T2J-BENCH: BENCHMARKING CODING AGENTS ON EXECUTION-GROUNDED PYTORCH-TO-JAX REPOSITORY MIGRATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Migrating a machine-learning codebase across frameworks—for example from PyTorch to JAX—is a routine but costly engineering task that is fundamentally *cross-paradigm*: eager, stateful, object-oriented modules must be rewritten as pure, jittable functions over an explicit parameter pytree. We ask whether coding agents can perform such whole-repository migrations *correctly*, and argue that existing code benchmarks, which grade unit-test pass rates or surface-level similarity to a reference translation, cannot answer this: they neither require the migrated system to *behave* the same nor prevent a JAX reimplementation from faking equivalence by secretly importing PyTorch itself. We introduce T2J-BENCH, a benchmark of 63 whole-repository PyTorch→JAX migration tasks spanning 18 model families across both training and serving, where torch-free checkpoint conversion is itself part of the deliverable. T2J-BENCH grades *execution-grounded functional equivalence*: each candidate is *run* inside a separate, `torch`-free, offline verifier and scored against references frozen from the PyTorch original, under a hard independence gate and a continuous, threshold-free reward. Across twenty off-the-shelf agent–model configurations, the strongest recovers only 0.676 of the reward against a 0.980 oracle, with substantial numeric and behavioral divergence despite high spec pass rates. Finally, we contribute TWINHARNESS, a reference-grounded agent harness that turns the agent's executable PyTorch source into legitimate self-verification feedback; applied to an `opus-5` agent it raises the reward from 0.649 to 0.723—the highest of any configuration—using only task-visible resources. We will release the benchmark to support reproducible study of cross-paradigm ML migration.

1 INTRODUCTION

Moving model repositories across the fragmented machine-learning ecosystem is routine but costly. Teams migrate PyTorch (Paszke et al., 2019) codebases to JAX (Bradbury et al., 2018; Heek et al., 2026) to reach new accelerators, reuse published models in different research stacks, or build functional serving engines. This *whole-repository* job spans model definitions, training or serving loops, data pipelines, and checkpoint formats. Repository-editing coding agents are plausible tools for this migration, but can they produce a JAX reimplementation that both runs and *behaves the same* as the original?

General-purpose code benchmarks do not fully capture this setting. They grade functional correctness through unit-test pass rates (Jimenez et al., 2024b; Chen et al., 2021) or, for transpilation, through surface-level similarity to a reference translation (Lu et al., 2021; Ren et al., 2020). Neither captures two key difficulties. First, migration is *cross-paradigm*: eager, stateful, object-oriented PyTorch modules must be re-expressed as pure, jittable functions over an explicit parameter pytree in `flax.linen`/`optax` or `flax.nnx` — a structural rewrite of the program's execution model, not a line-by-line transliteration. Second, "correct" means the migrated system reproduces the original's *execution* — its forward numerics, its gradients, its optimizer trajectory, the generalization of a model it trains, and the decode behavior of a serving engine — to floating-point tolerances, properties that aggregate test pass rates alone do not characterize. A JAX reimplementation can also fake equivalence by importing PyTorch, so evaluation must establish *independence*. Recent work directly addresses

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Representative benchmarks and systems. *Whole-repo*: full-repository tasks; *Cross-parad.*: eager/stateful-to-functional migration; *Exec-ground.*: execution-based evaluation; *Behav. equiv.*: reference-based training or serving behavior beyond isolated outputs; *Indep. gate*: source-framework exclusion during grading. A cross marks a criterion not targeted or explicitly enforced. Liang et al. (2026) and JAXBench (Tschand et al., 2026) evaluate output-level equivalence without systematic trajectory or serving-state evaluation; Liang et al.'s repository results are human-assessed.

Benchmark / system	Whole-repo	Cross-parad.	Exec-ground.	Behav. equiv.	Indep. gate
HumanEval (Chen et al., 2021)	✗	✗	✓	✗	✗
CodeXGLUE (Lu et al., 2021; Ren et al., 2020)	✗	✗	✗	✗	✗
SWE-bench (Jimenez et al., 2024a)	✓	✗	✓	✗	✗
MigrationBench (Liu et al., 2025d; Wang et al., 2024)	✓	✗	✓	✗	✗
Agentic migration (Liang et al., 2026)	✓	✓	✓	✗	✗
JAXBench (Tschand et al., 2026)	✗	✗	✓	✗	✗
T2J-BENCH (ours)	✓	✓	✓	✓	✓

adjacent settings. Liang et al. (2026) combine in-context examples, source-execution oracles, and self-debugging for PyTorch-to-JAX migration, including human-assessed repository conversions. JAXBench (Tschand et al., 2026) evaluates correctness and speedup of TPU kernels generated from JAX workloads. These efforts motivate a complementary benchmark of automatically graded whole-repository migration, including training and serving behavior under an explicit source-framework independence gate. Table 1 summarizes these distinctions.

We introduce T2J-BENCH: 63 whole-repository tasks requiring coding agents to convert real, self-contained PyTorch ML repositories into native, `torch`-free JAX/Flax repositories that preserve functional behavior (§3). Tasks cover training migration (SFT, DPO, and PPO across 15 model families) and serving migration (vLLM-style paged-attention engines), including weight conversion. Grading is based on *execution-grounded functional equivalence* (§3.2): every candidate is *run* inside a separate verifier with `torch` removed and network access disabled, and compared against references recorded from the original. Independence is a hard multiplicative gate — a JAX reimplementation that relies on PyTorch earns nothing — while numeric and behavioral parity are scored with continuous, threshold-free similarity measures. Thus, T2J-BENCH rewards equivalent *behavior*, not surface resemblance.

Our baseline study shows this setting is far from saturated (§5): across twenty baseline agent–model configurations evaluated with three random seeds over the full suite, the strongest off-the-shelf configuration reaches a reward of only 0.676 against a 0.980 faithful JAX reimplementation ceiling — a gap of 0.304. High spec pass rates do not ensure numerical or behavioral fidelity: strong agents can produce structurally valid repositories whose forward numerics, gradients, or training dynamics diverge from the original. The reward–cost analysis further shows that continued baseline refinement remains below TWINHARNESS at similar plotted API expenditure (§5.3). These results identify a deficiency: standard harnesses rarely turn the executable source implementation into structured feedback on behavioral preservation. Our *reference-grounded migration harness*, TWINHARNESS, treats the task-visible PyTorch repository as an executable specification within an iterative *diagnose–migrate–verify* loop (§4): reference-grounded diagnostics localize discrepancies before further edits, while an explicit information boundary keeps the harness from accessing hidden tests, frozen references, verifier code, or benchmark rewards — converting *reference availability* into *reference-grounded feedback* without weakening the evaluation contract.

This additional feedback substantially improves migration quality. With the underlying agent–model configuration held fixed, TWINHARNESS raises the reward from 0.649 to 0.723, an 11.4% relative improvement over that off-the-shelf scaffold and the highest reward of any configuration (§5). These gains suggest that a meaningful part of the remaining difficulty concerns how systematically agents use the available executable behavioral specification, alongside model capability.

Our contributions are:

- **A cross-paradigm, whole-repository migration benchmark.** We introduce T2J-BENCH, comprising 63 PyTorch→JAX migration tasks spanning 18 model families across training

and serving, with native checkpoint conversion included as part of the migration contract (§3).

- **An execution-grounded functional-equivalence evaluation.** We evaluate migrated repositories by executing them in a separate `torch`-free, offline verifier against references recorded from the original PyTorch repository, combining an independence gate with continuous numeric and behavioral similarity (§3.2).
- **TWINHARNESS, a reference-grounded migration harness.** We introduce a diagnose–migrate–verify loop that turns the task-visible PyTorch repository into legitimate development-time feedback while maintaining a strict separation from hidden benchmark state, improving reward from 0.649 to 0.723 (§4; §5).

2 RELATED WORK

LLM-Based Code Conversion Benchmarks. MigrationBench (Liu et al., 2025d) and RepoTrans-Bench (Wang et al., 2024) evaluate repository-level migration and codebase translation; §B surveys related work. These benchmarks largely target API, language, version, or general codebase conversion through reference similarity (Lu et al., 2021; Ren et al., 2020) or small-program execution. Beyond surface- or snippet-level evaluation, T2J-BENCH measures execution-grounded functional equivalence after cross-paradigm migration of whole ML repositories across SFT, DPO, PPO, and paged-attention serving against frozen PyTorch references, with a hard independence gate.

Agentic Framework Migration. Liang et al. (2026) combine curated in-context translations, immutable PyTorch execution oracles, and iterative self-debugging, evaluating operators, neural modules, and ten human-assessed repositories with limited intervention. Building on this execution-grounded migration work, T2J-BENCH standardizes automated evaluation of complete training and serving repositories through continuous numeric and behavioral scores and a separate independence gate (Table 1). TWINHARNESS organizes source-execution feedback into diagnosis, migration, and verification while excluding hidden benchmark state.

Reference-Grounded Feedback and Self-Repair. TWINHARNESS builds on feedback-guided code repair. Intrinsic self-correction and self-verification remain unreliable (Huang et al., 2024; Hong et al., 2024); external signals support execution-trace self-debugging (Chen et al., 2024), tool-interactive critique (Gou et al., 2024), and multi-agent review (Qian et al., 2024). Claude Code (Anthropic, 2026) and Codex (OpenAI, 2025) implement similar diagnose-and-repair loops. TWINHARNESS grounds cross-framework diagnosis, migration, and verification in task-visible PyTorch behavior, preserving the evaluation boundary by excluding frozen references, hidden tests, verifier code, and rewards.

3 T2J-BENCH

T2J-BENCH comprises 63 whole-repository tasks that re-express eager, stateful PyTorch modules as pure, compilable JAX/Flax functions (Bradbury et al., 2018; Heek et al., 2026; DeepMind et al., 2020) for *training* or *inference (serving)* migration. Candidates are graded by *execution-grounded functional equivalence* against recorded PyTorch references (Fig. 1).

3.1 THE MIGRATION TASK

Each task provides a self-contained, pure-PyTorch source tree $\mathcal{P}$ (roughly 15–20 modules from a real codebase) and the raw, publicly released checkpoint. The agent must produce a native JAX reimplementation $\mathcal{J}$ that mirrors $\mathcal{P}$'s file layout and public interface and writes its own PyTorch-free checkpoint-conversion routine — *weight conversion is part of the deliverable*. The hard *independence* constraint prohibits PyTorch use anywhere, even offline; grading enforces it in a separate, offline, PyTorch-free environment. The *training* class is a 15×3 grid of 15 model families crossed with SFT, DPO (Rafailov et al., 2023), and PPO (Schulman et al., 2017), giving 45 LoRA (Hu et al., 2021) fine-tuning tasks (24 text, 21 vision); the *inference* class is a 6×3 grid of six families crossed with three paged-attention engine modes of increasing scheduling complexity (single-sequence, ragged-batch, chunked-prefill), giving 18 serving tasks (15 text, 3 vision). Full family lists, source repositories, per-task requirements, and modality details are in §C.1.

