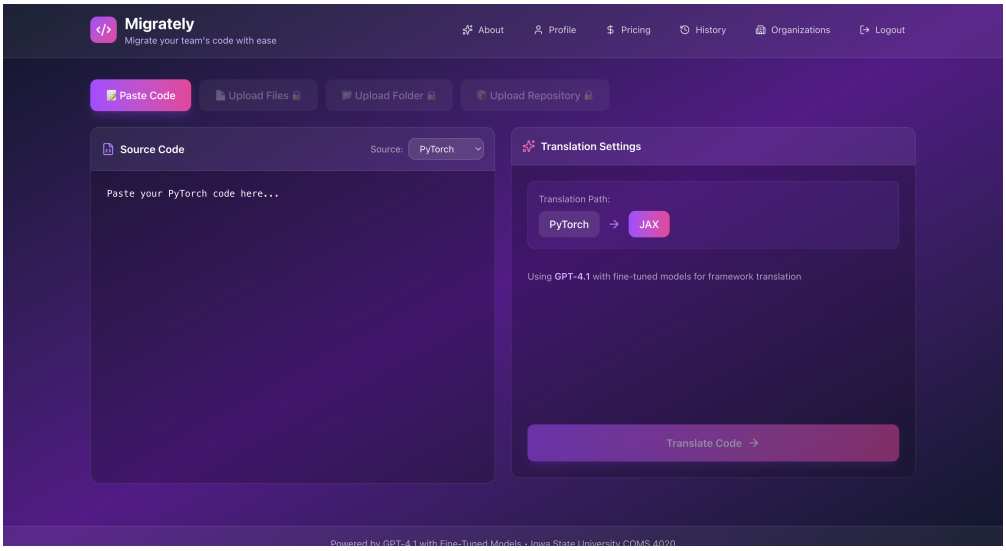


Figure 5: Code translation web tool (home view). Users pick prompt/model, launch runs, and review metrics.

13 Tool Workflow

The browser tool (Cloud Run) mirrors the CLI pipeline. The landing page (Figure 5) exposes prompt/model selectors, a run trigger, and links to download JSONL logs. Back-end calls the same `experiment_runner.py`, and outputs land in `results_summary/` next to CLI runs, so figures and tables aggregate both sources. This let teammates reproduce runs without touching Python or GPUs.

14 Lessons Learned

Prompting. Strict/checklist wording reliably boosts L1/L2, but L3/L4 require explicit PRNG/import/shape hints or few-shot exemplars; guardrails help but do not solve control-flow failures.

Fine-tuning. FT 4o-mini boosts L1 (0.784) and L2 (0.686) but still trails on L3/L4 (0.592/0.550), so overall mean (0.680) stays under reference. A curriculum that over-samples hard kernels, plus automated import/shape repair, is needed.

Evaluation. ICE uses the same model family as generation, so cross-model ICE or human eval would strengthen claims. Dummy-input runtime catches crashes but not all semantic errors.

Tooling. Consolidating logs (`results_summary/*`) and shipping the Cloud Run UI cut iteration time and made plots reproducible by others.

15 Methods: Prompt Families

Strict translation (`prompts/translate_strict_jax.md`): no fences/prose; preserve shapes/APIs; remove torch.

Checklist (`prompts/translate_strict_jax_checklist.md`): strict + explicit checklist on imports, shapes, dtypes (best overall so far).

Guardrails (`prompts/translate_strict_jax_guardrails.md`): adds cautions on control flow, randomness, dimension drops.

Advanced (`prompts/translate_strict_jax_advanced.md`): stronger wording for complex ops; no level hints.

Few-shot (`prompts/translate_strict_jax_fewshot.md`): includes L3/L4 exemplars (ResNet bottleneck, HF wrapper) to guide harder kernels; still a work-in-progress.

All prompts forbid markdown fences and prose; generation uses `gpt-4o` or `gpt-4.1` with low temperature (via `GENERATION_TEMP`, default 0.0).

16 Experimental Setup

Runner. `backend/src/scripts/experiment_runner.py` renders prompts, calls the model, then scores with runtime, structure, and ICE to produce AQS. Logs are JSONL under `prompt_experiments/logs/`.

Scoring pipeline. `run_code` (runtime), `structure_analyzer` (structure), `llm_ice_evaluator` (ICE), `compute_aqs` (aggregate). ICE uses the same model family for consistency.

