VECTRA: VERIFIED EPISODIC CONTROL FOR REPOSITORY-LEVEL CODE TRANSLATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Repository-level code translation is a long-horizon engineering process that requires establishing a coherent target architecture and cross-file interfaces, reconstructing build configurations, and preserving program behavior. Modern coding agents can perform much of this work, but effective execution also requires deciding which intermediate repository states to retain, which failures to address next under a limited budget, and what knowledge to carry across sessions. We present VECTRA, a control framework that advances repository translation through a sequence of verifier-admitted states. Four coordinated mechanisms form its execution loop: a holistic bootstrap establishes the overall target structure; a frontier scheduler turns current build and test evidence into bounded translation missions; a verification gate accepts candidate changes or rolls them back; and a persistent translation ledger carries verified facts and cross-language decisions across fresh coding-agent episodes. The underlying coding agent remains unmodified and can use its usual tools and subagents. Across 13 translation directions in RepoTrans-Bench, VECTRA achieves an average success rate of 67.9%, outperforming the evaluated agent-based baselines. Ablation results show that removing any of the four mechanisms reduces the success rate. These results suggest that explicit control over intermediate repository states and translation progress can complement advances in code generation.

1 INTRODUCTION

Repository-level code translation aims to reproduce an entire software project in another language while preserving its behavior. The target project must reconstruct cross-file interfaces, replace language-specific libraries, reconstruct its build configuration, and keep earlier functionality intact as the implementation evolves.

Modern coding agents make this task considerably more tractable. Systems such as OpenHands and OpenCode can inspect a repository, edit multiple files, execute commands, interpret compiler and test feedback, and delegate focused work to subagents (Wang et al., 2025b; Anomaly, 2026). These abilities are well suited to the exploratory parts of translation: understanding unfamiliar code, choosing target-language libraries, implementing behavior, and debugging failures. A translation method can therefore build on a capable coding agent instead of reproducing these general software-engineering abilities.

The remaining difficulty is how to control a long sequence of repository changes. A repair may make one group of tests pass while breaking behavior that worked earlier. An agent may repeatedly pursue one stubborn failure while other parts of the repository remain unfinished. Decisions about type mappings, dependencies, or target interfaces may also disappear when the context is renewed. Without an external controller, the same agent that proposes code changes implicitly decides which repository state and translation decisions are retained. This makes accumulated progress vulnerable to regression, stalled debugging, and context loss.

We address this problem with *verified episodic control*. VECTRA (Verified Episodic Control for Translation via Repository Admission) retains an off-the-shelf coding agent for exploration, implementation, and debugging, and assigns scheduling, verification, and repository-state retention to an outer controller. A holistic bootstrap initializes the target structure and build configuration. A frontier scheduler converts current build and test failures into bounded missions. A verification gate

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.

checks candidates against the accepted build and test results and rejects detected regressions. A persistent ledger carries translation decisions and failed approaches into later episodes.

We evaluate VECTRA across all 13 translation directions in RepoTransBench. We compare it with RepoTransAgent, the benchmark's ReAct-style translation agent, and with OpenCode, which also serves as VECTRA's episode executor. The comparison with direct OpenCode evaluates VECTRA's overall control policy using the same coding agent and project budget. VECTRA achieves a 67.9% average success rate, versus 55.6% for OpenCode and 20.8% for RepoTransAgent, and improves compilation and partial-correctness metrics. An ablation study further finds a lower success rate after removing any of the four mechanisms. Our contributions are:

- a formulation of repository translation as an episodic process in which verified repository states provide durable progress;
- a four-mechanism control framework that combines holistic repository bootstrap, frontier scheduling, a verification gate, and a persistent translation ledger;
- an evaluation across all 13 translation directions in RepoTransBench, including a controlled comparison with the same coding agent used directly, an ablation study, and execution-level analyses.

2 BACKGROUND AND RELATED WORK

2.1 REPOSITORY TRANSLATION SYSTEMS

Early neural approaches focused on function-level translation, using multilingual pretraining and, later, automated tests to filter generated translations (Rozière et al., 2020; 2022). Repository translation also requires handling dependencies, cross-file interfaces, and build configurations. AlphaTrans uses static analysis to divide a source project into fragments, constructs a target skeleton, and translates the fragments with compilation and execution checks (Ibrahimzada et al., 2025). Skel mechanically constructs a target skeleton with implementation holes, preserving lexical scopes and symbol relationships to guide fragment generation (Wang et al., 2025a). Syzygy translates C code and tests to Rust using dynamic execution information (Shetty et al., 2024), while Oxidizer translates Go projects to Rust through type-guided translation followed by input-output validation (Zhang et al., 2025).