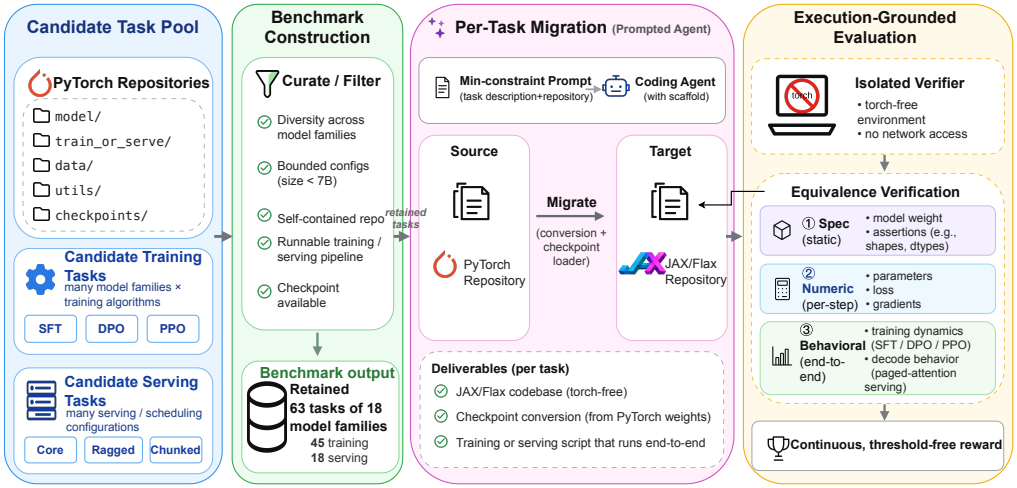


Figure 1: T2J-BENCH overview: 63 PyTorch-to-JAX repository migrations across 18 model families, covering 45 training and 18 paged-attention serving tasks. Agents deliver native, `torch`-free JAX/Flax code, checkpoint conversion, and executable training or serving paths. An offline verifier combines SPEC, NUMERIC, and BEHAVIORAL checks into a continuous, threshold-free reward.

Table 2: The five grading challenges, the principle that answers each, and the mechanism that realizes it in the grader. The challenges and principles are stated in full in §C.2 and §C.3.

	Challenge	Principle	Mechanism in the grader
C1 / P1	Fabricated equivalence	Independence-first	The grader runs offline with `torch` removed, so any hidden PyTorch import fails loudly; independence is a multiplicative gate that zeros the whole score
C2 / P2	"Looks right" ≠ "behaves right"	Execution-grounding	Every equivalence claim is settled by *running* the candidate on fixed inputs — weight conversion, forward, backward, and optimizer loop, or prefill and paged decode — never by reading its source
C3 / P3	Uncertain ground truth	Oracle provenance	References are recorded once from the pure-PyTorch original; the reference JAX solution is graded as a candidate, never used as the ground truth
C4 / P4	No clean pass/fail	Bounded, threshold-free metrics	Each numeric and behavioral check returns a continuous $[0, 1]$ similarity from an intrinsically bounded metric, with no tuned pass/fail cutoff
C5 / P5	Performance is part of correctness	Runtime as a graded dimension	The end-to-end levels (**Behavioral**) run under a uniform time budget; a JAX reimplementation that overruns it scores 0 on those levels

3.2 GRADING DESIGN AND MACHINERY

Correctness requires $\mathcal{J}$ to reproduce $\mathcal{P}$'s interface and behavior on the same inputs: forward passes, gradients, optimizer trajectories, and trained-model generalization for training; prefill/decode logits, sampling, and multi-request paged-cache behavior for serving. This *functional equivalence*, beyond test-pass counts, motivates the five challenge–principle pairs in Table 2 (Appendix C.2–C.3).

The pipeline (§C.4–§C.6) grades seven class-specific levels (L0–L6, the *lattice*): independence and structure (L0); module/operator and composed-forward numerics across input shapes (L1–L3); loss and gradients (training) or step-wise decoding (serving) at L4; and end-to-end execution and held-out behavior of trained artifacts or live decode engines (L5–L6). Each level requires the preceding levels; scores reflect the level reached rather than a single pass/fail verdict.

The levels feed three reward dimensions: L0 is **Spec** (structure, interface, and independence; pass/fail), the single-pass levels L1–L4 feed **Numeric**, and the whole-run levels L5–L6 feed **Behavioral** (L4's step-wise checks also contribute to **Behavioral**). The dimensions combine into a spec-gated,

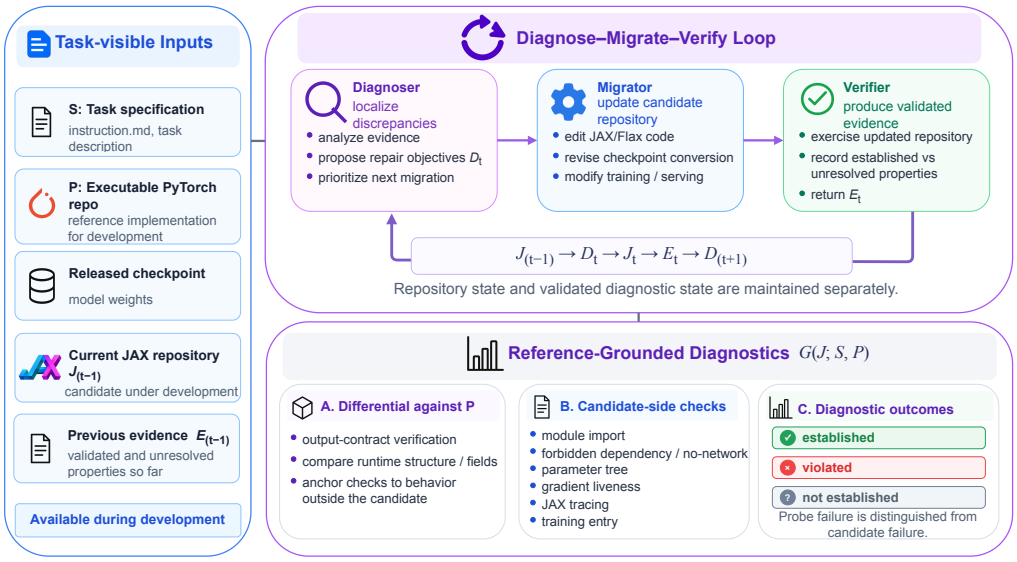


Figure 2: TWINHARNESS's diagnose–migrate–verify loop uses only task-visible artifacts. Diagnostics $\mathcal{G}(\mathcal{J}; \mathcal{S}, \mathcal{P})$ combine differential checks against the executable PyTorch source with candidate-side checks, labeling evidence as *established*, *violated*, or *not established*.

dimension-weighted reward:

$$R = \underbrace{s_{\text{spec}}}_{\text{multiplicative gate}} \times \left(w_{\text{num}}\, s_{\text{num}} + w_{\text{beh}}\, s_{\text{beh}}\right), \qquad w_{\text{num}} = w_{\text{beh}} = 0.5. \tag{1}$$

where $R, s_{\text{spec}}, s_{\text{num}}, s_{\text{beh}} \in [0, 1]$: s_{spec} is the fraction of **Spec** checks passed, and $s_{\text{num}}, s_{\text{beh}}$ are the mean per-check similarity in the **Numeric** and **Behavioral** dimensions. The multiplicative Spec gate gives zero credit to reimplementations that are not genuine, correctly shaped, and PyTorch-free, regardless of output similarity. Similarity measures are intrinsically bounded and threshold-free, calibrated to the ≈ 0.999 floating-point noise ceiling. Any crash, PyTorch leak, NaN, or end-to-end run exceeding its wall-clock budget (roughly twice the original's runtime) scores 0 on the affected check (§C.4, §C.5, §C.6).

3.3 BENCHMARK VALIDITY AND SCOPE

We validate each task using its author-written, known-good *reference JAX solution*, graded as a candidate, not ground truth. The full-pipeline *oracle solvability check* attains 0.980 overall reward and ≈ 0.999 per-pass similarity, establishing task solvability and reward calibration. A *seeded-defect* sweep confirms failure on broken code. High reward covers only graded properties, excluding fully-attended multimodal numerics, exact token-by-token generation, and cross-hardware portability. Both safeguards and the full non-goals list appear in §C.7.

4 TWINHARNESS: A REFERENCE-GROUNDED HARNESS

TWINHARNESS grounds JAX migration in executable PyTorch behavior through a *diagnose–migrate–verify* loop (Figure 2). It combines source comparisons with candidate-side checks and maintains repository state separately from diagnostic evidence.

4.1 REFERENCE-GROUNDED MIGRATION LOOP

Let $\mathcal{S}$ be the task specification, $\mathcal{P}$ the executable PyTorch repository, and $\mathcal{J}_t$ the candidate JAX repository after iteration t. At each iteration, the *Diagnoser* uses the current repository and prior evidence to identify remaining migration gaps D_t; the *Migrator* updates the candidate; and the *Verifier* executes the result to produce validated evidence E_t, following $\mathcal{J}_{t-1} \rightarrow D_t \rightarrow \mathcal{J}_t \rightarrow E_t \rightarrow D_{t+1}$.

The loop separates implementation state $\mathcal{J}_t$ from evidence E_t of established, violated, or unresolved properties, avoiding diagnostic reconstruction solely from code or interaction history. Unlike open-ended harness evolution, iteration reduces discrepancies between $\mathcal{P}$ and $\mathcal{J}_t$ against fixed, executable source behavior. Stage definitions appear in Appendix D.1.

4.2 Reference-Grounded Self-Verification

The development report $\mathcal{G}(\mathcal{J}; \mathcal{S}, \mathcal{P})$ records runtime observations, structural evidence, and failure traces from task-visible execution, not benchmark scores. Frozen PyTorch outputs belong exclusively to grading (§3.2); development executes $\mathcal{P}$ itself. Matched probes compare $\mathcal{P}$ and $\mathcal{J}$'s output structure, exposed fields, and behavior, anchoring verification outside the candidate to avoid circular self-validation. Candidate-side checks cover executability, independence, parameter structure, gradient liveness, JAX tracing, and integrated training or serving. Together these checks localize divergence from repository validity through runtime interfaces to end-to-end behavior. Outcomes are *established*, *violated*, or NOT ESTABLISHED, separating inconclusive probes from candidate errors (Appendix D.2).

5 Experiments

5.1 Setup

Agents. We evaluate twenty combinations of five coding-agent scaffolds and their natively supported frontier models: *claude-code* (Anthropic's CLI agent) with `claude-opus-5` and `claude-sonnet-5`; *OpenHands*, a model-agnostic scaffold, with all six models; *Gemini-CLI* (Google's CLI agent) with `gemini-3.1-pro`, `gemini-3.7-flash`, and `gemini-3.8-flash`; *OpenCode*, an open-source CLI agent (OpenCode Team, 2025), with all six models; and *Terminus-2*, Harbor's Terminus agent (Harbor Framework Team, 2026), with the three Gemini models. For compactness, Appendix A.3 (Table 6) reports OpenHands and Terminus-2; Table 3 reports the remaining eleven off-the-shelf configurations, TwinHarness, and the oracle. Each agent receives only `instruction.md`, the pure-PyTorch source `torch_orig/`, and the raw checkpoint (§3); it works in a networked container with one NVIDIA H100, 24 CPU cores, and 200 GiB of RAM, under a 3-hour wall-clock budget per task, and emits `jax_repo/`.

Grading and protocol. Every candidate is graded in the separate, no-network, `torch`-free verifier on a single pinned H100 (§3). We evaluate every experimental setting over the full 63-task suite with *three random seeds*, including all baseline and TwinHarness configurations at each reported refinement budget. Results are seed averages; reported standard deviations are sample standard deviations across seeds. The unmodified reference JAX solution serves as the *oracle* candidate (§3.3), defining the faithful reimplementation ceiling.

Metrics. The primary metric is the continuous, spec-gated **reward** $R \in [0, 1]$ of Eq. 1, averaged over tasks. The reference JAX solution's aggregate reward of 0.980 differs from the approximately 0.999 similarity in individual floating-point comparisons. Partial agreement earns partial credit subject to the spec gate. Three component scores localize failures: the **spec** pass rate (structural and independence correctness, the multiplicative gate), the **numeric** score (mean single-forward/backward similarity), and the **behavior** score (mean training-dynamics and invariant similarity). The leaderboard reports three-seed means; higher reward is better, while component scores are diagnostics, not separately gated targets.

