LaDiM: Layered Diagnosis for Multi-Agent Code Migration

Anonymous authors
Paper under double-blind review

Abstract

Migrating code across deep learning frameworks enables the reuse of training programs in new software and hardware environments. Different frameworks, however, might compute gradients or update parameters in different ways, leading to wrong results even when a translated program runs without error. Current methods are mostly designed for language transfer, neglecting the subtle differences in framework level. To address this, LaDiM follows the underlying training computation order, e.g., forward, gradient computation, update, and uses the dependencies among these stages to detect and localize silent computational errors. LaDiM includes Translator, Verifier, and Repair agents working under a shared Orchestrator. The Translator produces an initial output. The Verifier traces discrepancies along the computation order and hands its findings to the Repair Agent, which revises the candidate using this evidence and a persistent repair history. For repository level migrations, LaDiM further introduces repository context management which tracks cross file dependencies and preserves diagnostic context as agents move across the repository. Evaluation across framework migration, language migration, and repository migration tasks shows that LaDiM achieves comparable or better migration results with up to 80.8% fewer tokens than state-of-the-art methods, demonstrating its effectiveness.

1 Introduction

Code development environments and languages have grown increasingly diverse. Focusing on deep learning, popular frameworks such as PyTorch, TensorFlow, and MindSpore provide programming interfaces, execution semantics, and hardware backends (Paszke et al., 2019; Abadi et al., 2016; MindSpore, 2020). To enable easy adaptation of deep models and training pipelines on new software and hardware environments, migrating existing code across languages and frameworks can reduce development cost (Guan et al., 2025). This need has driven the development of numerous code translation methods. Large language models (LLMs) reduce the manual effort involved in code translation (He et al., 2025; Macedo et al., 2025b), while automated tests and interactive repair help assess and correct translated behavior (Rozière et al., 2022; Yang et al., 2024; Ibrahimzada et al., 2026).

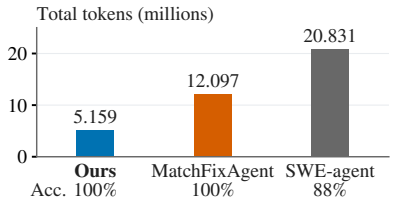


Figure 1: Tokens used on 50 PyTorch to MindSpore migration tasks.

However, success at translating code does not guarantee equal translation performance across deep learning frameworks, for example, migrating from PyTorch to MindSpore. Different frameworks use different interfaces and execution mechanisms for forward computation, gradient computation, and parameter updates (Paszke et al., 2019; MindSpore, 2024). Further differences in operator semantics, automatic differentiation, and optimizer interfaces can change model behavior after migration (Paszke et al., 2019; MindSpore, 2020). As a result, a translated program can run without raising error while still producing incorrect results due to silent failures in computing losses, gradients or parameter updates. We use semantic drift to describe the changes in such training behavior introduced by migration. Existing translation methods primarily target language level translation

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.

and test based functional correctness, leaving framework specific computational drift less directly addressed (He et al., 2025; Macedo et al., 2025b; Ibrahimzada et al., 2026).

To address this, we aim to locate and repair the true cause of semantic drift after initial translation. We propose LaDiM (Layered Diagnosis for Multi-Agent Code Migration), a multi-agent framework that follows the training computation order of deep learning programs. We propose Layered Diagnosis, inspecting the translated candidate stage by stage, e.g., forward computation, gradient computation, and parameter update, in the order of how training actually executes. The observed discrepancy during inspection is handed to a separate Repair Agent and saved to local repair history for future use in later rounds. For repository-scale migration, we propose repository context management that first organizes code files into a dependency graph and repairs them in topological order to avoid redundant inspection and improve effectiveness. An Orchestrator schedules overall process and returns verification results to guide further repair.

LaDiM achieves comparable or better translation results with substantially lower tokens consumed. As shown in Fig. 1, LaDiM completes all 50 migrations from PyTorch to MindSpore with 57.4% fewer total tokens than MatchFixAgent, and 75.2% fewer than SWE-agent while obtaining higher acceptance. On repository-level migration, LaDiM saves more than 68% of tokens on the time series repository at full acceptance and achieves the highest behavioral coverage on the recommendation repository. Our contributions are as follows:

- We develop Layered Diagnosis, which uses dependencies among execution, forward values, gradients, and parameter updates to locate the causes of changed training behavior.
- We design an agent framework with independent evidence handoff and repair history to connect investigation, edits, and verification. Repository context management coordinates this process across dependent files through Repository Structural Analysis and Dependency Graph Planning.
- We demonstrate LaDiM's effectiveness and token efficiency across framework, language, and repository migration, and identify the contributions of training signals and agent components through ablation studies.

2 Related Work

Code Translation and Framework Migration. Neural translation methods learn code mappings from multilingual corpora and use automated tests or compiler representations to improve translations (Rozière et al., 2020; 2022; Szafraniec et al., 2023). ExeCoder encodes functional semantics, syntax structure, and variable dependencies (He et al., 2025), while InterTrans explores intermediate translation paths (Macedo et al., 2025b). MindConverter converts model definitions and weights and maps supported APIs (MindSpore, 2023). RepoTransAgent uses repository context, test failures, and reflection to refine translated code (Guan et al., 2025). LaDiM investigates how translated training code computes forward values and gradients and updates model parameters. At repository scale, it coordinates changes to shared interfaces and checks their effects on the repository entry points that use them.

Differential Testing and Deep Learning Verification. Differential testing exposes compiler and program faults through disagreement between related executions (Yang et al., 2011; Le et al., 2014). DeepXplore detects behavioral differences between deep learning systems (Pei et al., 2017), while DeepGauge measures test coverage in these systems (Ma et al., 2018). More directly related to framework migration, TensorScope tests corresponding APIs across frameworks to identify inconsistent behavior (Deng et al., 2023). NablaFuzz compares execution scenarios to test automatic differentiation, including first-order and higher-order gradients (Yang et al., 2023). LaDiM uses training differences as evidence for code repair. Its Verifier Agent inspects code to explain discrepancies in forward values, gradients, and parameter updates, then passes the supporting observations and hypotheses to the Repair Agent for testing during repair.

Automated Program Repair and LLM Agents. Automated program repair uses tests or semantic constraints to guide patch generation (Nguyen et al., 2013). Pretrained language models generate patches from buggy code and its context, with tests used to evaluate candidate patches (Xia et al.,

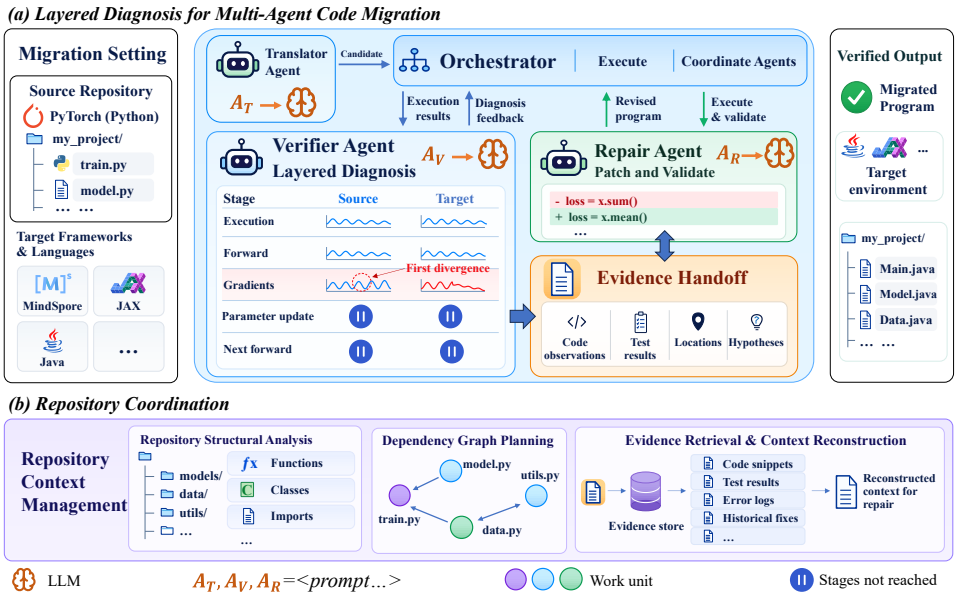


Figure 2: Overview of LaDiM. (a) The Orchestrator evaluates the Translator's candidate across execution, forward values, gradients, and parameter updates. The Verifier investigates the first divergence (e.g., gradients), handing code observations, test results, locations, and hypotheses to the Repair Agent. Repair continues with testing, editing, and Orchestrator feedback until acceptance or budget exhaustion. (b) Repository context management analyzes code structure, orders dependent repairs, and reconstructs compact contexts from archived evidence.

2023). SWE-agent supports repository navigation, editing, and testing (Yang et al., 2024). Reflexion and Self-Debugging use execution feedback to support reflection and self-correction (Shinn et al., 2023; Chen et al., 2024b). CodeAct uses executable code actions and their results to guide subsequent actions (Wang et al., 2024). ChatDev and MetaGPT assign software development activities to specialized roles (Qian et al., 2024; Hong et al., 2024). AutoGen and AgentVerse provide general frameworks for multi-agent collaboration (Wu et al., 2024; Chen et al., 2024a). MatchFixAgent takes a source program and an existing translation and uses several semantic analyses to guide test generation and repair (Ibrahimzada et al., 2026). LaDiM includes initial translation and organizes diagnosis around dependencies in training computations. The Verifier Agent passes its findings and supporting observations to a separate Repair Agent, which uses measurements from each submitted revision to revise its hypotheses.

3 Methods

3.1 Preliminary

A migration task $\mathcal{T} = (P, L_s, F_s, L_t, F_t, W)$ specifies a source program P, its language and framework (L_s, F_s), the requested target environment (L_t, F_t), and an editable file scope W. Migration produces a target program $\hat{P}$ that preserves the source's training behavior. The scope can include the translated program, its supporting library, or a repository. An edit modifies code within W.

