BEHAVIORDISTILL: DISTILLING BLACK-BOX SOFTWARE VIA EVOLVING, VERIFIABLE SPECIFICATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Black-box software replication builds a student program from scratch to match a teacher's behavior using documentation and execution observations without source access. Recent methods capture observed behavior in specifications to guide construction, but natural-language specifications cannot be executed to verify consistency or locate discrepancies. Tests enable such verification, yet handcrafted generation rules restrict exploration and risk biased, incomplete coverage, while unstructured suites obscure behavioral relations and coverage gaps. We introduce BEHAVIORDISTILL, a method connecting exploration and construction through evolving, verifiable behavioral specifications. Evolving Class-Structured Coverage (ECC) synthesizes these specifications by linking executable tests to an evolving class-structured behavior model, where high-level classes guide feature discovery, fine-grained classes guide deeper exploration, and new tests refine the model. Building on these specifications, Plan-Guided Coverage-Expanding Construction (PCC) uses dynamic plans to organize test constraints into staged implementation and repair tasks, while construction feedback drives test expansion and revisions to plans and code. Across multiple benchmarks and models, BEHAVIORDISTILL achieves Raw Pass Rate and Almost Resolved gains reaching 15.8 and 18.2 percentage points, respectively, illuminating a path toward autonomous software reconstruction through evolving, verifiable behavioral specifications.

1 INTRODUCTION

Agents based on large language models (LLMs) now autonomously handle complex software engineering tasks in existing codebases (Yang et al., 2024; Wang et al., 2025). Their growing use in building complete software from scratch raises broader challenges in behavioral understanding, software design, and implementation (Yang et al., 2026). Black-box software replication brings these challenges together: agents must build a student program from scratch that matches a teacher program's behavior using only documentation and execution observations, without source code access (Yang et al., 2026; Adamczewski et al., 2026). Since documentation may leave boundary cases, state changes, and feature interactions unspecified, agents must actively explore the teacher to discover these behaviors and reproduce them in the student.

Limitations of Existing Approaches. Existing approaches face two main limitations. First, *reliance on human priors.* Veldt extends a supplied implementation scaffold (hdresearch, 2026), while ReBuilder and related methods rely on manually designed workflows (zbl1998-sdjn, 2026; Zhao et al., 2025a; Wang et al., 2026; Liu et al., 2025), limiting their flexibility in adapting to different task requirements (Zhang et al., 2025). Second, *limited specification verifiability and coverage.* Recent methods capture knowledge from exploration in behavioral specifications to guide construction. SpecFirst derives natural-language specifications through black-box probing (Chen et al., 2026), but these specifications cannot be executed to verify behavioral consistency or locate discrepancies. Executable tests enable such verification, and broader test coverage of features, boundary cases, and feature interactions helps detect more errors and improve behavioral consistency. MirrorCode provides handcrafted visible tests for each task (Adamczewski et al., 2026); Gongent probes the teacher to generate tests using predefined dimensions and combination rules (qucheng-gong, 2026), which limit exploration and risk coverage gaps, bias, and student overfitting.

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.

