

Kozuchi Agent: A Language-Agnostic Open-Weight Agent for Software Repair

Mehdi Bahrami
Fujitsu Research of
America
Santa Clara, CA, USA

Kosaku Kimura
Fujitsu Research
Kawasaki, Japan

Satoshi Munakata
Fujitsu Research
Kawasaki, Japan

Satoshi Nakashima
Fujitsu Research
Kawasaki, Japan

Yu Ishikawa
Fujitsu Research
Kawasaki, Japan

Kosuke Maeda
Fujitsu Research
Kawasaki, Japan

Nao Soma
Fujitsu Research
Kawasaki, Japan

Kenichi Kobayashi
Fujitsu Research
Kawasaki, Japan

Keisuke Miyazaki
Fujitsu Research
Kawasaki, Japan

Keizo Kato
Fujitsu Research
Kawasaki, Japan

Shigeki Fukuta
Fujitsu Research
Kawasaki, Japan

Tatsuo Kumano
Fujitsu Research
Kawasaki, Japan

Nobutaka Imamura
Fujitsu Research
Kawasaki, Japan

Kevin Musgrave
Fujitsu Research of
America
Santa Clara, CA, USA

Shahbaz Abdul Khader
Fujitsu Research of Europe
Slough, UK

Kwun Ho Ngan
Fujitsu Research of Europe
Slough, UK

Joe Townsend
Fujitsu Research of Europe
Slough, UK

Fayas Asharindavida
Fujitsu Research of Europe
Slough, UK

Matthieu Parizy
Fujitsu Research of Europe
Slough, UK

Akira Sakai
Fujitsu Research
Kawasaki, Japan

Yuma Ichikawa
Fujitsu Research
Kawasaki, Japan

Yang Zhao
Fujitsu Research &
Development
Beijing, China

Michiaki Takizawa
Fujitsu Research
Kawasaki, Japan

Taku Fukui
Fujitsu Research
Kawasaki, Japan

Hiroki Ohtsuji
Fujitsu Research
Kawasaki, Japan

Wei-Peng Chen
Fujitsu Research of
America
Santa Clara, CA, USA

Hiromichi Kobashi
Fujitsu Research
Kawasaki, Japan

Abstract

Industrial software-engineering teams increasingly need LLM agents that turn bug reports into correct patches, yet benchmark-scale operation adds long horizons, tool-use discipline, context persistence, heterogeneous clusters, and evaluation reuse. We present Kozuchi Agent, a *language-agnostic open-weight repair agent* and *CI-operated evaluation pipeline*. Explicit phases, persistent state, deterministic tools, a model-independent action interface, and cross-agent test-time selection make runs auditable and repeatable. With locally hosted Qwen3.5-27B, no fine-tuning, and TTS@8, Kozuchi resolves 374/500 SWE-bench Verified instances on the official evaluator. Unchanged on Multi-SWE-bench Java, the same 27-billion-parameter agent resolves 41/128 instances (32.03 %), ranking first

among strict open-weight submissions and fourth of 42 overall; on Python it ranks 12th of 135 and first among open-weight systems. Per-phase behavior remains within ± 5 percentage points across languages. Remaining failures mainly reflect semantic correctness, Java-specific harness issues, and selection errors. Across both tracks, results compare favorably with open/local peers by parameter count. Analysis of candidate diversity, selector regret, and patch reliability shows that the remaining gap is primarily semantic correctness and selection rather than edit formatting or proprietary-model access. Operationally, reusable CI stages reduce operator touch-points from five to one across heterogeneous internal clusters.

CCS Concepts

• **Software and its engineering** → **Software defect analysis**; • **Computing methodologies** → *Natural language processing*.

Keywords

AI agent, automated program repair, SWE-bench, experience report



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3834531>

clusters. Its reuse mode records which expensive stages can be short-circuited using previously validated outputs, while the published analysis scripts in `src/` regenerate every empirical value cited in RQ1–RQ6 directly from the SWE-bench and Multi-SWE-bench Java harness reports, trajectories, and cross-agent selection bundles in the public artifact (Data Availability Statement).

C3. A cross-agent test-time selector with trajectory-level audit. We merge $K = 8$ candidate inference runs through cross-agent testing: each run contributes both a patch and independent tests, and the tests are cross-applied to score candidate patches without using the hidden benchmark tests or a learned verifier. We also analyze the eight candidate streams and 495 full message trajectories, separating selector regret from instance hardness (408 instances solved by at least one run, 234 by all runs, 92 by none) and surfacing content-level selector signals such as `THOUGHT: -block-to-action` balance, rollback language, and error-marker asymmetries.

C4. Competitive open-weight repair results with statistical and failure-mode analysis. On the official SWE-bench Verified cloud evaluator, our 27-billion-parameter open-weight Python configuration resolves 374 of 500 instances (74.80 %, Wilson 95 % CI [70.82 %, 78.41 %]). The internal Docker re-grade resolves 376 (75.2 %); the cross-agent selector contributes +14 resolved instances over a size-ordered first-candidate baseline. On a frozen snapshot of 135 publicly catalogued SWE-bench Verified submissions the configuration ranks 12th overall and is the highest-ranked open-weight system; under Benjamini–Hochberg FDR correction (target $q \leq 0.05$) the configuration significantly outperforms 16 of 17 curated open-weight peers on matched per-instance McNemar tests. Of 126 unresolved instances, 91.3 % are patches that apply cleanly but do not flip the hidden tests; 0 % are malformed-diff or empty-patch failures, isolating the residual gap as a semantic-correctness problem rather than an editing-format problem. Patch LOC churn is the strongest single feature signal (point-biserial $r = -0.197$, $p = 1.1 \times 10^{-5}$), and 494/495 produced patches apply cleanly through the SWE-bench harness. On Multi-SWE-bench Java, the unchanged 27-billion-parameter Java configuration resolves 41/128 instances (32.03 %), ranks 4 of 42 overall, and is first among strict open-weight submissions; under Benjamini–Hochberg FDR control ($q \leq 0.05$), it significantly outperforms 34 of 41 Java peer systems on paired per-instance outcomes.

C5. Cross-language transfer of the same harness to Java. The phase decomposition, action contract, persistent shared state, and cross-agent selector transfer from Python to Java unchanged. Per-phase share of assistant messages matches the Python run within ± 5 percentage points on every phase, six of seven per-instance effort medians sit within 20 % of Python parity, and the same 27-billion-parameter Qwen3.5 backbone beats the strongest same-class open-weight Java peer (Qwen2.5-72B) by 38 instances, compared with a 26-instance same-family margin on Python.

Evidence boundary. Table 1 separates what this experience report establishes through controlled comparisons from what it supports only through operational signatures. “Ablated” rows have a matched comparison in the published artifact; “operational” rows are evidenced by trajectory audits and inventories without component-removal reruns; “estimate” and “out of scope” rows are disclosed as such wherever they are cited.

Table 1: Evidence map: mechanisms and claims by type.

Mechanism / claim	Evidence type	Where
Candidate count $K=8$	Ablated: pass@1 334–343 vs. oracle 408	Tab. 7
Cross-agent selector	Ablated: 376 vs. 362 (order) vs. 346 (self-tests)	Tab. 7
Selector weights w_B, w_R	Sensitivity sweep: $\leq 5/500$ outcome change	§7
Phase graph, handover state, formatter, tool sandbox, CI reuse	Operational signatures; not ablated	§6, §9
Engineer-minute saving	Derived order-of-magnitude estimate	Tab. 2
GPU-hours	Wall-clock-derived estimate	§8.5
Production deployment, developer pilot, GPU accounting	Out of scope; future work	§10

Roadmap. §2–§4 cover prior systems, industrial background, and the runtime definitions. §5–§7 describe the observations, harness design, and cross-agent selector. §8 reports six scripted analyses over the published Python TTS@8 and Java xcheck@8 artifacts. §9 states operational lessons, §10 concludes, and the Data Availability Statement defines the public artifact boundary.

2 Related Work

Agentic SWE Systems. SWE-agent [42] established the now-standard pattern of an LLM observing a shell and emitting commands, building on broader reason-act and reflective-agent patterns [25, 34, 43, 44]. We build on a minimal open-source runtime and add a phase-driven contract, a fixed tool sandbox, and a multi-cluster pipeline. AGENTLESS [40] uses a simpler localization-fix-verify pipeline; OpenHands and AutoCodeRover provide complementary open software-agent platforms and program-improvement systems [37, 47]. Our system is closer to a long agentic loop but constrains it through phases and shared state. The clearest difference is how test assets are handled: AGENTLESS selects one reproduction test and one regression set per issue, whereas each Kozuchi agent run independently generates tests that are later cross-applied to other candidates.