At step t, we record source observations $(e_t, v_t, g_t, \theta_{t+1} - \theta_t)$ and target observations $(\hat{e}_t, \hat{v}_t, \hat{g}_t, \hat{\theta}_{t+1} - \hat{\theta}_t)$, with numerical values recorded when available. Here e_t records the execution outcome, including success or an exception, v_t contains selected forward values and losses, g_t contains gradients, and θ_t denotes the parameters before the update. After aligning corresponding tensors and parameters, we define

$$d_{\text{execution},t} = \mathbf{1}[e_t \neq \hat{e}_t], \qquad d_{\text{forward},t} = v_t - \hat{v}_t, \tag{1}$$

$$d_{\text{gradient},t} = g_t - \hat{g}_t, \qquad d_{\text{update},t} = (\theta_{t+1} - \theta_t) - (\hat{\theta}_{t+1} - \hat{\theta}_t). \tag{2}$$

For a numerical comparison $d = x - \hat{x}$, $d \simeq 0$ denotes agreement within its tolerance, such as $|d_j| \leq \alpha + \beta|x_j|$ for absolute tolerance α and relative tolerance β, or a specified norm bound. Forward values determine the loss, differentiation produces the gradients, and the optimizer uses those gradients and its state to update parameters. The updated parameters then affect subsequent forward computations.

The LLM M performs translation, investigation, and repair under role instructions A_T, A_V, and A_R. The Orchestrator $\mathcal{O}$ evaluates candidates and manages their shared investigation and repair budget B. In the algorithms, $S.\hat{P}$ is the candidate, $S.m$ contains its measurements, q is its verification status, and H is a conversation. Context C retains evidence and provides tools. For repository tasks, it also stores the dependency plan and local check results. A submission sends the complete candidate for external verification after a repair stage that can contain several LLM calls, tests, and edits. Each $\text{LLMCALL}(M, A, H, b, B)$ invokes M with instructions A and conversation H, charging its usage to the stage allowance b and shared budget B. The Orchestrator counts each submission against B.

3.2 VERIFIER AGENT WITH LAYERED DIAGNOSIS

Algorithm 1 Layered Diagnosis

Require: $\mathcal{T}, S, M, B, C$
Ensure: Evidence E

function LAYEREDDIAGNOSIS($\mathcal{T}, S, M, B, C$)
  $H \leftarrow$ STARTCONVERSATION($A_V, \mathcal{T}, S$)
  $b \leftarrow$ DIAGNOSISALLOWANCE(B)
  $\mathcal{D}$: execution $\rightarrow v_t \rightarrow g_t \rightarrow \Delta\theta_t \rightarrow v_{t+1}$
  while WITHINLIMITS(b, B) **do**
    $H \leftarrow C$.PREPARECONTEXT(H, S)
    $(\ell, h, a) \leftarrow$ LLMCALL(M, A_V, H, b, B)
    $H \leftarrow$ TOOLFEEDBACK(a, H, S, A_V, C)
    if a.complete **then break**
  return PUBLICEVIDENCE(H, C)
function TOOLFEEDBACK(a, H, S, A, C)
  $(C, S, r) \leftarrow C$.EXECUTETOOLS(a, S, A)
  return $H \parallel (a, r)$

The Verifier Agent receives the Translator Agent's candidate and the Orchestrator's measurements (Fig. 2a). It uses Layered Diagnosis to focus its investigation on the computations responsible for observed discrepancies.

For an **execution discrepancy** ($d_{\text{execution},t} = 1$), it inspects the exception and the calls that produced it. For a **forward discrepancy** ($d_{\text{forward},t} \not\simeq 0$), it traces the operators that produce different values and tests the suspected computation. When forward values agree but a **gradient discrepancy** ($d_{\text{gradient},t} \not\simeq 0$) remains, it examines parameter registration and the path from the parameters to the loss. When both forward values and gradients agree, it investigates an **update discrepancy** ($d_{\text{update},t} \not\simeq 0$) through the optimizer's rule and state. The first step with an observed discrepancy further guides this investigation because a changed update can alter later forward computations. The LLM selects code inspections and tests that distinguish possible causes, then records the code locations and observations supporting its hypotheses. These records form the evidence for the independent handoff to the Repair Agent.

Algorithm 1 expresses this reasoning through the training dependency relation $\mathcal{D}$, where $\Delta\theta_t = \theta_{t+1} - \theta_t$. The symbols ℓ and h summarize the LLM's choice of computation and hypothesized cause within a call, and a is its tool request or completion response. The Verifier uses both agreement and disagreement between layers to select ℓ, then tests h against code and execution evidence. The shared TOOLFEEDBACK operation executes requests in order and adds their results to the conversation. During investigation, the Verifier Agent can create and run scratch tests while candidate and supporting library code remain unchanged.

3.3 REPAIR AGENT WITH EVIDENCE HANDOFF AND HISTORY

Independent evidence handoff gives the Repair Agent the Verifier Agent's recorded code observations, test results, and hypotheses, identified as the Verifier Agent's findings. The Repair Agent reassesses this evidence, runs further tests, and edits files in W, including supporting library code when the task permits it. The Orchestrator verifies each submission and returns the remaining discrepancies to guide the Repair Agent's next investigation and edits.

Repair history preserves earlier edits, observations, and hypotheses across attempts. The continuing conversation includes failed tests and verification feedback, allowing the agent to relate a changed result to the preceding modification and revise explanations that did not resolve the fault. The current candidate also persists, so later attempts build on applied changes.

Algorithm 2 REPAIR($\mathcal{T}, S, E, q, M, B, C$)

$H \leftarrow$ INDEPENDENTHANDOFF($A_R, \mathcal{T}, E, C$)
while SUBMISSIONSLEFT(B) $\wedge$ BUDGETLEFT(B) **do**
  $H \leftarrow H \parallel S.m$; $b \leftarrow$ REPAIRALLOWANCE(B)
  while WITHINLIMITS(b, B) **do**
    $H \leftarrow C$.PREPARECONTEXT(H, S)
    $a \leftarrow$ LLMCALL(M, A_R, H, b, B) ▷ Test and edit
    $H \leftarrow$ TOOLFEEDBACK(a, H, S, A_R, C)
    if REPAIRCOMPLETE(a, H) **then break**
  $(S.m, q) \leftarrow \mathcal{O}$.SUBMIT($S.\hat{P}, B$)
  if $q =$ accepted **then return** $(S.\hat{P}, q)$
return $(S.\hat{P}, q)$

Algorithm 2 reuses TOOLFEEDBACK from Algorithm 1, with $H \parallel x$ denoting the addition of a record x to the repair history. Its REPAIRCOMPLETE condition requires a completion response, an applied edit in the current stage, and a test executed after the last edit. If an edit or test is missing, the conversation requests that action within the remaining budget. The Orchestrator evaluates the candidate when the repair stage ends.

Algorithm 3 Repository migration and repair coordination

function REPOSITORYMIGRATE($\mathcal{T}, M, B$)
  $C \leftarrow$ REPOSITORYCONTEXT($\mathcal{T}$)
  $S.\hat{P} \leftarrow$ LLMTRANSLATE($M, A_T, \mathcal{T}$)
  $(S.m, q) \leftarrow \mathcal{O}$.VERIFY($S.\hat{P}$) ▷ Complete repository
  if $q =$ accepted **then return** $(S.\hat{P}, q)$
  $E \leftarrow$ LAYEREDDIAGNOSIS($\mathcal{T}, S, M, B, C$) ▷ Algorithm 1
  return REPAIR($\mathcal{T}, S, E, q, M, B, C$) ▷ Algorithm 2
function C.EXECUTETOOLS(a, S, A)
  $r \leftarrow [\,]$
  for all $c \in a$.tools **do**
    $U \leftarrow \varnothing$
    if c.kind $=$ structure **then** $o \leftarrow$ REPOSITORYSTRUCTURALANALYSIS(C, S)
    else if c.kind $=$ plan **then** $(C.G, U, o) \leftarrow$ UPDATEDEPENDENCYGRAPH(c, C)
    else $(C, S, o, U) \leftarrow$ EXECUTE(c, C, S, A)
    $C.E \leftarrow C.E \parallel (o,$ CODEVERSION(S)$)$; $r \leftarrow r \parallel o$
    $C.Q[U \cup$ DEPENDENTS$(C.G, U)] \leftarrow$ invalid
  return (C, S, r)

3.4 REPOSITORY COORDINATION

Repository migration raises two problems. A repair may span several relevant files, and evidence gathered on one file must remain available while the agent works on others. Repository context management addresses both through three components (Fig. 2b).

Repository Structural Analysis. This tool returns an inventory of the target repository: its files, the imports among them, the functions and classes each file defines, and the cells of each notebook. It also lists files not yet assigned to any work unit. The agent invokes it on demand.

Dependency Graph Planning. Based on this inventory, the agent partitions the target files into *work units*. Each unit specifies its editable files, a repair goal, its tests, and the units it depends on. Together, the units form a directed acyclic graph $C.G$. A unit may start only when the checks of its prerequisites remain valid, so shared implementations are repaired before the units that use them. $C.Q$ stores each unit's syntax check and test results with the code version on which they ran. When a file or the plan changes, the results of the affected units and their dependents are invalidated. Unit checks only guide the agent. The Orchestrator decides acceptance by verifying the complete repository.

Evidence archive and context reconstruction. Tool observations, code read by the agent, measurements, hypotheses, and conversations are stored in an archive $C.E$ outside the editable workspace, each tagged with the code version it describes. At evidence handoff, when the agent switches work units, or when the conversation exceeds 75% of the context capacity, PREPARECONTEXT replaces the conversation with a compact one containing the current plan and progress, the structured findings, the latest measurements, and previously read code that is relevant to the active unit and unchanged since it was read.

Table 1: Migration acceptance and token use on common inputs: (a,b) individual programs, (c,d) repositories. Tokens are in millions. Calls count all LLM calls in (a,b) and investigation and repair calls in (c,d).