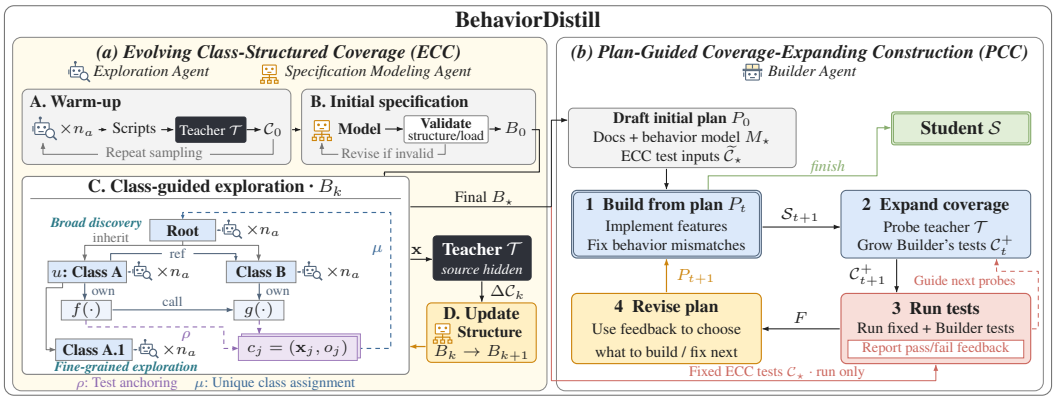

Figure 1: Overview of BEHAVIORDISTILL. BEHAVIORDISTILL combines behavioral specification discovery with specification-guided construction for black-box software replication. (a) ECC coordinates broad functionality discovery and fine-grained behavioral exploration through an evolving class structure, organizing observations into a structured, verifiable behavioral specification. New tests refine the structure, which in turn guides further exploration. (b) PCC uses the resulting specification to guide construction through a dynamic plan, validating the implementation with fixed ECC tests and tests added during construction. Implementation state and test feedback drive targeted probing, test augmentation, and revisions to both the plan and implementation, extending coverage throughout construction.

Challenges and Design Requirements. These limitations raise two challenges. First, *how can tests better cover the teacher's behavior space?* Software behavior has a fractal-like hierarchy: subspaces can be progressively subdivided by input type, boundary case, state change, and feature interaction. Coverage thus depends on granularity and budget: finite tests cannot exhaust all behavior, so exploration must balance broad feature discovery with fine-grained analysis. Yet this balance requires understanding behavioral subspaces and their relations, which test inputs and outputs alone do not reveal. A behavior model is therefore needed to organize evidence across types and granularities, locate coverage gaps, and guide exploration. Models such as state machines (Aarts et al., 2012) and input classification trees (Grochtmann & Grimm, 1993) offer limited support for this unified organization, as they focus on specific behavioral dimensions and often rely on manual abstraction. This calls for a structured yet flexible behavior model that organizes tests and balances exploration breadth and depth without handcrafted exploration rules. Second, *how can tests guide replication more effectively?* A Builder Agent struggles to harness large test suites at once, while pregenerated tests may still leave coverage gaps and bias. Effective construction therefore requires satisfying existing test constraints while continually generating new tests from feedback.

Our Approach. To this end, we propose BEHAVIORDISTILL, a black-box software replication method with two stages: Evolving Class-Structured Coverage (ECC) and Plan-Guided Coverage-Expanding Construction (PCC), as illustrated in Figure 1. ECC expands behavioral coverage by organizing verifiable tests in an evolving class-structured behavior model and balancing exploration breadth and depth. Building on the ECC model and its tests, PCC follows a dynamic plan, adds tests through active exploration, and uses test feedback to refine the plan and implementation.

During ECC, BEHAVIORDISTILL organizes tests from black-box exploration in an evolving class-structured behavior model to guide further exploration, while linked tests verify student behavior. Exploration Agents first query the teacher to generate executable tests, from which BEHAVIORDISTILL synthesizes a code skeleton with class and function declarations but no function bodies. Within this skeleton, classes represent behavior subspaces, and the structure forms a typed multirelational graph combining state machines, input classification trees, and dependency graphs while preserving their objects, operations, and relations. Each test belongs to exactly one class but may link to multiple classes or functions, anchoring concrete behavior and helping locate coverage gaps. The model, tests, links, and assignments together form a behavioral specification. Guided by this specification, Exploration Agents work in parallel across class levels: high-level classes guide major feature discovery, while fine-grained classes guide deeper exploration of input boundaries, state changes, and

feature interactions. New tests, in turn, drive updates to classes, functions, and relations, allowing modeling and test generation to reinforce each other and improve coverage.

Building on this specification, PCC guides iterative replication with dynamic plans, a fixed test suite from ECC, and an expanding self-generated suite. The fixed suite, including expected outputs, is stored in a validation environment and can only be executed for feedback. The Builder Agent drafts an initial natural-language plan using the teacher, class-structured model, and the fixed suite with expected outputs removed. It then (1) implements or refines the student according to the plan; (2) probes the teacher using the implementation and test feedback, adding tests that expose discrepancies and their expected outputs to the self-generated suite; (3) runs both suites for feedback; and (4) revises implementation and repair tasks in the plan, repeating this cycle until it chooses to finish. Dynamic plans thus organize test constraints into staged tasks, while construction feedback guides test generation to reduce coverage gaps and bias.