Agent-based methods organize translation and repair at several levels. The RepoTransAgent released with RepoTransBench uses a ReAct loop over repository tools (Yao et al., 2023; Wang et al., 2026). A separate framework with the same name uses retrieval, context analysis, and refinement agents to translate individual methods using repository context (Guan et al., 2025). TransAgent uses fine-grained execution alignment to repair translated programs and is evaluated on programming problems (Yuan et al., 2026). MatchFixAgent validates and repairs existing translations (Ibrahimzada et al., 2026a). ReCodeAgent coordinates analysis, planning, translation, and validation agents around a target design, skeleton, dependency-aware plan, and validation reports (Ibrahimzada et al., 2026b). VECTRA delegates implementation to a general coding agent and controls which repository states and translation decisions are retained across episodes.

2.2 CODING AGENTS AND AGENT WORKFLOWS

Coding agents provide repository inspection, editing, and command execution. SWE-agent studies how the agent–computer interface shapes repository repair (Yang et al., 2024); CodeAct represents actions as executable code (Wang et al., 2024); and OpenHands provides a general runtime with tools, sandboxes, delegation, and evaluation support (Wang et al., 2025b). OpenCode is an open-source coding agent with repository tools and native primary-agent/subagent orchestration (Anomaly, 2026). For repeated attempts, Reflexion retains textual feedback from earlier runs (Shinn et al., 2023). VECTRA couples persistent handoff notes with verified repository checkpoints and test-driven task selection.

2.3 Feedback-Guided Repair and Planning

GenProg uses tests to guide repair search (Le Goues et al., 2012), while SWT-Bench studies generated tests for filtering candidate fixes (Mündler et al., 2024). CodePlan combines dependency analysis with checker feedback to update repository edit plans (Bairi et al., 2024); SWE-Search explores alternative solutions through Monte Carlo tree search (Antoniades et al., 2025). TraceCoder coordinates instrumentation, analysis, and repair agents to debug generated code using runtime traces, failure history, and rollback (Huang et al., 2026).

VECTRA focuses on progress across repository translation episodes executed by an unmodified coding agent. Its controller uses the latest accepted repository state to select tasks from target-side build and test failures and to check whether candidate changes preserve previously verified behavior.

3 Method

3.1 Problem Formulation

A repository-translation instance contains a source repository S, a target project scaffold, and a target-language test suite T. A candidate target repository R must satisfy both structural requirements, such as a valid build and consistent cross-file interfaces, and behavioral requirements evaluated using T. When individual test results are available, we write $\mathcal{P}(R)$ for the *set* of tests that pass on R.

At control step k after bootstrap, the controller maintains

$$\Omega_k = \langle R_k, P_k, L_k, F_k \rangle, \tag{1}$$

where R_k is the accepted target repository, $P_k = \mathcal{P}(R_k)$ is its verified set of passing tests, L_k is the translation ledger, and F_k contains the unresolved build or test obstacles.

3.2 Control Framework Overview

VECTRA treats repository translation as budgeted control of an evolving target repository. The coding agent proposes implementation changes, while an outer controller determines which repository is retained and which work is attempted next.

At the start of translation, the target may lack usable build configurations and cross-file interfaces, so focused repair first requires an initial repository-wide implementation. Once failures become observable, a limited budget must be distributed among unresolved problems, including problems that repeatedly resist repair. Selecting subsequent work therefore requires both current verification evidence and the history of earlier attempts. Because edits can undo previous successes, generating a candidate must be separated from deciding whether its code should persist. Fresh episodes and rollback also interrupt continuity, so the retained code and the record of translation decisions and attempted repairs need separate persistence.

VECTRA realizes these responsibilities through a control framework which comprises four mechanisms. Holistic repository bootstrap produces an initial target candidate, after which Frontier scheduling turns the retained repository's build and test evidence into bounded missions. Each episode produces a provisional repository, and the Verification gate determines whether to retain it or restore the preceding checkpoint. The Persistent translation ledger preserves decisions and observations across either outcome, allowing subsequent missions to use information from rejected attempts as well as accepted changes.

This feedback loop coordinates initialization, subsequent work, repository retention, and cross-episode context through one evolving control state (Figure 1). Implementation and debugging remain within coding-agent episodes; scheduling, admission, and structured-record updates follow deterministic rules. Execution ends when the tests pass, no mission remains schedulable, or the remaining budget cannot support another episode. Algorithm 1 in Appendix A summarizes this control loop.