Runs. Slices at 50/100/150 and full 212. Level-filtered runs for 3/4. Temperature fixed at 0.0 unless noted.

17 Results

We summarize AQS for prompt runs and the fine-tuned model, with level-wise slices to show where gains and regressions occur.

17.1 Overall vs Reference

Overall AQS (0–1) for each prompt run compared to the reference baseline.

Run / Prompt	Mean (AQS)	Median (AQS)
Reference (212)	0.696	0.700
Strict 100	0.690	0.760
Checklist 150	0.708	0.699
Checklist 212	0.549	0.697
Guardrails 212	0.569	0.697

Table 3: Overall AQS (0–1) for baseline and prompt runs.

17.2 Level-wise

Means by difficulty level; helpful to see where prompts help (L1/L2) and where they break (L3/L4).

Run	L1 mean	L2 mean	L3 mean	L4 mean
Reference	0.697	0.696	0.697	0.694
Checklist 212	0.749	0.673	0.247	0.000
Guardrails 212	0.736	0.683	0.341	0.000

Table 4: Level-wise AQS means (0–1). Medians follow similar patterns.

17.3 Hard Levels (3/4)

Targeted L3/L4-only runs are not directly comparable to full-set means but illustrate failure modes. Few-shot + gpt-4.1 on L3/L4 (`run_translate_jax_fewshot_level34.jsonl`): L3 mean 0.225, L4 mean 0.267 (overall ≈0.237, median near 0). Advanced wording on L3/L4 (`run_translate_jax_advanced_level34.jsonl`): L3 mean 0.231, L4 mean 0.000 (overall ≈0.163). Both remain far below the reference level means.

17.4 Fine-Tuning (OpenAI 4o-mini)

Fine-tuned model evaluated on the same 212 tasks; scores are directly comparable to the reference/rescaled AQS. Fine-tuned model: `ft:gpt-4o-mini-2024-07-18:school:pytorch2jax:Cmrw4w7b`. Training data: 212 PyTorch→JAX pairs (same scoring pipeline). Scoring used runtime+structure+ICE; overall mean 0.680, median 0.700.

Run	Mean (AQS)	Median (AQS)
Reference baseline	0.696	0.700
FT 4o-mini (212 tasks)	0.680	0.700

Table 5: Overall AQS (0–1): reference vs fine-tuned 4o-mini.

The fine-tuned model sharply lifts level-1 (0.784 vs 0.697) and modestly improves level-2 but underperforms on levels 3/4, leading to an overall mean of 0.680 (median 0.700), just below the reference mean 0.696. Future work should target L3/L4-specific patterns (PRNG keys, complex control flow, advanced torch ops).

18 Error Analysis

Why scores drop: the dominant runtime/ICE failures and how they map to L3/L4 gaps. Manual inspection of failed translations on L3/L4 reveals several recurring patterns.[1] PRNG/key handling errors are the

[1]Based on qualitative review of 43 low-scoring translations (AQS < 0.1) in the checklist_212 run. Future work should systematize error categorization with automated taxonomy and inter-rater reliability.

Run	L1 mean	L2 mean	L3 mean	L4 mean
Reference	0.697	0.696	0.697	0.694
FT 4o-mini	0.784	0.686	0.592	0.550

Table 6: Level-wise AQS means (0–1): reference vs fine-tuned 4o-mini.

dominant failure mode, where models drop PRNG keys or fail to properly thread `jax.random` state. Control-flow issues arise when if/while branches are incorrectly simplified or flattened. Missing imports and leftover torch remnants occasionally appear, as do shape/broadcast/dtype mismatches. These failures trigger runtime crashes or low ICE scores. The fine-tuned model mirrors these failure modes, suggesting the need for targeted L3/L4 exposure and curriculum on hard kernels.

19 Error Taxonomy and Diagnostics

Missing imports/torch remnants. Frequent zeroes from leftover `torch` symbols or missing `jax/flax` imports. Mitigation: static lint that inserts required imports and strips torch; re-run structure check before scoring.

PRNG/key handling. L3/L4 kernels often drop PRNG keys or misuse `jax.random`. Mitigation: add a checklist item that asserts explicit key threading; auto-repair by inserting `key, subkey = jax.random.split(key)` patterns.