5.2 Results

Higher reward, with substantial headroom. TwinHarness leads five of six columns in Table 3 in reported mean, including reward; OpenCode with `sonnet-5` leads inference reward (0.677 vs. 0.622). TwinHarness exceeds the strongest off-the-shelf configuration, OpenCode with `opus-5`, by 0.047 (7.0%), from 0.676 to 0.723, and improves on Claude-Code with the same model by 0.074 (11.4%) across spec ($+0.007$), numeric ($+0.112$), behavior ($+0.064$), training ($+0.039$), and inference ($+0.165$). The gap to the 0.980 oracle remains 0.257.

Table 3: Leaderboard on 63 tasks. R is reward (Eq. 1); SPEC/NUM./BEH. are full-suite dimension scores; TRAIN/INFER. are rewards on 45 training/18 inference tasks. All agent values, including TWINHARNESS, are means over three random seeds; R subscripts give sample standard deviations. The oracle is the reference JAX solution. Bold/underline mark the two highest distinct means per column, excluding the oracle; ties share the same mark. Additional baselines (OpenHands and Terminus-2) are reported in Appendix A.3.

Harness	Model	R	SPEC	NUM.	BEH.	TRAIN	INFER.
Claude-Code	opus-5	$0.649_{\pm.008}$	0.924	0.618	0.728	0.725	0.457
Claude-Code	sonnet-5	$0.408_{\pm.032}$	0.732	0.424	0.479	0.329	0.605
OpenCode	opus-5	$\underline{0.676}_{\pm.014}$	<u>0.927</u>	<u>0.670</u>	<u>0.741</u>	<u>0.747</u>	0.499
OpenCode	gemini-3.7-flash	$0.526_{\pm.033}$	0.883	0.592	0.535	0.595	0.351
OpenCode	sonnet-5	$0.472_{\pm.042}$	0.760	0.467	0.554	0.390	**0.677**
OpenCode	glm-5.2	$0.470_{\pm.051}$	0.732	0.495	0.544	0.491	0.418
OpenCode	gemini-3.8-flash	$0.361_{\pm.013}$	0.633	0.423	0.381	0.448	0.144
OpenCode	gemini-3.1-pro	$0.064_{\pm.023}$	0.355	0.069	0.083	0.034	0.139
Gemini-CLI	gemini-3.7-flash	$0.102_{\pm.020}$	0.346	0.120	0.111	0.081	0.155
Gemini-CLI	gemini-3.8-flash	$0.087_{\pm.012}$	0.324	0.100	0.102	0.069	0.131
Gemini-CLI	gemini-3.1-pro	$0.057_{\pm.032}$	0.345	0.066	0.068	0.027	0.133
TWINHARNESS	opus-5	$\mathbf{0.723}_{\pm.022}$	**0.931**	**0.730**	**0.792**	**0.764**	<u>0.622</u>
Oracle		0.980	1.000	0.998	0.961	0.973	0.997

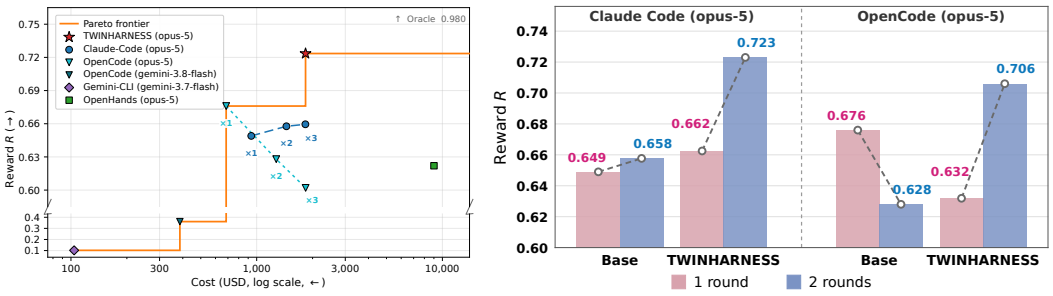
Table 4: Reward by training method (15 tasks each) and inference mode (6 each). Bold/underline mark the two highest distinct means per column, excluding the oracle; ties share the same mark.

		Training method			Inference mode		
Harness	Model	SFT	DPO	PPO	core	ragged	chunked
Claude-Code	opus-5	**0.854**	0.784	0.538	0.489	0.472	0.409
Claude-Code	sonnet-5	0.483	0.289	0.215	<u>0.626</u>	**0.604**	<u>0.585</u>
OpenCode	opus-5	0.807	<u>0.839</u>	<u>0.597</u>	**0.742**	0.344	0.410
OpenCode	gemini-3.8-flash	0.438	0.484	0.422	0.168	0.085	0.179
TWINHARNESS	opus-5	<u>0.821</u>	**0.851**	**0.620**	0.623	<u>0.517</u>	**0.725**
Oracle		0.966	0.965	0.988	0.999	0.996	0.995

Scaffold choice affects migration quality. With `opus-5`, OpenCode leads Claude-Code (0.676 vs. 0.649), and with `gemini-3.8-flash` it leads Gemini-CLI (0.361 vs. 0.087). Gemini-CLI's low spec scores (0.324–0.346) indicate frequent structural or independence failures. Section 5.3 examines the corresponding cost trade-offs.

Performance depends on both model and scaffold. `gemini-3.7-flash` outperforms `gemini-3.1-pro` under both Gemini-CLI and OpenCode, and model rankings depend on the task class: with Claude-Code, `opus-5` leads `sonnet-5` on training reward (0.725 vs. 0.329), whereas `sonnet-5` leads on inference (0.605 vs. 0.457). Performance thus depends on configuration; these comparisons do not separate reasoning from tool-use effects.

Gains vary across task subtypes. Table 4 shows TWINHARNESS leading on DPO, PPO, and chunked inference, while Claude-Code retains the highest SFT (`opus-5`) and ragged (`sonnet-5`) scores and OpenCode with `opus-5` leads on core inference (0.742 vs. 0.623). Relative to OpenCode with `opus-5`, TWINHARNESS is higher on every subtype except core inference; relative to Claude-Code with `opus-5`, it improves five of six subtypes, most on chunked inference (from 0.409 to 0.725), but SFT decreases from 0.854 to 0.821. PPO remains the weakest training subtype for every configuration despite its 0.988 oracle score, exposing a persistent gap in reproducing reinforcement-learning dynamics.



(a) Reward–cost trade-offs.

(b) One versus two refinement rounds.

Figure 3: **Cost and iteration trade-offs on T2J-BENCH.** Rewards are means over three random seeds per setting. **(a)** Reward versus API cost per 63-task run. Connected Claude-Code/OpenCode points use `opus-5`; $\times 1$ is the initial run, $\times 2$/$\times 3$ add one/two continuations. Red star: TWINHARNESS with `opus-5`; orange staircase: best observed reward within each budget. Cost is logarithmic; the reward axis is broken, with the 0.980 oracle above range. **(b)** One-round (pink) and two-round (blue) rewards with `opus-5` on Claude-Code/OpenCode. Base uses ordinary continuation; TWINHARNESS uses diagnose–migrate–verify. Dashed lines connect each method's own trajectory.

Takeaways. TWINHARNESS improves aggregate training and inference reward, but its subtype gains are uneven and numeric and behavioral fidelity remain below the reference ceiling. We next examine these gains across costs, agent configurations, and refinement rounds.

5.3 REWARD–COST TRADE-OFFS UNDER CONTINUED REFINEMENT

Figure 3a compares reward and API cost per 63-task run, including Claude-Code and OpenCode continuations with `opus-5`. Markers $\times 1$, $\times 2$, and $\times 3$ denote the initial run and one or two added continuations, respectively, not the three random seeds used for statistical averaging or token-budget multipliers. Baselines continue without TWINHARNESS's structured reference-grounded protocol.

A favorable trade-off beyond longer continuation. Initial OpenCode (0.676) has lower plotted API cost than TWINHARNESS (0.723); both are non-dominated in Figure 3a. At an API cost close to that of TWINHARNESS, two additional baseline continuations yield approximately 0.66 for Claude-Code and 0.60 for OpenCode, leaving reward gaps of about 0.06 and 0.12, respectively. Continuation does not close the gap, while initial OpenCode remains competitive at lower cost.

More refinement does not guarantee better migrations. Claude-Code improves modestly, whereas OpenCode falls from 0.676 to 0.628 after one continuation and to about 0.60 after two, despite increasing API cost. Continued editing can thus degrade candidates, motivating structured verification feedback. This does not isolate token-limit effects: continuation changes both edits and computation.

5.4 ABLATIONS OF AGENT CONFIGURATION AND ITERATION BUDGET

Applicability across scaffolds and model backbones. Across four matched scaffold–model pairs, TWINHARNESS improves reward by 0.059–0.130 (Appendix A.1, Figure 4). This supports applicability across the evaluated T2J-BENCH configurations, not transfer to unseen task distributions.

Structured iteration versus ordinary continuation. Figure 3b compares one- and two-round rewards for base agents and TWINHARNESS, fixing `opus-5` across Claude-Code and OpenCode. Base uses ordinary continuation; TWINHARNESS uses diagnose–migrate–verify. With Claude-Code, the base changes from 0.649 to 0.658 ($+0.009$), while TWINHARNESS increases from 0.662 to 0.723 ($+0.061$). With OpenCode, the base decreases from 0.676 to 0.628 (-0.048), while TWINHARNESS increases from 0.632 to 0.706 ($+0.074$). Thus, TWINHARNESS trails the OpenCode base by 0.044 after one round but leads by 0.078 after two, also exceeding the initial base by 0.030. The advantage emerges through iteration, not at every round budget. Larger one-to-two-round gains are consistent

Table 5: Failure taxonomy on the same 63 tasks and verifier. Counts are three-seed means of task attempts with at least one failed root-cause check per category; categories may overlap. Downstream checks blocked by runtime errors are excluded. Δ is TWINHARNESS minus Claude Code; negative values indicate fewer failures.

Category	What goes wrong	TWINHARNESS	Claude Code	Δ
	Interface / admissibility (Spec stage)			
Init failure	Candidate cannot be constructed under real configs	0	0.7	-0.7
Parameter-tree mismatch	State-dict topology drifts from reference	30	28.7	$+1.3$
Batch-schema mismatch	Collator output does not satisfy contract	10	7.7	$+2.3$
	Cross-paradigm runtime errors (Numeric stage)			
JAX/Torch boundary	Type confusion at the JAX↔Torch seam	4	1.7	$+2.3$
Dtype unsupported	Wrapper drops bf16 / mixed precision contract	6	5.3	$+0.7$
Artifact-contract drift	Field exists but has wrong type/shape	0	1.0	-1.0
Missing artifact	Required field absent from output	15	13.3	$+1.7$
Shape mismatch	Token/label dims collapsed or expanded	1	7.3	-6.3
	Numeric disagreement (passes interface, disagrees on numbers)			
Forward mismatch	`forward_logits` or `forward_loss` differs	4	2.7	$+1.3$
Method mismatch	DPO/PPO/SFT method-specific tensors differ	15	17.0	-2.0
Gradient mismatch	`gradient_loss` / `gradient_norm` differs	0	1.7	-1.7
	Behavioral disagreement / generation			
Generation mismatch / KV cache	Sampled trajectory or KV state diverges	23	29.7	-6.7

with diagnostics aiding subsequent repairs, but neither establish monotonic gains over further rounds nor isolate individual components. Section 5.3 complements this comparison with API costs.

Task-level refinement stability. With OpenCode, second-round regressions affect 2 of 63 tasks under TWINHARNESS versus 22 under ordinary continuation (Appendix A.2, Figure 5). This is consistent with structured verification preserving achieved behavior, but does not isolate component effects: each method starts from its own first-round candidate.

