Agentic Framework for Deep Learning workload migration via In-Context Learning

Qiyue Liang[†] **Steven Ingram** **George Vanica**[*] **Andi Gavrilescu**[*]
Newfel Harrat **Hassan Sipra**[†] **Sethuraman Sankaran**[‡]
Google, Sunnyvale, CA [*]Google, Kirkland, WA [†]Google, Mountain View, CA [‡]Google, New York, NY
{jqiyueliang, stingram, gvanica, andig, newfel, hisipra, sethusankaran}@google.com

***Abstract*—Translating deep learning models from PyTorch's flexible, object-oriented design to JAX's functional, stateless setup is usually a manual and error-prone task. Automated migration is challenging because Large Language Models (LLMs) struggle with strict and dynamic API alignment and are prone to mistakes for exacting operations. We propose a fully automated system that combines In-Context Learning (ICL) with oracle-driven self-debugging. First, we curated an ICL context that serves as a strict reference for idiomatic JAX styling and test case generation. Second, instead of depending on the LLM to deduce mathematical outputs, we run the source PyTorch modules to get their actual dynamic tensor states. This creates an unchangeable execution oracle. We then use an autonomous agentic loop to synthesize tests based on the oracle data. The test cases are executed repeatedly, and the traceback is sent back to the LLM for self-correction. We use Gemini-2.5-Pro for this study (model can be swapped in the code link available below). Ablations show that combining ICL references with oracle grounding and self-debugging greatly outperforms pure instructional and basic agentic baselines. This improvement does not add an excessive computational overhead. Our lightweight pipeline achieves 91% numerical equivalence (compared to baseline: 9%, instruction + self-debugging: 27%) on neural modules, providing a highly reliable, scalable blueprint for cross-framework migration. The code is available at https://github.com/AI-Hypercomputer/accelerator-agents/tree/main/MaxCode and includes the configuration settings.**

I. Introduction

PyTorch and Jax are popular deep learning frameworks, but have some fundamental differences such as PyTorch's eager-execution style against Jax's JIT (just in time) compilation leveraging XLA (Accelerated linear algebra). This has created a split in modeling ecosystems with unique advantages to each of the ecosystems. Migration of code from one framework to another is often daunting, creating increasing silos over time. This migration is mainly driven by cost and unique advantages to each accelerator-based ecosystem. Google's Tensor Processing Units (TPUs) offer competitive computation per dollar and highly optimized matrix multiplication speeds. However, moving from PyTorch research code to TPU-optimized JAX code creates a significant engineering challenge. Developers must convert object-oriented, mutable states into JAX's functional, stateless setup using `flax.linen`. They also need to replace standard Python control flow with compiler-friendly structures like `jax.lax.scan`. Additionally, developers must address default discrepancies that affect numerical equivalence, such as different tensor layouts (PyTorch uses NCHW while JAX uses NHWC), default precisions (PyTorch uses fp32 while JAX uses bfloat16), and fundamentally different paddings in attention layers.

Modern Large Language Models (LLMs) have shown remarkable proficiency in generating general-purpose code [6]. However, even with improvements in reasoning approaches like Chain of Thought [8] and Self-Consistency [7], LLMs still struggle with strict framework translation. We believe that LLMs cannot natively handle complex, high-dimensional tensor arithmetic. Without a reliable execution-based verification loop, LLMs tend to hallucinate incorrect mathematical and semantic judgements. To overcome this, we propose a fully automated, agentic pipeline specifically designed for PyTorch-to-JAX translation. We view framework translation as a grounded generation problem. By giving the LLM syntactic anchors for formatting and dynamic execution anchors to define the absolute mathematical ground truth, we create a self-correction loop that ensures that its output aligns with the target system.

The main contributions of this paper are threefold:

- **Structural Anchoring via In-Context Learning:** We demonstrate that giving a language model a small, carefully chosen set of reference translations and tests for ICL effectively anchors the generation process and lowers the risks of hallucination.
- **The Dynamic Execution Oracle:** We introduce a method to work around the arithmetic limitations of LLMs. By running the source PyTorch modules and saving the dynamic computational state to a fixed "Oracle," we align with the shift toward execution-based code evaluation [1] to ensure mathematical fidelity.
- **Ablation Guidance for Agentic Workflows:** We conduct ablation studies that isolate the effects of instructions, ICL, and execution-conditioned self-debugging. By measuring the impact of each part, we offer practical guidance for developing future agentic code-translation systems.