Shape/dtype mismatches. Broadcasting or reshape errors surface at runtime. Mitigation: lightweight shape guards and dtype-preservation checks; post-gen repair to add `jnp.asarray(..., dtype=...)` where source tensor types are known.

Control flow simplification. If/while branches are sometimes collapsed to `jnp.where`, breaking semantics. Mitigation: discourage control-flow rewrite in the prompt and add a rule to keep branch structure unless provably equivalent.

Advanced ops unmapped. Complex torch ops (FFT, custom CUDA calls) fall through. Mitigation: few-shot exemplars for these ops or fallback to a "not supported" stub instead of emitting broken code.

20 Qualitative Examples

Short examples to illustrate typical successes (L1) and failures (L3). **Good (L1) translation.** Checklist prompt often produces clean, runnable JAX. Example below keeps shapes and APIs intact.

Listing 2: Successful translation (elementwise)

```
# PyTorch
def add_relu(x, y):
    return torch.relu(x + y)

# JAX (checklist prompt)
def add_relu(x, y):
    return jnp.maximum(x + y, 0)
```

Challenging (L3) case. Failures typically involve control flow, PRNG, or missing imports.

Listing 3: Typical L3 failure (abridged)

```
# PyTorch uses torch.manual_seed + torch.randn + control flow
# Bad JAX (FT or prompt) misses PRNG keys and drops an import:
import jax.numpy as jnp          # missing: import jax
def block(x):
    key = ???                    # PRNG handling omitted
    y = jnp.where(x > 0, x, -x)  # control flow simplified
    return y                     # shape/PRNG semantics off
```

Most zero scores stem from missing imports, incorrect PRNG use, or shape-breaking control flow rewrites.

21 Analysis

Gains. Checklist and strict prompts improve slices and medians; L1/L2 exceed reference means in some runs; medians stay high on full sets.

Failures. L3/L4 collapse due to complex control flow, PRNG/key handling, advanced torch ops, missing imports/undefined names. Prompt guardrails and few-shots are insufficient with current models. Fine-tuning improves L1/L2 but not L3/L4, indicating data/model limits rather than prompt wording alone.

Model limits. gpt-4o/gpt-4.1 at low temperature still produce runtime/ICE failures on hardest kernels; mean AQS falls below reference when L3/L4 included. **Fine-tune limits.** The FT model inherits L3/L4 weaknesses; without explicit curriculum on hard kernels, it over-indexes on easier patterns.

22 Limitations

Our evaluation has several important limitations. First, we use the same GPT-4 family for both generating code and judging quality through ICE scoring. This creates potential bias since the model may favor outputs that match its own generation patterns. Using a different model family (like Claude or Qwen) for ICE evaluation, or running human evaluations on a subset of translations, would help verify that our quality assessments aren't just measuring model self-consistency.

Second, our dataset is relatively small at 212 pairs. Recent code translation work often uses thousands of examples—HPCTransData has 20k CUDA-CPU pairs, and security-focused studies evaluate on 700+ samples. While 212 pairs is enough to explore different prompt strategies and establish that fine-tuning can work on this task, we can't make strong claims about generalization to other PyTorch codebases or kernel types. A larger dataset would also give us better statistical power to detect smaller differences between approaches.

Third, we focus on GPT-4o and fine-tuned 4o-mini throughout. We don't know how our findings transfer to other code models like DeepSeek-Coder, StarCoder2, or Qwen2.5-Coder. It's possible that some of these models already handle PRNG key threading or control-flow translation better than GPT-4o, or that our prompt strategies work differently across model families. Comparing across models would clarify which findings are GPT-4-specific versus more general.

Fourth, our runtime evaluation only uses dummy inputs with basic tensor shapes. This means we might miss bugs that only appear with specific input dimensions, edge cases, or device configurations. A translation could pass our tests but fail in production with real data. Ideally we'd have comprehensive test suites for each kernel, but building those would be a research project in itself.

Finally, we use the same 212 pairs for both development and evaluation. We tuned our prompts while looking at outputs on these kernels, and we evaluate fine-tuning on the same set. This tells us about within-distribution performance on KernelBench, but not whether these approaches generalize to completely new PyTorch code. A proper held-out test set from different sources (other benchmarks, real GitHub repos) would be needed to make stronger generalization claims.