(a) PyTorch to MindSpore

Method	Calls	Tokens	Accepted
Direct LLM	29	0.429	29/50
CodeTransEngine	29	0.280	31/50
MSAdapter	–	–	15/50
SWE-agent	1,097	20.831	44/50
MatchFixAgent	619	12.097	**50/50**
LaDiM (ours)	326	5.159	**50/50**

(b) Java/DJL to Python/PyTorch

Method	Calls	Tokens	Accepted
Direct LLM	18	0.694	1/18
InterTrans	89	2.286	1/18
Test-guided repair	71	4.475	7/18
SWE-agent	682	24.321	8/18
MatchFixAgent	524	28.355	8/18
LaDiM (ours)	345	21.479	**9/18**

	(c) Time series repository				*(d) Recommendation repository*			
Method	**Checks**	**Entries**	**Calls**	**Tokens**	**Checks**	**Tests**	**Calls**	**Tokens**
SWE-agent	69/69	3/3	80	8.200	104/145	10/10	80	8.317
MatchFixAgent	69/69	3/3	69	13.598	129/145	0/10	80	8.679
LaDiM (ours)	69/69	3/3	**35**	2.614	**130/145**	10/10	80	8.811

Algorithm 3 connects these components to Algorithms 1 and 2. EXECUTE handles unit selection, file and notebook-cell operations, tests, and evidence retrieval, and returns an observation o with the set U of units affected by file or plan changes. All units share the repository files, the repair history, and the total budget.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Tasks. The main comparison contains 50 migration tasks from PyTorch to MindSpore, covering tensor operations, CNNs, MLPs, Transformers, language models, and parameter and optimizer interfaces. These 50 task identifiers correspond to 29 distinct combinations of source program and evaluation settings (Appendix A.1). We also migrate 18 Java/DJL programs to Python/PyTorch, comprising eight textbook examples and ten applications (DJL Contributors, 2026b;a), and two public repositories for time series (jinglescode, 2026) and recommendation (gauravchak, 2026). The repositories contain 13 and 24 code files or notebooks, totaling 865 and 2,086 nonblank lines.

To examine repair of actual translation errors, we use ten programs produced by the initial translation, of which five fail evaluation and five pass before repair. We use 16 faulty candidates to study training signals and six faulty MLP and CNN candidates to study repair on JAX. A separate detection study examines how early gradient, update, and loss checks reveal training faults. Appendix A.1 summarizes the program collections in Table 4.

Baselines. We compare LaDiM with SWE-agent (Yang et al., 2024) and MatchFixAgent (Ibrahimzada et al., 2026) using a common DeepSeek backend and each baseline's native repair procedure. The main table also includes Direct LLM for initial translation, CodeTransEngine's direct translation configuration (Macedo et al., 2025a), MSAdapter (OpenI, 2025), and InterTrans (Macedo et al., 2025b). Test-guided repair rewrites the complete candidate using test feedback. Direct repair in the analysis studies receives the same candidate as LaDiM and uses LaDiM's inspection, testing, and editing tools. For JAX, Ivy converts code between frameworks by transpiling the supplied model (Ivy Contributors, 2026), while `torch2jax` maps PyTorch operations and tensors to JAX through native conversion (torch2jax Contributors, 2026). We apply each converter once to the supplied faulty candidates and measure whether the existing faults are removed, without LLM repair. (Appendix A.5).

Evaluation Protocol. Methods use common source programs, evaluation inputs, and corresponding initial parameters. Repair methods also receive the same initial translations. Acceptance requires agreement in execution, forward values, gradients, and parameter updates, together with the task's

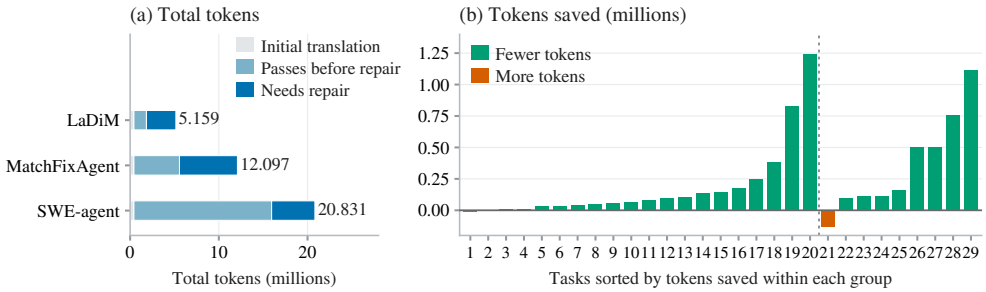


Figure 3: MindSpore migration token costs. (a) Initial translation and subsequent calls, grouped by whether repair is needed. (b) MatchFixAgent minus LaDiM tokens for 29 distinct inputs. Tasks 1–20 pass before repair, while tasks 21–29 require repair.

optimizer and interface checks. For repositories, *Checks* counts successful execution, numerical, and interface checks. *Entries* counts successful execution of the training script, notebook, and teaching examples. *Tests* counts passing tests supplied by the repository. Candidates that pass the evaluation available during repair are checked on additional seeds. Investigation and repair allow up to 40 LLM calls per program and 80 per repository, with at most four submissions. End-to-end costs include initial translation once and all subsequent calls, including failed attempts. Appendix A gives the numerical tolerances, seeds, and complete budgets.

4.2 Main Results

Migration to MindSpore. LaDiM completes all 50 MindSpore migrations with 57.4% fewer tokens than MatchFixAgent (Table 1a). The savings cover both programs that require repair and programs that pass evaluation before repair (Fig. 3). Reduced input token usage accounts for 93.0% of the savings. Most of the reduction comes from processing less context during investigation and repair. With submission limits of 1, 2, and 4, LaDiM accepts 46/50, 50/50, and 50/50 tasks, respectively.

Migration across languages. LaDiM accepts all programs completed by MatchFixAgent and an additional recurrent network example in the Java/DJL collection (Table 1b). Its 9/18 accepted programs comprise all eight textbook examples and one application, compared with 8/18 for SWE-agent and MatchFixAgent. LaDiM uses 11.7% fewer tokens than SWE-agent and 24.2% fewer than MatchFixAgent.

Repository migration. All three agents complete the time series migration, preserving the shared model across a training script, notebook, and teaching examples. At the same complete acceptance of 69/69 checks, LaDiM uses 68.1% fewer end-to-end tokens than SWE-agent and 80.8% fewer than MatchFixAgent.

On recommendation, LaDiM restores reward model inference and retrieval, which remain unresolved in both baselines, and executes the training command. It achieves the highest behavioral coverage, passing 130/145 checks and all ten original tests. Appendix E gives the remaining numerical differences and detailed checks.

4.3 Analysis Experiments

Repairing Translation Errors. Methods may edit both the ten translated programs and their supporting library. Table 2a reports how many failed programs are *Repaired*, how many passing programs are *Preserved*, and how many programs are finally *Accepted*. LaDiM repairs four of the five failed programs and preserves all five programs that pass evaluation before repair, raising final acceptance from five to nine. Direct repair, SWE-agent, and MatchFixAgent retain the initial five successes but recover none of the failed programs. Appendix D details a repaired LSTM whose recurrent operators and tensor indexing in the supporting library required two further submissions.

Table 2: Repair of (a) ten initial translations, five failing and five passing, and (b) six faulty JAX candidates. LaDiM and Direct repair allow four submissions. Ivy and `torch2jax` perform one native conversion. Tokens are in millions for all investigation and repair calls.

(a) Repairing translation errors

Method	Repaired	Preserved	Tokens	Accepted
Direct repair	0/5	5/5	3.751	5/10
SWE-agent	0/5	5/5	12.678	5/10
MatchFixAgent	0/5	5/5	14.351	5/10
LaDiM	**4/5**	5/5	13.207	**9/10**

(b) JAX repair and conversion

Method	Accepted	Calls	Tokens
Direct repair	6/6	57	0.353
Ivy	0/6	0	0
`torch2jax`	0/6	0	0
LaDiM	6/6	76	0.674

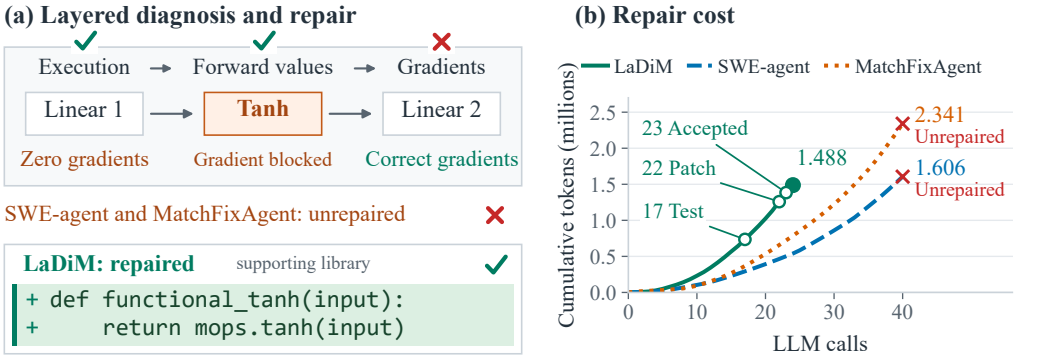


Figure 4: Repairing a silent gradient fault. (a) Layered Diagnosis identifies Tanh as the gradient break. The highlighted code adds its missing MindSpore mapping (registration decorators omitted). (b) Cumulative investigation and repair tokens. LaDiM repairs the program, while both baselines remain unrepaired after 40 calls.

Case Study: From a Silent Gradient Fault to an Accepted Repair. Fig. 4 traces a failing translation from Table 2a: an autoencoder-style classifier whose encoder applies Tanh between two linear layers. Execution and forward values match the source and the loss differs by only 1.4×10^{-6}, yet the first linear layer receives zero gradients. Because the second layer's gradients are correct, the Verifier isolates Tanh, and a targeted test confirms that the supporting library lacks a MindSpore mapping for it, so Tanh silently runs in PyTorch outside MindSpore differentiation. LaDiM registers MindSpore's `tanh` and passes public verification at call 23, with acceptance confirmed on three seeds with 1.488 million token consumption over all 24 calls. SWE-agent and MatchFixAgent also suspect Tanh but never turn the suspicion into a tested patch, exhausting the 40-call limit after 1.606 and 2.341 million tokens. Neither the loss nor the gradient-norm difference (0.046, below its 0.05 threshold) flags this fault; the per-parameter gradient comparison exposes it, and agreement in the neighboring layers localizes it to Tanh. A loss comparison can show that training has drifted, but not where. Appendix A.6 reports further detection measurements.