Industry-Facing Coding Agents. Commercial systems such as GitHub Copilot’s cloud coding agent [13], Amazon Q Developer [2], Google Jules [20], OpenAI Codex [30], Anthropic Claude Code [3], and Devin [39] validate industrial demand for asynchronous issue-to-patch agents. Their public descriptions are necessarily product-oriented; our focus is complementary harness evidence: phase boundaries, deterministic tools, queue-aware CI orchestration, reusable benchmark artifacts, and inspectable selection.

Tools, Training, and CI. Toolformer-style work injects tool calls into model training [33]; generated-test selection and pass@k-style code evaluation provide related precedents for using execution evidence around generated code [6, 7]. Our approach is the dual, holding the model fixed while injecting classic SE tools (line tracing, caller discovery, guarded editing) at runtime. SFT and RL recipes for code agents remain active directions, and systems such as Nemotron-CORTEXA [35] use fine-tuned retrievers and graph localization. R2E-Gym [17] explores verifiers for BEST@K; our cross-agent testing is execution-based but deliberately model-free. Finally, deployed LLM-for-CI systems such as LogSage [41] share our concern for artifact availability and reproducibility, although in our case CI orchestrates the agent workload itself.

Table 2: Per-evaluation-cycle workflow comparison.

Aspect	Pre-Kozuchi agent	Kozuchi agent
Operator touch-points / cycle	5 (vLLM, launch, wait, grade, log)	1 (CI push)
Operator effort / cycle (est. min)	≈75	≈5
Backend integrations	separate integration effort	4 action formats × 17 model configs
Reusable pipeline stages	0 of 9	6 of 9
Cluster targets / abstraction	per-cluster scripts	5 (ENV_NAME)
Score provenance	manual score log	495 traj. + selection JSON
Patch applicability	not measured	494/495 clean apply

In sum, the individual harness mechanisms—phased workflows, persistent state, runtime tools, CI orchestration—draw on established practice. The originality of this work lies in the cross-agent selector formulation, in which independent runs generate tests that are cross-applied as selection evidence under a constraint that no hidden or production-only feedback is available at selection time, and in the integration and industrial-scale evaluation of these mechanisms as one auditable harness.

3 Background and Industrial Context

3.1 SWE-bench and Multi-SWE-bench Java

SWE-bench is a benchmark of real GitHub issues paired with the affected repository state and a hidden test suite; an entry counts as *resolved* only when the model’s patch passes the issue-revealing tests while preserving the existing regression tests [18]. It extends a longer code-generation and program-repair evaluation line from function-level benchmarks to repository-level issues [7, 12, 14, 22–24]. SWE-bench Verified is a 500-instance human-curated subset with cleaner problem statements and reliable evaluation tests [29]; it is the Python benchmark used in RQ1–RQ5. Multi-SWE-bench Java is the corresponding Java track used in RQ6: it keeps the same issue-to-patch resolved metric, but evaluates Java repositories through Maven/Gradle-style harnesses and a separate public leaderboard [46]. We report the Python result on the sb-cli cloud stack [36] and the Java result on the Multi-SWE-bench Java leaderboard, keeping each benchmark’s denominator and peer set separate.

3.2 Agentic Loops and Base Runtime

Kozuchi Agent uses mini-swe-agent [9] as its base tool-using repair loop. Its contribution is the harness around that loop: phase control, persistent shared state, deterministic tools, and model-independent action formatting for long repairs.

3.3 The Workflow We Set Out to Replace

Before Kozuchi Agent, each iteration crossed heterogeneous internal clusters and required manually starting a vLLM server, launching a one-off repair-benchmark run, waiting hours, running Docker grading, and recording scores in a spreadsheet. The harness replaces this workflow with a CI-driven pipeline, one configurable agent, a fixed runtime contract, and a publishable artifact bundle. Table 2 summarizes this workflow inventory; the engineer-minute estimate is an order-of-magnitude operational estimate bounded by published SWE-bench Docker-runtime and CI-adoption studies [1, 15].

4 Definitions and Design Invariants

We adopt the following definitions for the rest of the paper. Some invariants are mechanically enforced by parsing and configured transitions; others are prompt/workflow contracts audited through trajectories.

D1 (Agent Runtime $\mathcal{R}$). The runtime is $\mathcal{R} = \langle \mathcal{P}, \mathcal{S}, \mathcal{T}, \mathcal{A}, \mathcal{F}, \mathcal{H} \rangle$: phases, shared state, tools, action format, parser, and handover function. This tuple is declared by configuration rather than recovered from ad hoc scripts.

D2 (Trajectory). A trajectory is the ordered record of prompts, assistant completions, parsed tool calls, tool responses, format/-timeout feedback, phase handovers, and shared-state snapshots. Trajectories are published per instance and are the main audit object.

D3 (Phase Transition System). The phase transition system is a directed graph $G_\Phi = (V_\Phi, E_\Phi)$. The evaluated graph has eight phases and fifteen normal/fallback edges (Algorithm 1).

D4 (Tool-Call Contract). Each assistant turn must contain exactly one parseable action. Four surface formats let the same logical contract serve different model families while keeping parser failures explicit in the trajectory.

D5 (Verifier/Selector). A verifier returns a binary or numeric pass-rate signal for a candidate patch. We use an intra-trajectory verifier for tests generated inside one run, and a cross-agent selector (§7) that cross-applies each run’s generated tests to every candidate patch.

D6 (Artifact Boundary). The public artifact includes trajectories, patches, per-run reports, selection tables, and official submission files. Transient scratch state, internal cluster configuration, and pipeline source are outside the release boundary because they contain internal paths, runners, tokens, or licence-sensitive dependencies.

Runtime Invariants. Every assistant turn yields one parseable action or a recorded format error; phase exits follow configured edges; generated notes and logs stay outside the submitted patch; long observations are summarized with explicit elision; and repeated failures or repeated identical actions trigger a strategy-change warning.

5 Motivating Empirical Observations

Early prototypes exposed recurring operational failures: long repair contexts lost evidence and repeated work, model-specific tool syntax made backend swaps brittle, shell-only debugging localized defects poorly, and multi-hour benchmark stages made recomputation expensive. The system design below turns these observations into phase boundaries, durable shared state, deterministic tools, configurable action formats, and CI artifact reuse.

6 System Design

Figure 2 summarizes the architecture at two connected resolutions. The broad path shows how the pipeline turns one issue into K auditable candidate trajectories and a final submitted patch. The

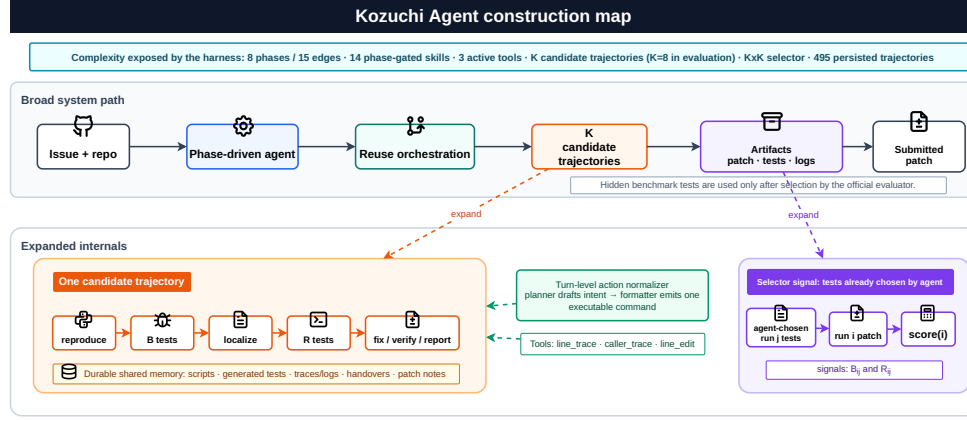


Figure 2: Kozuchi Agent workflow from candidate generation to cross-agent selection.