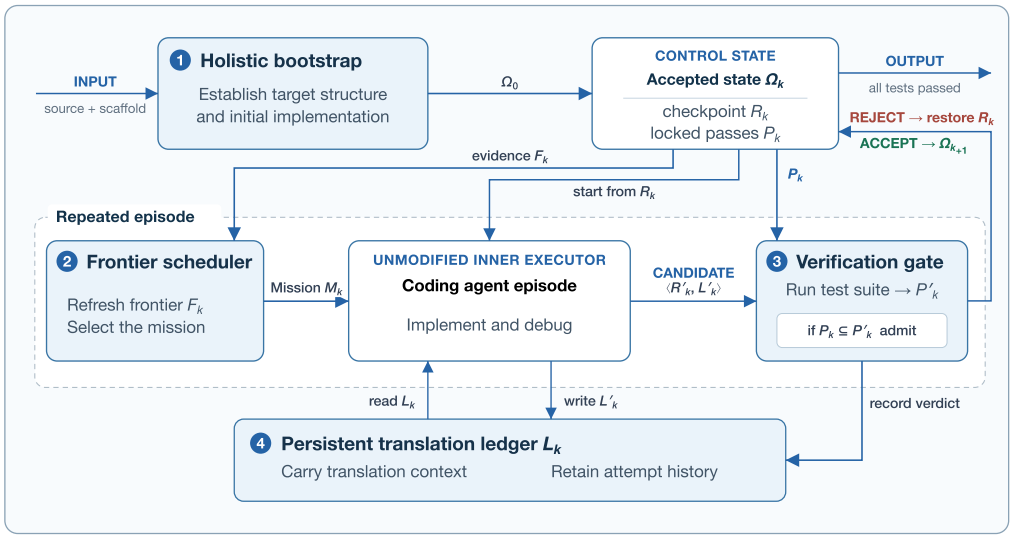


Figure 1: VECTRA coordinates bounded coding-agent episodes through an evolving control state.

3.3 HOLISTIC REPOSITORY BOOTSTRAP

Holistic repository bootstrap gives the coding agent a repository-wide translation mission, together with project context, the supplied target scaffold, the test runner and tests, and workspace rules. Within the initial episode's budget, the agent can inspect the source repository, choose the target package structure, reconstruct dependencies and build configuration, and implement the major interfaces. This opening mission covers the repository as a whole because the target may not yet build or expose useful repair units. After the episode, the Verification gate evaluates the candidate repository, and the Persistent translation ledger retains controller records and agent notes. The accepted state after bootstrap is $\Omega_0 = \langle R_0, P_0, L_0, F_0 \rangle$. Frontier scheduling then uses unresolved build obstacles or remaining test failures to select work for subsequent episodes.

3.4 FRONTIER-BASED MISSION SELECTION

The frontier scheduler derives F_k from the latest accepted verification report. When the target cannot yet run its tests, it assigns a scaffold mission to resolve blockers such as missing build files, dependencies, or unresolved interfaces. Once tests run, failures are grouped into units using framework structure—for example, a pytest file, JUnit class, gtest suite, or Go package. In both cases, the scheduler uses target-side evidence rather than a static partition of the source repository. Accepted changes refresh the frontier to reflect the remaining failures.

Selection uses two fields in the unit record. The scheduler first chooses among non-frozen units with the fewest consecutive non-improving attempts, breaking ties in favor of units covering more failing tests. A verified improvement clears the attempt counter. After two non-improving attempts on a unit, its next episode receives a longer debugging allocation. If that episode also yields no verified improvement, the unit is frozen so that further budget can go to other units. This policy distributes attempts across units while reserving longer runs for persistent failures. Episode allowances are reported in Section 4.

Candidate execution. Each follow-up coding-agent episode starts from the accepted target tree R_k in its workspace. The coding agent receives a stable context—the project structure, source files, benchmark runner, tests, and workspace rules—and a dynamic context for the episode: the selected frontier unit, reported failures, locked-test constraints, a time allowance, and a bounded view of L_k. Detailed diagnostics are available by running the supplied test command.

The mission limits the objective and budget while allowing repository-wide implementation edits. The coding agent can inspect the entire source repository, modify implementation and build files, run

tools, and invoke its own subagents when a failure crosses module or interface boundaries. When the episode finishes or reaches its time limit, the controller collects any notes written during the attempt and passes the edited tree R'_k to the verification gate. The gate determines whether this candidate becomes the next accepted checkpoint.

3.5 VERIFICATION-GATED STATE ADMISSION

To evaluate R'_k, the verification gate restores the supplied runner and protected test files while retaining the target-side build support needed to execute the suite. Framework normalization preserves the test assertions. The suite then runs outside the coding-agent episode, so admission depends on the resulting test report. The candidate's passing-test set is $P'_k = \mathcal{P}(R'_k)$.

When individual test results are comparable across checkpoints, the candidate must preserve all currently locked passing tests. The gate applies the following admission rule:

$$\langle R_{k+1}, P_{k+1} \rangle = \begin{cases} \langle R'_k, P'_k \rangle, & P_k \subseteq P'_k, \\ \langle R_k, P_k \rangle, & \text{otherwise.} \end{cases} \tag{2}$$

The controller maintains checkpoints of accepted repository states. Acceptance saves the candidate as the new checkpoint; rejection restores the preceding checkpoint before another mission is formed. Therefore, each episode starts from the repository state retained by the gate.

The report determines the evidence available for admission. Individual test results support the set-preservation rule, which distinguishes new successes from regressions in previously passing tests. When only aggregate results are available, the gate requires that compilation status not regress and that the pass count not decrease. The controller also recognizes structural progress: making the target buildable or making its tests runnable can advance translation before any test passes. This progress informs subsequent scheduling, alongside gains in passing tests, while the admission rule determines which candidate is retained.