5.5 A CROSS-AGENT FAILURE TAXONOMY

Under the same verifier and task set, TWINHARNESS's improvements over Claude Code concentrate on execution-visible failures (Table 5): shape mismatches decrease by approximately 86%, generation/KV-cache failures by 23%, and no gradient mismatches are recorded for TWINHARNESS. These runtime properties—tensor shapes, generation states, and gradient statistics—align with the harness's additional reference-grounded feedback. Improvements are weaker for latent structural or semantic contracts: parameter-tree mismatches remain comparable, batch-schema and JAX/Torch-boundary failures increase, and method-specific numeric mismatches remain substantial. This suggests that TWINHARNESS is most effective when correctness yields a locally executable signal; hidden interface conventions and fine-grained semantic equivalence appear harder to address with current feedback. The pattern is consistent with improved detection and repair of execution-visible incompatibilities, not uniform error reductions, without isolating individual feedback components' causal contributions.

6 CONCLUSION

T2J-BENCH grades *execution-grounded functional equivalence* in 63 whole-repository PyTorch→JAX migrations across 18 model families, covering training and serving. A separate, `torch`-free verifier executes candidates against frozen PyTorch references. The best of twenty off-the-shelf agent–model configurations scores 0.676 against the 0.980 oracle: high spec pass rates still leave substantial numeric and behavioral gaps. Using task-visible PyTorch execution for legitimate self-verification, TWINHARNESS raises its `opus-5` scaffold's reward from 0.649 to 0.723, the highest of any configuration (§4). By evaluating whether migrated repositories preserve executable behavior, T2J-BENCH provides a foundation for measuring and advancing reliable cross-framework migration by coding agents.

AI USE STATEMENT

We used generative AI tools to assist with language polishing and the search for relevant literature. The authors manually reviewed and verified all AI-assisted text and retrieved references and take full responsibility for the final manuscript, including its text, claims, results, and citations.

ETHICS STATEMENT

This work introduces a benchmark and an agent harness for automated PyTorch→JAX repository migration. This work does not involve human subjects, personal data, or user-facing deployment. All tasks are built from publicly available repositories and checkpoints distributed under licenses permitting research use, with original licenses and attribution preserved. We foresee limited potential for misuse and declare no conflicts of interest or sponsorship influencing this study.

REPRODUCIBILITY STATEMENT

We describe the benchmark, grading pipeline, TWINHARNESS, the full evaluation protocol, and the prompt templates in the main paper and appendix. Because agent rollouts are stochastic, we evaluate every experimental setting, including TWINHARNESS and all baselines, with three random seeds and report seed averages with sample standard deviations. Refinement rounds are specified separately for each setting. We will release the benchmark suite to support reproduction under the same protocol.

REFERENCES

Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Automatic library migration using large language models: First results. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 427–433, 2024. doi: 10.1145/3674805.3690746. URL http://dx.doi.org/10.1145/3674805.3690746.

Nishil Amin, Zhiwei Fei, Xiang Li, Justyna Petke, and He Ye. JMigBench: A benchmark for evaluating LLMs on source code migration (Java 8 to Java 11). In *Proceedings of the 1st Workshop on Code Translation, Transformation, and Modernization (ReCode '26)*. ACM, 2026. doi: 10.1145/3786180.3788316. URL https://discovery.ucl.ac.uk/id/eprint/10222370/. In press.

Anthropic. Claude Code overview. Anthropic Documentation, 2026. URL https://docs.anthropic.com/en/docs/claude-code/overview.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

Keyuan Cheng, Xudong Shen, Yihao Yang, Tengyue Wang, Yang Cao, Muhammad Asif Ali, Hanbin Wang, Lijie Hu, and Di Wang. CODEMENV: Benchmarking large language models on code migration. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 2719–2744. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.140. URL https://aclanthology.org/2025.findings-acl.140/.

DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL http://github.com/google-deepmind.

Junjia Du, Yadi Liu, Hongcheng Guo, Jiawei Wang, Haojian Huang, Yunyi Ni, and Zhoujun Li. DependEval: Benchmarking LLMs for repository dependency understanding. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 7150–7179. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.373. URL https://aclanthology.org/2025.findings-acl.373/.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Sx038qxjek.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Minghua He, Yue Chen, Fangkai Yang, Pu Zhao, Wenjie Yin, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. ExeCoder: Empowering large language models with executability representation for code translation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 7099–7125. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.362. URL https://aclanthology.org/2025.emnlp-main.362/.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2026. URL http://github.com/google/flax.

Ruixin Hong, Hongming Zhang, Xinyu Pang, Dong Yu, and Changshui Zhang. A closer look at the self-verification abilities of large language models in logical reasoning. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 900–925. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-long.52. URL https://aclanthology.org/2024.naacl-long.52/.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, 2024.

Mohayeminul Islam, Ajay Kumar Jha, May Mahmoud, Ildar Akhmetov, and Sarah Nadi. An empirical study of python library migration using large language models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 867–879. IEEE, 2025. doi: 10.1109/ASE63991.2025.00077. URL https://doi.org/10.1109/ASE63991.2025.00077.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024b.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6766–6805. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.367. URL https://aclanthology.org/2024.acl-long.367/.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.839. URL https://aclanthology.org/2025.acl-long.839/.

Qiyue Liang, Steven Ingram, George Vanica, Andi Gavrilescu, Newfel Harrat, Hassan Sipra, and Sethuraman Sankaran. Agentic framework for deep learning workload migration via in-context learning. *arXiv preprint arXiv:2606.15994*, 2026.

Shanchao Liang, Nan Jiang, Yiran Hu, and Lin Tan. Can language models replace programmers for coding? REPOCOD says "not yet". In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24698–24717. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1204. URL https://aclanthology.org/2025.acl-long.1204/.

Jiaheng Liu, Ken Deng, Congnan Liu, Jian Yang, Shukai Liu, He Zhu, Peng Zhao, Linzheng Chai, Yanan Wu, JinKe JinKe, Ge Zhang, Zekun Moore Wang, Guoan Zhang, Yingshui Tan, Bangyu Xiang, Zhaoxiang Zhang, Wenbo Su, and Bo Zheng. M2RC-EVAL: Massively multilingual repository-level code completion evaluation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15661–15684. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.acl-long.763. URL https://aclanthology.org/2025.acl-long.763/.

Jingjing Liu, Zeming Liu, Zihao Cheng, Mengliang He, Xiaoming Shi, Yuhang Guo, Xiangrong Zhu, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. RepoDebug: Repository-level multi-task and multi-language debugging evaluation of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 23784–23813. Association for Computational Linguistics, 2025b. doi: 10.18653/v1/2025.findings-emnlp.1294. URL https://aclanthology.org/2025.findings-emnlp.1294/.

Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. ProjectEval: A benchmark for programming agents automated evaluation on project-level code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 20205–20221. Association for Computational Linguistics, 2025c. doi: 10.18653/v1/2025.findings-acl.1036. URL https://aclanthology.org/2025.findings-acl.1036/.

Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar-Tehrani, Xi Shen, Jun Huan, Omer Tripp, and Anoop Deoras. MigrationBench: Repository-level code migration benchmark from Java 8, 2025d. URL https://arxiv.org/abs/2505.09569.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems, 2023. URL https://arxiv.org/abs/2306.03091.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Introducing Codex. OpenAI, 2025. URL https://openai.com/index/introducing-codex/.

OpenCode Team. OpenCode: The open source AI coding agent, 2025. URL https://opencode.ai/.

Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Yanlin Wang, Xin Peng, and Zibin Zheng. RustRepoTrans: Repository-level context code translation benchmark targeting Rust. In *40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)*, pp. 610–622. IEEE, 2025. doi: 10.1109/ASE63991.2025.00057. URL https://doi.org/10.1109/ASE63991.2025.00057.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. CodeAgent: Autonomous communicative agents for code review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11313, 2024.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Arya Tschand, Charles Hong, Julian Walker, Nina Cai, Shangkun Wang, Suvinay Subramanian, Sundar Dev, Vijay Janapa Reddi, Amir Yazdanbakhsh, and Sethu Sankaran. Jaxbench: Benchmarking autonomous tpu kernel optimization. *arXiv preprint arXiv:2607.20466*, 2026.

vLLM Project. vllm tpu inference: A unified backend supporting pytorch and jax on tpu. https://github.com/vllm-project/tpu-inference, 2026.

Peiran Wang, Xiaogeng Liu, and Chaowei Xiao. CVE-Bench: Benchmarking LLM-based software engineering agents' ability to repair real-world CVE vulnerabilities. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 4207–4224. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.naacl-long.212. URL https://aclanthology.org/2025.naacl-long.212/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. RepoTransBench: A real-world multilingual benchmark for repository-level code translation, 2024. URL https://arxiv.org/abs/2412.17744.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5067–5089. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.337. URL https://aclanthology.org/2023.findings-emnlp.337/.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604. ACM, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pp. 400–410, 2024.

Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. Migrating code at scale with LLMs at Google. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 162–173. ACM, 2025. doi: 10.1145/3696630.3728542. URL https://doi.org/10.1145/3696630.3728542.

Appendix

Contents of Appendix

A Additional Experimental Analyses **16**

A.1 Applicability Across Agent Configurations 16

A.2 Task-Level Refinement Stability 16

A.3 Additional Off-the-Shelf Baselines 17

B Extended Related Work **18**

B.1 Coding Agents and Codebase-Level Benchmarks 18

B.2 Autonomous Kernel Optimization 18

C Benchmark design details **18**

C.1 Task classes in detail 19

C.2 Design challenges 19

C.3 Design principles 20

C.4 Equivalence lattice and aspects 21

C.5 Continuous-fidelity reward 21

C.6 Behavioral gates and the runtime dimension 22

C.7 Calibration and documented boundaries 23

D Additional Details of TWINHARNESS **23**

D.1 Detailed Diagnose–Migrate–Verify Loop 23

D.2 Reference-Grounded Diagnostic Taxonomy 24

D.3 Detailed Information Boundary 26

D.4 Role Prompts for Diagnose–Migrate–Verify 27

A Additional Experimental Analyses

A.1 Applicability Across Agent Configurations

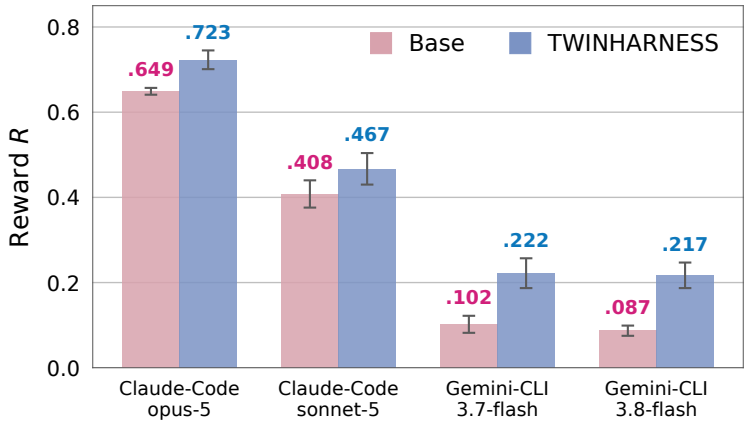


Figure 4: **Reward across four scaffold–model pairs on T2J-BENCH.** Each pair compares the base agent (pink) with TWINHARNESS (blue), holding the underlying scaffold and model fixed. Labels give mean rewards over three random seeds.

Figure 4 compares the base agent with TWINHARNESS for four scaffold–model combinations. On Claude-Code, reward increases from 0.649 to 0.723 with `opus-5` and from 0.408 to 0.467 with `sonnet-5`, absolute gains of 0.074 and 0.059, respectively. The Gemini-CLI configurations also show higher reported rewards, from 0.102 to 0.222 with `gemini-3.7-flash` and from 0.087 to 0.217 with `gemini-3.8-flash`. Taken together, the comparisons support applicability beyond a single scaffold–model pair on T2J-BENCH, without establishing generalization to unseen task distributions.