Generalization Across Frameworks. LaDiM repairs all six JAX cases spanning execution, forward, and gradient faults in MLP and CNN models (Table 2b). Every case passes single-step SGD verification on three seeds after the first submission. Direct repair also accepts all six cases and uses fewer tokens. Ivy and `torch2jax` accept none of the faulty candidates after native conversion. Both converters pass evaluation on the corresponding MLP and CNN programs without these faults. In these cases, native conversion preserves the supplied execution or numerical faults, and explicit repair restores agreement in training behavior. The same diagnosis and repair procedure thus works beyond MindSpore with JAX differentiation and parameter updates. Appendix A.5 details the converter evaluations. Appendix H reports the separate study of native JAX translations evaluated over consecutive training steps.

4.4 Ablation Studies

Training signals. We compare cumulative feedback on 16 fixed candidates from a CNN, an image MLP, a Transformer classifier, and a small causal language model. Each model contributes four

Table 3: Training signals and repair components. (a) Cumulative signals on 16 faulty candidates, scored with complete verification. Each + adds a signal. (b) Repair history and evidence handoff on ten translations, where parentheses count repairs among five failing programs. (c) Repository components on time series. All settings pass 69 checks across three seeds. Tokens are in millions.

(a) Training signals

Feedback	Accepted	Tokens
Execution and basic checks	4/16	2.089
+ Forward values	8/16	2.603
+ Gradients	12/16	3.303
+ Parameter updates	**16/16**	3.419

(b) Repair components

Condition	Accepted	Tokens
LaDiM	**9/10 (4/5)**	15.548
Continuous conversation	8/10 (3/5)	14.516
Without repair history	5/10 (0/5)	10.852

(c) Components on the time series repository

	Without component		With component		
Component	**Calls**	**Tokens**	**Calls**	**Tokens**	**Saved**
Repository context management	26	6.047	22	1.937	**68.0%**
Repository Structural Analysis	42	2.651	30	2.127	**19.8%**

candidates with execution, forward, gradient, and update faults, respectively. Sources, candidates, tools, and budgets are fixed. Available feedback determines whether repair begins and stops. Each added signal exposes another fault class, increasing final acceptance under complete verification from 4/16 to 16/16 (Table 3a). Initially undetected faults decrease from 12 to 8 to 4 to 0 across the four settings. Forward values reveal changed computations, gradients reveal differentiation faults, and updates reveal optimizer faults.

Repair history and independent evidence handoff. Repair history and independent evidence handoff improve repair on the ten translated programs (Table 3b). The LaDiM row enables both components, and the other two rows change one component at a time. With repair history, LaDiM repairs four of five failed programs. Removing that history reduces the count to zero. Independent evidence handoff repairs one more program than a shared investigation and repair conversation, increasing final acceptance from 8/10 to 9/10. All three settings preserve the five programs that pass evaluation before repair. Repair history retains earlier attempts and their verification feedback, while independent evidence handoff lets the Repair Agent reassess the Verifier's evidence in its own conversation.

Repository context management and Repository Structural Analysis. On the time series repository, repository context management reduces end-to-end token use by 68.0% while preserving acceptance on all 69 checks (Table 3c). Both conditions retain investigation and independent evidence handoff. A separate comparison on the same repository evaluates LaDiM with and without Repository Structural Analysis. Enabling this component reduces end-to-end token use by 19.8% and LLM calls from 42 to 30, with all 69 checks still passing. Repository Structural Analysis identifies relevant implementations. Repository context management keeps plans, measurements, and inspected code available throughout the investigation. Appendix B gives the complete program component results, and Appendix G reports both repositories' cumulative comparisons and independent ablations of Repository Structural Analysis and Dependency Graph Planning.

5 Conclusion

We introduced LaDiM for preserving training behavior during code migration. Layered Diagnosis traces dependencies among execution, forward values, gradients, and parameter updates. Independent evidence handoff and repair history guide corrective edits, while repository coordination manages work across shared implementations and entry points. Experiments across frameworks, languages, and repositories show that LaDiM preserves training behavior with lower token costs and repairs translation errors that the repair baselines leave unresolved. By tracing training behavior stage by stage, LaDiM turns silent computational drift into localized, repairable evidence, making framework-level migration of deep learning programs reliable at practical cost.

AI USE STATEMENT

We used generative AI for preliminary literature research, method implementation, result analysis, and manuscript and figure editing. The authors checked the code, results, and citations and take responsibility for the final work.

REPRODUCIBILITY STATEMENT

The supplementary material contains the implementation of LaDiM, the evaluation scripts, and the configurations used to run all baselines, anonymized for review. Section 4.1 and Appendix A specify the task collections, numerical tolerances, random seeds, and call, token, and time budgets. Appendix I lists the LLM backend and decoding settings shared by all methods. All reported costs include failed calls and the shared initial translation, as described in Appendix A.1.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265–283, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=EHg5GDnyq1.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=KuPixIqPiq.

Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. Differential testing of cross deep learning framework APIs: Revealing inconsistencies and vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 7393–7410. USENIX Association, 2023. URL https://www.usenix.org/conference/usenixsecurity23/presentation/deng-zizhuang.

DJL Contributors. Dive into deep learning with java. https://github.com/deepjavalibrary/d2l-java, 2026a. Source repository, accessed September 21, 2026.

DJL Contributors. Deep java library. https://github.com/deepjavalibrary/djl, 2026b. Source repository, accessed September 21, 2026.

gauravchak. Two tower models. https://github.com/gauravchak/two_tower_models, 2026. Source repository, accessed September 21, 2026.

Google. torchax: A PyTorch frontend for JAX. https://google.github.io/torchax/, 2025. Accessed: 2026-06-02.

Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. RepoTransAgent: Multi-agent LLM framework for repository-aware code translation. *arXiv preprint arXiv:2508.17720*, 2025. doi: 10.48550/arXiv.2508.17720. URL https://arxiv.org/abs/2508.17720.

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, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.362. URL https://aclanthology.org/2025.emnlp-main.362/.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. MatchFixAgent: Language-agnostic autonomous repository-level code translation validation and repair. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=MuyXpH3GL1.

Ivy Contributors. Ivy: Convert machine learning code between frameworks. https://github.com/unifyai/ivy, 2026. Accessed: 2026-09-22.

jinglescode. Time series forecasting with pytorch. https://github.com/jinglescode/time-series-forecasting-pytorch, 2026. Source repository, accessed September 21, 2026.

Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 216–226. ACM, 2014. doi: 10.1145/2594291.2594334. URL https://doi.org/10.1145/2594291.2594334.

Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepGauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 120–131. ACM, 2018. doi: 10.1145/3238147.3238202. URL https://doi.org/10.1145/3238147.3238202.

Marcos Macedo, Yuan Tian, and Bram Adams. CodeTransEngine: Ready-to-use backend for LLM-based code translation. In *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025a. URL https://openreview.net/forum?id=9WHJuzjPJf.

Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation. In *Proceedings of the 47th International Conference on Software Engineering*, pp. 2819–2831. IEEE, 2025b. doi: 10.1109/ICSE55347.2025.00236. URL https://doi.org/10.1109/ICSE55347.2025.00236.

MindSpore. MindSpore: An all-scenario deep learning computing framework. https://mindspore-website.obs.cn-north-4.myhuaweicloud.com/white_paper/MindSpore_white_paper_enV1.1.pdf, 2020. White paper, version 1.1.

MindSpore. Migrating from third party frameworks with MindConverter. https://www.mindspore.cn/mindinsight/docs/en/r2.0.0-alpha/migrate_3rd_scripts_mindconverter.html, 2023. Accessed: 2026-06-02.

MindSpore. Gradient derivation. https://www.mindspore.cn/docs/en/r2.4.0/migration_guide/model_development/gradient.html, 2024. MindSpore 2.4.0 documentation, Principle Comparison. Accessed: 2026-09-24.

MindSpore. TroubleShooter loss_compare: Comparing loss curves of two training logs. https://github.com/mindspore-ai/toolkits/blob/e0486eee96a1893b5e1b59f02669804a2d796030/troubleshooter/docs/api/widget/loss_compare.md, 2025. Accessed: 2026-09-25.

MindSpore. Large model precision optimization guide. https://www.mindspore.cn/mindformers/docs/en/r1.7.0/advanced_development/precision_optimization.html, n.d. MindSpore Transformers r1.7.0 documentation. Accessed: 2026-09-25.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 772–781. IEEE, 2013. doi: 10.1109/ICSE.2013.6606623. URL https://doi.org/10.1109/ICSE.2013.6606623.

OpenI. MSAdapter. https://openi.pcl.ac.cn/OpenI/MSAdapter/src/tag/v0.6.0, 2025. Version 0.6.0. Accessed: 2026-09-20.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary 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 *Advances in Neural Information Processing Systems*, volume 32, pp. 8024–8035, 2019. URL https://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1–18. ACM, 2017. doi: 10.1145/3132747.3132785. URL https://doi.org/10.1145/3132747.3132785.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 15174–15186. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pp. 20601–20611, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html.

Baptiste Rozière, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. In *Proceedings of the Tenth International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=cmt-6KtR4c4.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.11366.

Marc Szafraniec, Baptiste Rozière, Hugh James Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. In *Proceedings of the Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=XomEU3eNeSQ.

torch2jax Contributors. torch2jax: Run PyTorch in JAX. https://github.com/samuela/torch2jax, 2026. Accessed: 2026-09-22.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232, Vienna, Austria, 2024. PMLR. URL https://proceedings.mlr.press/v235/wang24h.html.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Proceedings of the First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=uAjxFFing2.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, pp. 1482–1494. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00129. URL https://doi.org/10.1109/ICSE48619.2023.00129.

Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023. doi: 10.1109/ICSE48619.2023.00105. URL https://arxiv.org/abs/2302.04351.

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. *arXiv preprint arXiv:2405.15793*, 2024. doi: 10.48550/arXiv.2405.15793. URL https://arxiv.org/abs/2405.15793.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 283–294. ACM, 2011. doi: 10.1145/1993498.1993532. URL https://doi.org/10.1145/1993498.1993532.

A Additional Experimental Protocols

A.1 Task Construction and Accounting

Table 4: Program collections for migration and repair, with their candidate construction, model coverage, and task counts.