Across benchmarks and models, BEHAVIORDISTILL uses black-box exploration alone to surpass human-revised official tests in source coverage and approach the coverage of source-assisted official tests, with Raw Pass Rate and Almost Resolved gains over Vanilla reaching 15.8 and 18.2 percentage points, respectively (Sec. 3.2). Scaling experiments, ablations, and case studies further show that dynamic planning and feedback-driven test expansion close behavioral gaps, enabling more complete replication in less construction time (Secs. 3.3–3.4, Appendix B.4). These results open a path to automatically acquiring verifiable behavioral specifications and reconstructing software without source access or complete initial specifications. Our main contributions are as follows.

1. We introduce BEHAVIORDISTILL, a method that bridges black-box exploration and software construction through evolving, verifiable behavioral specifications, unifying the discovery, organization, and use of behavioral knowledge.
2. ECC jointly evolves class structures and tests to balance exploration breadth and depth, while PCC uses dynamic plans to couple test expansion with implementation repair, turning behavioral discoveries into code improvements.
3. Experiments demonstrate Raw Pass Rate and Almost Resolved gains reaching 15.8 and 18.2 percentage points, respectively, and coverage beyond source-generated official tests under extended exploration, illuminating a path toward autonomous software reconstruction through evolving, verifiable behavioral specifications.

2 METHOD

BEHAVIORDISTILL constructs a student replica of a black-box teacher program through the two-stage workflow shown in Figure 1. ECC organizes tests in an evolving class-structured model, combining hierarchical exploration with structural updates to expand coverage. Building on the ECC model and its linked tests, PCC uses dynamic plans to cycle through implementation, test generation, execution-based validation, and plan revision based on feedback to improve the student's behavioral consistency.

2.1 TEACHER SOFTWARE

Let $\mathcal{T}$ denote the black-box teacher, I its executable input-script space, and O its observable output space. Each script contains one or more operations and runs in isolation from a common initial state. Agents observe behavior only through execution, without access to internal state or the execution function. For a script $\mathbf{x} \in I$, the observed output is $\mathrm{exec}_{\mathcal{T}}(\mathbf{x}) = o \in O$. The teacher's behavior is $\mathrm{Beh}(\mathcal{T}) = \big\{(\mathbf{x}, \mathrm{exec}_{\mathcal{T}}(\mathbf{x})) \mid \mathbf{x} \in I\big\}$. The student $\mathcal{S}$ shares the teacher's input and output interfaces. The ideal replication objective is $\mathrm{Beh}(\mathcal{S}) = \mathrm{Beh}(\mathcal{T})$.

2.2 EVOLVING CLASS-STRUCTURED COVERAGE (ECC)

2.2.1 CLASS-STRUCTURED BEHAVIOR MODELS AND BEHAVIORAL SPECIFICATIONS

In this setting, we study behavior observable through finite scripts with deterministic outcomes in isolation, as in MirrorCode and ProgramBench. We record N executions in the test set $\mathcal{C} = \{c_j\}_{j=1}^{N}$, where $c_j = (\mathbf{x}_j, o_j)$ and $o_j = \mathrm{exec}_{\mathcal{T}}(\mathbf{x}_j)$. These tests check whether $\mathcal{S}$ reproduces observed behavior but reveal little about subspaces and their relations within the vast behavior space. We therefore

organize $\mathcal{C}$ in a behavior model to locate coverage gaps and guide exploration. Conventional formal modeling, however, typically requires manual choices of model forms and abstraction rules. For example, state machines focus on state transitions, whereas input classification trees partition inputs; neither readily captures diverse behavior structures. Abstraction may also omit objects, operations, or relations, while the lack of a common representation hinders composition and refinement. A unified representation must therefore preserve these elements and support combinations of different behavior structures.