A.2 Task-Level Refinement Stability

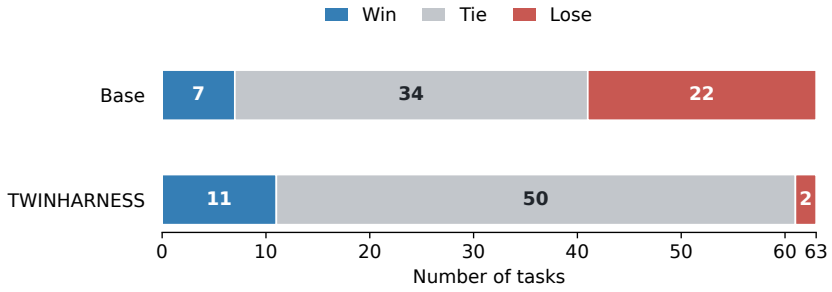


Figure 5: **Task-level reward changes after a second round.** Results cover 63 tasks with OpenCode and `opus-5`, with $\Delta R = R^{(2)} - R^{(1)}$. Win: $\Delta R > 0.01$; Tie: $|\Delta R| \leq 0.01$; Lose: $\Delta R < -0.01$. Labels are task counts. Each method is compared with its own first-round candidate.

Figure 5 examines task-level changes from the first to the second round with OpenCode and `opus-5`, where $\Delta R = R^{(2)} - R^{(1)}$. Ordinary continuation improves 7 of 63 tasks, leaves 34 stable, and degrades 22 (34.9%). In comparison, TWINHARNESS improves 11 tasks, leaves 50 stable, and degrades only 2 (3.2%). Its advantage therefore lies not only in improving more tasks, but also in substantially reducing observed regressions during further refinement.

Execution-grounded feedback localizes discrepancies, while verification checks repairs and informs subsequent diagnosis (Section 4.1). The results are consistent with this feedback-guided refinement helping preserve correct behavior. Because methods start from their own first-round artifacts, these observations do not isolate individual components' causal effects.

A.3 Additional Off-the-Shelf Baselines

To make the evaluation comprehensive, we provide additional off-the-shelf baseline results in this section. Table 6 reports two further scaffolds under the same protocol and metrics as §5.2: OpenHands (Wang et al., 2025b) with all six models, and Terminus-2, Harbor's Terminus agent (Harbor Framework Team, 2026), with the three Gemini models. All values are means over three random seeds on the full 63-task suite, produced by the same unmodified grading pipeline (§3.2). With `opus-5`, OpenHands reaches 0.622, below OpenCode (0.676) and Claude-Code (0.649) with the same model (Table 3); Terminus-2 peaks at 0.369 with `gemini-3.8-flash`. Every OpenHands and Terminus-2 configuration scores below both the top main-leaderboard baseline, OpenCode with `opus-5` (0.676), and TwinHarness (0.723). Including these baselines therefore leaves the conclusions of §5.2 unchanged: across all twenty off-the-shelf configurations the highest reward is still 0.676, and TwinHarness remains the highest of any configuration.

Table 6: Additional off-the-shelf baselines on 63 tasks, reported separately from the main leaderboard (Table 3) under the same protocol and metrics. R is reward (Eq. 1); Spec/Num./Beh. are full-suite dimension scores; Train/Infer. are rewards on the 45 training/18 inference tasks. Values are means over three random seeds; R subscripts give sample standard deviations.

Harness	Model	R	Spec	Num.	Beh.	Train	Infer.
OpenHands	opus-5	$0.622_{\pm.029}$	0.907	0.618	0.681	0.709	0.403
OpenHands	gemini-3.8-flash	$0.443_{\pm.016}$	0.828	0.555	0.435	0.525	0.235
OpenHands	gemini-3.7-flash	$0.416_{\pm.030}$	0.851	0.537	0.377	0.488	0.234
OpenHands	sonnet-5	$0.401_{\pm.023}$	0.755	0.428	0.459	0.321	0.602
OpenHands	glm-5.2	$0.247_{\pm.026}$	0.572	0.279	0.279	0.223	0.306
OpenHands	gemini-3.1-pro	$0.086_{\pm.031}$	0.364	0.108	0.092	0.040	0.203
Terminus-2	gemini-3.8-flash	$0.369_{\pm.015}$	0.793	0.496	0.353	0.453	0.160
Terminus-2	gemini-3.7-flash	$0.358_{\pm.012}$	0.786	0.479	0.331	0.442	0.147
Terminus-2	gemini-3.1-pro	$0.006_{\pm.004}$	0.223	0.008	0.008	0.001	0.019

B Extended Related Work

This appendix expands §2, enumerating the benchmarks and systems the main text cites only representatively.

LLM-based code conversion benchmarks. Conversion-oriented evaluation spans environment/version migration in CODEMENV (Cheng et al., 2025), Java migration in JMigBench (Amin et al., 2026), and codebase-scale conversion in RustRepoTrans (Ou et al., 2025), alongside the repository-level MigrationBench (Liu et al., 2025d) and RepoTransBench (Wang et al., 2024) cited in the main text. Adjacent work has examined library conversion with LLMs through early empirical studies (Almeida et al., 2024; Islam et al., 2025) and large-scale industrial deployment settings (Ziftci et al., 2025). Related evaluation has also grown through multilingual and execution-based benchmarks such as CodeTransOcean (Yan et al., 2023) and XCodeEval (Khan et al., 2024), as well as conversion methods evaluated at benchmark scale such as ExeCoder (He et al., 2025). These efforts substantially broaden evaluation beyond generic code generation, but they largely target API, language, version, or general codebase conversion, and grade it either by surface-level similarity to a reference (Lu et al., 2021; Ren et al., 2020) or by executing small, self-contained programs, rather than by execution-grounded equivalence of a whole running ML system.

B.1 Coding Agents and Codebase-Level Benchmarks

A parallel line studies LLM-based agents for codebase-level software engineering, established by issue resolution on SWE-bench (Jimenez et al., 2024a) and pursued by scaffolds such as SWE-agent (Yang et al., 2024) and OpenHands (Wang et al., 2025b). These evaluate completion, feature implementation, dependency understanding, debugging, and ML-engineering workflows, but do not center on cross-framework repository conversion under a fixed equivalence contract. T2J-Bench complements them by evaluating whether such agents can perform semantic-preserving, cross-paradigm conversion of whole PyTorch repositories—across both training and serving—graded by execution-grounded functional equivalence rather than issue resolution or code completion.

Additional scaffolds such as AutoCodeRover (Zhang et al., 2024) and Agentless (Xia et al., 2025) explore different interfaces and pipelines for codebase navigation, localization, editing, and execution. Related agentic settings include multi-agent competitive code generation in MapCoder (Islam et al., 2024), autonomous code review in CodeAgent (Tang et al., 2024), interactive agent benchmarking in AppWorld (Trivedi et al., 2024), and broader scientific and algorithmic discovery through coding agents in AlphaEvolve (Novikov et al., 2025). In parallel, a broader benchmark ecosystem has emerged around codebase-level coding ability, including RepoBench (Liu et al., 2023), FEA-Bench (Li et al., 2025), RepoCod (Liang et al., 2025), DependEval (Du et al., 2025), ProjectEval (Liu et al., 2025c), RepoDebug (Liu et al., 2025b), CVE-Bench (Wang et al., 2025a), M2RC-EVAL (Liu et al., 2025a), and MLE-bench (Chan et al., 2025). These evaluate codebase completion, feature implementation, dependency understanding, debugging, vulnerability repair, multilingual codebase reasoning, and ML engineering workflows, but do not center on cross-framework repository conversion under a fixed equivalence contract.

B.2 Autonomous Kernel Optimization

JAXBench (Tschand et al., 2026) evaluates autonomous Pallas kernel optimization on 50 JAX workloads for TPUs through numerical correctness and speedup over XLA. Its execution-feedback-driven operator optimization complements T2J-Bench's repository-level PyTorch-to-JAX migration, which requires checkpoint conversion and faithful training or serving behavior.

C Benchmark design details

This appendix expands §3: the per-class task mechanics (§C.1), the full statements of the grading challenges (§C.2) and design principles (§C.3), the equivalence lattice (§C.4), the continuous-fidelity reward (§C.5), the behavioral and runtime gates (§C.6), and the benchmark's calibration and documented boundaries (§C.7).

C.1 TASK CLASSES IN DETAIL

Each task gives the agent two things: (i) a self-contained, pure-PyTorch source tree — a legible artifact of roughly 15–20 modules (about 4k lines) extracted from a real codebase and stripped of heavy external dependencies — and (ii) the raw, publicly released model checkpoint, provided inside the task environment. These source trees are not synthetic snippets: the training-class sources are drawn from LLaMA-Factory (Zheng et al., 2024), and the serving-class sources from vLLM (Kwon et al., 2023) and its JAX/TPU counterpart, tpu-inference (vLLM Project, 2026), so a passing JAX reimplementation is a migration of code that practitioners actually run. Each task is delivered as a self-contained, containerized environment (Harbor Framework Team, 2026) in two halves. The agent works in a networked environment, with a GPU and both the original source and the checkpoint available. The finished JAX reimplementation is then graded in a *separate*, offline environment that contains no PyTorch at all and receives only the agent's JAX repository together with the same checkpoint. This split is the structural embodiment of the independence guarantee: the agent may run PyTorch as a reference while working, but the graded JAX reimplementation runs where PyTorch does not exist and the network is unavailable.

Training-migration class (45 tasks). The training class is a 15×3 grid: 15 model families crossed with three post-training methods — supervised fine-tuning (SFT), Direct Preference Optimization (DPO) (Rafailov et al., 2023), and Proximal Policy Optimization (PPO) (Schulman et al., 2017). Every task is a LoRA (Hu et al., 2021) fine-tune (rank 8 on all linear projections, base weights frozen; the trainable fraction is 0.28–0.88%). The 15 families span eight text-only architectures (BLOOM, Falcon, Llama 3, Mistral, Phi-4, Qwen2, Qwen3, StarCoder2) and seven multimodal ones (Gemma 3, InternVL, LLaVA-NeXT, PaliGemma, Qwen2-VL, Qwen3-VL, Qwen2-Omni), giving 24 text-only and 21 vision-language tasks. The benchmark grades only two modalities — text and vision (image) — and there is *no graded audio modality*: the single omni-modal family, Qwen2-Omni, is counted and graded as a vision-language task, and its audio pathway is merely checked for presence rather than numerically evaluated (§C.7). To succeed, a JAX reimplementation must reproduce — without PyTorch — the model's forward pass, the method-specific loss (shifted causal cross-entropy for SFT, the preference objective for DPO, the policy surrogate for PPO), the AdamW optimizer and its learning-rate schedule, the LoRA freeze/adapter structure, and the full data pipeline (prompt templating, batching, and label masking). PPO tasks additionally run a reward model inside the environment so that the whole loop stays self-contained. The three methods form a natural difficulty gradient: SFT exercises a single supervised loss, DPO a paired preference objective, and PPO a full sampling-and-update loop whose cross-framework reproducibility is the loosest.