Algorithm 1 Phase-driven repair orchestration

Input: Issue I , repository checkout R , phase graph Φ , skill library $\mathcal{K}$, tool set $\mathcal{T}$, action grammar A , shared artifact store S , and limits B .
Output: Candidate patch Δ , final report ρ , and trajectory bundle τ .

```

1:  $p \leftarrow p_{\text{reproduce}}; H \leftarrow \emptyset; M \leftarrow \emptyset; \tau \leftarrow \emptyset$ 
2:  $\triangleright$  In the evaluated graph,  $p$  ranges over issue reproduction, FAIL-to-PASS test synthesis, code localization, PASS-to-PASS test localization, code fixing, patch verification, issue closeout, and final reporting.
3: while  $p \neq p_{\text{final}}$  do
4:    $K_p \leftarrow \{k \in \mathcal{K} : p \in \text{phases}(k) \wedge \text{tools}(k) \subseteq \mathcal{T}\}$ 
5:    $C_p \leftarrow \text{Render}(I, R, \Phi[p], K_p, S, M, H)$ 
6:    $\triangleright C_p$  contains the phase definition, available tools, selected skill guidance, handover memos, and artifacts already written to  $S$ .
7:    $e \leftarrow \perp$ 
8:   while  $e = \perp$  do
9:      $y \leftarrow \text{LLM}(C_p, H)$ 
10:     $a \leftarrow \text{Parse}_A(y)$ 
11:    if  $a = \perp$  then
12:       $o \leftarrow \text{FormatError}(y, A)$ 
13:       $\triangleright$  Ambiguous or multi-action output is recorded but not executed.
14:    end if
15:    if  $a \neq \perp$  then
16:       $o \leftarrow \text{SandboxExec}(a, R, \mathcal{T})$ 
17:       $S \leftarrow \text{UpdateArtifacts}(S, p, a, o)$ 
18:       $\triangleright$  Phase artifacts include reproduction scripts, tests, traces, root-cause notes, patch diffs, and reports.
19:    end if
20:     $H \leftarrow H \circ (y, a, o); \tau \leftarrow \tau \circ (p, y, a, o, S)$ 
21:     $e \leftarrow \text{ExitMarker}(y, o)$ 
22:    if  $e = \perp$  and  $\text{OverBudget}(H, S, B, p)$  then
23:       $m \leftarrow \text{Compress}(H, S, p); S \leftarrow S \cup \{m\}; H \leftarrow \emptyset$ 
24:       $C_p \leftarrow \text{Render}(I, R, \Phi[p], K_p, S, M, H)$ 
25:    end if
26:  end while
27:   $(p, M) \leftarrow \text{PhaseHandover}(\Phi, p, e, H, S, M)$ 
28:   $\triangleright$  Handover extracts phase artifacts, follows the configured next/fallback edge, writes a memo, and preserves scratch state outside the patch.
29: end while
30:  $(\Delta, \rho) \leftarrow \text{CollectFinalArtifacts}(S, M)$ 
31: return  $(\Delta, \rho, \tau)$ 

```

in-depth path shows the internal structure that makes those trajectories useful evidence: phase boundaries define what each run must

produce, the agent chooses or synthesizes the tests during those phases, durable shared memory persists tests and handover state, deterministic tools make localization and edits inspectable, and cross-agent selection converts those independently chosen tests into signals as defined in Eq §1.

6.1 Agent Runtime Contract

The agent runtime is specified declaratively: one configuration file defines the phase graph, prompt templates, action-interface family, prompt-guidance skill library, and operational limits. The runtime contract is therefore explicit before execution, instead of being recoverable only from scattered scripts.

Phases. The evaluated phase graph has eight nodes and fifteen directed edges (Algorithm 1). Each phase encodes a local workflow and two possible exits: successful completion or explicit give-up. This structure does not make the agent more capable by itself; rather, it is designed to keep long trajectories from becoming unstructured conversations. The phase definitions and handover prompts make phase exits auditable: when a run leaves a phase, the next phase is selected from the configured edge and the handover memo records what evidence is carried forward. Concretely, PHASEHANDOVER extracts scripts, traces, tests, patch diffs, verification logs, and reports from the exiting phase; chooses either the configured next edge or the configured fallback edge based on the exit reason; writes a memo that summarizes carried evidence and open risks; and preserves scratch notes in shared state rather than in the submitted patch.

Proposition 1 (Workflow Well-Formedness). In the observed Python headline trajectories, phase visits use the configured phase names and the run terminates either in final reporting or in an explicit runtime outcome.

Argument. The phase graph is declared in configuration, and the trajectory-derived phase summaries show complete coverage of the eight declared phases for the 495 persisted trajectories. We do not claim liveness—the agent is not guaranteed to solve every instance—only that the published run is auditable against the configured phase graph.

Skills (Phase-Gated, Tool-Gated). Skills are reusable prompt-guidance blocks, not learned model capabilities. Each skill gives the agent a small procedure or tool-use pattern, and is injected only when the current phase and available tools make it relevant. The current configuration contains 14 such phase- and tool-gated skills. This deliberately conservative injection policy shortens the prompt and narrows the action space, which matters when a repair trajectory can span hundreds of turns.

Action Formats. The action interface separates the semantic requirement—one executable action per turn—from model-specific syntax. The same runtime can therefore evaluate several model families without changing the agent’s state machine or tool semantics.

Tool Sandbox. At runtime, the agent receives a fixed set of deterministic software-engineering operations. The current set covers dynamic line tracing, caller discovery, and guarded line editing. Keeping this set small made tool behavior inspectable and prevented the agent from depending on a large, brittle tool zoo.

Templates. Templates enforce the interaction contract: each turn contains an observable rationale field and exactly one action, and long observations are summarized with explicit elision rather than silently dropped. We treat that field as trajectory text, not as a faithful record of the model’s cognitive state; the goal is to make the observable interface stable.

Context Compression and Handover. State handover addresses a simple failure mode: facts stored only in the conversation can disappear when context is compressed. At phase changes, and when a phase grows too long, the agent writes a concise memo and continues with the relevant shared artifacts. The filesystem, not the raw conversation, becomes the durable memory.

Orchestra Runtime. For the Python headline submission, each turn is processed by two roles. A planning role drafts the next step, and a formatting role rewrites only the executable action to satisfy the active interface. Both roles use the same locally hosted model. This division is a harness-level substitute for fine-tuning: the planner can express the repair strategy naturally, while the formatter normalizes only the command surface before execution, reducing formatting failures without changing model weights.

6.2 Software-Engineering Tool Suite

The suite is deliberately small: **line tracing** records which statements and values appear under a reproduction test, turning localization into execution evidence; **caller tracing** shows the paths that reach suspect code and connects local symptoms to user-visible behavior; and **guarded editing** couples each edit to an expected target line, avoiding many free-form patch corruptions. The main policy is phase-gated injection: the agent sees only the tools and guidance relevant to its current phase, making tool availability part of the experimental contract.

6.3 Multi-Cluster Pipeline

The pipeline turns benchmark experimentation into a set of reusable stages: candidate generation, grading, data synthesis, optional post-training, test-time selection, and reporting. Each stage can run

locally for debugging or as a scheduled cluster job. The practical contribution is reuse: operators can re-run only the stage under study while holding earlier artifacts fixed. This is what made repeated TTS@8 sweeps feasible in day-to-day development.

6.4 Model-Agnostic Serving

The serving layer decouples model identity, chat formatting, and execution sandbox. The same agent can run against local open-weight models, multi-server vLLM deployments, and remote APIs [21]. The backend inventory is reported with the artifact materials.

6.5 CI as the Evaluation Substrate

CI is the evaluation substrate, not just a test runner. It records the stage graph, the cluster target, the reusable artifacts, and the conditions under which expensive work should be skipped. The stage graph covers preparation, candidate generation, grading, synthesis, optional training, post-training benchmarking, selection, and reporting; reuse short-circuits let operators iterate on one stage without invalidating the rest. This follows standard CI practice of keeping frequent integration feedback cheap and repeatable [10, 11].