A typed multirelational graph provides this representation: nodes represent behavioral objects and operations, and typed edges represent semantic relations. Connecting and extending local structures supports composition and refinement. Object-oriented class structures encode this graph as a code skeleton: class and function declarations form nodes, while inheritance, reference, call, and ownership relations form typed edges. This skeleton combines structures abstracted from finite execution observations while preserving their objects, operations, and relations (see supplementary Section 2 for lossless encoding and composition).

Formally, this skeleton encodes a Class-Structured Behavior Model $M = (V, E, \tau, \ell)$. Its node set is $V = V_c \cup V_f$, with $V_c \cap V_f = \varnothing$, where V_c and V_f contain class and function-declaration nodes. The directed edge set E has type map $\tau : E \to \{\mathsf{inherit}, \mathsf{ref}, \mathsf{call}, \mathsf{own}\}$. The node-attribute map $\ell : V \to \mathcal{L}$ records the declaration, name, and semantic role of node v in $\ell(v)$. For an edge $e \in E$, the four values of $\tau(e)$ denote inheritance, reference, call, and ownership, respectively. Inheritance edges connect classes, call edges connect functions, ownership edges connect classes to their functions, and reference edges represent dependencies between classes or functions. Each class represents a behavior subspace, and its functions represent operations within it. Inheritance edges must form a tree on V_c rooted at r, ensuring a unique class hierarchy. The parent map $\pi : \{v \in V_c \mid v \neq r\} \to V_c$ assigns each non-root class v its unique immediate parent $\pi(v)$. Inheritance edges in M uniquely determine this map; reference, call, and other edges represent the remaining class relations.

The class hierarchy explicitly decomposes behavior into units to organize tests $\mathcal{C}$, locate coverage gaps, guide exploration, and allocate tasks. For $\mathcal{C}$, define an anchoring map $\rho : V \to 2^{\mathcal{C}}$ and an assignment map $\mu : \mathcal{C} \to V_c$. Here, $\rho(v)$ contains tests relevant to node v; each test may link to multiple classes or functions. The value $\mu(c)$ is the unique class to which test c is assigned, with $c \in \rho(\mu(c))$. Each test thus belongs to one class but may link to other classes or functions through ρ. The Behavioral Specification $B = (M, \mathcal{C}, \rho, \mu)$ combines the model, tests, and their anchoring and assignment maps.

2.2.2 Behavior Coverage across Breadth and Depth

Software behavior has a fractal-like hierarchy: subspaces can be progressively subdivided by input type, boundary case, state change, and feature interaction. Coverage, like coastline length, depends on scale: finer exploration distinguishes more behaviors and requires more tests. Under a limited budget, favoring breadth risks missing fine-grained behavior, while favoring depth risks missing major features. To balance both, ECC explores classes across levels in parallel: high-level classes guide major feature discovery, while fine-grained classes guide deeper exploration of boundaries, states, and feature interactions. New tests, in turn, refine the class structure (Figure 1(a)). Parts A–E below detail the ECC steps: initial sampling, initial specification construction, hierarchical exploration, structural evolution, and termination.

A. Warm-up: Initial Sampling without Priors. Before building the class-structured model, ECC collects an initial test suite through black-box exploration to support structural synthesis. It runs n_a Exploration Agents in parallel, denoting agent i by $\mathrm{Explore}_i$. In warm-up round h, $\mathrm{Explore}_i$ uses the current suite $\mathcal{C}^{(h)}$ and black-box access to teacher $\mathcal{T}$ to generate candidate scripts $\mathcal{X}_h^i$: $\mathcal{X}_h^i = \mathrm{Explore}_i(\mathcal{C}^{(h)}, \mathcal{T}),\ 1 \le i \le n_a$.

