XStack: A Benchmark for ML Repo Migration

Linrong Cai Taiming Lu Haozhe Chen Zhuang Liu
Princeton University

Dataset Code Website

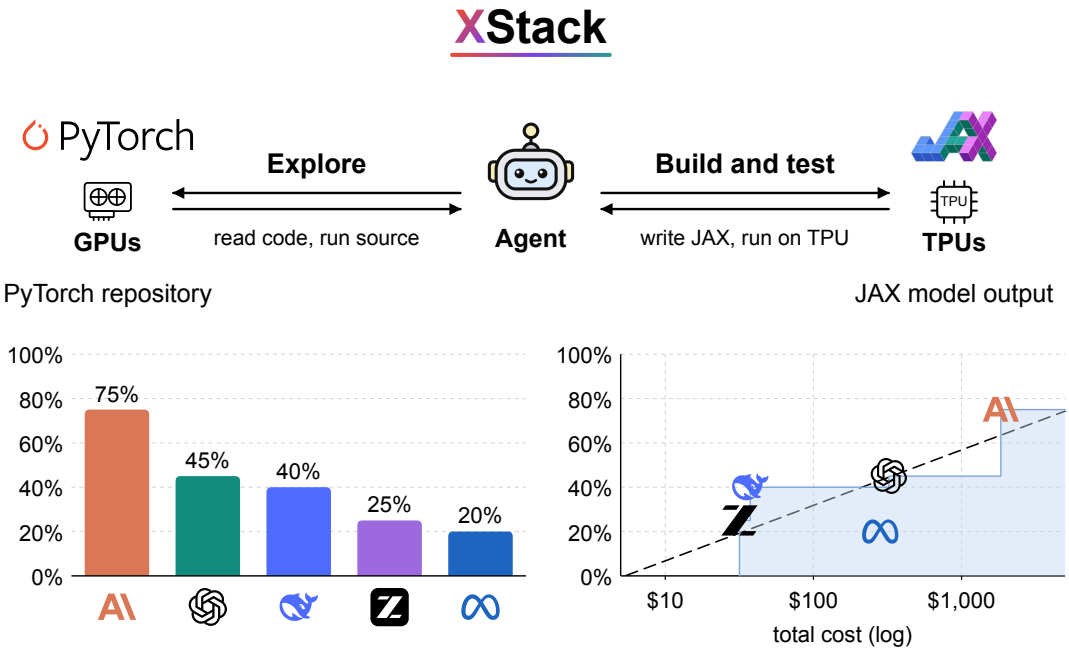


Figure 1. Overview of XStack. Top: in our benchmark, the agent explores a PyTorch repository built for GPUs, reads its code and runs the source, then builds and tests a JAX version on TPUs. Bottom left: pass rates of the 5 best-performing agents among the 7 we evaluate. Bottom right: the pass rates against total cost on a log scale; the dashed line is a linear fit, and the shaded steps are the Pareto frontier, the best pass rate reached at or below each cost.

Abstract

ML code should run where researchers choose, not where it was first written. Today, machine learning code mainly lives in two ecosystems, PyTorch on NVIDIA GPUs and JAX on Google TPUs. Migrating repositories between them means building and testing across frameworks and hardware. This migration is (1) a practical need, since it unlocks different hardware for researchers, and (2) a natural benchmark: it involves different frameworks and hardware, covers whole repositories, and is long-horizon and challenging. We introduce **XStack**, a benchmark that evaluates agents on migrating ML repositories between these two ecosystems. Each agent receives a PyTorch repository and a clean TPU machine and must deliver a JAX repository. Its output passes if it meets our grading criteria, based on the source repository's published result, authenticated and calibrated by us. Migration by agents remains unreliable: Claude Opus 5 passes 75%, GPT-5.6-sol 45%, and every other agent at most 40%. XStack is a step toward freeing ML code from its ecosystem. We release XStack to make progress toward reliable migration across frameworks and hardware measurable.

1 Introduction

Suppose you find a repository that does exactly what you need. Its paper reports strong results, the code is public, and the experiment has already been shown to work. There is only one problem: it was built for the wrong machine. Your lab has TPUs, but the repository assumes GPUs and PyTorch. Although, in principle, nothing about the scientific idea depends on this choice, in practice you are trapped. Using it means rewriting the model, optimizer, data pipeline, and evaluation in JAX, since that is the stack TPUs run, then making everything work under a different compiler, memory hierarchy, and distributed execution model. What looks like an implementation detail becomes a barrier to reusing the science. The hardware a team happens to have can determine which published ideas it can build on.

This is exactly the kind of barrier that coding agents should make disappear. Much published ML code is written in PyTorch for GPUs, while, for example, people also use TPUs, which are commonly programmed through JAX[1]. By the end of 2025, TPUs made up nearly a fifth (18.2%) of the world's AI compute: 3.81 million of 20.93 million H100-equivalents (Epoch AI, 2026). Yet only 4.2% of the framework-based code released with ML papers in 2024 used JAX (Papers with Code, 2025). This creates a growing need to move working systems across software and hardware stacks, a real-world task that researchers care about because it unlocks access to different hardware. Doing so means crossing not only code-level framework boundaries, but also machine-level differences. This is exactly the problem we want agents to solve; we want to bring pure science back to researchers, and provide a measurement for this progress.

	PyTorch on GPUs	**JAX on TPUs**
chips made by	NVIDIA and others	Google
where to get them	most clusters and clouds	Google Cloud or buyers' own sites
how the code runs	one step at a time	compiled by XLA before execution
how a model is written	objects that hold their weights	functions that take the weights

Table 1. PyTorch on GPUs versus JAX on TPUs. A migration changes the hardware as well as the code, so the agent must do more than translate code (Bradbury et al., 2018; Paszke et al., 2019).

We ask the following question directly:

Can an agent migrate a working ML repository to a new framework and new hardware correctly?

Existing benchmarks do not quite test this. Issue-resolution benchmarks (Jimenez et al., 2024; Zan et al., 2025) start from broken code and ask agents to fix it, while conversion benchmarks (Wang et al., 2026c; Song et al., 2026; Hong et al., 2026) compare translated code with the original on the same machine. In our case, the repository already works on the original machines. The challenge is to preserve that behavior while moving the entire system onto hardware the original code does not support. We introduce **XStack** (XStack stands for Cross-Stack Benchmark) to test this directly. Each task gives an agent a working PyTorch repository and a clean TPU machine, then asks it to rebuild the code on TPU with JAX and make an original experiment run end to end and reach its published result.

The answer to our question above is currently no: agents are not fully reliable for this task. Our results show that this barrier is beginning to break, but has not disappeared. The strongest agent successfully migrates 15 of 20 repositories. Agents can therefore already carry some ML systems across framework and hardware boundaries at repository scale. Yet this capability is far from reliable across agents and repositories. Agents are beginning to loosen the link between scientific code and the stack it was written for, but we are still not at a point where researchers can simply run published work wherever they choose.

A broader pattern emerges from the failures: agents can write a great deal of code, but they are still much less reliable at verifying that what they built is faithful, complete, and actually works in the real environment. This shows up across models as missed evaluations, fidelity gaps, crashes, and even attempts to work around the task when they get stuck.

[1] JAX is Google's Python library for array computation; it compiles NumPy-style code with the XLA compiler to run on TPUs and GPUs (Bradbury et al., 2018).

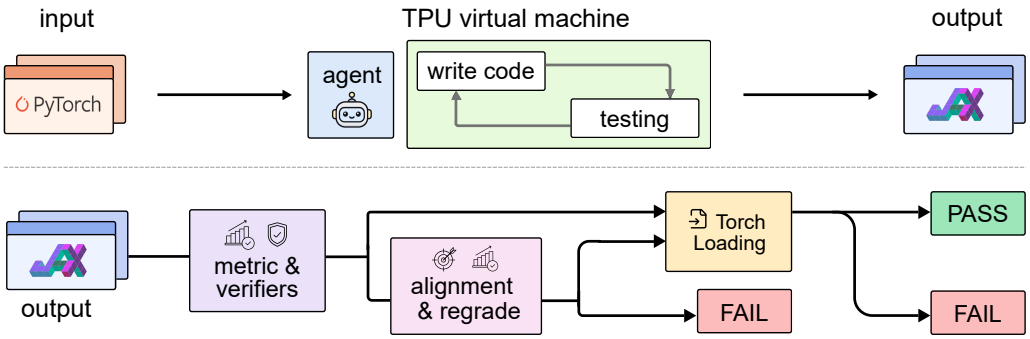


Figure 2. Pipeline of XStack. Top: the agent migrates a PyTorch repository to JAX on a bare TPU. Bottom: we rerun the output on a fresh TPU and check its number against the source's result and with two model judges. If they fail, the alignment step (phase 2, pink) gives the agent the main experiment's recipe, lets it change only the entry script, and regrades the result. A pass also needs the weights to load into the original PyTorch model.

Overall, our contributions are:

- We introduce **XStack**, a benchmark of 20 real repositories for testing whether agents can migrate working ML systems across frameworks and hardware.
- We evaluate seven coding agents on real TPUs and release the grading criteria and full agent trajectories, so progress on this capability is measurable.
- We find that agents can sometimes complete these migrations, but still struggle to reliably verify that what they built is faithful, complete, and actually works end to end. We also observe that pass rate grows linearly with the log of an agent's cost (Figure 6).