The verification gate controls which edits persist across episodes, so rollback does not depend on the coding agent noticing a regression. An accepted report refreshes the frontier and may resolve existing units or expose additional failures. When an attempt yields no verified improvement, the controller updates the selected unit's retry record. The next mission combines this scheduling history with the accepted repository state.

Evidence normalization across test frameworks. Runner adapters produce a common report containing compilation status, total and passing/failing counts, optional module counts, and a runner-output excerpt. Counts come from the benchmark runner. Framework-specific parsers recover individual test results from console output or structured reports where available, with supplementary verbose runs for supported frameworks. The verification gate uses this report to assess the candidate; after acceptance, the reported failures are used to update the frontier. Individual results support test preservation and failure grouping, while aggregate counts support verification when individual outcomes are unavailable. Appendix B describes the framework-specific extraction rules.

3.6 PERSISTENT TRANSLATION LEDGER

The Persistent translation ledger L_k carries information across fresh coding-agent episodes alongside the accepted repository. It separates controller-managed records of verification and scheduling from agent-written notes on translation decisions and debugging observations. The notes can include type and name mappings, library choices, interface conventions, build requirements, and failed repair hypotheses.

The controller records locked tests and each frontier unit's failing tests, attempt count, and frozen status. Episode records identify the selected task and store its time budget, verification outcome, pass count, and any broken locked tests. The controller collects agent notes before the verification gate accepts or rolls back the candidate, so observations from rejected work remain available to later episodes. After verification, it updates the structured records with the episode outcome and current scheduling state. The ledger thus retains a record of the attempt even when its code changes are rolled back.

Agent notes have a size limit; when it is exceeded, the controller retains their beginning and newest additions. Each mission prompt includes a bounded view of the ledger: the beginning and end of the ledger are retained when the full text is too long. This prompt truncation leaves the stored episode history intact. The agent writes notes during normal coding-agent execution, while the controller collects, stores, and renders the ledger deterministically, without additional model calls.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmark. We evaluate RepoTransAgent, OpenCode, and VECTRA on the full RepoTrans-Bench across all 13 translation directions. Each instance contains a source repository, target-language project template, target tests, and an executable runner. The directions include dynamic and static target languages, several build systems, and multiple test frameworks.

Model, budget, and environment. All methods use DeepSeek-V4-Flash with a 1,800-second budget for each project. For VECTRA, episode execution, verification, rollback, ledger updates, and orchestration are charged to this same budget. The bootstrap receives 45% of the budget, an ordinary follow-up episode receives at most 360 seconds, and two consecutive non-improving attempts trigger one longer debugging episode. That episode receives the larger of three ordinary episode allowances and 80% of the remaining project time, capped by the time still available. No new episode starts with fewer than 120 seconds remaining. Each verifier invocation has a 300-second timeout, and 60 seconds are reserved for final verification and summary generation. These settings are fixed across directions. Final scoring restores the benchmark-supplied test surface and executes it in an isolated target-language environment.

Compared methods. RepoTransAgent is a ReAct-style single-agent system designed for repository translation. OpenCode (version 1.15.10) is a general-purpose coding agent that can inspect files, edit a workspace, execute commands, and use its native subagents. VECTRA uses the same OpenCode agent as the executor inside each episode. VECTRA adds a verified episodic control framework around this executor. We additionally conduct an extension experiment with ReCodeAgent on Java→Python and C→Rust using its four-stage workflow with OpenCode as executor. All methods are evaluated under the same 1,800-second budget per project.

Metrics. Following RepoTransBench, we report success rate (SR), compilation rate (CR), average test pass rate (APR), and average module pass rate (AMPR). SR requires every target test of a project to pass. CR measures whether the target builds. APR and AMPR expose project- and module-level partial correctness.

4.2 FULL-BENCHMARK RESULTS

Table 1 reports the full-benchmark comparison. Among the evaluated methods, VECTRA achieves the highest SR in all 13 directions. Averaged equally across directions, its SR is 67.9%, compared with 55.6% for OpenCode and 20.8% for RepoTransAgent. Relative to the direct OpenCode baseline, the gains are 12.3 percentage points in SR, 16.3 in CR, 15.7 in APR, and 15.8 in AMPR. The joint movement of full-project and partial-correctness metrics shows broader progress among incomplete repositories as well as additional full successes.

The metric profile also separates build recovery from behavioral completion. VECTRA's 16.3-point CR gain over direct OpenCode is larger than its 12.3-point SR gain, while APR and AMPR rise by comparable amounts. Thus, the controller helps more projects reach an executable target and advances their tests and modules after compilation. The 25.3-point separation between CR (93.2%) and SR (67.9%) locates much of the residual difficulty after build recovery: these projects compile, but behavioral or interface mismatches still prevent full test success.