7 Cross-Agent Test-Time Selection

A central design choice is to separate *candidate generation* from *final selection*. With $K = 8$ independent runs of the same locally hosted model, each run produces a candidate patch together with its own bug-revealing and regression-preservation tests. The candidate agent decides which tests exist before final selection; the cross-agent selector does not choose tests, but only cross-applies the archived test runners. Each run contributes two kinds of evidence: tests intended to expose the reported bug, and tests intended to guard nearby behavior against regressions. These tests are fixed before final selection, so the selector can compare candidate patches without accessing hidden benchmark tests. At final selection time, we form a $K \times K$ matrix in which entry (i, j) is the result of running run j ’s archived tests against run i ’s patch, converting candidate diversity into a selection signal using only evidence archived by the candidate runs.

Selector. For each candidate i we compute

$$\text{score}(i) = \frac{1}{K} \sum_{j=1}^K (w_B B_{ij} + w_R R_{ij}) \quad (1)$$

Here B_{ij} and R_{ij} are the pass rates obtained by the cross-application process above: B_{ij} is the pass rate of candidate i on run j ’s bug-revealing (FAIL_TO_PASS) tests, and R_{ij} is the corresponding pass rate on its regression (PASS_TO_PASS) tests. The weights $w_B=0.3$ and $w_R=0.7$ are pragmatically selected fixed global constants in the submitted selector artifact; they are not learned and were not tuned on benchmark outcomes. The asymmetry encodes a deliberate engineering prior for a merge-gated setting: a patch that silently breaks existing behaviour (a regression that a PASS_TO_PASS suite would catch) is the costlier failure once merged, whereas agent-generated bug-revealing tests are sparser and noisier as evidence, so regression conformance is weighted higher than bug-revealing pass rate. Nevertheless, an artifact-only sweep over seven nearby

Table 3: Cross-agent selector summary.

Quantity	Value	Quantity	Value
Expected (Verified)	500	w_B	0.3
Selected (covered)	495	w_R	0.7
Resolved (xcheck selector)	376	Tie epsilon	0.001
Score over expected (xcheck)	0.752	Tie-breaker	shortest_patch_raw
Resolved (order baseline)	362	Tie-break applied	304/495
Score over expected (order)	0.724	Mean unique patches	5.59/8
Tie instances	304	Matrix executions	21,635
Reordered by selector	302		

weight pairs changes at most 35/495 selected candidates and 5/500 estimated outcomes (≤ 1.0 pp), so we report the submitted pair as a robust operating point rather than an optimized one; a systematic weight search on held-out tasks remains future work. We then pick $\arg \max_i \text{score}(i)$; when execution evidence is indistinguishable, we break ties by shorter patch length as a conservative engineering heuristic. A tie means that at least two deduplicated patches are within 10^{-3} of the best score; this occurs on 304/495 covered instances. The selector summary is reproduced as Table 3.

Why Cross-Agent Rather Than Oracle Best-of-K. We do not use the official benchmark tests at selection time; doing so would leak evaluation information. The same constraint matters in production: a bad merge can cause significant downstream damage, so the system needs a selection signal before any hidden or production-only feedback is available. Instead, each run contributes the tests that its own agent chose or generated while solving the issue. The selector treats those archived tests as fixed evidence and cross-applies them to other runs’ patches, turning trajectory diversity into selection evidence without any learned verifier or LLM judge. The matrix admits only archived suites with both `has_fail_to_pass` and `has_pass_to_pass`, so empty or invalid generated-test bundles are dropped before selection (12–21 drops per leg); candidate patches are deduplicated by content hash, and the 2/2,768 deduplicated patches with non-ok apply status are excluded from scoring. We do not infer flakiness or over-breadth from logs; after archive creation the selector reads only suite outcomes and apply status, never SWE-bench hidden-test results. Per-instance reordering data backs this up: on 302/495 instances the selector chose a candidate other than the order baseline, demonstrating that it actively re-ranks rather than acting as a no-op.

Diversity Ceiling. A simple bound on selector quality comes from the union and intersection of the eight runs: at least one run resolves 408 instances, while all runs resolve 234. The cross-agent selector reaches 376 on the internal Docker re-grade (374 on the official evaluator), or about $376/408 = 92.2\%$ of the oracle union ceiling under the internal Docker re-grade. Pairwise completion overlap is high, so the useful diversity is mostly in *which* instances are solved, not in whether runs cover different parts of SWE-bench Verified. Table 4 gives the candidate-level decomposition that sharpens this bound: for 92 instances, none of the eight runs produces a resolving candidate, so these instances cannot be recovered by a better selector. Only 174 instances are “marginal”—resolved by some but not all runs ($1 \leq r_i \leq 7$)—and therefore actually expose a selection choice. Measured against the official Python headline, the cross-agent selector leaves 34 instances (6.8 pp) below the oracle union; linear interpolation on the closed-form oracle pass@k curve

Table 4: Candidate-level TTS@8 decomposition.

Quantity	Interpretation	Value
Mean pass@1	Average over eight runs	67.7 %
All-run solved	$r_i = 8$ scaffold-easy instances	234 / 500
Zero-run solved	$r_i = 0$ scaffold-hard instances	92 / 500
Marginal instances	$1 \leq r_i \leq 7$ selector-sensitive	174 / 500
Oracle pass@8	Any run resolves	408 / 500 = 81.6 %
Cross-agent selector	Official-evaluator result	374 / 500 = 74.8 %
Selector regret	Oracle minus submitted result	34 / 500 = 6.8 pp

places the cross-agent selector near an oracle with $k \approx 2.2$, so much of the eight-way compute budget is currently spent on candidate generation that the selector does not fully monetise.

8 Evaluation

Evaluation setup. RQ1–RQ5 use SWE-bench Verified as the fixed Python task suite; RQ6 uses Multi-SWE-bench Java as a cross-language validation track. The Python headline configuration uses a locally hosted model out of the box: the approach does not require training or fine-tuning before connecting the model to the agent runtime. For Python we run the agent with the same model eight times, archive each run’s candidate patch and tests, and then apply the cross-agent selector described in §7. We report the official SWE-bench cloud evaluation as the primary Python result and use internal Docker re-grading only as a consistency check. The Java RQ6 result uses the same 27-billion-parameter agent and phase graph, but the Java xcheck@8 artifact and Java leaderboard rather than the Python Docker/cloud evaluator. This setup separates three questions for each track: how often the full system resolves issues, how it compares with published peers, and where remaining errors come from.

Reproducibility configuration. The public artifact records two reproducibility layers: the declarative runtime configuration (`configs/agent_sota.yaml`) and the executed Python sweep manifest (`provenance/sweep_manifest.json`). The run uses backbone Qwen/Qwen3.5-72B [32], $K=8$ independent candidate runs, selector weights $w_B=0.3$ and $w_R=0.7$ (§7), run labels `r01_s1001` through `r08_s1008`, and the tool set `line_trace`, `caller_trace`, and `line_edit`. The manifest records serving/run parameters including two-GPU tensor parallelism, 50 shards, `max_model_len=150000`, `max_new_tokens=16384`, `step_limit=1000`, `instance_timeout=06:00:00`, and the best-known repository commit before submission; the analysis scripts use fixed seed `SEED=20260427`. Model weights remain external licensed dependencies (Data Availability Statement), so the artifact pins the model identifier and run provenance rather than redistributing the checkpoint.

The Python analysis uses the published TTS@8 artifacts: 495 SWE-bench harness reports, 495 trajectories, the cross-agent selection bundle, and the per-instance outcome vectors of a curated competitor set. We adopt the canonical denominator $N=500$ for every SWE-bench Verified aggregate; the five trajectory/report pairs that are missing from the bundle are folded into the unresolved bucket as `MISSING_ARTEFACT` failures rather than dropped. The Java analysis uses the published Multi-SWE-bench Java submission bundle with canonical denominator $N=128$, described separately in RQ6.

Table 5: SWE-bench Verified resolution by repository.

Repository	Resolved/Total	Rate	Repository	Resolved/Total	Rate
astropy/astropy	13/22	59.1%	pydata/xarray	18/22	81.8%
django/django	177/231	76.6%	pylint-dev/pylint	3/10	30.0%
matplotlib/matplotlib	23/34	67.7%	pytest-dev/pytest	16/19	84.2%
mwaskom/seaborn	1/2	50.0%	scikit-learn/scikit-learn	27/32	84.4%
pallets/flask	1/1	100.0%	sphinx-doc/sphinx	30/44	68.2%
psf/requests	8/8	100.0%	sympy/sympy	57/75	76.0%
Overall: 374/500 (74.8%)					

8.1 RQ1: Headline Accuracy and Uncertainty

Result. The Python configuration resolves 374/500 instances on the official cloud evaluator (74.80 %, Wilson 95 % CI [70.82 %, 78.41 %] [38]). The corresponding internal Docker re-grade resolves 376/500 (75.2 %); the two-instance delta reflects evaluator differences and we report the official Python number as the Python headline. A repository-clustered bootstrap ($B=10,000$ replicates resampling at the repository level) returns a wider 95 % CI of [67.0 %, 79.8 %], reflecting the fact that resolution is correlated within repositories; we therefore report both intervals to make the headline comparable to peers under their own reporting conventions and conservative to clustering.