Inference-migration class (18 tasks). The inference class targets the serving end of the lifecycle: the agent reimplements a vLLM-style paged-attention inference engine (about 1,500 lines over 19 modules) as a native JAX implementation that preserves *serving* semantics — the paged key–value (KV) cache, the block table that indexes it, the sampler's ordering of transforms, and greedy decoding (Kwon et al., 2023). This is a distinct kind of migration object: state that PyTorch mutates in place (the KV cache) must become an explicit value that the JAX code threads through each step. The class is a 6×3 grid — six families (Llama 3, Qwen2, Qwen3, GPT-OSS, Gemma-4, and the vision-language Qwen2.5-VL) crossed with three engine modes of increasing scheduling complexity: *core* (single-sequence decode), *ragged* (a batched paged cache over requests of differing lengths), and *chunked* (chunked prefill with prefix reuse) — giving 15 text and 3 vision-language tasks. Because the two classes require different JAX versions (the serving stack needs a newer release), they build on separate base environments but share the checkpoints and grading utilities.

C.2 DESIGN CHALLENGES

Grading functional equivalence faithfully — rather than rewarding a JAX reimplementation that merely *looks* like the original — raises five challenges that ordinary code-translation benchmarks do not confront. Table 2 in the main text summarizes their corresponding principles and grading mechanisms.

(C1) Fabricated equivalence. A candidate can appear equivalent while secretly relying on PyTorch, the original modeling libraries, or the network — to load weights, borrow a reference computation, or fetch something at run time — so any numeric agreement it reports may be unearned.

(C2) "Looks right" is not "behaves right." A cross-paradigm rewrite turns eager, stateful, object-oriented code into pure, compilable functions, and in-place cache updates into explicitly threaded state; code that resembles the original, or that passes hand-written unit tests, need not reproduce the system's numerics or dynamics.

(C3) Uncertain ground truth. Grading a JAX reimplementation against a second, hand-written JAX implementation would compare one implementation's quirks against another's; the target of equivalence must instead be the original code that practitioners actually run.

(C4) No clean pass/fail. Two independent 32-bit floating-point stacks disagree slightly simply because they sum numbers in a different order, so exact equality is impossible and any fixed tolerance is arbitrary and tunable; grading must nonetheless separate a real bug from this floating-point noise and award partial credit across a large, multi-stage system.

(C5) Performance is part of correctness. A numerically perfect but too-slow JAX reimplementation is not a usable migration: production training and paged-attention serving are viable only within a compute budget, so a JAX reimplementation that reproduces every number yet cannot run at the original's speed has not truly migrated the system.

C.3 Design principles

We answer these challenges with five design principles that govern grading in *both* task classes (Table 2); the grading machinery each one drives is detailed in §C.4–§C.6.

(P1) Independence-first. Numeric agreement is meaningless if the candidate secretly calls PyTorch (C1), so independence is a precondition, not a checkbox. The grading environment has no PyTorch, none of the original modeling libraries, and no network, and the candidate is run in a setting where importing PyTorch simply fails. Any hidden dependency therefore fails loudly rather than silently inflating a score — whether it hides in a training loop or in a decoding kernel.

(P2) Execution-grounding. Because code that *looks* like the original need not *behave* like it (C2), every equivalence claim is settled by *running* the candidate on fixed inputs, never by reading its source. A training candidate runs its own weight conversion, forward pass, backward pass, and a real optimizer loop; a serving candidate runs its own weight conversion, prefill, and multi-step paged decoding through its own engine. In both cases the candidate is handed the reference's exact inputs and must rebuild the model weights itself, so any observed agreement is attributable to the JAX reimplementation, not to borrowed tensors or cached outputs.

(P3) Oracle provenance. To fix an unambiguous ground truth (C3), the target of equivalence is the *pure-PyTorch original*, not a reference JAX solution. Reference outputs are recorded from the PyTorch original on a fixed H100 GPU in true 32-bit precision — for training, the inputs, logits, gradients, loss trajectories, and held-out validation losses; for serving, the prefill logits, per-step decode logits, greedy-decoded sequences, and held-out-prompt log-probabilities — and any change to the original forces them to be re-recorded. The reference JAX solution is treated as a known-imperfect candidate, never as the source of truth, which keeps the benchmark from grading one JAX reimplementation against another's idiosyncrasies.

(P4) Bounded, threshold-free metrics. Because no fixed tolerance can honestly separate floating-point noise from a real bug (C4), every numeric and behavioral comparison yields a *continuous similarity in* $[0, 1]$ from an intrinsically bounded metric, rather than a tuned pass/fail cutoff (§C.5).

(P5) Runtime as a graded dimension. Because a numerically perfect but too-slow JAX reimplementation is not a usable migration (C5), speed is graded rather than assumed: the end-to-end levels run under a uniform time budget derived from the PyTorch original, and a JAX reimplementation that exceeds it scores 0 on the affected check, exactly as a wrong one does (§C.6). Correctness in T2J-BENCH therefore includes running at the original's speed, not merely returning its numbers.

Table 7: The seven-level functional-equivalence lattice, read per class. A task attains a prefix; the top levels grade a *trained model* (training) or a *live paged-decode engine* (serving), so a faithful JAX reimplementation must run end to end, not merely match a single forward pass.

Level	Name	Training reading	Serving reading	Aspect
L0	Independence & structure	PyTorch-free, compilable, mirrors the original's file and interface layout; converted parameters map one-to-one to the checkpoint.	PyTorch-free JAX engine mirroring the original's module and interface layout; converted parameters map one-to-one to the checkpoint.	Spec
L1	Module / operator parity	every reimplemented submodule matches its PyTorch counterpart on the same input.	every reimplemented operator — attention, rotary embeddings, the paged-attention kernel, sampler transforms — matches its PyTorch counterpart.	Numeric
L2	Composed-forward parity	the full forward pass (logits and hidden states) matches on the canonical batch.	the full prefill and a single decode step reproduce the reference logits on the canonical prompt.	Numeric
L3	Generalization	forward parity holds across input shapes (length, ragged or padded, multiple rows).	logit parity holds across serving regimes: varying lengths, ragged batches, and chunked-prefill and prefix-reuse boundaries.	Numeric
L4	Mechanism parity	the method's loss and its gradient match (value, authenticity, LoRA structure, learning-rate schedule).	greedy decoding reproduces the reference next-token distribution step by step (sampler ordering, decode/prefill consistency).	Num. / Beh.
L5	End-to-end run parity	the JAX reimplementation's full training run reproduces the loss trend, optimizer identity, and decoding behavior.	the engine serves end to end with correct paging: invariance to batching, chunking, and prefix caching, determinism, and no leakage across sequences.	Behavioral
L6	Held-out behavior parity	a model *trained* by the JAX reimplementation reaches a held-out validation loss close to the PyTorch-trained model's.	greedy-decoded sequences and held-out-prompt log-probabilities match the reference engine's on *unseen* prompts.	Behavioral

C.4 Equivalence lattice and aspects

Equivalence is decomposed along a seven-level lattice (Table 7), read once for each class: every level has a training reading and a serving reading, and a task earns some prefix of the levels. Independently, each individual check is classified into exactly one of three *aspects* by the kind of evidence it provides: **Spec** (static structure, public interface, parameter correspondence, and independence — hardware-invariant, pass/fail), **Numeric** (a single forward or backward pass, or an isolated module or operator, on a fixed input, scored by continuous similarity), and **Behavioral** (optimizer step loops, multi-step decode loops, reference-free invariants, and the real end-to-end training or serving run). The three aspects are mutually exclusive and cover every kind of evidence a check can give, and they are exactly the dimensions the spec-gated reward (Eq. 1) consumes. The lattice maps onto them by depth: level L0 is the multiplicative **Spec** gate, levels L1–L4 feed the **Numeric** dimension (L4's step-wise mechanism checks also feed the **Behavioral** one), and the whole-run levels L5–L6 feed **Behavioral**. In both classes, then, the reward scores exactly the lattice prefix a task attains.

C.5 Continuous-fidelity reward

Because two independent 32-bit floating-point stacks disagree only through differences in summation order, a *correct* JAX reimplementation does not score exactly 1.0: single-pass comparisons land at a measured noise ceiling of about 0.999, and a real bug drives the score smoothly below it, with no hand-set threshold separating "exact" from "loose." Every comparison is built from five intrinsically bounded similarity measures (Table 8), each of which maps any not-a-number, infinity, or shape mismatch to 0 so that a degenerate output cannot look similar. Both operands are first promoted to 32-bit precision; the small constant in each denominator ($\varepsilon = 10^{-12}$) only guards against division by

Table 8: The five similarity measures ($a \cdot b$ is a flattened dot product; $\|\cdot\|$ is the ℓ_2 norm; JSD is the Jensen–Shannon divergence). Every measure returns a score in $[0, 1]$ (1 = identical) and needs no threshold.

Measure	Definition	Applied to
$S_{\text{scalar}}(a, b)$	$1 - \dfrac{2\,\lvert a-b\rvert}{\lvert a\rvert + \lvert b\rvert + \varepsilon}$	losses, log-probabilities, gradient norms, validation loss
$S_{\ell_2}(a, b)$	$1 - \dfrac{\lVert a-b\rVert}{\lVert a\rVert + \lVert b\rVert + \varepsilon}$	magnitude-aware tensor closeness, schedules
$S_{\cos}(a, b)$	$\max\!\left(0,\ \frac{a\cdot b}{\lVert a\rVert\,\lVert b\rVert+\varepsilon}\right)$	direction only (used inside S_{tensor})
$S_{\text{tensor}}(a, b)$	$\sqrt{S_{\cos} \cdot S_{\ell_2}}$	hidden states, module inputs/outputs, gradients
$S_{\text{dist}}(p, q)$	$1 - \dfrac{\text{JSD}(\text{softmax}\, p,\ \text{softmax}\, q)}{\ln 2}$	logits, any distribution over a vocabulary

zero and cannot be tuned to change a verdict. The fixed factor of 2 in the scalar measure is likewise not a hidden tolerance: it rescales the symmetric difference to the true mean-relative error, so the score reaches 0 once two values differ by their own average magnitude.

The measure is chosen by the *shape* of the quantity being compared: distributions over a vocabulary use the distributional measure (a Jensen–Shannon distance on the softmax, which ignores the arbitrary per-token additive constant yet still reacts to temperature changes that a scale-blind cosine would miss); single numbers use the scalar measure; and arrays use the tensor measure, whose geometric mean of direction and magnitude ensures that a uniform-scaling bug — which a cosine alone would overlook — still lowers the score. Aggregation is two-level and threshold-free: within a multi-element check the score is the plain average of the element scores, and each dimension's score is the plain average of its checks' scores; nothing is rounded to pass/fail on the way up.

These per-check scores combine into the primary reward (Eq. 1), a *spec-gated, dimension-weighted similarity*. Spec acts as a multiplicative anti-fabrication gate: a JAX reimplementation that is not genuine, correctly shaped, and PyTorch-free earns nothing, however close its numbers happen to look, and a partially correct structure scales the entire reward by the fraction of spec checks it passes. The two similarity dimensions each contribute the average of their per-check $[0, 1]$ scores, renormalized over whichever dimensions actually ran, so partial credit is continuous rather than a series of stage pass/fails. Every failure mode maps to 0 rather than being skipped: a crash, a PyTorch leak, a not-a-number result, or an end-to-end run that exceeds the uniform time budget (§C.6) all score 0 on the affected check — too slow counts for exactly as little as wrong.

C.6 Behavioral gates and the runtime dimension

Realizing P5, the highest levels grade a *trained artifact*, not a single forward pass. Level L6 splits each task's bundled dataset into fixed training and validation subsets (using a fixed, framework- and platform-independent random shuffle of the indices), trains a model using the candidate's own training entry point — inside the PyTorch-blocked, offline environment — and scores the resulting held-out validation loss against the recorded PyTorch reference using a *banded* version of the scalar measure. The width of the band is set by the *reference's own* run-to-run variability (measured by re-running the PyTorch original), never by the PyTorch-versus-JAX gap, so a faithful JAX reimplementation lands inside the band and scores 1.0, and only divergence beyond the reference's own fluctuation pulls the score down. Because the small single-pass probes barely move the model, L6 shares one real training run with the end-to-end level L5, and that run is capped by a uniform time budget derived from the PyTorch reference's wall-clock time. A numerically perfect but too-slow JAX reimplementation therefore scores 0: clearing L5 and L6 requires the JAX reimplementation to be genuinely optimized for speed — compiled fixed-shape execution, fused kernels, and static-shape rewrites of eager sub-networks — which is what makes *runtime a first-class graded dimension* rather than an afterthought.