2 The XStack Benchmark

Each task in XStack is a PyTorch repository that already works and reports a result (Figure 2). The agent migrates it to JAX on a bare TPU machine, and we run the migrated code again on a fresh machine to check that it meets our grading criteria. For fairness, grading has two phases: if the first attempt fails, a second phase tells the agent which experiment we grade, so a correct migration is not failed only for running a different experiment (Section 2.2). The numerical target stays hidden, and only the entry script may change; all other code and setup remain frozen. Section 2.1 defines the task, Section 2.2 explains how we grade it, and Section 2.3 describes our repositories and agents.

2.1 Task Formulation

Input. The agent receives the repository, pinned to one git commit so that every agent sees the same code, and a TPU machine. One prompt template, the same for every repository, asks it to migrate the entire repository to JAX, with Flax for the model and Optax for the optimizer (Bradbury et al., 2018; Heek et al., 2020; DeepMind et al., 2020) (Table 1), and to run it the way the source runs its experiments, at full scale. For four repositories whose source offers several configurations, one or two added lines name the graded configuration or a flag the grader uses, so that every agent knows what it has to do without learning the grading target (Appendix B).

Machine setup. The machine has no machine learning software; setting it up is part of the task: the agent installs every package, fetches the data and decides how to use its hosts. It works from a sandbox on our cluster, reaches the machine over SSH and has no time limit while it produces output (Appendix C).

Output. The agent delivers the migrated repository with three scripts. `setup_env.sh` prepares a clean machine, the entry script `run_train.py` trains, evaluates and writes metrics, and `export_torch.py` saves the

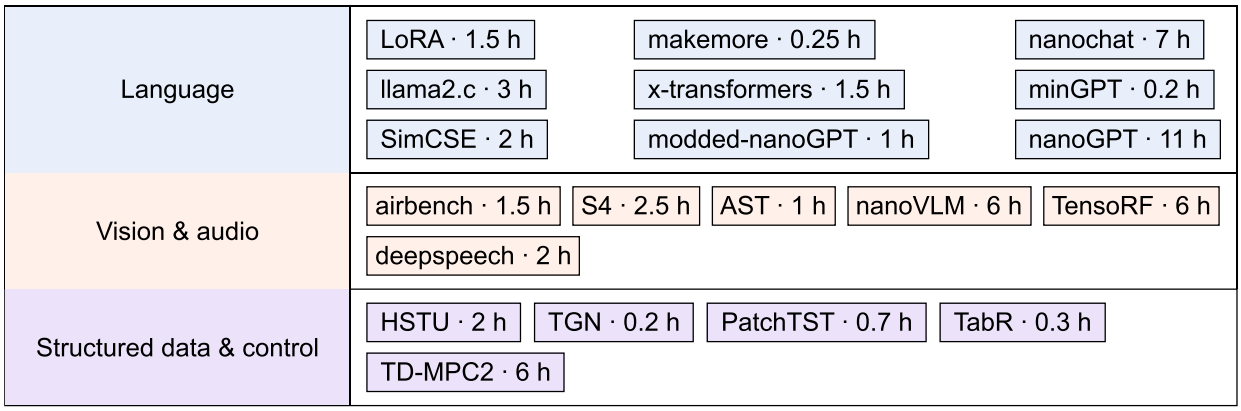


Figure 3. Repositories of XStack. Rows group the 20 repositories into three task families. The number beside each name is the expected duration, in hours, of one graded run, which trains the model at the source's full scale.

	agent		
	phase 1	phase 2	grader
source repository	✓	✓	✓
TPU machine	✓	✓	fresh
task prompt	✓	✓	×
experiment recipe	×	✓	✓
grading target	×	×	✓
grading code	×	×	✓
outcome	×	×	✓

Table 2. What the agent and the grader see. The agent never sees the grading target, so it has to migrate the whole repository; phase 2 adds the main experiment's recipe to ensure grading fairness.

weights under the source's parameter names. In phase 1 the agent is not told which number we grade, what value to reach or whether it passed (Table 2), so it has to migrate the whole repository, not one experiment.

2.2 Grading a Migration

Reaching the grading target. We copy the migrated repository, without its git history, to a freshly wiped machine and run `setup_env.sh` and `run_train.py` once (the purple stage of Figure 2). A script, the grader, compares the graded number with the grading target, usually the number the source published, with a tolerance (Appendix A). Each grading target follows the source's published result, which we authenticated against the pinned source; for 3 repositories we set it from our own runs of the unmodified source, and for minGPT as a rule (Section 2.3). A faithful migration reaches the grading target by reproducing the repository, without knowing the target. A run that writes no metrics within ten times its expected run time fails.

Judges. A number can also reach the grading target for the wrong reason, such as a smaller variant or another run's files. Two model judges, both GPT-5.6-sol, therefore review every result that reaches the grading target (the verifiers of Figure 2). The correctness judge checks that the number comes from this run's training of the main experiment, and the cheat judge checks that the agent did not reach it by cheating, for example by clamping the metric or using another run's files. A result passes only if it reaches the grading target, both judges accept it, and its weights load into the source model.

Two phases. With the criteria hidden, an agent can migrate the code correctly and still run an experiment we do not grade, such as AST's ESC-50 recipe instead of Speech Commands (Warden, 2018). Scoring that as a failure would penalize a choice the task never specified. Phase 2 removes this unfairness without making the

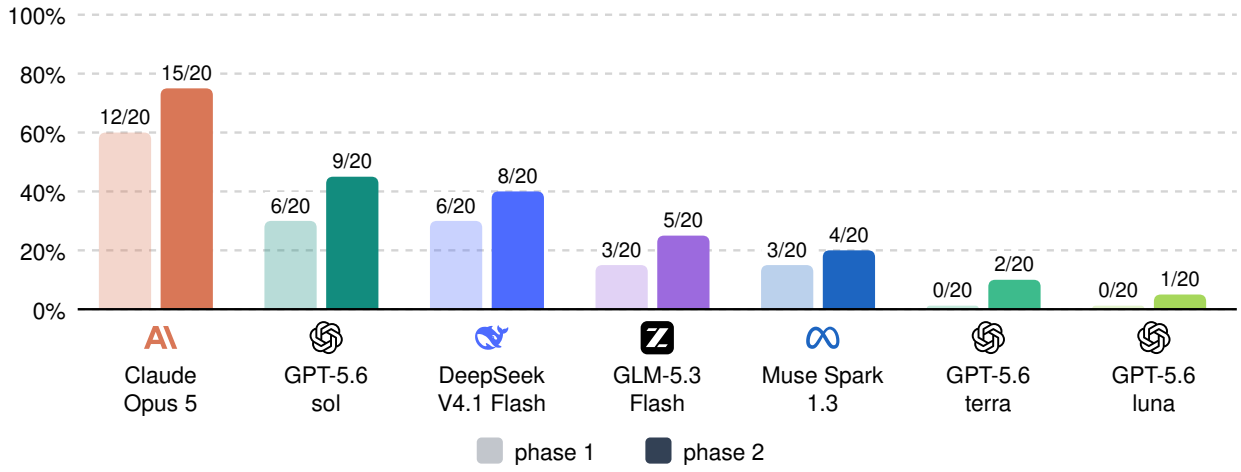


Figure 4. Pass rate per agent. Light bars: after phase 1; dark bars: after phase 2. Phase 2 adds a few passes for every agent: once aligned with the main experiment, each agent passes more repositories.

task easier: after a failed phase 1, one message (the alignment step of Figure 2) freezes everything except the entry script and gives the recipe of the main experiment, but not the grading target (Table 2). Changing any other file fails phase 2 as a *freeze violation*, and the judges fail a pass whose entry script rebuilds the frozen code, a broken freeze. We report the pass rate after phase 2 as the main result.

2.3 Repositories and Agents

Repositories. XStack contains 20 repositories from fourteen domains (Figure 3; Table A1 gives each grading target). Seven are language models: minGPT (Karpathy, 2020), nanoGPT (Karpathy, 2022b), makemore (Karpathy, 2022a), llama2.c (Karpathy, 2023), modded-nanoGPT (Jordan et al., 2024), nanochat (Karpathy, 2025) and x-transformers (Wang, 2020b). The others are LoRA (Hu et al., 2022), SimCSE (Gao et al., 2021), airbench (Jordan, 2024), AST (Gong et al., 2021), S4 (Gu et al., 2022), HSTU (Zhai et al., 2024), TD-MPC2 (Hansen et al., 2024), TensoRF (Chen et al., 2022), nanoVLM (Wiedmann et al., 2025), deepspeech (Naren, 2017), TGN (Rossi et al., 2020), PatchTST (Nie et al., 2023) and TabR (Gorishniy et al., 2024). We chose them from more than 200 PyTorch repositories, most of which we rejected because of their published number.