II. Methods

We suggest a step-by-step framework for translating PyTorch to JAX. This framework has two main parts: anchoring through In-Context Learning and refinement using feedback from an oracle-driven self-debugging process. To implement

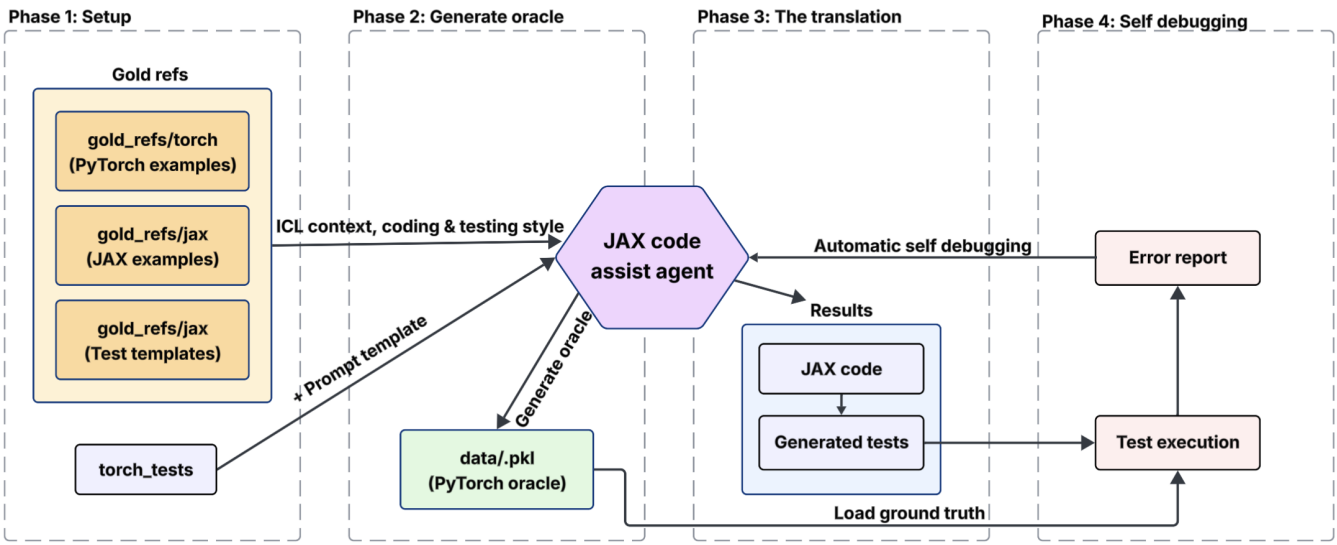

Fig. 1. Block diagram of the workflow

this, we divided the workflow into four automated phases, followed by a manual check that provides the final accuracy number we report. Figure 1 represents the general workflow of our approach. We use Gemini-2.5-pro in all phases whenever the agent needs to talk to an LLM. In the following, we describe each phase in detail.

A. Phase 1: Anchoring Translation via In-Context Learning (ICL)

The main challenge in translating PyTorch to JAX automatically is to move from mutable, object-oriented states to a functional, stateless design. We first tried a Retrieval-Augmented Generation (RAG) pipeline referencing Google's MaxText repository. This approach created significant problems. The fragments retrieved introduced complex issues related to distributed computing, such as hardware sharding. These distractions worsened the model's API hallucinations, illustrating the "Lost in the Middle" phenomenon ([4]). To reduce this repository-specific noise, we used a tightly controlled In-Context Learning (ICL) method. Research shows that large models can apply ICL to overwrite pre-trained patterns and compact demonstrations into task vectors ([3]) that act as structural anchors for clear syntactic formatting ([5]). Instead of a broad retrieval, we found that giving the right boundary and execution feedback greatly boosts accuracy. Thus, our context is limited to a small set of references. This includes the source PyTorch module, the idiomatic JAX translation, and the relevant test cases. This setup provides structural guidance to anchor the functional framework without overwhelming the context window.

B. Phase 2: Dynamic Execution and the PyTorch Oracle

To ensure code functional equivalence, our framework treats the original PyTorch implementation as the unquestioned ground truth. We assume the source module is mathematically correct and aim to mirror its functionality. Relying on an LLM to read code and predict high-dimensional tensor arithmetic can cause critical mistakes. Models might produce incorrect results for complex mathematical operations and intermediate shapes ([2]).

Instead of heuristics , we direct the agent to create a profiling script for dynamic execution. This script compiles the PyTorch module, passes random input tensors through the forward pass, and extracts the computational state. This state includes the initialized weights ($state_{dict}$), input tensors, and output activations. We serialize this state into an unchangeable artifact, which we call our "Oracle." Importantly, running the code on the PyTorch backend reveals behaviors that static analysis misses. These behaviors include silent padding methods and shape changes during execution. By basing our evaluation on this immutable Oracle, we avoid the limitations of the LLM's arithmetic. This ensures that the downstream JAX translation is validated against real computation instead of theoretical estimations.