C.7 CALIBRATION AND DOCUMENTED BOUNDARIES

Two safeguards protect the benchmark's own validity. First, a *mirror check* statically verifies that the reference JAX solution and the PyTorch original expose the same set of modules and that every generated task instruction matches its template, so the specification the agent reads never drifts from the reference. Second, an *oracle solvability check* submits the reference JAX solution itself as a candidate and grades it through the full pipeline. This oracle attains a reward of 0.973 on the training class (34 of 45 tasks pass every check strictly) and 0.997 on the inference class (18 of 18), for 0.980 overall (52 of 63) — with per-pass numeric similarity near 0.999. The overall reward aggregates structural, numeric, and behavioral checks and should not be identified with this per-pass similarity. The reference results provide an empirical baseline for faithful reimplementation, while the remaining strict-check failures delimit the claim of exact agreement. For the inference class, a *seeded-defect* sweep further shows that 84% of the collected checks are tripped by at least one deliberately injected bug in the reference, evidence that the checks have teeth rather than merely staying green on correct code.

We state non-goals explicitly rather than letting a high reward quietly absorb them. Multimodal grading pins the text-token forward pass and, by design, the vision tower and its merge step up to the point where they enter the causal mask; but *fully attended* multimodal numerics (image content flowing all the way into a graded language-model logit) and language-side submodule parity on multimodal families remain a scoped frontier, and audio (for the omni-modal family) is checked only for presence rather than graded. Several other items are deliberately out of scope: cache- and end-of-sequence-aware decoding within the training class; exact token-by-token generation (tie-breaking among equal-probability tokens and reduction-order flips make the exact token ids irreproducible, so the tie-robust next-token distribution is scored instead); LoRA gradient *magnitudes* beyond the first step (which depend on random initialization once it differs across frameworks, so they are scored structurally and through initialization-stable statistics); and portability of grading across arbitrary hardware (the suite fixes a single H100 GPU so the choice of device never enters a verdict). Naming these keeps the benchmark's coverage legible: what a high reward proves is exactly the graded lattice prefix, and nothing is smuggled in as unearned coverage.

D ADDITIONAL DETAILS OF TWINHARNESS

This appendix provides the complete stage-wise formalization of the *diagnose–migrate–verify* loop, the full taxonomy of reference-grounded diagnostics, and the information boundary enforced by TWINHARNESS.

D.1 DETAILED DIAGNOSE–MIGRATE–VERIFY LOOP

Let $\mathcal{S}$ denote the task specification, $\mathcal{P}$ the task-visible PyTorch repository, and $\mathcal{J}_t$ the candidate JAX repository after iteration t. We denote by $\mathcal{G}(\mathcal{J}; \mathcal{S}, \mathcal{P})$ the reference-grounded diagnostic report introduced in §4.2.

The *Diagnoser* first analyzes the current candidate together with previously validated evidence and identifies the behavioral discrepancies that should be addressed next:

$$D_t = \mathrm{Diagnose}(\mathcal{S}, E_{t-1}, \mathcal{G}(\mathcal{J}_{t-1}; \mathcal{S}, \mathcal{P}); \mathrm{ro}(\mathcal{J}_{t-1}))\,. \tag{2}$$

The *Migrator* then updates the JAX repository to resolve the diagnosed cross-framework discrepancies:

$$\mathcal{J}_t = \mathrm{Migrate}(\mathcal{J}_{t-1}; \mathcal{S}, D_t, \mathcal{G}(\mathcal{J}_{t-1}; \mathcal{S}, \mathcal{P}))\,. \tag{3}$$

Finally, the *Verifier* independently exercises the updated repository and records which behaviors have been established and which remain unresolved:

$$E_t = \mathrm{Verify}(\mathrm{ro}(\mathcal{J}_t); \mathcal{S}, D_t, \mathcal{G}(\mathcal{J}_t; \mathcal{S}, \mathcal{P}))\,. \tag{4}$$

Equations (2)–(4) formalize the three stages of the migration loop. Here D_t denotes a diagnosis of the remaining migration gaps together with the corresponding repair objectives, while E_t records validated evidence about the updated candidate.

The stages deliberately serve distinct purposes. *Diagnose* converts execution evidence into localized migration objectives; *Migrate* changes the candidate implementation; and *Verify* determines which properties have actually been established after the change. The Diagnoser and Verifier receive the candidate repository in read-only form, represented by $\mathrm{ro}(\mathcal{J})$, while only the Migrator is allowed to modify the repository.

This structure differs from open-ended iterative harness development. The desired capability is already specified by the executable PyTorch repository, so iteration is not used to discover an unknown objective. Instead, it progressively reduces the behavioral discrepancy between the source repository $\mathcal{P}$ and the migrated candidate $\mathcal{J}_t$.

Two persistent objects carry state across iterations. The repository $\mathcal{J}_t$ records what the current implementation *is*, whereas E_t records validated knowledge about what remains incorrect or unestablished. Keeping these objects separate prevents later iterations from having to reconstruct development state solely from implementation code or interaction history.

Evidence produced after one migration step is subsequently available to diagnosis of the next step, yielding the feedback flow

$$\mathcal{J}_{t-1} \longrightarrow D_t \longrightarrow \mathcal{J}_t \longrightarrow E_t \longrightarrow D_{t+1}. \tag{5}$$

Equation (5) makes explicit that repository state and validated diagnostic state are maintained separately while evidence is propagated across iterations.

D.2 Reference-Grounded Diagnostic Taxonomy

The diagnostic report $\mathcal{G}(\mathcal{J}; \mathcal{S}, \mathcal{P})$ is constructed exclusively from the task specification, the task-visible PyTorch repository, and the evolving JAX candidate. Its purpose is not to assign a benchmark score, but to convert ordinary development-time execution into localized evidence that can be consumed by the Diagnoser, Migrator, and Verifier.

Concretely, each diagnostic records (i) the property being examined, (ii) the execution context used to examine it, (iii) an outcome in {*established*, *violated*, NOT ESTABLISHED}, and (iv) supporting observations such as runtime structure, relevant values or metadata, and failure traces when execution does not complete. This evidence is retained across iterations through E_t and can therefore guide later diagnosis without requiring the harness to infer previous failures solely from the current repository state.

Development reference versus benchmark oracle. Under T2J-BENCH, benchmark reference outputs are recorded ahead of time from the pure-PyTorch original and consumed only by the separate grading pipeline (§3.2). Those frozen references never enter the development environment.

In contrast, TWINHARNESS executes the task-visible PyTorch repository $\mathcal{P}$ itself during development and derives diagnostic observations directly from that execution. The PyTorch repository therefore acts as an *executable specification*, not as a source of benchmark goldens. Development-time observations may be incomplete or inconclusive, and no frozen reference, hidden test, verifier result, reference JAX solution, or benchmark reward is exposed to the harness. The complete information boundary is detailed in Appendix D.3.

Differential behavioral grounding. The main reference-grounded diagnostic executes a matched probe against both $\mathcal{P}$ and $\mathcal{J}$ through their native execution paths. The probe constructs inputs through the repositories' exposed interfaces, performs the corresponding computation, and records the resulting runtime contract.

The comparison primarily establishes whether the migrated implementation preserves externally observable behavior that repository code depends on, including the structure of returned objects and their exposed fields. For example, the PyTorch implementation may return a structured model output while the candidate exposes only its underlying array. Although a candidate-authored numerical test could flatten both values and report agreement, downstream repository code expecting the original interface would still fail. Differential execution exposes this discrepancy because the behavioral reference is defined outside the candidate itself.

This distinction is important for avoiding circular self-validation. Tests authored only against $\mathcal{J}$ can accidentally encode assumptions introduced by the migration itself, whereas a matched execution against $\mathcal{P}$ anchors the claim to behavior of the original repository.

Candidate-side diagnostics. Not every migration property requires differential execution. A complementary set of diagnostics directly exercises properties of $\mathcal{J}$ that follow from the task contract or from the semantics required by its JAX execution path. These checks cover basic executability, independence from prohibited dependencies, parameter representation, differentiability, compatibility with JAX transformations, and integrated execution of the migrated training or serving entry point.

These diagnostics progressively narrow the source of a failure. For example, failure to import the repository is distinguished from an invalid parameter tree; successful construction is distinguished from a computation with no live gradient; and local execution is distinguished from an integrated training or serving path that fails only after components are composed. The objective is localization rather than reproduction of the benchmark's grading lattice.

Independence diagnostics. Independence is treated as a development-time property because it is already explicit in the migration specification. The candidate is exercised under conditions intended to expose reliance on dependencies or external resources excluded by the task, including PyTorch-dependent execution and network-dependent behavior. Such checks can reveal dependencies that ordinary development accidentally masks, for example through locally cached resources.

These checks instantiate only task-visible constraints. They neither invoke nor approximate the hidden independence checks used by the benchmark grader; the detailed separation between development and grading environments is given in Appendix D.3.

Progressive divergence localization. An end-to-end failure alone provides little information about where a cross-framework migration diverged. TWINHARNESS therefore combines diagnostics at multiple levels of execution, moving from repository validity and independence toward runtime contracts and integrated training or serving behavior.

This organization shares the high-level intuition of the functional-equivalence lattice in Appendix C.4: deeper execution can support stronger claims about the candidate. The two mechanisms are nevertheless separate. The development diagnostics neither execute the hidden grader nor reconstruct its checks, thresholds, references, or reward. They are independently constructed from task-visible artifacts and are used only to produce development-time evidence.

Table 9 summarizes the diagnostics used by TWINHARNESS. Differential checks use the executable PyTorch repository as an external behavioral anchor, whereas candidate-side checks establish properties of the evolving JAX repository directly.

Diagnostic outcome semantics. A diagnostic result describes what the corresponding probe actually establishes, rather than treating every unsuccessful execution as evidence against the candidate. For a property q, the report assigns one of three outcomes:

- *Established*: the diagnostic executed with sufficient evidence and observed behavior consistent with q.
- *Violated*: the diagnostic executed with sufficient evidence and observed behavior inconsistent with q.
- NOT ESTABLISHED: the diagnostic itself could not reliably determine whether q holds.

The distinction between *violated* and NOT ESTABLISHED is essential in cross-framework migration. Multiple correct JAX implementations can realize the same external behavior through different internal structures or calling conventions. A probe that assumes a convention not supported by a candidate may therefore fail before it has produced evidence about the behavioral property it was intended to test. Treating such a probe failure as a candidate failure would introduce false diagnostic evidence.

Evidence preservation under uncertainty. A NOT ESTABLISHED result is not discarded. The harness retains the execution context and failure trace that made the diagnostic inconclusive, allowing

Table 9: Development-time diagnostics used by TWINHARNESS. *Differential* checks compare observable behavior of the candidate with the task-visible PyTorch repository; *candidate* checks exercise properties implied by the migration contract or the exposed execution path. Returned evidence is used for localization and repair rather than benchmark scoring.