The largest SR differences from OpenCode occur on Python→Java (32.5 points), Matlab→Python (27.4), Java→C# (24.4), and C#→Java (23.6). The directions targeting Java or C# also show large compilation gains. These directions combine target build reconstruction with strong cross-file typing and dependency constraints, so they provide many opportunities for the bootstrap, frontier, gate, and

Table 1: Repository translation results (%). RTA: RepoTransAgent; OC: OpenCode; VEC: VECTRA. Best scores are bold.

		SR (%)			CR (%)		
Direction	N	RTA	OC	VEC	RTA	OC	VEC
C → Python	121	45.1	66.1	**68.6**	76.2	81.8	**92.6**
C++ → Python	180	37.2	71.1	**77.8**	56.7	89.4	**98.9**
Java → Python	144	51.7	66.7	**68.8**	77.2	89.6	**96.5**
JavaScript → Python	188	33.3	60.0	**66.0**	73.0	85.8	**93.6**
Matlab → Python	62	9.7	51.6	**79.0**	17.7	69.4	**95.2**
C# → Java	96	28.1	63.9	**87.5**	32.3	68.0	**93.8**
Python → Java	169	10.1	45.0	**77.5**	16.0	56.7	**97.0**
Java → C#	144	0.7	54.1	**78.5**	0.7	61.0	**89.6**
Python → C++	164	0.0	46.3	**57.9**	0.0	55.5	**77.4**
Java → Go	139	20.9	40.3	**45.3**	73.4	97.8	**98.6**
Python → Go	168	8.3	26.8	**28.6**	73.8	90.5	**95.8**
C → Rust	122	16.4	66.4	**74.6**	38.5	80.3	**94.3**
Python → Rust	167	8.4	64.3	**73.1**	16.8	74.4	**88.6**
Average (13 directions)	1,864	20.8	55.6	**67.9**	42.5	76.9	**93.2**

		APR (%)			AMPR (%)		
Direction	N	RTA	OC	VEC	RTA	OC	VEC
C → Python	121	64.2	78.6	**88.0**	57.6	78.6	**88.0**
C++ → Python	180	53.3	85.5	**95.2**	53.3	84.5	**95.2**
Java → Python	144	72.3	86.7	**92.9**	70.0	86.5	**92.9**
JavaScript → Python	188	56.4	79.6	**86.7**	48.5	78.6	**86.7**
Matlab → Python	62	14.0	66.9	**92.0**	14.0	66.9	**92.0**
C# → Java	96	31.8	67.5	**93.1**	31.8	67.5	**93.1**
Python → Java	169	14.5	54.7	**93.3**	14.5	54.7	**93.3**
Java → C#	144	0.7	59.3	**87.8**	0.7	59.3	**87.8**
Python → C++	164	0.0	53.3	**73.5**	0.0	51.2	**68.9**
Java → Go	139	25.0	49.4	**57.4**	25.0	49.4	**57.4**
Python → Go	168	11.8	35.3	**40.2**	11.8	35.3	**40.2**
C → Rust	122	23.5	74.0	**86.8**	23.5	72.8	**86.8**
Python → Rust	167	9.4	71.9	**80.1**	9.4	70.8	**80.1**
Average (13 directions)	1,864	29.0	66.4	**82.1**	27.7	65.9	**81.7**

ledger to interact over multiple episodes. The Go directions improve more modestly in SR—5.0 points for Java→Go and 1.8 for Python→Go—while still improving CR, APR, and AMPR.

We also conduct an extension experiment with ReCodeAgent under the same 1,800-second project budget as the other methods. We evaluate ReCodeAgent on two of RepoTransBench's 13 translation directions, Java→Python and C→Rust (Table 2), because its released implementation is tailored to specific language pairs and does not directly support the remaining directions. ReCodeAgent uses OpenCode as its coding agent and the same model as the other methods. ReCodeAgent obtains lower scores than both direct OpenCode and VECTRA in these two directions. Its analysis and planning stages consume a substantial share of the budget, leaving less time for implementation and repair. This time budget may constrain its more resource-intensive workflow. C→Rust additionally encounters interface and build-integration issues when adapting the workflow to RepoTransBench, where target-package construction and test-interface alignment remain part of the translation task. Appendix C reports stage-budget usage.

4.3 COMPONENT ABLATION

We conduct the component ablation on all 169 Python→Java projects in RepoTransBench. This dynamic-to-static direction has the largest SR gain over direct OpenCode in Table 1. The ablation

Table 2: ReCodeAgent results on RepoTransBench.

Direction	N	SR (%)	CR (%)	APR (%)	AMPR (%)
Java → Python	144	41.7	76.4	68.6	65.0
C → Rust	122	23.8	34.4	27.5	25.9

ReCodeAgent uses the same coding agent, model, and 1,800-second project budget as direct OpenCode and VECTRA.

Table 3: Ablation Results (Python→Java). ΔSR Is Relative to Full VECTRA.