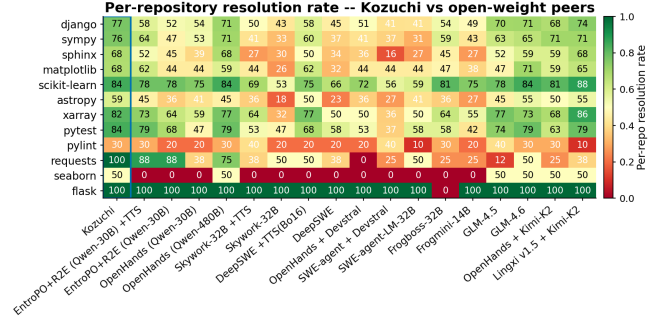
Per-repository breakdown. Table 5 gives the repository breakdown. The largest repository, django/django, accounts for 46.2 % of the Python benchmark and resolves at 76.6 %, just above the global mean. The lowest-rate repository is pylint-dev/pylint at 30.0 % ($n=10$). The auxiliary per-year CSV remains available in the artifact; across 2017–2023, year-level resolution stays in the 66 % to 94 % band, with the high end driven by the small 2017 slice. We report the structural repository pattern rather than mechanism because the workspace does not contain a controlled study of repository-internal API churn or year effects.

Threats. Python cross-system comparisons against published numbers from other research groups use heterogeneous evaluation stacks; we therefore restrict our Python headline number to the official evaluator and use paired per-instance outcome vectors (rather than aggregate rates) for all Python peer comparisons in RQ2. A run-to-run variance estimate beyond the eight-run sweep would require additional eight-run sweeps under the same Orchestra configuration; we instead report the per-run pass@1 dispersion in RQ5 (eight independent decoding seeds, $\sigma=3.12$ resolved instances, range 334–343/500), which we treat as the within-configuration variance estimate for the headline.

8.2 RQ2: Peer-Ranked Position With Multiplicity-Corrected Tests

Setup. We rank our system on a frozen snapshot of 135 publicly cataloged SWE-bench Verified as of April 28, 2026, and run paired McNemar exact tests [26] on the per-instance outcome vectors of a curated set of 17 open-weight and 7 closed-frontier peers. Multiplicity is controlled across the 24 paired tests with both Holm–Bonferroni (family-wise error rate) and Benjamini–Hochberg (false discovery rate) procedures [5, 16], each with target $\alpha = 0.05$ or $q = 0.05$.

Result. On the frozen Python leaderboard snapshot, the configuration ranks **12th overall** and is the highest-ranked open-weight system; the next open-weight submission, *Lingxi v1.5 + Kimi-K2*,

**Figure 3: Per-repository resolution rate for Kozuchi agent (first column on the left) and the curated open-weight peers.**

resolves 356/500 (71.2 %). Under Benjamini–Hochberg FDR control ($q \leq 0.05$), the configuration significantly outperforms 16 of 17 curated open-weight peers, including a 480-billion-parameter Qwen3-Coder configuration ($\Delta = +26$ instances, BH-FDR $q = 0.011$). The single peer the configuration does not significantly outperform under either FWER or FDR control is *Lingxi v1.5 + Kimi-K2* (raw $p = 0.054$, $\Delta = +18$). Among closed-frontier peers, the configuration is statistically indistinguishable at $\alpha = 0.05$ from *Sonar + Claude-Sonnet-4.5* ($\Delta = 0$), *OpenHands + Claude-Opus-4.5*, *live-SWE + Gemini-3-Pro*, and *OpenHands + GPT-5*; significantly outperforms *OpenHands + Claude-4-Sonnet* (FDR-significant); and is significantly behind the two top *Sonar / live-SWE + Claude-Opus-4.5* builds (FDR-significant; $\Delta = -22$ in both cases). The full pair-wise tables list raw McNemar p -values, Holm-corrected p -values, BH-corrected q -values, and conditional log-odds-ratio effect sizes with exact 95 % Clopper–Pearson confidence intervals [8].

Per-repository peer structure. Figure 3 shows that the open-weight lead is not only an aggregate leaderboard artifact. On the largest benchmark slice, django/django ($n = 231$), Kozuchi agent resolves 76.6 %, ahead of the next open-weight peer in the heatmap at 74.0 %; on sympy/sympy ($n = 75$) it resolves 76.0 %, compared with 70.7 % for the strongest peers; and on pytest-dev/pytest ($n = 19$) it reaches 84.2 %, above the nearest peers at 78.9 %. The same pattern appears on several medium-size repositories: sphinx-doc/sphinx is tied for best at 68.2 %, and pydata/xarray is near the top at 81.8 %. The main exception is pylint-dev/pylint, where Kozuchi agent resolves only 30.0 % (3/10) and several peers reach 40.0 %. We therefore interpret the peer advantage as broad but not uniform: it is strongest on the large repositories that dominate the Verified denominator, while low-count repositories should be read as diagnostic failure clusters rather than stable rank evidence. §8.6 extends the peer position analysis to Java, where the same 27-billion-parameter model configuration **ranked first** among strict open-weight submissions.

Threats. Ranks use a frozen snapshot of cataloged submissions.

8.3 RQ3: Where Does Performance Vary?

Setup. For the Python SWE-bench Verified run, we stratify resolution by patch size, by per-instance LLM-call count, and by phase-level events extracted from the 495 trajectory JSON files; we also report a multivariate logistic regression with cluster-robust

standard errors (clustering at the repository level) that adjusts trajectory features for external consensus hardness proxies. Because the consensus terms use peer outcomes, we treat this model as an explanatory diagnostic rather than as a deployable predictor.

Patch size. Resolution rate decreases monotonically with the line-of-code (LOC) churn of the produced patch: from 81.7 % at 1 LOC to 4 LOC to 41.7 % at >100 LOC. The point-biserial correlation between LOC churn and the binary resolution outcome is $r = -0.197$ ($p = 1.1 \times 10^{-5}$), the strongest single feature signal in the dataset. The mean LOC added per patch is 10.5 LOC (median 3 LOC) for resolved instances versus 21.9 LOC (median 8 LOC) for unresolved instances.

Trajectory effort. Trajectory-level effort metrics (per-instance LLM calls, runtime, prompt tokens, bash invocations) all carry *negative* correlations with resolution. We interpret this signal as a hardness-selection effect: harder instances cost more effort and are also more likely to fail. We do not interpret it as “effort hurts”; no effort metric is positively correlated with resolution at $\alpha = 0.05$, which we report as a structural ceiling on what further inference compute can buy at fixed scaffold and backbone.

Phase-level behaviour. All 495 trajectories visit every one of the eight phases; rework is concentrated in the CODE_FIX $\leftrightarrow$ VERIFY_PATCH loop. VERIFY_PATCH issues a GIVEUP in 18.2 % of trajectories (90/495, 95 % CI [15.0 %, 21.8 %]); CODE_FIX itself rolls back in 1.0 % of cases; every other phase rolls back zero times. Unresolved trajectories fire roughly 28 % more CODE_FIX messages than resolved ones (mean 170 vs. 132) and report 0.67 vs. 0.52 mean VERIFY_PATCH GIVEUPs per instance.

Multivariate adjustment. A logistic regression of the binary resolution outcome on log-transformed LLM calls, patch churn, runtime, and two external consensus hardness proxies (qwen_consensus and frontier_consensus) shows that the consensus terms dominate the multivariate signal. After this adjustment, the trajectory-effort and patch-churn coefficients are not significant at $\alpha = 0.05$. This supports the hardness-selection interpretation above and prevents the univariate patch-churn correlation from being read as an independent causal predictor.

Threats. The consensus-adjusted model requires other systems’ outcomes, and its coefficients should not be read causally.