Grading targets. For 15 repositories, the grading target is the number that the README or paper reports, which we authenticated against the source at its pinned commit; deepspeech's comes from its release notes. A fair target must be one that the original code itself reaches, and our experiments confirm that each of these targets can be reached. For 3 others, we set the target from our own GPU runs of the unmodified source. For example, the makemore README reports a loss of about 1.92, but eight runs of the unmodified source on our GPUs average 1.989 with a standard deviation of 0.017, so a target of 1.92 would fail the original code itself. We therefore set its target to a best test loss of at most 2.023, the mean plus two standard deviations. For minGPT, whose task trains a small GPT to add two-digit numbers, we require at least 495 of the 500 held-out sums to be correct: a trained model is expected to solve all 500, and the margin allows five misses for seed and numerical noise.

Agents and cells. We evaluate 7 agents from 5 labs (Table 3): GPT-5.6-sol, GPT-5.6-terra and GPT-5.6-luna (OpenAI, 2026), Claude Opus 5 (Anthropic, 2026), GLM-5.3-Flash (Z.ai, 2026; GLM-5 Team, 2026), DeepSeek V4.1 Flash (DeepSeek-AI, 2026) and Muse Spark 1.3 (Meta, 2026), all at high reasoning effort. We call a pair of an agent and a repository a *cell*. Each cell runs on a TPU v6e, v5p or v4 machine (Jouppi et al., 2023; Google Cloud, 2026a,b), and we ensure every repository has adequate compute (Appendix H; Figure H1).

agent	harness	reasoning effort	list price
GPT-5.6-sol	Codex CLI	high	5 / 0.5 / 30
GPT-5.6-terra	Codex CLI	high	2 / 0.2 / 12
GPT-5.6-luna	Codex CLI	high	0.2 / 0.02 / 1.2
Claude Opus 5	Claude Code CLI (Anthropic)	high	5 / 0.5 / 25
GLM-5.3-Flash	Claude Code CLI (Z.ai)	high	0.15 / 0.03 / 0.5
DeepSeek V4.1 Flash	Claude Code CLI (DeepSeek)	high	0.3 / 0.006 / 1.2
Muse Spark 1.3	Claude Code CLI (Meta)	high	1.25 / 0.15 / 4.25

Table 3. Harness, reasoning effort and price per agent. All agents run at high reasoning effort with the same settings, and every model is called through its developer's official API. Prices are the official list prices on September 14, 2026, USD per million tokens (input / cached input / output).

3 Results Analysis

3.1 Main Results

Phase 1. In phase 1, the agents pass 30/140 cells (21%; Figure 4). The grader passed 58 phase 1 results, and the judges rejected 28 of these. Claude Opus 5 leads by a wide margin with 12/20. DeepSeek V4.1 Flash follows with 6/20, and among the GPT-5.6 tiers only sol passes any repository (6/20). Migrating a whole repository is thus still hard for current agents: most attempts fail, even though every agent has the full source and its own TPU, on which it can test extensively.

Phase 2 runs on 108 cells that failed phase 1. The grader passed 44 of them, and the judges let 16 of these pass. These are the cases the phase was designed for, such as a fixed number of steps where the source trains until a stop rule, a training statistic reported where the source runs an evaluation, or another variant of the source's recipe. The judges rejected 28 phase 2 passes, 21 of them because the entry script rebuilt the frozen code, and the remaining phase 2 failures lie in the frozen code (Section 2.2). After phase 2, 46 cells pass the grader and the judges. The weights of 2 of them do not load under the source's parameter names, so the agents pass 44/140 cells (31%). Appendix E lists every cell's result.

Phase 2 aligns the agent with the recipe of the experiment we grade, so a correct migration that first ran a different experiment is not failed for that choice. Every agent gains at least one pass from it. It also helps ensure the remaining failures lie in the migrated code, not in an under-specified task (Section 2.2). Running all 140 cells took 22,802 TPU chip-hours (Figure H1), and the agents' trajectories sum to 12.5 billion tokens.

3.2 Judges, Cost and Behavior

Effect of the judges. The judges are needed, because they catch results that reach the grading target for the wrong reason, which the metric alone cannot detect. Across both phases, they rejected 56 of the 102 results that the grader passed (Figure 5; Table 4). We checked many rejections by hand and found them sound; the judge caught a llama2.c migration that wrote the published number into its metrics beside its own and a minGPT migration evaluated on its training data.

What the judges find. Apart from broken freezes, the judges catch two kinds of false positives. The grader reads whatever number the run reports, which may not be the recipe's own, such as a separate validation pass in place of the recipe's averaged loss; the correctness judge finds the recipe's number in the run's logs and checks it, so every pass is a true positive. The cheat judge catches cheating, such as training on another task's machine or reading another session's files.

Accuracy versus cost. Pass rate grows linearly with the logarithm of cost (Figure 6). We fit pass rate on log cost because the agents' costs for the 20 repositories span from \$10.0 to \$1,844. Every 10× in cost adds about 25 percentage points, and the line explains most of the difference between agents ($R^2 = 0.66$, $r = 0.81$,

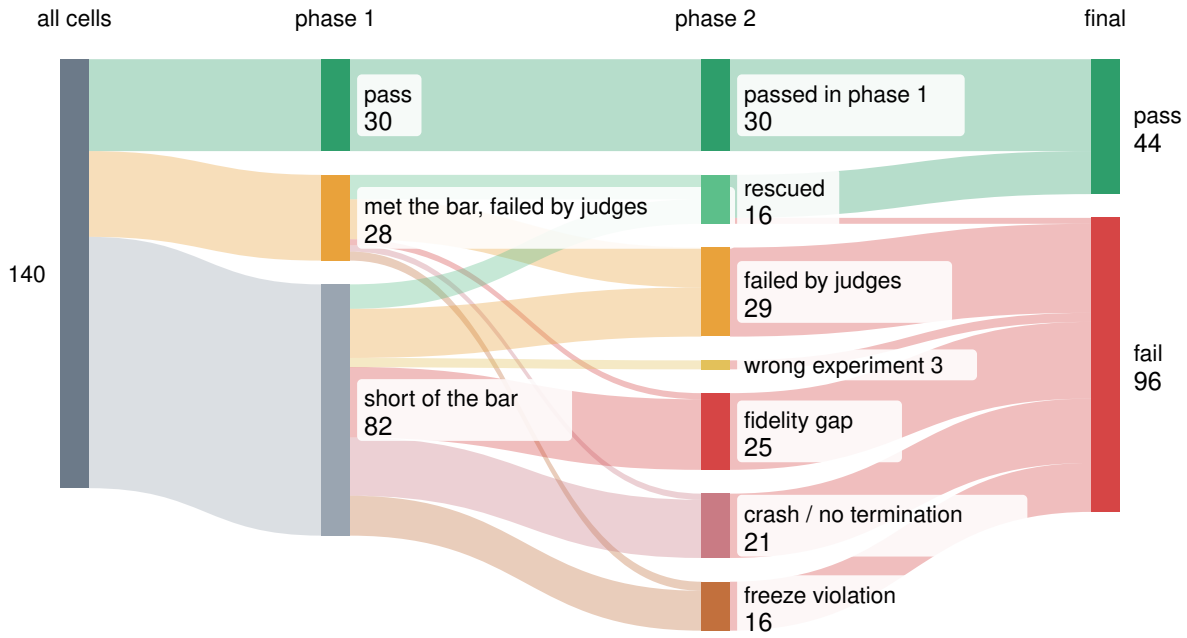


Figure 5. Outcomes by phase. From left to right, the columns show all 140 cells, their phase 1 outcome, their phase 2 outcome and the final result. Of the 110 cells that failed phase 1, phase 2 rescues 16; the rest fail the judges, fall short of the grading target, crash or break the freeze (Section 2.2). Two rescued cells fail the weight-loading check.

$p = 0.027$, $n = 7$ agents). The trend is statistically significant at the 5% level. The upper edge of the shaded area, the efficient frontier, runs through three agents: GPT-5.6-luna, the cheapest; DeepSeek V4.1 Flash, the cheapest per pass; and Claude Opus 5, the most accurate. Every other agent lies below it; Muse Spark 1.3, for example, costs more than seven times as much as DeepSeek V4.1 Flash and passes fewer repositories.

	pass rate				cost and time				
agent	target	+correctness	+cheat	+weights	pass	tokens	$/cell	$/pass	hours
Claude-Opus-5	**95%**	**80%**	**80%**	**75%**	**15/20**	**156M**	**92**	**123**	8.7
GPT-5.6-sol	75%	70%	45%	45%	9/20	23M	16	36	1.0
DS-4.1-Flash	75%	60%	40%	40%	8/20	106M	1.9	4.7	11.1
GLM-5.3-Flash	60%	50%	25%	25%	5/20	43M	1.6	6.3	**12.6**
Muse-Spark-1.3	50%	35%	20%	20%	4/20	32M	14	71	4.1
GPT-5.6-terra	35%	25%	15%	10%	2/20	5.3M	2.0	20	0.4
GPT-5.6-luna	20%	15%	5%	5%	1/20	16M	0.5	10.0	0.6
all	59%	48%	33%	31%	44/140	54M	18	58	5.5

Table 4. Checks and cost per agent. Left: share of repositories passing each check in turn (grading target, correctness and cheat judges, weight loading). **Right**: repositories passed, mean tokens per cell (input, cached input and output over all turns, as billed), list-price dollars per cell and per pass, and mean agent hours per cell.