Condition	SR (%)	ΔSR
Direct OpenCode	45.0	-32.5
Full VECTRA	**77.5**	—
w/o holistic bootstrap	59.2	-18.3
w/o frontier scheduling	49.7	-27.8
w/o verification gate	54.4	-23.1
w/o translation ledger	50.3	-27.2

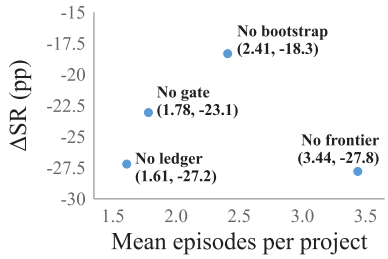


Figure 2: Success-rate change and mean episode effort for each component ablation. ΔSR is relative to full VECTRA; values closer to zero are better.

changes the controller configuration only. Direct OpenCode, full VECTRA, and the four single-mechanism removals use identical projects, model, executor, budget, and evaluator.

Table 3 presents the ablation results. Each single-mechanism removal reduces SR: 18.3 points for bootstrap, 27.8 for frontier scheduling, 23.1 for the gate, and 27.2 for the ledger. Removing frontier scheduling gives the lowest SR.

Figure 2 plots ΔSR against average episode effort for the four removals; a higher vertical position indicates a smaller loss. Removing frontier scheduling creates the largest gap from full VECTRA and also consumes the most episode effort, producing 581 episodes, an average of 3.44 per project. These counts explain why the scheduler affects both success and resource use: it changes the number and scope of follow-up attempts. Removing bootstrap averages 2.41 episodes per project, second only to no-frontier, and 94 projects continue beyond their first episode, more than under any other removal. The later loop can recover part of the missing initialization, but it has to spend follow-up episodes doing so. Across the three ablation conditions with the gate enabled, 30 candidate states are rejected in 26 runs. With the gate disabled, all 301 episode outputs move forward without an admission check, and SR falls by 23.1 points. Removing the ledger produces 272 episodes, the fewest among the four removals, and yields only 50.3% SR, close to the no-frontier result. This suggests that without the ledger, useful follow-up work can stop early.

4.4 CASE STUDY

In *alingse_jsoncsv* (Python→Go), bootstrap produced a compilable implementation, but a test still failed when encoding and decoding a JSON key containing backslashes (Figure 3). The source encoding joins path elements with a backslash-prefixed separator; literal backslashes must first be

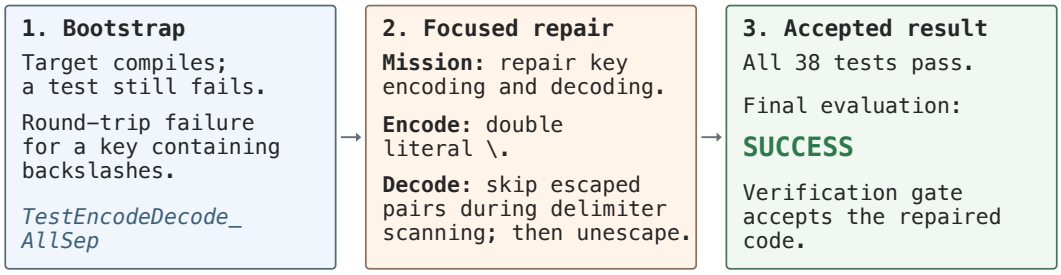


Figure 3: The *jsoncsv* Python→Go trace. A focused episode resolves a test failure left after repository translation.

escaped and then recognized during decoding. The coding agent saw the failure during bootstrap, but stopped after treating the encoding requirements as conflicting.

The verification gate recorded the test failure, and the frontier scheduler assigned it to a focused episode. The mission included `TestEncodeDecode_AllSep` and a ledger record that contained the first episode's implementation choices and mistaken diagnosis. The new episode reassessed the problem and changed two functions: `EncodeSafeKey` doubled literal backslashes before joining path elements, and `DecodeSafeKey` skipped escaped pairs while scanning for delimiters, then unescaped the recovered elements. The verification gate accepted the repaired implementation. All 38 tests passed, and the final evaluator marked the project successful.

The test encodes a path containing four keys, `A`, `B`, `..`, and `\.\ww`, and checks that decoding recovers the same keys. Both backslashes in the fourth key are part of its content. Direct OpenCode also compiled successfully and doubled these backslashes during encoding. With separator `.`, however, its `strings.Split` decoder matched the second backslash plus the dot as a delimiter. It returned only a backslash for the fourth key and failed the test. The repaired decoder skipped escaped pairs, preserving the dot as key content.