Collection	Candidate construction	Model or operation coverage	Tasks
PyTorch to MindSpore	Generated translations	Operators, CNNs, MLPs, Transformers, language models	50
Initial translations	Five fail and five pass their first evaluation	Recurrent, convolutional, attention, and set models	10
Training signal ablation	Controlled training faults	CNN, image MLP, Transformer, causal language model	16
JAX repair	Supplied faulty candidates	MLP and CNN	6

The main MindSpore collection contains 50 task identifiers. Twelve tasks reuse existing PyTorch source programs, and the remaining tasks use programs authored from public interfaces and declared semantics. The identifiers correspond to 29 distinct combinations of source program and evaluation settings drawn from 24 source files. Conditions with identical source, evaluation settings, initial candidate, method, and inputs are executed once and associated with the corresponding identifiers. Acceptance in the main table uses all 50 identifiers, while token totals count actual calls. Over the 29 distinct inputs, LaDiM and MatchFixAgent each accept 29/29, and SWE-agent accepts 25/29. The four remaining SWE-agent inputs comprise three failed outcomes and one interrupted run with unavailable final acceptance; all remain in the denominator. LaDiM, SWE-agent, and MatchFixAgent share the same initial translation for each distinct input. Methods may edit the candidate program and create separate temporary tests for investigation. Sources and task specifications remain fixed.

The comparison across languages contains ten DJL application programs and eight textbook examples. LaDiM accepts the fruit classification application using transfer learning and all eight examples. MatchFixAgent accepts the same application and seven examples, while SWE-agent accepts the MNIST LSTM application and seven examples. Test-guided repair accepts seven examples. All repair conditions share the programs produced by initial translation and their generation costs. InterTrans uses its direct Java-to-Python path and an intermediate JavaScript path, with native search and extraction procedures. Its 2.286 million tokens include 0.574 million from incomplete generation calls and 1.712 million from the completed search. The final evaluation compares outputs, losses, gradients, parameter updates, and observable optimizer state over two consecutive training steps.

The common MindSpore initial translations cost 0.429 million tokens. This amount is included once in each repair method's total. Subsequent LaDiM calls use 1.410 million tokens on programs that pass evaluation before repair and 3.320 million on programs that need repair. The corresponding MatchFixAgent totals are 5.136 million and 6.532 million. Across the 29 distinct inputs, the median ratio of LaDiM to MatchFixAgent total tokens is 0.588. LaDiM uses fewer tokens on 19 of the 20 programs that pass before repair and eight of the nine that fail, giving 27/29 overall. LaDiM repairs seven of the nine failing programs at the first submission. Shared initial translation costs cancel when their paired costs are subtracted. MatchFixAgent accepts all tasks at its first submission, which can include multiple analysis and repair calls.

All failed tasks remain in each study's declared denominator, and all failed calls remain in its cost totals. Acceptance at a submission budget scores the current candidate, including any regression after an earlier pass. An edit counts one successfully applied editing operation and can replace several code blocks across files in the declared scope. Investigation and repair share cumulative call, output token, and time budgets across submissions. Token usage is reported in millions and rounded to three decimal places, budgets and per-call limits are given as exact token counts.

Table 5: Complete input and output token costs by method for the two program migration collections. Input, output, and total tokens are in millions. Calls and tokens include initial translation and every subsequent call. Acceptance uses the declared task denominator for each collection.

Method	Calls	Input	Output	Tokens	Accepted
(a) PyTorch to MindSpore					
Direct LLM	29	0.018	0.411	0.429	29/50
CodeTransEngine (Macedo et al., 2025a)	29	0.022	0.257	0.280	31/50
MSAdapter (OpenI, 2025)	–	–	–	–	15/50
SWE-agent (Yang et al., 2024)	1,097	19.845	0.987	20.831	44/50
MatchFixAgent (Ibrahimzada et al., 2026)	619	10.750	1.347	12.097	**50/50**
LaDiM (ours)	326	4.298	0.861	5.159	**50/50**
(b) Java/DJL to Python/PyTorch					
Direct LLM	18	0.142	0.552	0.694	1/18
InterTrans (Macedo et al., 2025b)	89	0.680	1.606	2.286	1/18
Test-guided repair	71	1.594	2.881	4.475	7/18
SWE-agent	682	23.063	1.258	24.321	8/18
MatchFixAgent	524	26.335	2.020	28.355	8/18
LaDiM (ours)	345	19.470	2.009	21.479	**9/18**

A.2 Numerical Comparisons and Migration Settings

The main MindSpore collection uses absolute/relative tolerances of $10^{-4}/10^{-4}$ for forward values, $10^{-4}/10^{-3}$ for gradients, and $10^{-5}/10^{-3}$ for updates. BF16 matrix multiplication uses absolute tolerance 10^{-2}. Shapes, data types, initial values, and trainability must match. Parameter and optimizer tasks test their specified enumeration, gradient assignment, or update behavior. Source and target programs begin from corresponding initial states and retain their own state over two consecutive steps for individual migration and three for repositories. The source program and task requirements are available to each method. Acceptance also checks that the target framework executes the required computations.

SWE-agent and MatchFixAgent retain their native algorithms, prompts, tools, parsers, and retry policies. SWE-agent uses its native controller, and MatchFixAgent runs its analysis and repair procedure through the shared DeepSeek tool backend. CodeTransEngine generates its own translation through the native direct path, and MSAdapter uses its native interface mapping.

For each program, each agent has cumulative investigation and repair limits of 40 LLM calls, 120,000 output tokens, 1,800 seconds, and four submissions. Each call allows at most 16,384 output tokens, and LaDiM allows eight calls per investigation or repair stage. Initial translation is separate from this repair budget and allows at most 131,072 output tokens per call. Test-guided repair rewrites the complete candidate, with four repair rounds and at most 131,072 output tokens per call. For each repository, each agent has cumulative limits of 80 calls, 480,000 output tokens, 3,600 seconds, and four submissions, with at most 32,768 output tokens per call. Output budgets include reasoning tokens, and all input usage is recorded.

A.3 Random Seeds and Initial States

Individual migration. The main MindSpore and Java/DJL comparisons use seed 101 for the evaluation available during repair and seeds 202 and 303 to confirm a passing candidate. We refer to these as the public and confirmation seeds, respectively. The MindSpore evaluator uses NumPy's random generator to sample numerical inputs and training targets. Corresponding parameters are assigned identical initial values generated from a hash of each parameter name; these values remain fixed across the three input seeds. Discrete inputs, masks, and prescribed optimizer-test values retain their task-specific settings. For Java/DJL programs, each paired evaluation supplies the original initial parameters and two preprocessed batches to the candidate for consecutive training steps. The component comparison on the main MindSpore collection uses the same settings.

Initial translations and training signals. The study of ten programs produced by initial translation uses seed 42 for public evaluation and 1042 and 2042 for confirmation. The 16 training-signal

candidates use 4101, 5101, and 6101, with 4101 serving as the public seed. The paired evaluator sets the Python, NumPy, and PyTorch random states before constructing the source model and copies its initial state to both executions. A separate CPU PyTorch generator seeded with $s + 101$ produces inputs and labels for evaluation seed s, including token inputs for text models. The signal study uses all three seeds when checking fault construction and scoring the final candidate with all signals.

JAX repair. The MLP and CNN programs use public seed 6701 and confirmation seeds 6702 and 6703. The evaluator initializes PyTorch's random state with the evaluation seed before constructing the source or candidate model. Inputs and class labels use a separate CPU generator with seed $s + 101$.

Detection trajectories. Seeds 300, 301, and 302 define three repeated trajectories for each model. Each seed initializes the model's floating-point parameters, and its initial state is shared across the source and target executions. Image batch order uses seed $s + 5000$, text-classification batch order uses $s + 6000$, and language-model window selection uses $s + 7000$. The fault and synchronization conditions for a given model and seed share the same initial state and 50-step batch sequence.

Time series repository. The main comparison and component ablation use public seed 101 and confirmation seeds 202 and 303. The source model is initialized with the evaluation seed, and its parameters are copied to the target. NumPy generators use seed $s + 47$ for the three training batches and seed s for the synthetic price series. The target MindSpore random state also uses s. The training script, notebook, and teaching examples are exercised in the public evaluation; confirmation repeats the numerical checks on the remaining two seeds.

Recommendation repository. The public numerical evaluation uses seed 101 for source model initialization and NumPy generation of user identifiers, features, histories, item identifiers, positions, and labels. The source parameters and retrieval corpus are then supplied to the target, whose MindSpore random state uses the same seed. Confirmation with seeds 202 and 303 begins after all public checks pass. The reported recommendation runs do not reach this stage. The original tests use seed 42, including Python, NumPy, and framework random states. For exact comparisons involving the history encoder, the task supplies source parameters initialized with PyTorch seed 42. The one-epoch training command uses seed 101.

A.4 Training Signal Study

The training signal and JAX repair studies use the investigation and evidence handoff procedure with editing examples and feedback for correcting edit formats. The signal experiment introduces each of four faults into each of four model classes, producing 16 candidates from four source programs. The execution fault uses an invalid optimizer keyword, the forward fault adds a constant loss offset, the gradient fault scales gradients, and the update fault changes the learning rate.

All feedback conditions retain the initial investigation and current working files across submissions, with the same cumulative call budget. The checks visible to the agent determine whether repair starts and when it stops. Basic checks cover execution, output structure, use of the target framework, preservation of the source program, and the required training records. Final acceptance uses all training signals on three seeds. MindSpore measurements use our PyTorch-to-MindSpore bridge library (anonymized for review) for forward and backward computation, followed by optimizer updates on the target parameters.

A.5 JAX Repair and Native Conversion

The JAX repair study uses three faulty versions of an MLP and three of a CNN, with an execution, forward, or gradient fault in each model. LaDiM and Direct repair receive the same source programs and faulty candidates. Direct repair keeps investigation and repair in one conversation and permits two LLM calls and one editing operation per repair attempt. Both methods may edit the candidate and create temporary tests, while the source and evaluation settings remain fixed. For each task, both methods allow up to four submissions, 40 LLM calls, 120,000 output tokens, and 1,800 seconds, including investigation. Both methods accept every task at the first submission on seeds 6701, 6702, and 6703. Across the six tasks, LaDiM uses 76 calls and 0.674 million tokens, and Direct repair uses 57 calls and 0.353 million tokens.