8.4 RQ4: Failure Taxonomy and Operational Reliability

Setup. For the Python SWE-bench Verified run, we classify each unresolved instance into one of five mutually exclusive failure categories from the SWE-bench harness reports. We then report the operational profile of the 495 trajectories that produced a harness report: phase visit completeness, exit status, and patch-application reliability.

Failure taxonomy. Of 126 unresolved instances, 115 (91.3 %) are WRONG_FIX: the patch applies cleanly under the harness but does not flip the hidden FAIL_TO_PASS tests. 6 (4.8 %) are REGRESSION: the patch applies but breaks at least one PASS_TO_PASS test. 5 (4.0 %) are MISSING_ARTEFACT: no trajectory or harness report was persisted for the instance under the published bundle. The PATCH_DID_NOT_APPLY and EMPTY_PATCH categories are both 0. The taxonomy is summarised in Table 6.

Table 6: Failure taxonomy for unresolved instances.

Category	Count	Share
WRONG_FIX	115	91.3 %
REGRESSION	6	4.8 %
MISSING_ARTEFACT	5	4.0 %
PATCH_DID_NOT_APPLY	0	0.0 %
EMPTY_PATCH	0	0.0 %
Total	126	100.0 %

Operational reliability. All 495 trajectories report `exit_status = "Submitted"`; the agent does not crash mid-flight in this run. Of the 495 produced patches, 494 apply cleanly through the SWE-bench harness, giving a single edit-layer failure (0.20 %). Combined with the failure taxonomy, this isolates the residual gap as a semantic-correctness problem rather than an edit-format problem: the agent essentially always emits a syntactically acceptable patch, but in 91.3 % of unresolved cases the patch is semantically wrong.

Threats. MISSING_ARTEFACT cases are operational rather than modelling failures: under stricter persistence handling some of them might be recovered at no algorithmic cost. They are not removed from the Python headline denominator in this paper, but their existence inflates the unresolved bucket by up to 5/500 (1.0 pp).

8.5 RQ5: TTS@8 Selection Lift and Cost-Diversity Envelope

Setup. For Python, we compare per-run pass@1 patches against the cross-agent TTS@8 selection on the same eight runs. Cost is measured in additional compute: 8× candidate generation plus the cross-agent test cross-application matrix. For Java, the shipped bundle exposes the xcheck-selected source-run counts rather than complete per-leg pass@1 reports, so we use those counts only as a source-run audit of the selected Java outcome.

Per-run pass@1. The pass@1/pass@k terminology follows standard code-generation evaluation practice [7]. The eight Python pass@1 runs are tightly clustered: they resolve 334–343 of 500 instances (66.8 % to 68.6 %), with mean $338.6/500 = 67.7\%$ and population standard deviation 3.12 resolved instances (0.62 pp). The submitted cross-agent selection raises this average-run baseline to $374/500 = 74.8\%$ on the official evaluator and $376/500 = 75.2\%$ on the internal Docker re-grade, a gain of 35.4–37.4 resolved instances (7.1 pp to 7.5 pp) over mean pass@1. For Java, the xcheck artifact selects 15.9 source-run candidates on average and resolves 5.1 of them (population standard deviation 1.54), yielding $41/128 = 32.03\%$ overall and a 32.3 % conditional resolved rate over selected source-run contributions.

Selection lift. On Python, the cross-agent selector resolves $376/500 = 75.2\%$ on the internal Docker re-grade (Table 3) and $374/500 = 74.8\%$ on the official evaluator. On 302 of 495 covered instances the selector chooses a different candidate than the size-ordered first candidate. The order baseline (first-run-wins) resolves $362/500 = 72.4\%$; the selector therefore contributes +14 absolute (2.8 pp) over the order baseline and +33 to +42 absolute over individual per-run pass@1.

Selector ablation. Table 7 separates the measured Python TTS@8 gain into candidate generation and selection effects. Moving

Table 7: Artifact-based TTS@8 ablation. Scope: this table isolates candidate count (K) and selector design only; phase graph, formatter, persistent state, and tool sandbox are not ablated and are evidenced operationally in §6 and §9.

Variant	Resolved	Rate	Note
$K=1$, no selector (mean)	338.6	67.7%	eight-run mean
$K=1$, no selector (best)	343	68.6%	best leg
$K=8$, order baseline	362	72.4%	no cross-agent ranking
$K=8$, self-tests only	346	69.2%	estimated
$K=8$, cross-agent selector	376	75.2%	selected
$K=8$, oracle	408	81.6%	upper bound

from a single candidate to eight candidates creates the 408/500 oracle ceiling, while the cross-agent selector captures 376 of those instances on the internal Docker re-grade. A diagnostic self-tests-only selector, computed from the same archived cross-agent selection tables by scoring each candidate only on its own generated tests and then checking the chosen run’s per-run report, resolves 346/500. The gap between self-tests-only selection and the cross-agent selector’s 376/500 result supports the specific value of cross-agent test application, while the oracle gap leaves 32 internal Docker re-grade wins for better selection at fixed candidates. This ablation isolates candidate count and selector signals; the other harness mechanisms below are supported by operational signatures rather than by controlled component-removal reruns.

Diversity ceiling. Across the eight Python runs the union of resolved instances is 408 and the intersection is 234. The selector’s 376 corresponds to about 92 % of the internal Docker re-grade oracle ceiling (376/408). Pairwise Jaccard similarity over *completed* instances ranges from 0.943 to 0.965 (mean 0.957), so the diversity is in *which* instances are resolved rather than in coverage of the benchmark itself.

Compute envelope. On Python, TTS@8 buys the +14 absolute lift over the size-order baseline at $8\times$ candidate generation plus a separately logged selector matrix: 21,635 patch–suite executions over 495 covered instances (mean 5.59 deduplicated patches $\times$ 7.80 archived suites, versus a naive $8\times 8\times 495 = 31,680$ matrix), booked as 50 cross-check shards in `xcheck_manifest.json`. We measure candidate-generation cost from the published trajectories rather than from job logs. Across the 495 trajectories, per-instance medians are 490 LLM calls, 6.20M prompt tokens, 83,100 completion tokens, and 0.77 h of wall-clock; the p95 tail extends to 1,099 LLM calls, 16.6M prompt tokens, and 1.94 h. Aggregated over the full eight-run candidate generation, the configuration consumes approximately 3.8×10^9 prompt tokens and 5.8×10^7 completion tokens, so the workload is prompt-token dominated, consistent with the persistent handover and shared-state mechanism described in §6. Because the trajectory bundle records wall-clock but not GPU-utilisation traces, we can bound rather than measure GPU cost: mean per-instance wall-clock is 0.95 h on the two-GPU tensor-parallel serving configuration, so summing instance wall-clock across the full eight-run generation gives an upper bound of roughly 7.5×10^3 GPU-hours ($8\times 495\times 0.95\text{ h}\times 2$). This is an upper bound because the vLLM servers batch many concurrent instances on the same GPUs; per-instance wall-clock double-counts shared occupancy. Accounted GPU-hours would require cluster-side telemetry we did not collect (Table 1). The compute–resolution Pareto curve over the 374 resolved instances shows diminishing returns in the long tail:

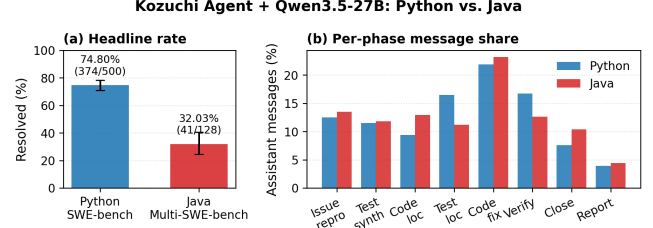


Figure 4: Cross-language transfer summary.

80 % of resolved instances finish within 653 LLM calls per instance and 95 % within 1,069, but moving from the 90 % point to the 99 % point requires increasing the budget from 820 to 1,547 calls. We report this Pareto as a descriptive cost curve over *this* run rather than as a prescription; an early-stopping policy that exploits it would need its own controlled study.

Scaling under tighter budgets. For organizations without the infrastructure to spend the full TTS@8 envelope, the measured cost structure suggests three policies, none of which we have validated with controlled runs: (i) a per-instance call or token budget that stops the long tail (cutting at the 90 % Pareto point of 820 calls sacrifices at most the last decile of resolved instances in this run); (ii) adaptive candidate count, since 234 of 500 instances are resolved by all eight runs and would be selected correctly from far fewer candidates; and (iii) skipping the cross-check matrix when deduplicated patches collapse to a single candidate. Each policy needs its own controlled study before deployment.