Behavior difference between agents. Reading the trajectories, we found that Claude Opus 5 differs from the other agents in how it checks its work: before handing in, it runs the source's experiment to the end and compares its number with the published one. This checking takes time and tokens, and Opus is both the most expensive agent and the one that passes the most (Table 4). On nanochat, GPT-5.6-sol estimated the

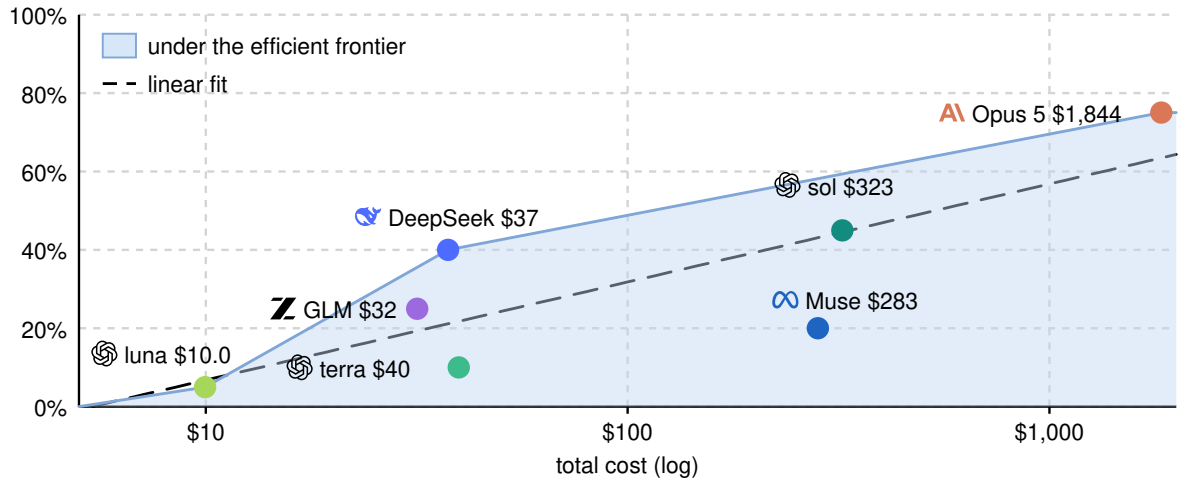


Figure 6. Accuracy against cost. Each dot is one agent's final pass rate against its cost at the list prices of September 14, 2026 (Appendix F). The dashed line is a least-squares fit of pass rate on log cost. The blue line is the efficient frontier, the upper convex hull of the agents with cost on a log scale. Pass rate rises by about 25 percentage points with every 10× in cost.

full run at a few hours and handed in without running it. Muse Spark 1.3 does run the full experiment, but on LoRA it handed in a BLEU score far below the paper's and called its migration "fully validated". We find that diligent testing and experimentation, together with careful comparison against the original code, are what a migration needs to succeed.

4 Why Do Agents Fail?

4.1 Failure Taxonomy

A third judge, the failure judge, assigns every graded run one class from a fixed taxonomy (Appendix D), based on its metrics, its log and the diff of the migrated code. The failure judge only reads the run; it cannot determine whether the run passes or fails. Figure 7 shows the failure judge's classification of phase 1 runs.

The code fails, not the choice of experiment. We group the classes by what could fix them. A wrong or incomplete experiment (a different pipeline, the right one at a different scale, or a run that never reaches the main evaluation) is a problem with what the entry script runs, and phase 2 can fix it. A fidelity gap, a crash or a run that never ends is a problem in the migrated code. In phase 1, 27% of the failures are wrong or incomplete experiments and 47% are code failures; the judges reject the rest. Phase 2 removes the first group. The cells it does not rescue still fail in the code, or they break the freeze by rebuilding the migrated code inside the entry script (Figure D1). Aligning the agent with the grading recipe does not make a cell pass if its migration is wrong in the first place.

Agents fail differently. DeepSeek V4.1 Flash rarely crashes (1 cell), and 15/20 of its cells reach the grading target, but the judges reject half of these, mostly for cheating. Terra's failures are spread over wrong experiments, fidelity gaps and crashes, and luna's over almost every class (Figure D1).

Translation alone is not the hard part. Because JAX also runs on GPUs, we can also test translation between PyTorch and JAX alone, on the same hardware. In the earlier development of our benchmark, we asked for a whole JAX repository and graded it on GPU. There the migration was nearly solved: GPT-5.5 at high reasoning effort migrated 12 PyTorch repositories to JAX and passed 99% of the checks, 116 of 117, with 11 of the 12 repositories passed in full (Appendix G, Table G1). The checks are strict: forward and gradient parity with PyTorch, checkpoints that load in both frameworks, decoding that matches token for token, and one miss fails the repository.

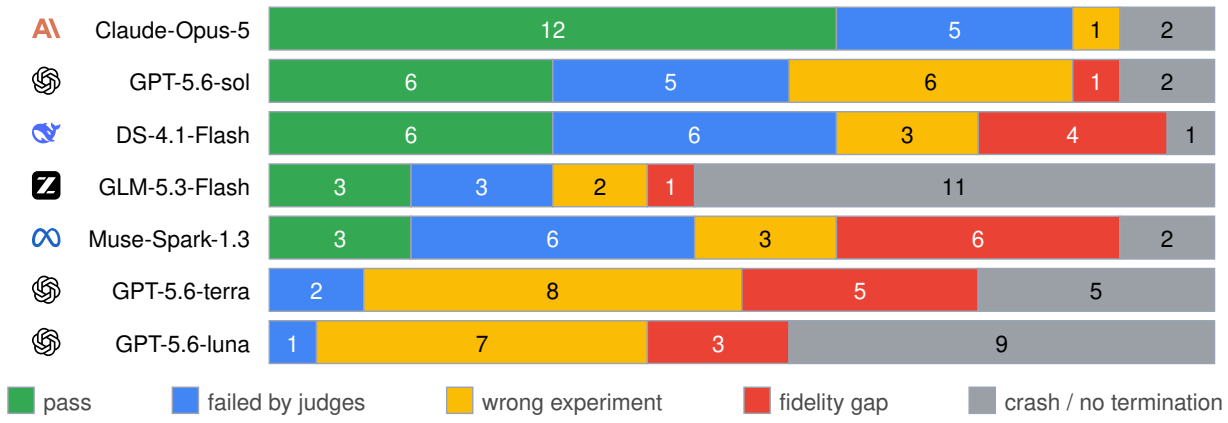


Figure 7. Failure breakdown after phase 1. Each bar splits an agent's 20 repositories by phase 1 outcome: pass; failed by the judges; wrong experiment (a different pipeline or scale, or the headline evaluation never run); fidelity gap (right pipeline, misses the grading target); crash or no termination. The last two, 47% of failures, are faults in the migrated code, which phase 2 cannot repair.

On TPUs, even after phase 2, the agents' average pass rate is only 31%, and about half the failures are code that crashes or falls short on a machine the agent set up. The comparison suggests the difficulty lies not in translation alone but in migrating across both framework and hardware. Testing this ability therefore requires migrating across hardware, which is exactly what XStack does.

4.2 Behavior and Integrity

Agents that pass follow the source's documentation, running the README's default recipe, reporting its numbers under its names, and getting results close to the source's. Agents that get stuck explore the machine instead, reading other cells' leftovers or using our credentials. We found these leaks in our initial experiments, hardened the sandbox, and reran the three affected cells cleanly. Two incidents that aimed at the grading target, a clamped metric and a copied checkpoint, also date from building the benchmark and affect neither the current benchmark nor its results. In the final benchmark, cheating incidents instead use what the source offers, such as a sibling variant or the published number, and our checks and the cheat judge catch them.

5 Related Work

Code migration. Code translation has grown from single functions and small programs (Roziere et al., 2020; Pan et al., 2024) to whole repositories (Ibrahimzada et al., 2025; Wang et al., 2026c; Li et al., 2026), although some repository benchmarks still grade one function in context (Ou et al., 2025) or fill in a skeleton (Zhang et al., 2025; Khatry et al., 2025). Migration benchmarks upgrade a language version (Liu et al., 2026) or move a repository onto another software stack (Hong et al., 2026).

Framework conversion. For machine learning code, framework conversion has been studied for single snippets (Phan et al., 2026), short programs (Yan et al., 2023) and training codebases (Song et al., 2026), or avoided with a portability layer (Lenton et al., 2021). T2J-Bench (Song et al., 2026) is the closest: whole PyTorch training codebases to JAX under a hidden evaluator, but on one GPU, the framework-only setting our GPU experiments found nearly solved (Section 4.1). Kernel benchmarks write kernels for one accelerator (Ouyang et al., 2025; Tschand et al., 2026; Wang et al., 2026b), one operator at a time against a reference that already runs on it. Models do worse on hardware rare in their training data (Wen et al., 2025). In all of these the accelerator stays the same or the unit is one kernel. XStack moves a whole repository to a new framework and hardware and forbids PyTorch at runtime, so no agent passes by doing nothing.