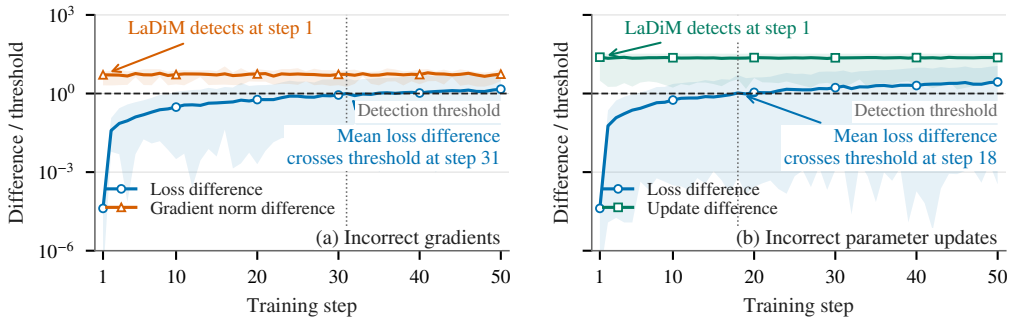


Figure 5: Detection of (a) gradient scaling and (b) partial update suppression over 50 training steps. Solid lines show the mean over 12 runs (four models $\times$ three seeds), and shaded bands show the minimum and maximum across these runs at each step. Each difference is divided by its detection threshold, so values above the dashed line count as detected. Gradient and update checks detect both faults at step 1 in every run; with the loss threshold of 0.02, the mean loss difference first crosses at steps 31 and 18, respectively.

TorchAX, a PyTorch frontend for JAX (Google, 2025), executes the candidates for both repair methods with JAX arrays. JAX computes gradients, and Optax applies one SGD step. The thresholds are 0.02 for the absolute loss difference, 0.05 for the absolute difference between gradient norms, and 0.03 for the relative parameter update difference. The accepted repairs match the reference initial parameters exactly. The maximum final loss, gradient norm, and relative update differences are 1.192×10^{-7}, 1.510×10^{-8}, and 1.193×10^{-6}.

Ivy produces a JAX/Flax implementation of each supplied model. The `samuela/torch2jax` implementation converts models and tensors to JAX, followed by JAX differentiation and Optax updates. Each performs a single native conversion of the already faulty candidate. For both converters, the two execution faults retain an extra argument to a linear layer and raise `TypeError`, leaving numerical measurements n/a. The two forward faults preserve the output scaling error and fail loss, gradient, and update checks. The two gradient faults preserve the interrupted differentiation path: loss passes, while gradients and updates fail. Both converters pass evaluation on the corresponding MLP and CNN programs without these faults. Ivy takes 561.935 seconds for the six faulty candidates and 191.663 seconds for the two programs without faults. The corresponding times for `torch2jax` are 39.234 and 15.930 seconds. Their LLM calls and tokens are zero.

A.6 Detection Measurements

The detection study uses four models, three seeds, and 50 training steps. Gradient scaling and partial update suppression are evaluated both with per-step synchronization to the source state and with each execution retaining its own state. With synchronization, each target starts the checked step from the source state. The detection thresholds for the absolute loss difference, absolute difference between gradient norms, and relative parameter update difference are 0.02, 0.05, and 0.03.

Fig. 5 shows trajectories in which the source and target retain their own states. It reports the mean and range over 12 runs per fault, comprising four models with three seeds each. The mean loss difference crosses its threshold at step 31 for gradient errors and step 18 for update errors. Within 50 steps, the loss difference exceeds its threshold in three of the 12 runs with gradient errors and four of the 12 runs with update errors. Direct gradient or update checks detect the corresponding faults at step 1 in all 12 runs. Translations without these faults pass the recorded checks, with maximum loss difference 3.815×10^{-6}.

Loss comparison is an established check in framework migration. MindSpore TroubleShooter's `loss_compare` tool plots the loss curves of two training logs and reports their error statistics (MindSpore, 2025), and the MindSpore Transformers precision guide compares the step-1 loss before gradient norms and updated weights (MindSpore, n.d.). Neither prescribes an automatic threshold, so the loss-only check in Fig. 5 applies our threshold of 0.02 to the recorded loss differences. Because both faults take effect after the loss is computed, every faulty run has exactly the

same loss difference at step 1 as its fault-free counterpart. Lowering the threshold detects the faults earlier (Table 6): at 10^{-5}, the loss check detects all 24 faulty runs by step 8 without false alarms, since the largest loss difference among fault-free runs is 3.815×10^{-6}. All thresholds are applied post hoc to the same saved trajectories.

Table 6: Loss-only detection at different thresholds, computed post hoc on the saved trajectories (12 runs per fault, 50 steps). First detection gives the range of first-crossing steps over detected runs; mean crossing is the first step at which the mean loss difference crosses the threshold; false alarms count fault-free runs that cross the threshold.

	(a) Gradient scaling				(b) Partial update suppression			
Threshold	Detected	**First detection**	**Mean crossing**	**False alarms**	Detected	**First detection**	**Mean crossing**	**False alarms**
0.02	3/12	10–11	31	0/12	4/12	6–20	18	0/12
0.005	12/12	3–37	9	0/12	8/12	3–42	6	0/12
0.001	12/12	2–8	3	0/12	9/12	2–14	2	0/12
10^{-5}	12/12	2	2	0/12	12/12	2–8	2	0/12

The detection study uses the following scalar differences:

$$\Delta_{\text{loss}} = |L(P) - L(\hat{P})|, \tag{3}$$

$$\Delta_{\text{grad}} = \left| \|\nabla_\theta L(P)\|_2 - \|\nabla_{\hat{\theta}} L(\hat{P})\|_2 \right|, \tag{4}$$

$$\Delta_{\text{param}} = \frac{\|\Delta\theta - \Delta\hat{\theta}\|_2}{\|\Delta\theta\|_2 + \epsilon}. \tag{5}$$

Here $\Delta\theta$ and $\Delta\hat{\theta}$ denote source and target parameter updates, and ϵ prevents division by zero. Fig. 5 divides each difference by its detection threshold. The main migration comparison uses the task-specific tensor comparisons defined above.

B Evidence Handoff and Agent Integration

The component study uses the same 50 MindSpore task identifiers, initial translations, acceptance checks, and budgets as the main comparison. The continuous conversation condition keeps investigation and repair in one conversation, including earlier messages and reasoning history. Independent evidence handoff supplies the Verifier's evidence to a separate repair conversation. Without repair history, the context for each repair attempt after the first starts from the initial investigation and latest observation. The current files and cumulative budget are retained. Disabling progress reminders leaves the call and submission limits unchanged. LaDiM accepts 50/50 task identifiers, compared with 49/50 when investigation and repair share one conversation. The variants without repair history and without progress reminders each accept 50/50 while using fewer tokens than LaDiM. Among

Table 7: Component variants and added investigation for baseline agents on the 50 MindSpore task identifiers. Tokens are in millions and include initial translation and all subsequent calls.

Setting	**Accepted**	**Tokens**
LaDiM	50/50	5.159
Continuous conversation	49/50	5.603
Without repair history	50/50	4.566
Without progress reminders	50/50	4.562
MatchFixAgent	50/50	12.097
MatchFixAgent with independent investigation	50/50	19.480
SWE-agent	44/50†	20.831
SWE-agent with independent investigation	43/50	36.547

† SWE-agent without added investigation has one interrupted task with unavailable final acceptance. It remains in the denominator and cost totals. Tasks with identical inputs are executed once as described in Appendix A.1.

the 29 distinct inputs represented by these identifiers, 20 translated programs pass evaluation before repair. Only two of the nine failing programs require a second repair submission under LaDiM. Most inputs are therefore accepted before a later repair attempt can use earlier repair history.

The second component study uses the same ten programs produced by initial translation as Table 2a, of which five fail evaluation and five pass before repair. Methods may edit these programs and their supporting libraries. LaDiM in Table 8 uses independent evidence handoff and repair history, with examples of the edit format and corrective feedback for malformed edits. Each other condition changes one component while retaining the candidates, workspace access, acceptance checks, and repair budgets. Removing the edit examples and format correction gives the LaDiM configuration reported in Table 2a. The edit-format assistance in Table 3b therefore accounts for the different LaDiM token totals in the two tables. All conditions also pass evaluation on the five programs that pass before repair. For SWE-agent and MatchFixAgent, we add the same investigation before repair, with code editing disabled during investigation. The investigation counts toward each baseline agent's total call, token, and time limits. Each baseline agent retains its own history and repair policy.

Table 8: Complete component results on ten programs produced by initial translation. Repaired counts failed programs corrected, and Preserved counts passing programs that remain accepted. Tokens are in millions and cover investigation and repair on all ten programs, including failed attempts.

Condition	**Repaired**	**Preserved**	**Tokens**	**Accepted**
LaDiM	4/5	5/5	15.548	9/10
Continuous conversation	3/5	5/5	14.516	8/10
Without repair history	0/5	5/5	10.852	5/10
Without progress reminders	2/5	5/5	17.325	7/10
Without editing format assistance	4/5	5/5	13.207	9/10

MatchFixAgent with investigation accepts 50/50 at a higher token total. SWE-agent accepts 43/50 with the added investigation and 44/50 without it. Among the nine distinct programs that fail evaluation before repair, SWE-agent repairs seven without the added investigation and eight with it. With the added investigation, four of the 20 programs that pass before repair fail the final evaluation. Final acceptance in Table 7 counts both repaired programs and programs that continue to pass after repair.

C Additional Repair Collections

The controlled MindSpore repair collection contains 50 injected faults in supplied target programs, with task-specific acceptance checks. Direct repair uses the same inspection, testing, and editing tools as LaDiM, while SWE-agent uses its native tools. LaDiM accepts 45/50 within four submissions, compared with 29/50 for SWE-agent and 35/50 for Direct repair. LaDiM's cumulative accepted counts are 29, 41, and 45 with submission limits of one, two, and four, respectively. Total token use is 39.983, 42.579, and 12.459 million for LaDiM, SWE-agent, and Direct repair, respectively.