Threats. Cross-agent testing requires that each candidate run produces high-quality agent-generated tests; weak tests directly weaken the selector’s discrimination. A controlled comparison of direct versus debate-style selection on the same trajectory bundle would require a paired selector run with otherwise identical inputs. The compute envelope is reported in LLM calls, prompt/completion tokens, and wall-clock seconds, with GPU-hours given only as a wall-clock-derived estimate, because the trajectory bundle does not record GPU-utilisation traces; accounted GPU-hours require cluster-side telemetry we did not collect. The ablation in Table 7 isolates candidate count and selector signals, but it does not isolate the causal contribution of the phase graph, persistent handover, action formatter, tracing tools, or guarded editing; those components require separate controlled reruns (Table 1). The auditable selector boundary is the archived suite outcome and patch apply-status table, so hidden SWE-bench outcomes are unavailable until after patch selection.

8.6 RQ6: Cross-Language Transfer to Multi-SWE-bench Java

We re-ran the same Qwen3.5-27B agent, eight-phase scaffold, and $K=8$ selection pattern on Multi-SWE-bench Java. Only the corpus and harness change: Java uses Maven/Gradle and strict `xcheck@8`, while the Python headline uses `pytest` and TTS@8 (§7). The phase graph, action contract, persistent state, and selection family remain the same. Figure 4 shows headline resolution with Wilson intervals [38] and per-phase assistant-message share, while Table 8 summarizes the matched Python–Java evidence.

Table 8: Python–Java cross-track evidence for the same 27-billion-parameter Kozuchi agent configuration.

Signal	Python	Java
Benchmark	SWE-bench ($n=500$)	Multi-SWE-bench ($n=128$)
Resolved (Wilson 95% CI)	374/500 = 74.80% [70.82, 78.41]	41/128 = 32.03% [24.57, 40.54]
Leaderboard position	12/135 overall	4/42 overall
Open-weight rank	1st curated open-weight	1st strict open-weight
BH-FDR peer wins	16/17 open-weight peers	34/41 Java peers
Same-family open margin	+26 vs. Qwen3-Coder-480B baseline	+38 vs. Qwen2.5-72B CODE_FIX +1.3 pp; VERIFY_PATCH -4.1 pp
Rework-loop phase delta		
Median LLM calls	490	529
Nonempty applying patch output	494/495 = 99.80%	124/128 = 96.9%
Globally novel solves	1	1

The Java run resolves 41/128 instances (32.03 %, Wilson 95 % CI [24.57 %, 40.54 %]), ranks 4 of 42 overall, and is first among strict open-weight submissions. Under BH-FDR control ($q \leq 0.05$), it significantly outperforms 34 of 41 Java peers, including all 72-billion-parameter Qwen2.5 open-weight builds ($\Delta = +27$ to $+38$). The only FDR-significant adverse comparison is *CodeArts-Agent + CodeArts-MiniMax-M2.5* ($\Delta = -15$, $q = 0.011$), with a much larger closed-weight backbone.

The scaffold behaviour transfers more strongly than the absolute score: per-phase message share is within ± 5 pp on every phase, rework still centres on CODE_FIX $\leftrightarrow$ VERIFY_PATCH, and six of seven effort medians sit within ± 20 % of Python parity. Thus the 43 pp headline gap is not a compute gap. The Java failure profile is instead language/harness-specific: no_fix_test_results (25 %), report_valid_not_leaderboard_resolved (9 %), and anomalous_test_pattern (6 %) do not appear on Python; edit-layer reliability is lower (96.9 % vs. 99.8 %); and elastic/logstash accounts for 38/128 Java instances, with 0/38 solved by Kozuchi agent and 32/38 unresolved by every catalogued peer. We therefore claim language-agnostic harness transfer, not equal language fluency.

The Java leaderboard mixes open-weight, semi-open, undeclared, and closed-frontier peers; our open-weight claim uses the strict “decision-LLM weights publicly released” subset. As in RQ5, the Java run does not separately ablate scaffold, and selector contributions.

9 Operational Planning and Lessons Learned

The operational lessons from benchmark-scale use are practical rather than architectural. First, phase boundaries kept long-horizon runs structured and auditable: every Python TTS@8 trajectory visits all eight phases, rework concentrates in CODE_FIX $\leftrightarrow$ VERIFY_PATCH, and the same phase shape transfers to Java within ± 5 percentage points per phase (§8.6). Second, model-family variation is cheapest to handle at the action-format boundary: switching tool-call syntax is a configuration change rather than an agent fork. Third, CI reuse is a core evaluation lever, because operators can hold trajectories, selection bundles, or environments fixed while changing one stage. The same principle applies to filesystem-backed state: reproduction notes, generated tests, traces, and handover memos survive context compression and make runs reviewable after the fact. Finally, cross-agent testing turns candidate diversity into accuracy without a learned verifier, while one cluster abstraction keeps multi-site runs reviewable.

The artifact-derived operational indicators are consistent with those lessons: in the inventoried workflow, five operator touch-points become one CI push; the about-70-engineer-minute-per-cycle figure is an estimate derived from the published per-stage inventory and the SWE-bench Docker-runtime / CI-adoption anchors in prior work [1, 15]. It is not a measured before/after from a controlled developer pilot, and we report it as an order-of-magnitude operational claim rather than a productivity metric. The runtime exposes four action formats across 17 model configs, six of nine CI stages are reusable, and five cluster environments sit behind ENV_NAME. On Python, 494/495 produced patches apply cleanly and the cloud-vs-Docker score drift is only 374 vs. 376 out of 500. On Java, the same runtime contract reaches 41/128 (32.03 %) and ranks first among strict open-weight submissions, making cross-language transfer a measured operational outcome rather than an anecdotal consistency check.

10 Conclusion

Operational readiness evidence. Kozuchi Agent targets conservative pre-production evaluation of issue-to-patch agents on large repositories, with evidence from SWE-bench (Python) and Multi-SWE-bench Java. Using a locally hosted open-weight model, no fine-tuning, eight candidate runs, and cross-agent testing, it resolves 374/500 SWE-bench Verified instances (74.80 %, Wilson 95 % CI [70.82 %, 78.41 %]; repository-clustered bootstrap [67.0 %, 79.8 %]) and 41/128 Multi-SWE-bench Java instances (32.03 %) under the same agent, backbone, and scaffold. It ranks 12th among 135 catalogued Python submissions, is the highest-ranked open-weight Python system, ranks first among strict open-weight Java, and recovers 80 % of resolved Python instances within 653 LLM calls.

What we learned. The results point to harness engineering—phase decomposition, single-command grammar, persistent shared filesystem, deterministic SE tools, cross-agent test selection, cluster indirection, and CI reuse—while ablations isolate only candidate count and selector signals; we leave controlled per-mechanism ablations (no-phase, no-formatter, no-Orchestra, no-tools) to a follow-up that can re-use the published trajectory bundle without re-running TTS@8. The phase decomposition, action grammar, persistent state, and cross-agent selector transfer from Python to Java unchanged, supporting the narrower cross-language harness-transfer claim. The selector adds +14 resolved Python instances; 494/495 Python patches apply cleanly; and remaining Python errors are mainly semantic or selection failures: 91.3 % of unresolved instances have clean-applying patches that miss hidden tests, and 34 instances are solved by some run but not selected.

Next. Future work should ablate tools, selector design, and post-training; use conversation audits for content-aware selection; connect failure-pattern detection to harness updates; and use the compute-resolution Pareto curve for controlled early stopping and adaptive candidate counts (§8.5). Industrial-impact claims are deliberately scoped to internal evaluation-pipeline replacement (Table 2, §9), cross-evaluator consistency (374/376 agreement), and Python–Java benchmark transfer of the same agent. Production deployment against proprietary repositories, end-user developer studies, field A/B telemetry, and broader multilingual benchmarks are explicit future work.

Contributions of Each Team

Fujitsu Research, Japan.

- Led the overall design and implementation of Kozuchi Agent.
- Conducted evaluations on SWE-bench.

Fujitsu Research of America, USA.