Agent benchmarks. Since SWE-bench (Jimenez et al., 2024), issue-resolution benchmarks have graded a change with tests that fail before it and pass after it. Later benchmarks ask for a codebase built from scratch (Zhao et al., 2025; Yang et al., 2026). A working repository, however, has no failing test in its own environment. Most also give the agent a ready environment; SetupBench (Arora et al., 2025) and DeployBench (Wang et al., 2026d) test setup alone, but stop once the code runs. Closer to our setting are benchmarks graded against research results. Agents reproduce a paper's results from the authors' code (Siegel et al., 2024), replicate papers against rubrics or reference code (Starace et al., 2025; Seo et al., 2026), or maximize a score on machine learning tasks (Chan et al., 2025; Wijk et al., 2025). Our grading target, in contrast, is the source repository's own result (Section 2.3), which a faithful migration will naturally reach.

Evaluation integrity. Agents exploit underspecified objectives (Krakovna et al., 2020; Skalse et al., 2022) and game checks they can see or tamper with (Zhu et al., 2025; Wang et al., 2026a); benchmarks built from public code risk memorization (Jain et al., 2025; Liang et al., 2026). In XStack, the grading target is hidden: the agent is never told it or whether it passed. For fairness, after a failed first attempt a second phase aligns the agent with the graded experiment while its code, except the entry script, stays frozen. We rerun the code on a fresh machine, so remembering an answer is not enough, and model judges review every result that reaches the target (Section 2.2).

6 Conclusion

We introduce **XStack**, a benchmark that evaluates agents' ability to migrate working ML repositories from PyTorch on GPUs to JAX on TPUs. Across 7 agents from 5 labs, the average pass rate is 31%, and pass rate grows linearly with the logarithm of cost. Our analysis shows that verification separates success from failure: agents that run the source's experiment to the end and compare their result with the original tend to pass, and agents that hand in untested code tend to fail. We encourage other benchmarks with hidden criteria to adopt a similar frozen second phase that aligns the agent with what is graded, so that they stay fair to agents. Although current agents are not yet reliable at migrating across both frameworks and hardware, we hope they will one day handle this task reliably and unlock access to different hardware for researchers. In the meantime, XStack will help measure progress toward this hardware freedom.

Acknowledgements

This work was supported by Princeton Research Computing resources, including the Della high-performance computing cluster. We thank Princeton Language and Intelligence (PLI) for their support of this research, as well as Google's TPU Research Cloud (TRC) program for providing TPU compute.

References

Anthropic. System card: Claude Opus 5. https://www.anthropic.com/claude-opus-5-system-card, 2026. Accessed 2026-09-25.

Avi Arora, Jinu Jang, and Roshanak Zilouchian Moghaddam. SetupBench: Assessing software engineering agents' ability to bootstrap development environments. *arXiv preprint arXiv:2507.09063*, 2025.

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. http://github.com/jax-ml/jax, 2018.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *ICLR*, 2025. arXiv:2410.07095.

Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. TensoRF: Tensorial radiance fields. In *ECCV*, 2022.

DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, et al. The DeepMind JAX ecosystem. http://github.com/google-deepmind, 2020.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. *arXiv preprint arXiv:2609.19969*, 2026.

Epoch AI. Data on AI chip owners. Epoch AI, https://epoch.ai/data/ai-chip-owners, 2026. Accessed 2026-09-25.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *EMNLP*, 2021.

GLM-5 Team. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Yuan Gong, Yu-An Chung, and James Glass. AST: Audio spectrogram transformer. In *Interspeech*, 2021.

Google Cloud. TPU v5p. https://docs.cloud.google.com/tpu/docs/v5p, 2026a. Accessed 2026-09-23.

Google Cloud. TPU v6e. https://docs.cloud.google.com/tpu/docs/v6e, 2026b. Accessed 2026-09-23.

Yury Gorishniy, Ivan Rubachev, Nikolay Kartashev, Daniil Shlenskii, Akim Kotelnikov, and Artem Babenko. TabR: Tabular deep learning meets nearest neighbors. In *ICLR*, 2024.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *ICLR*, 2022.

Nicklas Hansen, Hao Su, and Xiaolong Wang. TD-MPC2: Scalable, robust world models for continuous control. In *ICLR*, 2024.

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. http://github.com/google/flax, 2020.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. SWE refactor bench: Can coding agents complete a long-horizon, whole-repository stack migration? *arXiv preprint arXiv:2608.23564*, 2026.

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. In *ICLR*, 2022.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. AlphaTrans: A neuro-symbolic compositional approach for repository-level code translation and validation. In *FSE*, 2025. arXiv:2410.24117.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *ICLR*, 2025. arXiv:2403.07974.

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 *ICLR*, 2024.

Keller Jordan. 94% on CIFAR-10 in 3.29 seconds on a single GPU. *arXiv preprint arXiv:2404.00498*, 2024.

Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, et al. modded-nanoGPT: Speedrunning the NanoGPT baseline. https://github.com/KellerJordan/modded-nanogpt, 2024.

Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *ISCA*, 2023.

Andrej Karpathy. minGPT. https://github.com/karpathy/minGPT, 2020.

Andrej Karpathy. makemore. https://github.com/karpathy/makemore, 2022a.

Andrej Karpathy. nanoGPT. https://github.com/karpathy/nanoGPT, 2022b.

Andrej Karpathy. llama2.c. https://github.com/karpathy/llama2.c, 2023.

Andrej Karpathy. nanochat. https://github.com/karpathy/nanochat, 2025.

Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. CRUST-Bench: A comprehensive benchmark for C-to-safe-Rust transpilation. In *COLM*, 2025. arXiv:2504.15254.

Thomas Kipf. pygcn: Graph convolutional networks in PyTorch. https://github.com/tkipf/pygcn, 2017.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: The flip side of AI ingenuity. DeepMind blog, https://deepmind.google/blog/specification-gaming-the-flip-side-of-ai-ingenuity/, 2020.

Daniel Lenton, Fabio Pardo, Fabian Falck, Stephen James, and Ronald Clark. Ivy: Templated deep learning for inter-framework portability. *arXiv preprint arXiv:2102.02886*, 2021.

Xuefeng Li, Nir Ben-Israel, Yotam Raz, Belal Ahmed, Doron Serebro, and Antoine Raux. RepoMod-Bench: A benchmark for code repository modernization via implementation-agnostic testing. In *KDD*, 2026. arXiv:2602.22518.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-bench illusion: When state-of-the-art LLMs remember instead of reason. In *ICSE-SEIP*, 2026. arXiv:2506.12286.

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. In *KDD*, 2026. arXiv:2505.09569.

Meta. Introducing Muse Spark 1.3. https://research.meta.ai/blog/introducing-muse-spark-1-3, 2026. Accessed 2026-09-25.

Sean Naren. deepspeech.pytorch. https://github.com/SeanNaren/deepspeech.pytorch, 2017.

Yuqi Nie, Nam H. Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers. In *ICLR*, 2023.

OpenAI. GPT-5.6 system card. https://deploymentsafety.openai.com/gpt-5-6, 2026. Accessed 2026-09-25.

Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Yanlin Wang, Xin Peng, and Zibin Zheng. RustRepoTrans: Repository-level context code translation benchmark targeting Rust. In *ASE*, 2025. arXiv:2411.13990.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *ICML*, 2025. arXiv:2502.10517.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *ICSE*, 2024.

Papers with Code. Trends: Paper implementations grouped by framework. Papers with Code, archived 2025-06-09, https://web.archive.org/web/20250609230938/https://paperswithcode.com/api/stats/?type=quarter&from=2021-03-01&to=2025-03-31, 2025. Accessed 2026-09-25.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Hung Phan, Son Vu, Tuan Dinh, Nesreen Ahmed, Ali Payani, and Ali Jannesari. Learning bug context for PyTorch-to-JAX translation with LLMs. In *ICML Workshop on Deep Learning for Code (DL4C)*, 2026. arXiv:2510.09898.

PyTorch Contributors. PyTorch examples. https://github.com/pytorch/examples, 2016.

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *NeurIPS*, 2020.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating code generation from scientific papers in machine learning. In *ICLR*, 2026. arXiv:2504.17192.

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-Bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *TMLR*, 2024. arXiv:2409.11363.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *NeurIPS*, 2022.

Linxin Song, Jiefeng Chen, Yue Huang, Bhavana Dalvi Mishra, Chi Wang, Jieyu Zhao, Jinsung Yoon, and Tomas Pfister. Converted, not equivalent: Benchmarking codebase conversion via observational equivalence. *arXiv preprint arXiv:2605.29054v2*, 2026.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, et al. PaperBench: Evaluating AI's ability to replicate AI research. In *ICML*, 2025. arXiv:2504.01848.

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. In *ICML Workshop on Deep Learning for Code (DL4C)*, 2026. arXiv:2607.20466.

Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? systematically auditing AI agent benchmarks with BenchJack. *arXiv preprint arXiv:2605.12673*, 2026a.

Phil Wang. denoising-diffusion-pytorch. https://github.com/lucidrains/denoising-diffusion-pytorch, 2020a.