C. Phase 3: Agentic Translation and Oracle-Conditioned Test Generation

Grounded by the structural anchors from Phase 1, the agent translates the source PyTorch modules into standard flax.linen abstractions. Constrained by the selected ICL examples, the model carefully navigates the shift in framework, producing functional forward passes. However, simply generating syntactically correct code is not enough; verifying semantic equivalence is still the main challenge. To automate this verification, we direct the agent to create custom test harnesses based directly on the Phase 2 Oracle and Phase 1 ICL examples.

The evaluation follows a hierarchical three-stage process. First, the script checks if the code can compile, acting as a strict pass/fail test. It ensures the JAX module initializes and traces correctly without runtime errors. Second, it checks for

TABLE I
ABLATION STUDY RESULTS ACROSS COMPLEXITY LEVELS

Configuration	Detailed Instruction Prompts	ICL Anchor	Iterative Debugging	Level 1 (Mathematical Ops)			Level 2 (Neural Modules)		
				Comp.	Shape	Num. Eq.	Comp.	Shape	Num. Eq.
Baseline Only	×	×	×	44%	44%	44%	18%	18%	9%
Instruction Only	✓	×	×	67%	67%	44%	36%	36%	18%
Instruction + Self-Debugging	✓	×	✓	89%	89%	89%	73%	55%	27%
Full Pipeline (Ours)	✓	✓	✓	**100%**	**100%**	**100%**	**91%**	**91%**	**91%**

shape correctness (pass/fail), confirming dimensional consistency and addressing framework layout changes (for example, mapping PyTorch's NCHW to JAX's NHWC). Finally, the script programmatically deserializes the .pkl files, maps the PyTorch weights into the JAX model, and checks for numerical equivalence. For this last stage, we set a strict absolute error limit of 10^{-7} using numpy.allclose. This specific tolerance separates structural translation errors from the natural limits of standard 32-bit floating-point numbers (Higham, 2002), resulting in a reliable testing harness we refer to as the "Silver" test cases.

D. Phase 4: Iterative Self-Debugging and Silver Reference Validation

Initial zero-shot translations sometimes fail due to JIT compilation errors, unhandled PRNG routing, or strict numpy.allclose mismatches. To prevent these issues, we embed structural instructions, such as PRNG key splitting constraints, into the generation prompt. However, static instructions alone do not catch every edge case. An iterative self-debugging loop can effectively manage the challenges of cross-framework migration. When a generated module fails the Oracle-conditioned test, the execution feedback, including standard error, stack traces, and precise numerical differences, are fed directly to the agent's context. With this objective feedback and guided by the ICL templates, the LLM addresses its initial errors and refines the code from a broken state into a reliable, mathematically verified JAX module.

While this automated test generation and debugging loop drives our system's capability, we must protect our final metrics from reward hacking, such as the agent creating trivial tests to pass. To validate our approach scientifically, we separate the debugging capabilities from the final evaluation benchmark. The translation accuracies presented in this work are measured only against a manual evaluation suite. Furthermore, the modules used for ICL do not appear in this evaluation set. This strict separation ensures our reported metrics reflect true, uncontaminated generalization, demonstrating that our grounded, agent-driven pipeline effectively addresses the framework translation bottleneck.

III. BENCHMARK

To quantify the impact of our architectural anchors, we evaluated four pipeline configurations on a holdout set of human-crafted tests. To rigorously test scaling complexity, we categorize our evaluation into three difficulty tiers. **Level 1** comprises 9 fundamental mathematical and training operations (*data mixup, early stopping, gradient clipping, positional encoding, moving average, and dice, focal, label smoothing, and tversky losses*). **Level 2** evaluates 11 architectural abstractions and neural modules (*MLPs, VAEs, GANs, GRUs, LSTMs, attention mechanisms, transformer encoders, temporal convolutional units, graph convolutions, and ResNet blocks*). **Level 3** evaluates migration of an entire repository (given as a Github or HuggingFace link) to JAX. Since it is complex to evaluate Level 3 automatically, we leverage human experts to evaluate for this task. Ten repositories were included for level 3 which include Code-Whisper, Multi-modal transformers (MMTF), simple-moe, time-series forecasting model, FinGPT model (finance GPT), StyleGAN, T5 transformer, two-tower recommendation model, SAM (segment anything) model, detr-resnet50.

For level1 and level 2, we measure structural validity (compilation success), layout consistency (shape correctness), and mathematical fidelity. Given a source PyTorch module f_{pt}, a generated JAX module f_{jax}, identically initialized and mapped weights (W_{pt}, W_{jax}), and an input tensor $X \sim \mathcal{D}$, numerical equivalence is formally defined as:

$$\max |f_{pt}(X, W_{pt}) - f_{jax}(X, W_{jax})| < \epsilon$$

where $\epsilon = 1 \times 10^{-7}$ accounts for standard floating-point disparities between backend compilers.