5 Conclusion

Repository translation requires capable code generation together with reliable control over a long sequence of partial repository states. VECTRA organizes that sequence through a control framework comprising holistic bootstrap, frontier scheduling, verification-gated admission, and a persistent translation ledger, while leaving implementation and debugging to an unmodified coding agent. Across all 13 translation directions in RepoTransBench, VECTRA achieves an average success rate of 67.9%, outperforming the other evaluated methods in both full-project success and partial correctness. The ablation results and execution traces illustrate how the four mechanisms guide budget use, reject detected regressions, and preserve progress across episodes. Future work will evaluate VECTRA across additional benchmarks and coding agents to examine the generality of its control mechanisms.

AI USE STATEMENT

Generative AI tools assisted with research planning and methodology discussions, code for experiments, analysis of execution records, literature discovery, drafting and editing manuscript text, and preparation of figures and tables. The evaluated translation systems also use language models as described in Section 4. The authors take responsibility for the final content, including AI-assisted text, claims, and artifacts.

REPRODUCIBILITY STATEMENT

Section 3 describes the verified episodic control framework, with its control algorithm in Appendix A. Section 4 reports the model, execution budget, benchmark coverage, scoring procedure, and the ablation study. Appendices B and C describe framework normalization and the extension experiment. Appendices D and E report the budget-sensitivity study and full-benchmark resource usage.

REFERENCES

Anomaly. OpenCode: The open source coding agent. GitHub repository, 2026. URL https://github.com/anomalyco/opencode. Accessed May 23, 2026.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. In *International Conference on Learning Representations (ICLR)*, pp. 64485–64515, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a1e6783e4d739196cad3336f12d402bf-Abstract-Conference.html.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-Level Coding using LLMs and Planning. *Proceedings of the ACM on Software Engineering*, 1(FSE): 675–698, 2024. doi: 10.1145/3643757. URL https://doi.org/10.1145/3643757.

Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. RepoTransAgent: Multi-Agent LLM Framework for Repository-Aware Code Translation, 2025. URL https://arxiv.org/abs/2508.17720. arXiv:2508.17720.

Jiangping Huang, Wenguang Ye, Weisong Sun, Jian Zhang, Mingyue Zhang, and Yang Liu. TraceCoder: A Trace-Driven Multi-Agent Framework for Automated Debugging of LLM-Generated Code. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1083–1095, 2026. doi: 10.1145/3744916.3773187. URL https://doi.org/10.1145/3744916.3773187.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proceedings of the ACM on Software Engineering*, 2(FSE):2454–2476, 2025. doi: 10.1145/3729379. URL https://doi.org/10.1145/3729379. Article FSE109, 23 pages.

Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026a. URL https://openreview.net/forum?id=MuyXpH3GL1.

Ali Reza Ibrahimzada, Brandon Paulsen, Daniel Kroening, and Reyhaneh Jabbarvand. ReCodeAgent: A Multi-agent Workflow for Language-Agnostic Translation and Validation of Large-Scale Repositories. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026b. URL https://arxiv.org/abs/2604.07341. Accepted; forthcoming. arXiv:2604.07341.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi: 10.1109/TSE.2011.104. URL https://doi.org/10.1109/TSE.2011.104.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *Advances in Neural Information Processing Systems*, volume 37, pp. 81857–81887, 2024. doi: 10.52202/079017-2601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

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, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=cmt-6KtR4c4.

Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A. Seshia, and Koushik Sen. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis, 2024. URL https://arxiv.org/abs/2412.14234. arXiv:2412.14234.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Bo Wang, Tianyu Li, Ruishi Li, Umang Mathur, and Prateek Saxena. Program Skeletons for Automated Program Translation. *Proceedings of the ACM on Programming Languages*, 9(PLDI): 920–944, 2025a. doi: 10.1145/3729287. URL https://doi.org/10.1145/3729287. Article 184.

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 International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations (ICLR)*, pp. 65882–65919, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. RepoTransBench: A Real-World Multilingual Benchmark for Repository-Level Code Translation. *IEEE Transactions on Software Engineering*, 52(2):675–690, 2026. doi: 10.1109/TSE.2025.3645056. URL https://doi.org/10.1109/TSE.2025.3645056.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.03629.

Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Xin Peng, Zhenpeng Chen, and Yiling Lou. TransAgent: Enhancing LLM-Based Code Translation via Fine-Grained Execution Alignment. *Proceedings of the ACM on Software Engineering*, 3(FSE):1582–1604, 2026. doi: 10.1145/3797099. URL https://doi.org/10.1145/3797099.

Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1616–1641, 2025. doi: 10.1145/3729315. URL https://doi.org/10.1145/3729315. Article 212.

A VECTRA CONTROL ALGORITHM

The controller state is $\Omega_k = \langle R_k, P_k, L_k, F_k \rangle$. The mission M_k includes the task and episode allowance.