Phil Wang. x-transformers. https://github.com/lucidrains/x-transformers, 2020b.

Shangkun Wang, Nina Cai, Charles Hong, Julian Walker, Gerson Kroiz, George Vanica, Deepak Patil, Andi Gavrilescu, Hassan Sipra, and Sethu Sankaran. MaxKernel: Agentic kernel generation for TPUs. *arXiv preprint arXiv:2609.04523*, 2026b.

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. *IEEE Transactions on Software Engineering*, 2026c. arXiv:2412.17744.

Yuanli Wang, Yaoyao Qian, Yue Zhang, Hanhan Zhou, Jindan Huang, Tianfu Fu, et al. DeployBench: Benchmarking LLM agents for research artifact deployment. *arXiv preprint arXiv:2606.05238*, 2026d.

Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025.

Luis Wiedmann, Aritra Roy Gosthipaty, and Andrés Marafioti. nanoVLM. https://github.com/huggingface/nanoVLM, 2025. Hugging Face.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, et al. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts. In *ICML*, 2025. arXiv:2411.15114.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of EMNLP*, 2023.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch? In *ICML Workshop on Deep Learning for Code (DL4C)*, 2026. arXiv:2605.03546.

Xingkai Yu. Nano-vllm. https://github.com/GeeeekExplorer/nano-vllm, 2025.

Z.ai. GLM-5.3-Flash: Frontier intelligence, flash cost. https://z.ai/blog/glm-5.3-flash, 2026. Accessed 2026-09-25.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *NeurIPS Datasets and Benchmarks Track*, 2025. arXiv:2504.02605.

Jiaqi Zhai, Lucy Liao, Xing Liu, Yueming Wang, Rui Li, Xuan Cao, Leon Gao, Zhaojie Gong, Fangda Gu, Jiayuan He, Yinghai Lu, and Yu Shi. Actions speak louder than words: Trillion-parameter sequential transducers for generative recommendations. In *ICML*, 2024.

Xing Zhang, Jiaheng Wen, Fangkai Yang, Yu Kang, Pu Zhao, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Skeleton-guided-translation: A benchmarking framework for code repository translation with fine-grained quality evaluation. In *Findings of EMNLP*, 2025. arXiv:2501.16050.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *ICLR*, 2025. arXiv:2412.01769.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, et al. Establishing best practices for building rigorous agentic benchmarks. In *NeurIPS Datasets and Benchmarks Track*, 2025. arXiv:2507.02825.

Appendix

The appendix holds the repositories, headline metrics and grading targets (A); the agent's inputs, verbatim (B); the sandboxing (C); the failure taxonomy definitions (D); per-cell results (E); cost (F); the GPU-only experiment (G); and code size and compute (H).

A Repositories, Headline Metrics and Grading Targets **15**

B Agent Inputs, Verbatim **17**

B.1 Prompt Template 17

B.2 Access Document 18

B.3 Frozen Phase 2 Message 19

C Sandbox and Isolation **19**

D Failure Taxonomy Definitions **19**

E Per-Cell Results **20**

F Cost **20**

G GPU-Only Experiment **20**

H Code Size and Compute **21**

A Repositories, Headline Metrics and Grading Targets

Sourcing. Hard requirements: a training-centric pipeline that is the repository's point; a readable, vendorable tree at a pinnable commit under a permissive license read from the LICENSE file, not the GitHub label; a headline metric that the repository's own code computes, for a configuration the tree actually runs, with a published value; anonymously acquirable data; a full-scale run of about ten hours or less on one TPU host.

Notation. Table A1 is generated from the criteria files the grader reads. *Grading target* uses the comparator's notation: $\leq$ and $\geq$ are one-sided, $[x, y]$ is a band whose upper bound is the metric's ceiling or, on a loss, a leakage floor. *Origin*: *readme* or *paper*, the source's own published number for the configuration in the tree; *anchor*, a pre-registered band around our runs of the unmodified source; *rule*, a floor we set for a repository that publishes no number; *released-checkpoint*, the number the source's release notes report for its pre-trained checkpoint. The table in the release quotes each statement verbatim with its line pointer.

repository	domain	headline metric	grading target	origin
Language				
llama2.c	language modeling	a best validation loss of at most 1.130 at the source's own 100,000-iteration default horizon	≤ 1.13	anchor
LoRA	fine-tuning	a corpus BLEU of at least 0.680 on the E2E test set	$[0.68, 1.0]$	readme

repository	domain	headline metric	grading target	origin
makemore	language modeling	a best test NLL of at most 2.023	≤ 2.023	anchor
minGPT	language modeling	at least 495 of 500 correct adder test answers	≥ 495	rule
modded-nanoGPT	language modeling	a FineWeb validation cross-entropy loss of at most 3.285	≤ 3.285	readme
nanochat	language modeling	a base-evaluation-suite score of at least 0.256525	$[0.2565, 1.0]$	readme
nanoGPT	language modeling	a validation loss at the graded iteration budget	≤ 3.12	readme
SimCSE	sentence embeddings	an average Spearman correlation of at least 73.5 across the seven STS test sets	$[73.5, 100.0]$	readme
x-transformers	language modeling	a validation loss between 0.8788 and 0.9788	$[0.8788, 0.9788]$	anchor
Vision				
airbench	image classification	a test-time-augmented CIFAR-10 test accuracy of at least 0.9393, taken as the mean over the source's own 25-trial loop	$[0.9393, 1.0]$	paper
Generative & multimodal				
nanoVLM	vision-language	a best MMStar accuracy of at least 0.353	$[0.353, 1.0]$	readme
Structured data				
HSTU	recommendation	a MovieLens-20M HR@10 of at least 0.3450	$[0.345, 1.0]$	readme
PatchTST	time-series forecasting	a test mean squared error of at most 0.380 in standardized units	≤ 0.38	paper
TabR	tabular learning	a test accuracy of at least 0.9745 on the Covertype test split of 116,203 rows	$[0.9745, 1.0]$	paper
TGN	temporal graph learning	a transductive test average precision of at least 0.980 for future edge prediction on JODIE Wikipedia	$[0.98, 1.0]$	paper
Science & 3D				
TensoRF	neural fields	a mean test-view PSNR of at least 36.10 dB on Synthetic-NeRF lego	$[36.1, 45.0]$	paper
Control & sequence				
S4	long-sequence modeling	a top-1 accuracy of at least 87.0 percent on a held-out CIFAR-10 split	$[87.0, 100.0]$	readme

repository	domain	headline metric	grading target	origin
TD-MPC2	reinforcement learning	a mean evaluation return of at least 750.0 over the source's 10-episode evaluation on DMControl walker-run	$[750.0, 1000.0]$	readme
Audio & speech				
AST	audio classification	a test-split top-1 accuracy of at least 0.975	$[0.975, 1.0]$	readme
deepspeech	speech recognition	a word error rate of at most 20.0 percent on the AN4 test set	$[1.5, 20.0]$	released-checkpoint

Table A1. The 20 repositories by domain family, with the graded quantity and its grading target.

B Agent Inputs, Verbatim

A cell's input is the rendered prompt, the access document with live addresses, and the vendored repository. Below, typography is normalized and lines re-wrapped to fit the page; the words are verbatim. The recipe paragraph of every repository, the three judge prompts and every judge answer are released with the benchmark; the prompts are in the code at github.com/zlab-princeton/XStack under `prompts/`.

B.1 Prompt Template

Phase 1 Prompt Template

```
# PyTorch –> JAX on TPU: whole–repo transfer

Port the fixed–commit PyTorch repository under `source_torch/` entirely to a working JAX/Flax/Optax repository, and make it work at real scale on the TPU machine described in `TPU_ACCESS.md` –– a clean machine: set it up yourself. You have full access to that machine to develop, run, and test your port. The deliverable is the working repository itself: produce a faithful port and run it the way the source does.

Deliverables at this workspace root:

– `setup_env.sh` –– prepares a worker of the TPU machine from clean state, including any data the repository's pipeline needs, obtained and prepared the way the source repository does. It is sourced before the entry point runs.
– `run_train.py ––seed S ––out DIR ––metrics PATH` –– runs the repository's training pipelines the way the repository runs them, at their own scale, and writes the numbers the repository itself reports to the metrics JSON. It checkpoints to `––out` as it trains and resumes from an existing checkpoint in `––out`.
– `export_torch.py ––ckpt DIR ––out PATH` –– writes the trained weights as an `.npz` of numpy arrays keyed by the source model's own parameter names (exact names and shapes).
{{REPO_CONTRACT}}
Your submission is collected from this workspace and executed on a fresh machine of the same type. It is judged on behavior that is fundamental to the source repository, not all of which is spelled out here.

Rules:

– Do not import or call PyTorch anywhere in your submission at runtime.
– Data, library, and package downloads are allowed; do not fetch, clone, or copy model or porting code from the internet.

## Goal

Keep working until your submission passes grading.
```

The slot `{{REPO_CONTRACT}}` renders empty for sixteen of the 20 repositories. For nanoGPT it renders two

lines: a `-max-iters N` knob the entry must honor (the grader passes 120,000) and the line naming GPT-2 124M on OpenWebText as the graded configuration. For HSTU it renders one configuration line (HSTU-large on MovieLens-20M). For TD-MPC2 it renders `-task NAME` and `-max-steps N` knobs (the grader passes walker-run and 500,000; the prompt names neither). For nanochat, the same `-max-iters` knob, dormant.