A separate collection supplies twelve source programs together with faulty target programs, providing the program pairs required by MatchFixAgent. LaDiM accepts 12/12 and MatchFixAgent accepts 11/12, using 6.643 and 14.506 million tokens, respectively. Both receive the complete source program, faulty target program, task specification, and target library. Each method has up to four submissions, 40 LLM calls, 120,000 output tokens, and 1,800 seconds per task. Both repair collections permit edits to the candidate and its supporting library. Their original tests check execution and computed values, without an additional confirmation stage across random seeds.

D Initial Translation Repair Protocol

Of the ten programs produced by initial translation, five fail evaluation and five pass before repair. Each method receives the same source, candidate, task requirements, and supporting library. The candidate and library may be edited. The source and acceptance criteria remain fixed.

Target forward computation and differentiation execute in MindSpore through this bridge library. PyTorch then applies optimizer updates to the target parameters. The comparison uses matching inputs and initial parameters, with acceptance confirmed on three seeds. The thresholds are 0.02 for the absolute loss difference, 0.05 for the absolute difference between gradient norms, and 0.03 for the relative parameter update difference. The L2 difference between gradient vectors has an additional threshold of 0.05, alongside the specified layer and parameter checks.

The LLM call counts in Table 2a cover all ten programs, including failed repairs and investigation of programs that pass before repair. LaDiM uses 193 calls, Direct repair uses 120, SWE-agent uses 386, and MatchFixAgent uses 265. LaDiM uses 13.207 million tokens.

In the LSTM example in Section 4.3, LaDiM repairs two errors in the supporting library. The first submission leaves an unregistered recurrent operator unresolved. LaDiM reproduces the failure, implements the recurrence in MindSpore, and registers the recurrent operator. The second submission exposes recursive tensor indexing. A focused test identifies the repeated calls in tensor indexing, which the third submission corrects. On the public evaluation seed, the final absolute loss difference is 2.38×10^{-7}, gradient vector L2 difference is 4.19×10^{-7}, and relative parameter update difference is 8.11×10^{-7}. The example uses 32 LLM calls.

In the autoencoder example of Fig. 4, the supporting library registers the low-level Tanh operator but not `torch.tanh` or `torch.nn.functional.tanh`. Unregistered calls fall back to PyTorch on copied tensors, which preserves forward values but disconnects the MindSpore differentiation graph. Before repair, on the public evaluation seed, layer outputs differ by at most 1.5×10^{-8} and the absolute loss difference is 1.43×10^{-6}. The gradients of the first linear layer's weight and bias are zero, against norms of 0.342 and 0.109 in the source, while the other six parameter gradients differ by at most 6.2×10^{-7}. The gradient vector L2 difference is 0.359 and the relative parameter update difference is 0.771, whereas the absolute gradient-norm difference, 0.046, stays below its threshold. LaDiM's first submission contains no edits and fails. Its diagnostic test compares summed absolute input gradients under MindSpore differentiation: 4.894 for native `tanh` and zero for dispatched Tanh, while dispatched ReLU, sigmoid, and multiplication all pass nonzero gradients. The patch adds a four-line registration to the library, and the second submission is accepted on all three seeds, with gradient vector L2 difference 1.01×10^{-6} and relative parameter update difference 6.42×10^{-6} on the public seed. SWE-agent and MatchFixAgent both attribute the zero gradients to Tanh, and MatchFixAgent even drafts the same registration, yet neither applies an edit to the library within 40 calls, so both leave the program unrepaired. This case illustrates the completion condition in Section 3.3: the Repair Agent must apply an edit and test after that edit before completing the repair stage.

E Repository Protocols and Detailed Results

All methods receive identical copies of the source repositories and the same initial translations, task requirements, and acceptance procedures. The time series translation has three generation calls totaling 0.164 million tokens. The recommendation translation uses 20 calls totaling 0.368 million tokens. Translation follows the source repository's imports, and its cost includes failed or truncated attempts. In Table 9, each method's total includes the common initial translation once and all subsequent investigation and repair calls. The time series protocol compares the source and translated core models for three consecutive Adam steps from corresponding parameters, with dropout disabled and input shape $4 \times 20 \times 1$. Preprocessing and prediction absolute tolerances are 10^{-9} and 10^{-4}. The tolerances are 0.02 for the absolute loss difference, 0.05 for the L2 difference between gradient vectors, and 0.03 for the relative parameter update difference. Learning rate behavior is checked around its schedule boundaries. The public evaluation executes the 100-epoch training workflow, notebook cells, and teaching snippets in their source order. Each entry point must produce the required predictions and plots and compute gradients through the target framework's native differentiation. The notebook and snippets test execution, while the core model provides the numerical comparison of source and target training. All three methods pass 23 behavior checks on each of the public and two confirmation seeds, giving 69/69 overall.

The recommendation protocol compares six model execution paths in the source and translated programs over three sequential Adam updates. It also checks whether the translated program reproduces source exceptions and empty returns. Both the numerical evaluation and training command use the

Table 9: Training checks and complete costs for repository migration. Each training signal has nine checks for time series, from three steps on three seeds, and 18 for recommendation, from six model execution paths and three steps on the public seed. Calls, input tokens, and output tokens cover investigation and repair. Tokens reports the total including the common initial translation. Tokens are in millions.

Method	Gradients	Parameter updates	Calls	Submissions	Input	Output	Tokens
Time series							
LaDiM	9/9	9/9	35	2	2.345	0.105	2.614
SWE-agent	9/9	9/9	80	1	7.985	0.051	8.200
MatchFixAgent	9/9	9/9	69	2	13.306	0.128	13.598
Recommendation							
LaDiM	18/18	15/18	80	4	8.148	0.295	8.811
SWE-agent	17/18	12/18	80	1	7.890	0.058	8.317
MatchFixAgent	18/18	16/18	80	1	8.215	0.095	8.679

candidate's function for constructing the optimizer, with learning rate 0.001. The source Adam settings use decay coefficients 0.9 and 0.999 and epsilon 10^{-8}. Representation absolute tolerance is 10^{-4}. Loss comparisons use absolute tolerance 0.002 and relative tolerance 10^{-4}; gradient comparisons use 10^{-4} and 0.005, respectively. Relative parameter update tolerance is 0.03. The evaluation includes all ten original tests and one epoch of the training command on 16 synthetic examples.

Table 10 reports the recommendation checks on the public seed. The 114 numerical comparisons comprise 108 checks over six paths, three steps, and six training signals, plus six inference retrieval checks. Forward loss is returned by the model's forward computation; training return loss is returned by the training function on the same batch. Each has 18 checks. Twenty-four execution, step count, and native differentiation checks and seven source behavior checks bring the behavior check total to 145.

SWE-agent and MatchFixAgent each lack one retrieval measurement after reward model inference fails. These values are n/a, and the missing checks remain in the retrieval, numerical, and behavior denominators of six, 114, and 145. MatchFixAgent has no confirmed passes among the ten original tests because an unmigrated PyTorch import in its history encoder test prevents test collection under the native MindSpore requirement. This is reported as 0/10. No method passes all public checks, so the two confirmation seeds are not evaluated on this repository.

File coverage checks that the translated repository contains the corresponding Python files and public function and class names, preserves the original test function names, and has valid Python syntax in files and README code blocks. It also checks use of MindSpore in place of the source or alternate frameworks. Documentation and dependency checks require the README and dependency list to identify MindSpore, and the dependency list to exclude the source and alternate frameworks. These repository checks are reported separately from the 145 behavior checks.

LaDiM passes all 18 forward loss, 18 training return loss, and 18 gradient checks, together with 15/18 parameter update checks, in the main recommendation comparison. Item representations exceed the comparison tolerance in all six model execution paths at steps two and three. LaDiM also completes reward model inference, retrieval, the training command, and all ten original tests.

F Repository Context and Evidence

Repository context management tracks changes to the dependency plan and candidate files separately. Every recorded test identifies the code version on which it ran. After a code change, measurements from the old version no longer describe the current candidate. A new result replaces the previous result for the same test, including when the new test fails. The validity of each work unit's checks also depends on its files, selected tests, and prerequisite units. Changes to these files or dependencies invalidate the affected checks and those of declared dependent units. A unit checked without a selected test records syntax validation only. Final acceptance requires evaluation of the complete repository.

Table 10: Recommendation repository checks passed on the public seed. The 114 numerical checks are included in the 145 behavior checks. The six training signal rows divide the 108 training checks, with 18 per signal. One unmeasured retrieval check remains in the denominators for SWE-agent and MatchFixAgent.

Check	**LaDiM**	**SWE-agent**	**MatchFixAgent**
Behavior checks passed	130/145	104/145	129/145
Numerical checks	99/114	74/114	99/114
Training checks	93/108	69/108	94/108
Inference retrieval	6/6	5/6	5/6
User representations	18/18	16/18	18/18
Item representations	6/18	6/18	6/18
Forward loss	18/18	18/18	18/18
Training return loss	18/18	0/18	18/18
Gradients	18/18	17/18	18/18
Parameter updates	15/18	12/18	16/18
Original tests	10/10	10/10	0/10
Training command	1/1	1/1	1/1
File coverage	1/1	1/1	0/1
Documentation and dependencies	1/1	1/1	0/1

Full tool observations, previously read code, and conversations are stored outside the editable workspace and remain available for later retrieval. When rebuilding the conversation context, LaDiM includes the current dependency plan and translation progress, the complete structured findings, hypotheses and next steps, the latest measurements, and the most recent tool results. It also includes up to 120,000 characters of previously read code that still matches the current files and is relevant to the active work unit. Context is rebuilt for independent evidence handoff, when switching between work units, or when messages exceed 75% of the configured context capacity. Continuing a stage or selecting the first work unit does not itself trigger reconstruction.

For the repository experiments, investigation allows up to six LLM calls and 64,000 output tokens. The remaining calls and output tokens are allocated over the remaining permitted repair submissions, with at most 16 calls in the first repair stage. Investigation and all repair stages share the cumulative limits for the whole repository.