For level 3, since the evaluations are by human evaluators, we only report the results of the full pipeline (ours). The completeness of modules/files converted and the numerical equivalence are reported. In addition, they also report an average readability score on a Likert scale of 1-5 (averaged across all the files in a repository). Since readability is subjective, we only report the scores without claims of broader implications.

IV. RESULTS AND ABLATION STUDY

Table I summarizes results comparing our pipeline to the baselines, instructions and the self-debugging loops. While self debugging significantly improved performance over the instruction only prompts, there was still a wide gap for usability of the code. Table II summarizes results for the level 3 repository migration dataset.

Taxonomy of Translation Failures: Across the evaluation levels, zero-shot and static-instruction baselines showed three main types of failures. First, Syntactic Hallucinations: the LLM often did not generate the necessary wrapper classes, causing frequent `NameErrors` or missing `init` attributes. Second, API Signature Mismatches: the model mistakenly applied PyTorch-specific kwargs, such as passing `train=True` directly into a `__call__` method). Finally, State Mapping

Discrepancies: even when modules compiled, even when modules compiled, complicated structural translations led to `KeyErrors` during the precise weight-transfer phase like mismatched `Dense` layers in attention blocks. This highlighted the need for a self debugging loop.

TABLE II
LEVEL3 HUMAN EVALUATION ACROSS 10 GITHUB REPOSITORIES

Repository	Completeness	Num. Eq.	Readability
Code Whisper	100%	100%	4
Multimodal transformer	100%	100%	4.3
MOE	100%	93.3%	4.5
Time-series forecasting	100%	96.5%	4.8
FinGPT	100%	86%	4.9
StyleGAN2	100%	96%	4.4
T5	100%	100%	3.75
Two-tower	100%	100%	4.9
Segment Anything (SAM2)	86%	80%	4.8
DETR	100%	57%	3.8

- **Baseline:** Simple prompting predictably fails due to the previously mentioned syntactic hallucinations. The LLM creates PyTorch-like mutable states in JAX syntax, resulting in immediate compilation failures and almost no equivalence.
- **Instruction Only:** Providing detailed translation prompts, such as clear rules on PRNG routing and `flax.linen` requirements, helps reduce basic `NameErrors` and slightly improves compilation rates. However, relying only on written instructions does not guarantee all the necessary structural understanding. This approach works partially for Level 1 tasks, which are mainly stateless and need only localized tensor manipulations, but it was not that helpful for Level 2 modules. The model has trouble with complex state management, hierarchical initialization, and hidden dimensional shifts needed for neural architectures. As a result, mathematical accuracy stalls at 44% for Level 1 and only 18% for Level 2.
- **Detailed Prompt Instruction + Self Debugging:** Adding a compiler-driven feedback loop without the Oracle creates a false sense of success. The agent fixes syntactical errors, increasing the compilation rate. However, without a clear mathematical truth, the LLM resorts to reward hacking to achieve a compilation pass, which leaves numerical equivalence disproportionately low, especially on complex Level 2 architectures.
- **Full Pipeline (Ours):** The complete framework shows that structural anchoring must be combined with empirical truth. By requiring the generated test harnesses to physically load the serialized Oracle '.pkl' data, the debugging loop must iterate on its architecture and weight-mapping logic until it meets the strict mathematical condition ($< \epsilon$). This leads to the highest overall fidelity, achieving 100% fidelity on Level 1 and 91% on Level 2, demonstrating the need for execution-backed constraints.

V. CONCLUSION

Translating deep learning research from PyTorch to optimized JAX environments remains a persistent challenge. We demonstrate that while LLMs natively struggle with this shift, they are highly capable code generators when given clear rules and good references. By combining structural In-Context Learning anchors with dynamic feedback from an oracle, our fully automated system can handle the complexities of functional state management and hidden numerical issues. While our evaluation shows high fidelity on independent modules, we acknowledge two main limitations: our method has not yet been expanded to fully automatic repository level migrations, which will need complete structural dependency management, and we have not systematically explored how changing the specific examples used in the ICL affects results. Still, this grounded, agentic approach eliminates the need for manual test and offers a scalable blueprint for high-fidelity cross-framework migration.

REFERENCES

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] N. Dziri, X. Lu, M. Sclar, X. L. Li, J. Li, B. Y. Lin, P. West, C. Bhagavatula, R. Le Bras, J. D. Hwang *et al.*, "Faith and fate: Limits of transformers on compositionality," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[3] R. Hendel, M. Geva, and A. Globerson, "In-context learning creates task vectors," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 9318–9333.

[4] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the association for computational linguistics*, vol. 12, pp. 157–173, 2024.

[5] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 11 048–11 064.

[6] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[7] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[8] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.