B.2 Access Document

The multi-host document; a single-host twin exists for one-host machines:

Access Document (multi-host)

TPU access

A multi–host TPU machine is allocated to this task for its duration. It is a standing machine that must stay up (rules at the end).

– Google Cloud v5p slice with 8 worker hosts, 4 TPU chips each (32 chips total). A JAX program launched simultaneously on ALL workers sees all 32 devices as one machine.
– Each worker: Ubuntu 22.04, remote user `<user>`, stock Python 3.10, no ML stack preinstalled, its own ~100 GB local disk (set up every worker; large datasets and repeated checkpoints can exhaust it).
– TPU name `<name>`, zone `<zone>`. Worker external IPs: <list>

Run a command on worker N (this form is the reliable one):

```
gcloud alpha compute tpus tpu–vm ssh <user>@<name> \
  ––zone <zone> ––worker=N ––command 'command' ––quiet \
  ––ssh–flag=–i ––ssh–flag=$HOME/.ssh/<key>
```

Copy files to worker N:

```
gcloud alpha compute tpus tpu–vm scp ––recurse local_path \
  <user>@<name>:remote_path ––zone <zone> ––worker=N ––quiet \
  ––scp–flag=–i ––scp–flag=$HOME/.ssh/<key>
```

Direct `ssh –i ~/.ssh/<key> <user>@<worker–ip>` also works but times out intermittently in this zone; if you use it, set generous connect timeouts and retry. Prefer the gcloud form.

Notes:

– To use the whole machine, launch the SAME program on ALL workers at the same time (e.g. loop the command over workers with `&`); on TPU the JAX runtime discovers the other hosts automatically.
– Every worker has internet access; installing packages works on each.
– A dropped ssh session kills its foreground child: run long remote commands under `nohup` (or `tmux`) and poll their logs.
– Your final submission is collected from THIS workspace, then run on that machine (launched on all workers simultaneously).

Rules (the machine is shared infrastructure and must stay up):

– Never reboot, shut down or power off a worker, and never stop or restart system services. If the TPU looks stuck, a process of yours is still holding the devices: kill your own processes on every worker and wait for them to exit.
– All workers are yours to use. On each of them, keep your files under the `<user>` home directory, /tmp and /dev/shm, and only touch processes you started.

The document is re-rendered with live addresses before every turn.

B.3 Frozen Phase 2 Message

After a phase 1 fail the agent receives nothing but this message: no failure sentence and no pass or fail. The frame is one sentence followed by the repository's recipe paragraph:

Frozen Phase 2 Message

Your submission is now frozen: from this point do not change your ported code, setup, or any other file. Edit only `run_train.py` so that one invocation does exactly the following, using your ported code as it is:

{{RECIPE}}

C Sandbox and Isolation

The agent runs on our cluster inside a sandbox that hides the benchmark repository, our reference migrations, the grading criteria, the results archive and the session files of every other cell. The sandbox also turns off the agent's memory across sessions. The TPU machine itself stores no results: every graded turn is archived elsewhere, and the hosts are wiped before every cell. An agent turn has no time limit while output flows: thirty minutes of silence after activity ends the turn as a hang, and thirty minutes with no output at all counts as a provider outage and the turn is retried.

D Failure Taxonomy Definitions

The failure judge assigns one of the classes below.

class	meaning
pass	grading target reached (a pass in the figures also needs the judges)
scope miss	a different pipeline than the headline one, or only part of the repository (nanoGPT's Shakespeare quick-start instead of GPT-2 on OpenWebText; minGPT's chargpt instead of the adder; HSTU's ML-1M example)
incomplete transfer	the right pipeline trained, the headline evaluation never run (nanochat without the CORE suite; LoRA without beam decoding and BLEU)
fidelity gap	the right pipeline and evaluation, the model short of the grading target
scale deviation	the recipe run at a different scale than the source (batch, steps, model size, data ratio)
runtime crash	the graded run exited non-zero or hung
no termination	the migration has no stop rule; the run reached the cap and wrote no metrics
no TPU	the graded run never acquired the TPU (CPU fallback)
wrong fix	a later turn that changed something other than the defect
unchanged tree	the agent returned without editing the deliverable
grader artifact	the grade was the harness's error, not the migration's; the class describes the grader
fabricated result	the headline number was not produced by running the pipeline (copied, hard-coded, or evaluated on downloaded finished weights)
freeze violation	a phase 2 turn changed something other than the entry file
infra	machine or harness outage; the turn was not consumed

A turn may carry a secondary class. Rows are added when the grade lands and never change it.

Figure D1 repeats the phase-1 breakdown of Figure 7 after phase 2. The wrong-experiment group disappears, because phase 2 states the recipe: a cell that ran the wrong pipeline in phase 1 either passes now or fails in its migrated code. What remains is the two code groups, fidelity gaps and crashes, which no entry script can fix, the judges' rejections, and the 16 cells that broke the freeze by rebuilding the migration inside the entry script. Phase 2 turns the wrong experiments of the stronger agents mostly into passes, and those of the weaker agents into judge rejections, fidelity gaps and freeze violations. GLM-5.3-Flash is the clearest case: its eleven phase-1 crashes become two, because in phase 2 it rewrites the migration inside the entry script, which the freeze forbids, and six of its cells fail on that rule instead.

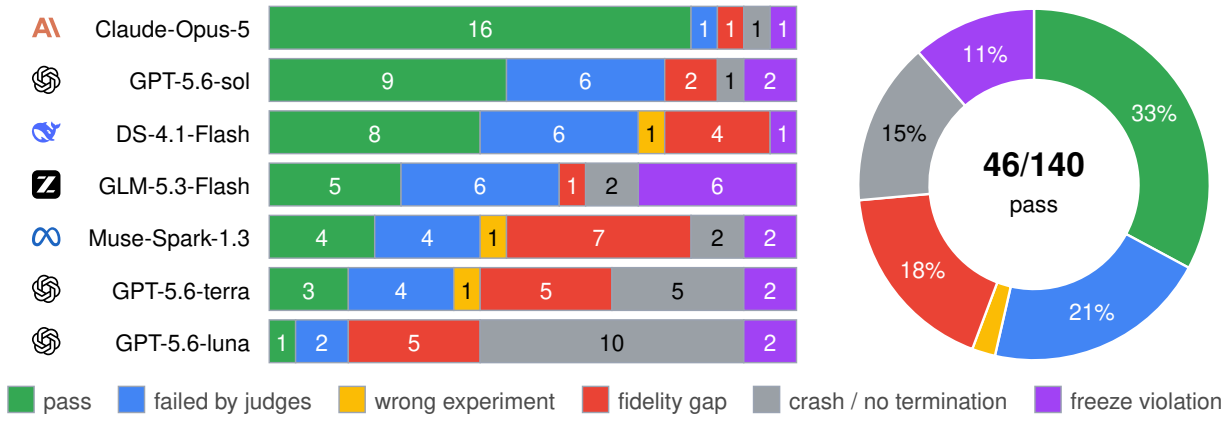


Figure D1. Failure breakdown after phase 2. Outcome of each cell after phase 2, grouped as in Figure 7 plus freeze violations: runs that changed more than the entry script or rebuilt the migration in it. The ring at the right sums the bars over the agents. Passing in either phase counts as a pass. A cell without a phase 2, or whose phase 2 left the entry script unchanged, keeps its phase 1 outcome.

E Per-Cell Results

Figure E1 gives the result of every cell. Rows are the 20 slate repositories and columns the 7 agents; the last two rows repeat the per-agent rates of Figure 4. Table 4 gives the pass rate per agent after each check. Claude Opus 5 on PatchTST and GPT-5.6-terra on deepspeech reach the target and pass the judges, but their exported weights do not load under the source's parameter names, so the figure draws them as fails. Two repositories bracket the matrix. minGPT, the smallest source, passes for every agent but GLM-5.3-Flash; nanoVLM, the longest pipeline, passes for none. Five repositories (LoRA, SimCSE, nanochat, airbench, AST) pass for Claude Opus 5 alone, and that is where its lead over the field comes from. The mid-field agents pass the small language-model repositories (minGPT, makemore, nanoGPT, llama2.c) and a few others each; GPT-5.6-terra and GPT-5.6-luna pass little beyond minGPT and makemore. Phase 2 adds between one and three passes per agent and breaks the three phase-1 ties: GPT-5.6-sol above DeepSeek, GLM-5.3-Flash above Muse Spark, and GPT-5.6-terra above GPT-5.6-luna.

F Cost

Cost and time. At list prices as of September 14, 2026 (Table 3), the 20 cells cost $10.0 for luna, $40 for terra, $32 for GLM-5.3-Flash, $37 for DeepSeek, $283 for Muse Spark, $323 for sol and $1,844 for Claude Opus 5 ($92.18 per cell, of which phase 1 is $49.42); 97% of input tokens are cache hits. The list prices of Table 3 are all undiscounted, and we used no promotional price when computing costs.