Check	Grounding	Property examined	Evidence returned
Module import	candidate	Required modules can be loaded through the migrated repository's intended import paths.	Import context together with successful construction or the exception that prevents the repository from being initialized.
Forbidden dependency	candidate	Execution does not rely on dependencies or external resources excluded by the migration contract.	The dependency or resource access exposed during execution, or evidence that the tested path executes without it.
Output contract	differential	The migrated implementation preserves the externally visible runtime structure and fields exposed by the PyTorch original.	Matched observations from $\mathcal{P}$ and $\mathcal{J}$, including output structure, available fields, and any discrepancy between the two execution paths.
Parameter tree	candidate	The migrated parameter representation forms a valid JAX pytree compatible with the repository's execution path.	Observed tree structure and the construction or lookup failure associated with an incompatible parameter representation.
Gradient liveness	candidate	The training objective remains differentiable through the parameters intended to be optimized.	Whether gradients can be produced for the relevant parameter path, together with the failure context when differentiation is broken.
JAX tracing	candidate	The relevant computation is compatible with the tracing assumptions required by the JAX implementation.	Successful transformed execution or the trace/concretization failure that identifies the incompatible computation.
Training / serving entry	candidate	The repository's exposed end-to-end entry path executes coherently after migrated components are composed.	Integrated execution status and the runtime trace at which the composed training or serving path fails.

Table 10: Information boundary of TWINHARNESS. The harness uses only artifacts available to an ordinary agent during task execution and never accesses hidden benchmark state.

Agent-visible: used by TWINHARNESS	**Agent-hidden: never read**
`instruction.md`; `torch_orig/`; the released checkpoint; the candidate repository; outputs and exceptions produced by executing task-visible artifacts	hidden tests; frozen reference outputs; the verifier implementation; reference JAX solutions; benchmark rewards, scores, and per-check grading results

the next Diagnoser to decide whether the unresolved property requires a different probe, a repository repair, or no immediate action. Likewise, an *established* result records the evidence supporting the property so that later iterations need not repeatedly infer it from the repository.

Consequently, E_t represents a partial and revisable account of the candidate rather than a scalar notion of correctness. The harness may know that some properties have been established, have concrete evidence that others are violated, and remain uncertain about the rest. This explicit representation of uncertainty is what allows execution evidence to be carried forward through the diagnose–migrate–verify loop without conflating diagnostic failure with migration failure.

Scope of a diagnostic claim. Development-time diagnostics are intentionally local. Establishing one property does not certify functional equivalence of the repository as a whole, and absence of a detected violation does not imply that the candidate will pass the benchmark. Only the separate, unchanged T2J-BENCH grading pipeline produces the final benchmark measurement.

Accordingly, $\mathcal{G}(\mathcal{J}; \mathcal{S}, \mathcal{P})$ should be interpreted as a collection of execution-grounded claims about properties that could be established from task-visible information at the current iteration. Its role is to improve localization and repair, not to serve as a surrogate benchmark oracle.

D.3 DETAILED INFORMATION BOUNDARY

Because T2J-BENCH explicitly guards against fabricated equivalence (§C.2), a reference-grounded harness must make its information boundary explicit. TWINHARNESS uses exactly the information granted to an ordinary agent by the task and nothing from the benchmark's hidden grading state. Table 10 summarizes this boundary.

We enforce this boundary at both the filesystem and execution levels. Development environments expose only task-visible components needed to solve the migration rather than the hidden grading implementation. Candidate state is exchanged through the evolving repository rather than through verifier-generated grading artifacts. Execution credentials and intermediate state are likewise restricted so that the development environment cannot serve as an indirect route to hidden benchmark information.

This distinction is important because information that is technically reachable can affect the legitimacy of self-verification even when it does not appear in an observed agent trace. We therefore define the boundary according to what the harness can access, rather than only according to what the agent happens to consume during one execution.

The same principle constrains the diagnostics themselves. A diagnostic is included only when the property it examines can be derived from the task specification, the task-visible PyTorch repository, or the ordinary semantics of the exposed execution interface. Diagnostics whose design depends on hidden verifier behavior are excluded from TWINHARNESS.

Unchanged final grading. TWINHARNESS changes only the procedure used to construct the candidate repository. Once migration terminates, the resulting repository is evaluated through the same unmodified T2J-BENCH grading pipeline as every off-the-shelf baseline. Neither the verifier nor the reward function is modified for TWINHARNESS.

D.4 ROLE PROMPTS FOR DIAGNOSE–MIGRATE–VERIFY

For reproducibility, we provide the complete role prompts used by TWINHARNESS. The prompts instantiate the three stages of the *diagnose–migrate–verify* loop described in Appendix D.1. They differ primarily in their responsibilities and permissions: the *Diagnoser* analyzes the current candidate and available execution evidence to localize discrepancies and formulate repair objectives; the *Migrator* is the only role permitted to modify the evolving JAX repository; and the *Verifier* independently exercises the updated candidate and records which properties are established, violated, or remain unresolved.

All three roles operate only on task-visible artifacts and development-time evidence available to TWINHARNESS. The prompts do not expose hidden tests, frozen benchmark references, verifier internals, reference JAX solutions, benchmark scores, or rewards. Separating the roles in this way keeps diagnosis, repository modification, and post-update verification explicit rather than conflating them within a single open-ended coding-agent instruction.

The following listings give the exact prompts used by the three roles.

```
/role/diagnoser
You are the Diagnoser for iteration {loop} of an iterative port-
    development run. This is a
planning-only invocation. Do not implement, edit, or repair anything, and
     do not run the port. Read
whatever you need to ground the plan, then return only the planning
    document described below.

/source-of-truth
The task document and the PyTorch program it refers to are the complete
    source of truth:

  * `/workspace/torch_orig` -- the original program. It is the
    specification: it can be read AND
    executed, and what it does when run is the behaviour the port must
    reproduce.
  * the task document reproduced below.

Do not use, request, or speculate about hidden tests, scores, rubrics,
    evaluator feedback, or any
other non-public information. None of it exists in this container.

{instruction}
```

```
/previous-iteration-evidence
{evidence}

/planning-policy
Prioritise blockers and regressions before new behaviour. A port that
    cannot be imported, cannot
load its checkpoint, or cannot complete a training step blocks every
    other claim about it.

Select at most three priorities. Each must be:

  * bounded -- it does not drag in unrelated rewrites of the port's
    architecture;
  * locally complete -- it includes whatever related changes are needed
    to make the behaviour
    actually work and actually observable, even when those span several
    files;
  * observable -- you can state, now, what one would run to see whether
    it holds.

Each priority is a concrete implementation target plus the observation
    that would validate it.
Behaviour already verified in the evidence above is a preservation
    constraint, not a target: name it
so the Migrator does not trade it away.

/output-contract
Return only this Markdown structure and nothing else:

## Priority Order
1. **name** - what to change, and the observation that shows it worked
2. ...

## Preservation Gate
- behaviour verified in earlier iterations that must not regress

## Acceptance Gate
- the smallest end-to-end run that demonstrates the selected priorities
```

Listing 1: Diagnoser prompt.

```
/role/migrator
You are the Migrator for iteration {loop}. The port lives at `/workspace/
    jax_repo`. Treat the task
document as the requirement and the development document below as this
    iteration's implementation
and validation brief.

/iteration-context
{warm_start}

/source-of-truth
`/workspace/torch_orig` is the original program and the specification.
    Read it and RUN it: it is the
task input, not an answer key, and the fastest way to settle a question
    about intended behaviour is
usually to execute the original and look. Hidden tests, scores and
    rubrics do not exist in this
container; do not look for them.

{instruction}

/development-document
```

```
{document}

/runtime-checks
Deterministic checks already run against the current port:

{runtime_checks}
{skills}
/development-policy
Repair build and runtime blockers first, then work the ordered targets in
     the document.

Establish a baseline before you edit: run the behaviour you are about to
    change, on the original and
on the port, and record what each does. After each meaningful change,
    rerun that path and check the
code adjacent to it for regressions. Keeping the distance between a
    change and its test short is what
makes a failure diagnosable.

Your own testing decides whether the port is ready to be presented as a
    candidate. It does not
establish that the requirements are met -- an independent verifier
    assesses that afterwards from the
frozen artifact. So do not claim completion you have not observed; an
    unsupported claim costs the
next iteration more than an honest gap does.

A comparison harness you write around your own port cannot detect a
    divergence in the port's own
interface: it unwraps what the port returns, compares the numbers inside,
     and reports a match no
matter what container those numbers arrived in. When you compare against
    the original, reach into
the result the way the ORIGINAL's caller does -- same attribute, same key,
     same unpacking -- and let
that access fail if the port does not support it. A port whose numbers
    are perfect and whose return
shape is not the original's is a broken port, and it is the shape that
    your own harness is blind to.

/output-contract
Leave the updated port in `/workspace/jax_repo`, importable and runnable.
     Keep every public name the
original defines, and keep what each of them returns: same container type,
     same fields, reachable by
the same access. Write `/workspace/dev_notes.md` recording, for each
    target: what you changed, what
you ran, and what you observed.
```

Listing 2: Migrator prompt.

```
/role/verifier
You are the Verifier for iteration {loop}. A candidate port has been
    frozen at
`/workspace/jax_repo`; you may read and execute it, but you may not
    modify it. You did not write it,
and its author's claims about it are not evidence.

/source-of-truth
`/workspace/torch_orig` is the original program. It runs. Anything you
    want to know about the
required behaviour, you can find out by running the original and
    comparing.
```

```
{instruction}

/development-document
{document}

/runtime-checks
Deterministic checks run against this frozen candidate:

{runtime_checks}

/migrator-notes
{dev_notes}

/assessment-policy
Derive the claims worth checking from the task document, the original's
    behaviour, and this
iteration's targets and preservation constraints. Do not apply a fixed
    checklist -- what is worth
checking depends on what this port does and what this iteration set out
    to do.

Use both views. Black-box: drive the port through its own entry points
    and compare what comes back
against the original driven the same way -- batches, shapes, dtypes,
    logits, losses, gradients,
parameter trees, whatever the claim needs. White-box: read the source,
    the configuration and the
logs to diagnose a failure or to corroborate an observation whose cause
    is not visible from outputs.

Two things about "driven the same way". Drive the ORIGINAL first and
    write down how its caller gets
at the result -- the type it hands back and the attribute or key each
    value is read from -- then
require the port to answer the same access. Values that match after you
    unwrap each side differently
are not a match; the unwrapping is part of what is being compared. And
    build that comparison
yourself from the original: tooling left in the candidate tree was
    written by its author, around the
author's assumptions, and running it re-checks those assumptions rather
    than the port.

Mark a claim `verified` only when the execution records you cite visibly
    support it. Source code
containing the right-looking implementation is NOT verification. Observed
     failures, regressions,
unmet requirements, and claims you could not gather enough evidence for
    are all `gap` -- never infer
completion from absence of evidence.

/restrictions
Do not read, list, infer from, or summarise hidden tests, benchmark
    scores, private evaluator files,
grading formulas, or any material outside the task document, the original
     program, and the frozen
candidate. None of it is present in this container.

/output-contract
Write `/out/qa_report.json` with exactly this shape (the candidate
    directory is read-only; `/out` is yours):

{
  "iteration": {loop},
  "qa_status": "pass" | "partial" | "fail",
```

```
  "verified_records": [
    {"claim_id": "...", "claim": "...",
      "execution_records": [{"type": "...", "command": "...", "
    observation": "..."}],
      "status": "verified"}
  ],
  "gap_records": [
    {"claim_id": "...", "claim": "...",
      "execution_records": [{"type": "...", "command": "...", "
    observation": "..."}],
      "status": "gap", "impact": "...", "recommended_update": "..."}
  ],
  "diagnoser_handoff": {
    "preservation_constraints": ["..."],
    "update_targets": ["..."],
    "validation_requirements": ["..."]
  }
}

Also write `/out/qa_report.md` with the same findings in prose, and print
     the JSON as your final message so it survives even if the file does
    not. Every record must cite the
command you actually ran and what you actually saw.
```

Listing 3: Verifier prompt.