- Led the writing of this paper.
- Designed the Verifier.
- Developed and prepared tools for Java.
- Conducted evaluations on Multi-SWE-bench Java.

Fujitsu Research of Europe, UK.

- Developed the Software Engineering Tool Suite.

Fujitsu Research & Development, China.

- Conducted evaluations of test-time selection.

Data Availability Statement

The public review artifact, including all analysis, is available at <https://github.com/marscod/kozuchi-agent-artifact> and Kozuchi Agent source code at: <https://github.com/FujitsuResearch/kozuchi-mini-swe-agent>.

The complete agent execution trajectories are available in the Zenodo archival record (DOI: 10.5281/zenodo.19941649), together with the agent source code, submission folders, logs, predictions, selector bundle, generated summaries, figures, audits, and scripts needed to regenerate RQ1–RQ6.

References

- [1] Tom Adamczewski. 2025. How to Run SWE-bench Verified in One Hour on One Machine. Epoch AI engineering blog. Accessed 2026-04-29. <https://epoch.ai/blog/swebench-docker>
- [2] Amazon Web Services. 2024. Reinventing the Amazon Q Developer Agent for Software Development. Public technical announcement. <https://aws.amazon.com/blogs/devops/reinventing-the-amazon-q-developer-agent-for-software-development/>
- [3] Anthropic. 2026. Claude Code Overview. Claude Code Documentation. Accessed 2026-04-28. <https://code.claude.com/docs/en/overview>
- [4] Anthropic. 2026. Tool Use with Claude. Claude API Documentation. Accessed 2026-04-29. <https://platform.claude.com/docs/en/agents-and-tools/tool-use/overview>
- [5] Yoav Benjamini and Yoel Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B (Methodological)* 57, 1 (1995), 289–300. doi:10.1111/j.2517-6161.1995.tb02031.x
- [6] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT5: Code Generation with Generated Tests. International Conference on Learning Representations (ICLR). <https://openreview.net/forum?id=ktrw68Cmu9c>
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebggen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374. <https://arxiv.org/abs/2107.03374>
- [8] Charles J. Clopper and Egon S. Pearson. 1934. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika* 26, 4 (1934), 404–413. doi:10.1093/biomet/26.4.404
- [9] mini-swe-agent maintainers. 2026. mini-swe-agent: A Minimal Agent Runtime for SWE Tasks. Vendored in this repository under `pip_modules/mini-swe-agent/`; upstream is the open-source mini-swe-agent project from the SWE-agent authors. Accessed 2026-04-28. <https://github.com/SWE-agent/mini-swe-agent>
- [10] Paul M. Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, Boston, MA, USA. <https://www.informit.com/store/continuous-integration-improving-software-quality-and-9780321336385>
- [11] Martin Fowler and Matthew Foemmel. 2006. Continuous Integration. MartinFowler.com article. Original article published 2000; updated 2006. <https://martinfowler.com/articles/continuousIntegration.html>
- [12] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. doi:10.1109/TSE.2017.2755013
- [13] GitHub. 2025. About GitHub Copilot Cloud Agent. GitHub Documentation. Accessed 2026-04-27. <https://docs.github.com/copilot/concepts/agents/coding-agent/about-coding-agent>
- [14] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. Advances in Neural Information Processing Systems, Datasets and Benchmarks Track. <https://arxiv.org/abs/2105.09938>
- [15] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 426–437. doi:10.1145/2970276.2970358
- [16] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70. <https://www.jstor.org/stable/4615733>
- [17] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. arXiv preprint arXiv:2504.07164. <https://arxiv.org/abs/2504.07164>
- [18] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World Github Issues? The Twelfth International Conference on Learning Representations (ICLR). <https://openreview.net/forum?id=VTF8yNQm66>
- [19] Sameer Komoravolu and Khalil Mrini. 2026. Agent-testing agent: A meta-agent for automated testing and evaluation of conversational AI agents. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Rabat, Morocco, 7199–7214. doi:10.18653/v1/2026.eacl-long.339
- [20] Kathy Korevec. 2025. Build with Jules, Your Asynchronous Coding Agent. Google public product announcement. <https://blog.google/innovation-and-ai/models-and-research/google-labs/jules/>
- [21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165
- [22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104
- [23] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158
- [24] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 298–312. doi:10.1145/2837614.2837617
- [25] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Karl Moritz Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. Advances in Neural Information Processing Systems. <https://arxiv.org/abs/2303.17651>
- [26] Quinn McNemar. 1947. Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages. *Psychometrika* 12, 2 (1947), 153–157. doi:10.1007/BF02295996
- [27] Mistral AI. 2026. Devstral-Small-2-24B-Instruct-2512. Hugging Face model card. Accessed 2026-04-29. <https://huggingface.co/mistralai/Devstral-Small-2-24B-Instruct-2512>
- [28] NVIDIA. 2026. Llama-3.3 Nemotron Super 49B v1.5 and Llama-3.1 Nemotron Nano 8B v1. Hugging Face model cards. Accessed 2026-04-29.

- Primary URL is the Super model card; Nano model card: <https://huggingface.co/nvidia/Llama-3.1-Nemotron-Nano-8B-v1>. https://huggingface.co/nvidia/Llama-3.3-Nemotron-Super-49B-v1_5
- [29] OpenAI. 2024. Introducing SWE-bench Verified. Technical announcement. <https://openai.com/index/introducing-swe-bench-verified/>
 - [30] OpenAI. 2025. Introducing Codex. Public product announcement. <https://openai.com/index/introducing-codex/>
 - [31] OpenAI. 2026. Function Calling. OpenAI API Documentation. Accessed 2026-04-29. <https://developers.openai.com/api/docs/guides/function-calling>
 - [32] Qwen Team. 2026. Qwen3.5–27B. Hugging Face model card. Accessed 2026-04-29. Used as the backbone LLM for the headline run reported in this paper; the public artifact records the model identifier, run manifest, and serving provenance, while model weights remain external licensed dependencies. <https://huggingface.co/Qwen/Qwen3.5-27B>
 - [33] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., New Orleans, LA, USA, 68539–68551. https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html
 - [34] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *Advances in Neural Information Processing Systems*. <https://openreview.net/forum?id=vAElhFcKW6>
 - [35] Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro. 2025. Nemotron-CORTEXA: Enhancing LLM Agents for Software Engineering Tasks via Improved Localization and Solution Diversity. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, Vancouver, Canada, 56085–56100. <https://proceedings.mlr.press/v267/sohrabizadeh25a.html>
 - [36] SWE-bench Maintainers. 2026. sb-cli: Command-Line Interface for SWE-bench Cloud Evaluation. SWE-bench Documentation. Accessed 2026-04-06. <https://www.swebench.com/sb-cli/>
 - [37] Xingyao Wang et al. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv preprint arXiv:2407.16741. <https://arxiv.org/abs/2407.16741>
 - [38] Edwin B. Wilson. 1927. Probable Inference, the Law of Succession, and Statistical Inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212. doi:10.1080/01621459.1927.10502953
 - [39] Scott Wu. 2024. Introducing Devin, the First AI Software Engineer. Cognition public product announcement. <https://www.cognition.ai/blog/introducing-devin>
 - [40] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv preprint arXiv:2407.01489. <https://arxiv.org/abs/2407.01489>
 - [41] Weiyuan Xu, Juntao Luo, Tao Huang, Kaixin Sui, Jie Geng, Qijun Ma, Isami Akasaka, Xiaoxue Shi, Jing Tang, and Peng Cai. 2025. LogSage: An LLM-Based Framework for CI/CD Failure Detection and Remediation with Industrial Validation. *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, Industry Showcase Track (ASE Industry)*. <https://arxiv.org/abs/2506.03691>
 - [42] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html
 - [43] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *Advances in Neural Information Processing Systems*. <https://arxiv.org/abs/2305.10601>
 - [44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2210.03629>
 - [45] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*. Lecture Notes in Computer Science, Vol. 2862. Springer, Berlin, Heidelberg, 44–60. doi:10.1007/10968987_3
 - [46] Daoguang Zan et al. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. arXiv preprint arXiv:2504.02605. <https://arxiv.org/abs/2504.02605>
 - [47] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.1145/3650212.3680384
 - [48] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, Bangkok, Thailand, 12834–12859. doi:10.18653/v1/2024.findings-acl.762

Received 2026-05-01; accepted 2026-07-01