G Repository Component Results

G.1 Cumulative Components

We add Verifier investigation, independent evidence handoff, and repository context management cumulatively to the Repair Agent on the time series and recommendation repositories (Table 11). Each condition runs once with the same source, initial candidate, evaluation inputs, seeds, thresholds, full training signals, and total budgets. The base condition retains the repair prompts, ordinary inspection and editing tools, syntax checks, and progress reminders. After adding investigation, investigation and repair share one conversation until independent evidence handoff is also added. All four conditions accept the time series repository with 69/69 checks and pass 131/145 checks on recommendation. Each condition therefore fully accepts one of the two repositories. On time series, LaDiM with all three added components uses 1.937 million total tokens, 68.4% fewer than the base Repair Agent. Adding repository context management to investigation and independent evidence handoff reduces tokens from 6.047 million to 1.937 million. Investigation and repair input tokens fall from 5.731 million to 1.700 million. The two truncated LLM responses in the condition without repository context management count toward its call and token totals.

On recommendation, adding the components increases token use while all four conditions pass 131/145 checks and all ten original tests. The 14 unresolved checks comprise 12 item representation checks over six model execution paths at steps two and three, and two parameter update checks. The first three conditions in the table reach the conversation context limit after 62, 59, and 68 calls,

Table 11: Cumulative agent components on both repositories. Checks reports passed behavior checks. Calls cover investigation and repair, and tokens also include the shared initial translation. Each row adds a component to the preceding row. Tokens are in millions.

	Time series			Recommendation		
Condition	**Checks**	**Calls**	**Tokens**	**Checks**	**Calls**	**Tokens**
Repair Agent	69/69	26	6.128	131/145	62	8.336
+ Verifier investigation	69/69	18	3.896	131/145	59	8.913
+ Independent evidence handoff	69/69	26	6.047	131/145	68	9.556
+ Repository context management	69/69	22	1.937	131/145	80	10.191

respectively. LaDiM with all components rebuilds the conversation context six times and continues to the 80-call limit.

G.2 Repository Structural Analysis

The comparison with and without Repository Structural Analysis retains Dependency Graph Planning, notebook tools, ordinary file inspection, independent evidence handoff, and evidence management in both conditions. Removing Repository Structural Analysis removes the inventory of files, imports, symbols, and notebook cells, together with the list of files not yet assigned to a work unit. With the component enabled, the agent requests structural analysis when needed. Initialization and context reconstruction do not perform this analysis. Each condition runs once on each repository with the same initial translation and repository budgets. Repository Structural Analysis reduces

Table 12: Repository comparisons that remove one component at a time. Both comparisons use the same LaDiM run with all components for each repository. Checks reports passed behavior checks. Calls cover investigation and repair, and tokens include initial translation. Tokens are in millions.

	Time series			Recommendation		
Condition	**Checks**	**Calls**	**Tokens**	**Checks**	**Calls**	**Tokens**
LaDiM	69/69	30	2.127	131/145	80	9.962
Without Repository Structural Analysis	69/69	42	2.651	131/145	80	7.172
Without Dependency Graph Planning	69/69	21	1.804	130/145	69	9.214

time series calls from 42 to 30 and total tokens by 19.8%, with both conditions passing all 69 checks after two submissions. On recommendation, both pass 131/145 checks. With structural analysis, this result is reached at the third submission instead of the fourth. Enabling structural analysis uses 9.962 million total tokens, 38.9% more than the 7.172 million used without it. The two conditions have the same final 14 failures described above. The runs with all components in this study and the cumulative study were executed separately, so each table reports the costs of its own comparison.

G.3 Dependency Graph Planning

Removing Dependency Graph Planning removes the work-unit plan, checks that prerequisite units are ready, restrictions on which files belong to each unit, invalidation of checks through dependencies, and context reconstruction when the active unit changes. The condition retains Repository Structural Analysis, ordinary file and notebook tools, code editing and syntax checks, independent evidence handoff, evidence associated with code versions, and reconstruction at the context limit. The entire permitted target repository remains editable. The conditions with and without planning in Table 12 use the same source, initial candidate, inputs, seeds, thresholds, underlying method implementation, and budgets. Each condition runs once on each repository. The run with all components is shared with the structural analysis comparison.

Without Dependency Graph Planning, time series completes all 69 checks after one submission, using 21 calls and 1.804 million total tokens. This is 15.2% fewer tokens than LaDiM with all components. On recommendation, removing planning uses 7.5% fewer tokens but passes 130/145 checks, one fewer than LaDiM with all components. The additional failure is the parameter update

at step three in the model with position debiasing. Both conditions fully accept one of the two repositories.

Without planning, submissions one through four pass 128, 103, 133, and 130 of the 145 recommendation checks. A change to a shared training entry point before the second submission causes the first regression. The final submission again reduces the number of passed checks from 133 to 130. With all components, the corresponding trajectory is n/a, 122, 131, and 131. The first submission reports eight execution failures separately from the 145 protocol checks, leaving all protocol measurements unavailable. The first submission without planning and the second submission with all components each lack one retrieval measurement; both final candidates have all 145 measurements. Removing planning exhausts the budget of 480,000 output tokens after 69 calls, while LaDiM with all components uses all 80 calls. Final scores use the last submitted candidate, including these regressions.

H Native JAX Migration with Consecutive Training Steps

This study uses twelve public model workloads from PyTorch examples, minGPT, and torchvision.[1] They cover image classification, super-resolution, a variational autoencoder, image generation, recurrent and Transformer language models, time series, two policy learning models, graph attention, GPT, and ResNet. Initial translation returns code for ten workloads. Eight translated programs pass evaluation on all three seeds before repair, and two fail. Initial translation of the image generation and Transformer language modeling workloads exhausts the output budget without returning code. All four methods receive the same two failing translations, for the time series model and ResNet, for repair. The translated programs implement the forward computation in native JAX and explicitly represent model buffers, which store state other than trainable parameters. The evaluator independently invokes JAX differentiation and the specified Optax optimizer for three consecutive steps on seeds 8101, 8102, and 8103. Source parameters, buffers, and batches are supplied to each method. The source and JAX programs start from exactly matching states. The evaluator compares their full outputs, losses, gradients, parameter updates, parameters, and buffers elementwise in float64, with absolute tolerance 10^{-7} and relative tolerance 10^{-5}. Tensor names, shapes, and data types must also match. Each method on each repair task has a cumulative budget of 40 LLM calls, four submissions, 120,000 output tokens, and 1,800 seconds, including investigation. Each initial translation has one call with at most 16,384 output tokens.

Table 13: Repair of two failing native JAX translations from twelve source workloads. Calls and Tokens cover repair. Two tasks and All sources report total tokens including 0.031 and 0.140 million shared initial translation tokens, respectively, counted once. Each method accepts 1/2 repair tasks and 9/12 source workloads overall. Tokens are in millions.

Method	Calls	Tokens	Two tasks	All sources
LaDiM	35	1.840	1.871	1.980
Direct repair	28	0.761	0.792	0.901
MatchFixAgent	23	0.259	0.290	0.399
SWE-agent	66	1.014	1.045	1.153

All four methods repair shape errors in the time series candidate's input slicing and output concatenation, after which the programs pass all three steps on all three seeds. LaDiM, MatchFixAgent, and SWE-agent pass at their first submission, and Direct repair passes at its second. LaDiM, Direct repair, and SWE-agent also turn the truncated ResNet candidate into executable code. These three ResNet programs pass all first-step numerical checks. Later steps contain gradient and parameter update differences, followed by output and buffer differences. MatchFixAgent stops when parsing the truncated ResNet candidate encounters a syntax error. Its numerical results are n/a, and its four completed calls remain in the costs.

The eight programs that pass before repair are CNN classification, super-resolution, the variational autoencoder, the recurrent language model, both policy learning models, graph attention, and GPT.

[1] `https://github.com/pytorch/examples`, `https://github.com/karpathy/minGPT`, and `https://github.com/pytorch/vision`.

Adding the repaired time series model gives 9/12 accepted source workloads for every method. Initial translation of all twelve sources uses 0.029 million input and 0.110 million output tokens, totaling 0.140 million, including both generation failures. The ResNet results show why consecutive training checks are necessary even when the first step agrees.

I LLM Backend and Decoding Settings

All LLM-based methods use the request model identifier `deepseek-v4-flash`: LaDiM, SWE-agent, MatchFixAgent, InterTrans, CodeTransEngine, Direct LLM, Direct repair, and test-guided repair. The recorded provider responses identify the model as `deepseek-flash`. Requests use an HTTP chat-completion API, and the experiments were recorded in September 2026. The archives preserve the request and response identifiers rather than an immutable model checkpoint. Table 14 reports the recorded settings and their scope. Baselines retain their native prompts, tools, parsers, and retry policies as described in Appendix A.2.

Table 14: LLM backend and request settings. Output limits include reasoning tokens. Settings that differ across generation and repair stages are specified separately.

Setting	**Recorded value**
Requested model	`deepseek-v4-flash`
Returned model	`deepseek-flash`
Reasoning mode	Enabled with high reasoning effort for the agent repair protocols; inherited provider settings for the reused Java initial translations and native InterTrans generation.
Temperature	Omitted by the agent client when reasoning is enabled; 0 in MindSpore initial-generation requests; 0.1 in reused Java initial translations, native InterTrans, and test-guided repair.
Top-p	0.95 for native InterTrans; omitted in the inspected shared-generation and agent requests.
Context capacity	1,000,000 serialized-message characters in the agent controller; the provider independently enforces its token window.
Maximum output per call	131,072 for program initial translation and test-guided repair; 16,384 for program agent repair; 32,768 for repository agent repair. Native JAX initial translation uses 16,384 (Appendix H).
Context reconstruction	At 75% of the configured character capacity for LaDiM on repositories, as well as at evidence handoff and work-unit changes.
API-error handling	One HTTP attempt per call in the shared agent client; baseline-native retry policies are retained. CodeTransEngine allows up to six transport retries.
Access period	September 2026.

The reused Java initial translations specify temperature 0.1 and inherit the provider's reasoning settings; their responses report reasoning tokens. Seventeen of these 18 requests omit an explicit maximum output limit, while one specifies 131,072. Native InterTrans also retains temperature 0.1. Its 36 earlier incomplete calls used an 8,192-token output limit, and the completed search used 131,072; both are included in the costs in Appendix A.1. These recorded settings are retained with the corresponding outcomes. Repair calls use the remaining cumulative budget when it is smaller than the per-call limit.