Algorithm 1 VECTRA: verified episodic control

Input: Source S; target scaffold R_{init}; target tests T
Output: Last accepted target repository R_k

$\langle R_0, P_0, L_0, F_0 \rangle \leftarrow$ **Holistic bootstrap**(S, R_{init}, T)
$k \leftarrow 0$
while not all tests pass and there is still budget remaining **do**
  $M_k \leftarrow$ **Frontier scheduler**(F_k)
  $\langle R'_k, L'_k \rangle \leftarrow$ Coding agent episode(R_k, M_k, S, T, L_k)
  $\langle P'_k, F_{k+1}, L_{k+1} \rangle \leftarrow$ **Verification gate**$(R'_k, T; \Omega_k, M_k, L'_k)$
  if $P_k \subseteq P'_k$ **then** ▷ No previously passing test regresses
    $\langle R_{k+1}, P_{k+1} \rangle \leftarrow \langle R'_k, P'_k \rangle$ ▷ ACCEPT
  else
    $\langle R_{k+1}, P_{k+1} \rangle \leftarrow \langle R_k, P_k \rangle$ ▷ REJECT
  end if
  $k \leftarrow k + 1$
end while
return R_k

B VERIFIER NORMALIZATION

Runner adapters report compilation status, total, passing and failing counts, optional module counts, and a runner-output excerpt. Aggregate counts come from the benchmark runner. Framework parsers extract test results from pytest, gtest, Cargo, and Go console output or Maven Surefire XML, checking the recovered total against the runner's total with a small tolerance. A supplementary verbose pytest run is used only when its total and passing counts match the benchmark run. For Go, a verbose run refines package-level results into individual test outcomes and retains package-level build failures; the benchmark counts remain unchanged. When only aggregate results are available, the gate compares compilation status and pass counts, and the scheduler groups the remaining failures into a single unit. If a new report switches between package-level and individual-test results, the gate checks for compilation or pass-count regression before replacing an existing locked set on acceptance.

C RECODEAGENT EXTENSION EXPERIMENT DETAILS

The scores aggregate all 144 Java→Python and 122 C→Rust projects in this evaluation. The four stages share the budget of 1800s. In the recorded stage summaries, analysis and planning together average 947.2 seconds for Java→Python and 611.1 seconds for C→Rust. The budget is exhausted in 143 and 98 runs, respectively. Validator execution is recorded in 81 and 96 runs.

D BUDGET SENSITIVITY

We conduct this study on 60 projects drawn equally from C→Rust, Java→Python, and Python→Java. The three directions cover static-to-static, static-to-dynamic, and dynamic-to-static translation. Within each direction, projects are stratified by source size and fixed before execution.

The project-level budget covers coding-agent episodes, verification, rollback, ledger maintenance, and orchestration. It determines how much translation and repair work the complete system can perform. We evaluate budgets of 1,200, 1,800, and 2,400 seconds on the 60-project study set.

Table 4: Project-Budget Sensitivity. ΔSR Is Relative to the Default (1,800 s).

Budget (s)	SR (%)	CR (%)	APR (%)	AMPR (%)	ΔSR
1,200	60.0	83.3	82.1	82.1	−1.7
1,800 (default)	61.7	81.7	80.5	80.5	—
2,400	61.7	83.3	81.9	81.9	0.0

The 1,800-second setting is the default used in the main and ablation experiments. The evaluator reports no separate module decomposition for these three directions, so AMPR falls back to APR and the two columns are identical here.

Reducing the budget to 1,200 seconds lowers SR by 1.7 points, corresponding to one net full-project success. Increasing it to 2,400 seconds yields no net increase in full-project successes. APR remains within 1.6 points of the default at both alternative budgets, and AMPR follows the same pattern. CR is 83.3% at both alternative budgets, compared with 81.7% at the default. The 1,800-second setting therefore captures the observed full-project effectiveness without relying on the largest execution allowance, while the 1,200-second result shows that most of the effectiveness is retained under a tighter limit.

E FULL-BENCHMARK RESOURCE USAGE

Table 5: Full-benchmark mean resource usage per project.

Method	Time (s)	Tokens (M)			
		Total	Input	Cached	Output
Direct OpenCode	445.9	0.877	0.174	0.695	0.008
VECTRA	816.5	2.837	0.080	2.727	0.030

Table 5 reports mean resource usage per project over the full benchmark. Time measures model-bearing execution within each project. Relative to direct OpenCode, VECTRA increases the mean time from 445.9 to 816.5 seconds (83.1%). Both methods use the default 1,800-second project budget. Recorded token use rises from 0.877M to 2.837M. Cached input accounts for most of the increase (0.695M to 2.727M); output rises from 0.008M to 0.030M, while non-cached input falls from 0.174M to 0.080M.

The added computation accompanies a 12.3-point gain in SR over direct OpenCode, while the resource usage of the two methods remains within the same order of magnitude. Under the common budget ceiling, realized computation also reflects each method's control policy. VECTRA directs additional work toward verified failures, with frontier selection, verification-gated admission, and the ledger organizing follow-up efforts and preserving accepted progress. The comparison thus evaluates the complete control policy, including how it uses the available budget to advance translation. The increase in time and token use remains substantial, making reduced context processing and execution overhead important directions for future work.