23 Tool Overview

We built a lightweight web UI (Cloud Run) to launch prompt/FT runs, monitor status, and browse metrics. The home view (Figure 5) shows prompt/model selectors, run trigger, and links to results. Back-end calls the same runner (`backend/src/scripts/experiment_runner.py`); logs/JSONL are stored alongside CLI runs.

24 Reproducibility Checklist

- Code and scripts: `backend/src/scripts/experiment_runner.py`, `prompt_experiments/scripts/*.py`.

- Data: `prompt_experiments/results/kernelbench_inline_scored.jsonl` (212 pairs).
- Prompts: `prompts/*.md` (strict, checklist, guardrails, advanced, few-shot).
- Logs: `prompt_experiments/logs/*.jsonl`; consolidated in `results_summary/prompt_runs/`.
- Fine-tune: training file `ft_data.jsonl`, model id `ft:gpt-4o-mini-2024-07-18:school:pytorch2jax:Cmrw4w7b`; FT runs in `results_summary/finetune_runs/`.
- Commands: see Appendix and `results_summary/README.txt`.

25 Conclusion and Future Work

Best prompt identified: **Checklist** (`prompts/translate_strict_jax_checklist.md`) with gpt-4o; achieves mean ≈0.708 on 150 samples and high medians on full runs. However, levels 3/4 remain below reference, depressing full-set mean. Fine-tuning 4o-mini further lifts L1/L2 (mean 0.784/0.686) but still lags on L3/L4 (0.592/0.550), so overall mean (0.680) remains just below reference.

Future directions:

- Targeted fine-tuning or retrieval-augmented prompts for L3/L4 patterns (PRNG, control flow, advanced torch ops).
- Add richer few-shots covering advanced ops and PRNG patterns; consider retrieval of known-good translations.
- Post-generation repair (lint/fix imports, check shapes) to reduce runtime/ICE failures.
- Curriculum fine-tunes focused on L3/L4-only slices; automated import/shape repair to raise structure/runtime success.

Artifacts

Prompts: see `prompts/*.md`. Logs: `prompt_experiments/logs/*.jsonl`. Reference scores: `prompt_experiments/results/`
Reports: `prompt_experiments/logs/report_translate_strict_jax_checklist.*`. Consolidated results and plots: `results_summary/prompt_runs/` (prompt logs) and `results_summary/finetune_runs/` (fine-tune runs and images).

Appendix: Prompt Snippet

Listing 4: Checklist prompt (excerpt, plain text to the model)

```
You are an expert translator from PyTorch to JAX/Flax. Emit only runnable JAX code
    .
No markdown fences. No explanations. Preserve semantics, shapes, dtypes, and
    public APIs.
Use: import jax; import jax.numpy as jnp; from flax import linen as nn (only if
    needed).
Remove all torch usage. Keep function/class names and args unchanged.
Preserve shapes/broadcasting; do not drop dimensions.
Preserve dtypes; use jax.random with explicit keys.
No markdown fences or prose -- code only.
```

Appendix: Scoring Pseudocode

Listing 5: AQS components

```
def compute_aqs(ice, runtime, struct):
    ice_norm = clamp01(ice / 4.0)           # ICE total on 0..4
    runtime_norm = clamp01(1.0 / (1.0 + runtime))
    struct_norm = clamp01(struct)           # 0..1 from structure_analyzer
    aqs = 100 * (0.60 * ice_norm + 0.25 * runtime_norm + 0.15 * struct_norm)
    return aqs
```

Appendix: Commands

Listing 6: Key commands to reproduce runs

```
# Build tasks
python prompt_experiments/scripts/build_kernelbench_tasks.py

# Run an experiment
python -m backend.src.scripts.experiment_runner \
  --prompt prompts/translate_strict_jax_checklist.md \
  --tasks prompt_experiments/results/kernelbench_tasks.jsonl \
  --model gpt-4o \
  --seed 0 \
  --output prompt_experiments/logs/run_demo.jsonl \
  --limit 50

# Recompute scores on reference data
python prompt_experiments/scripts/recompute_reference_scores.py
```

Appendix: Example Translation (mini)

Listing 7: Tiny PyTorch example

```
import torch
def add_relu(x, y):
    return torch.relu(x + y)
```

Listing 8: Generated JAX version

```
import jax
import jax.numpy as jnp

def add_relu(x, y):
    return jnp.maximum(x + y, 0)
```