G GPU-Only Experiment

Before the TPU benchmark we ran a GPU-only generation of the study: twelve small PyTorch repositories, each migrated whole to JAX by GPT-5.5 at high reasoning effort and graded on the same GPU by ten

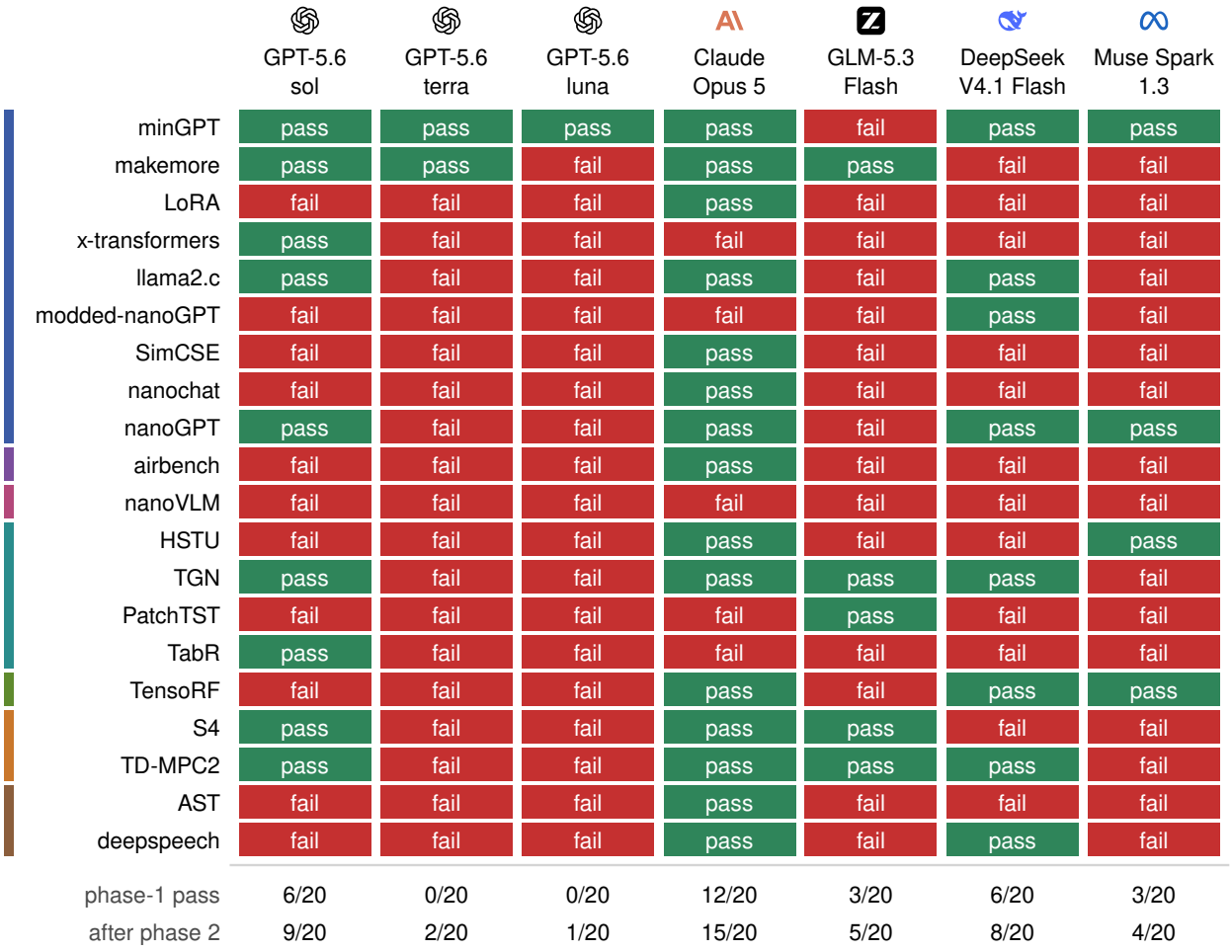


Figure E1. Every cell, repositories grouped by family: the final result, pass or fail.

checks (Table G1; Section 4.1). The twelve are DDPM (Wang, 2020a), makemore (Karpathy, 2022a), minGPT (Karpathy, 2020), nanochat (Karpathy, 2025), nanoGPT (Karpathy, 2022b), nano-vLLM (Yu, 2025), pyGCN (Kipf, 2017), x-transformers (Wang, 2020b), and the MNIST, REINFORCE, VAE and word language model examples of PyTorch (PyTorch Contributors, 2016). The checks are what a careful reviewer of a translation would write: the migration has its own entry points and no PyTorch, the same weights give the same logits, loss and gradients, optimizer and data pipeline match, checkpoints round-trip, greedy decoding matches token for token, and a restarted run resumes. The agent met 116 of 117, and the single miss is a convergence band (x-transformers, A6).

H Code Size and Compute

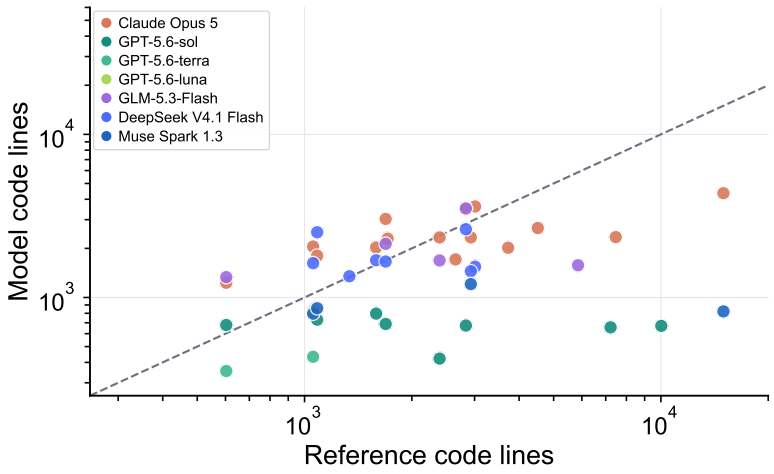
Every cell had enough compute: Figure H1 gives the TPU chip-hours of all 140 cells, split into the agents' turns and our grading. Grading, not generation, is where the compute goes: our full-scale grades took more than twice the chip-hours of the agents' own turns, and the v6e machines carried nearly three quarters of the total. The table converts chip-hours to H100-equivalents at each chip's peak bf16 throughput relative to an H100, so the campaign is comparable with GPU budgets.

The left panel of Figure H1 sets each passing migration's size against its source: 16 of the 44 passes are larger than the repository they came from. Most passing migrations sit within a factor of two of their source's size, and the agents differ systematically: Claude Opus 5, DeepSeek and GLM write migrations about as long as

Repository	Checks
DDPM	10/10
makemore	10/10
minGPT	10/10
MNIST	9/9
nano-vLLM	9/9
nanochat	10/10
nanoGPT	10/10
pyGCN	9/9
REINFORCE	10/10
VAE	10/10
word language model	10/10
x-transformers	9/10 [A6]
All	116/117

	Check	Passes if the migration has ...
A0	Completeness	a JAX code base with its own entry points and no PyTorch inside
A1	Forward parity	same weights give the same logits and loss as PyTorch
A2	Gradient parity	same gradients as PyTorch autograd
A3	Optimizer	the learning-rate schedule and optimizer step match the source
A4	Data pipeline	batches are the source's token windows
A5	Checkpoints	weights round-trip between the two frameworks
A6	Convergence	its own trainer reaches the source's validation-loss band
A7	Resume	a restarted training run continues from its checkpoint
A8	Generation	greedy decoding matches the source token for token
A9	Runnability	its own train and sample scripts run stand-alone

Table G1. The GPU-only experiment. Left: checks passed per repository by GPT-5.5 at high reasoning effort, whole-repository PyTorch to JAX migrations graded on an A100 against the running source (the GPU generation's last run); 11 of 12 repositories passed every check (x-transformers missed A6). Right: the ten checks, each a test against the running PyTorch source. MNIST and pyGCN have no generation and nano-vLLM no training resume, so they have nine. Five of the twelve repositories are in XStack.



	Generation	Grading	Total
TPU chip-hours			
v4	71	455	526
v5p	633	5,163	5,796
v6e	5,940	10,540	16,480
all	6,644	16,158	22,802
H100-equivalent hours			
v4	20	126	146
v5p	294	2,396	2,690
v6e	5,514	9,782	15,296
all	5,828	12,304	18,132

Figure H1. Code size and compute. Left: Python lines of each passing migration against the source it came from, one mark per cell (44 cells; the dashed line is equal size). Right: TPU chip-hours of all cells, split into the agents' turns (generation) and our grading of their code, and the same in H100-equivalent hours at peak bf16 throughput.

the source; GPT-5.6-sol and GPT-5.6-terra write compact ones, under a thousand lines even for sources ten times that size. The two largest sources that were migrated successfully, x-transformers and HSTU, came out at a fraction of their size: both are libraries, and their passing migrations preserve the model, training and evaluation paths the headline experiment runs rather than every option the library offers.