Each Exploration Agent independently generates candidates from the current suite. BehaviorDistill merges and deduplicates them, then runs them on the teacher $\mathcal{T}$. A script yields a test, as defined in Section 2.2.1, if it finishes within the time limit and its outputs and exception information

are fully captured. Round h produces the test set

$$\Delta\mathcal{C}^{(h)} = \left\{ (\mathbf{x}, o) \;\middle|\; \mathbf{x} \in \bigcup_{i=1}^{n_a} \mathcal{X}_h^i,\ o = \mathrm{exec}_{\mathcal{T}}(\mathbf{x}) \right\},$$

where $\Delta\mathcal{C}^{(h)}$ excludes scripts that do not finish or yield incomplete observations. The suite is updated as $\mathcal{C}^{(h+1)} = \mathcal{C}^{(h)} \cup \Delta\mathcal{C}^{(h)},\ \mathcal{C}^{(0)} = \varnothing$. Warm-up continues until the target of N_0 tests is reached, i.e., $|\mathcal{C}^{(h)}| \geq N_0$. The resulting suite $\mathcal{C}_0$ is used to synthesize the initial class structure.

B. Initial Behavioral Specification Construction. The warm-up suite $\mathcal{C}_0$ is unstructured. A Specification Modeling Agent builds a class-structured model and establishes test links and assignments. From the empty specification $B_{-1} = \varnothing$ and suite $\mathcal{C}_0$, it produces the initial specification $B_0 = \mathrm{Structure}(B_{-1}, \mathcal{C}_0) = (M_0, \mathcal{C}_0, \rho_0, \mu_0)$.

Components follow Section 2.2.1, with subscript 0 denoting the initial specification. BEHAVIORDISTILL checks structural validity and class loads. The default per-class limit N_c bounds the combined count of assigned tests and direct subclasses. For specification $B_k = (M_k, \mathcal{C}_k, \rho_k, \mu_k)$ at iteration k, let $V_{c,k}$ be the class-node set and π_k the parent map induced by inheritance edges in M_k. Class $u \in V_{c,k}$ has direct subclasses $\mathrm{child}_k(u) = \{v \in V_{c,k} \mid \pi_k(v) = u\}$ and load $\mathrm{load}_k(u) = |\{c \in \mathcal{C}_k \mid \mu_k(c) = u\}| + |\mathrm{child}_k(u)|$. The first term counts tests uniquely assigned to u; the second counts direct subclasses. Tests linked to u through ρ_k but not assigned to u do not contribute to the load. Validation of B_0 requires a complete, type-consistent, parseable code skeleton; linked tests for every class and function; a unique assignment for every test; and $\mathrm{load}_0(u) \leq N_c$ for all classes, except for the internal nodes permitted in D. If B_0 fails validation, the Specification Modeling Agent uses the feedback to revise M_0, ρ_0, and μ_0 while preserving $\mathcal{C}_0$.

C. Hierarchical Exploration Guided by Coverage Gaps. From the initial specification, BEHAVIORDISTILL uses the class structure as a coverage map to explore classes at different levels in parallel. Let $d_k(u)$ be the depth of class u at iteration k, and L the maximum exploration depth. Depths, ancestors, and subclasses follow the inheritance tree defined by π_k. Each class with $d_k(u) < L$ receives n_a Exploration Agents. The exploration context for class u is $\Gamma_k(u) = \big(M_k[u], \rho_k(u), \mu_k^{-1}(u)\big)$, where $M_k[u]$ includes class u, its ancestors, subclasses, and relevant interfaces; $\rho_k(u)$ contains linked tests; and $\mu_k^{-1}(u)$ contains tests uniquely assigned to class u. Agent i generates candidate scripts $\mathcal{X}_{k,u}^i = \mathrm{Explore}_i(\Gamma_k(u), \mathcal{T})$.

High-level classes target undiscovered major features; fine-grained classes target input boundaries, state changes, and feature interactions. Candidates from all classes with $d_k(u) < L$ are merged, deduplicated, and converted into tests as in A. The resulting test set $\Delta\mathcal{C}_k$ is passed to D to update the model and test links, producing a new specification. Subscript u identifies the exploration source; μ_{k+1} in D determines the final test assignment.

D. Evidence-Driven Structural Evolution. The Specification Modeling Agent uses the new tests $\Delta\mathcal{C}_k$ from C as additional execution evidence to update the specification: $B_{k+1} = \mathrm{Structure}(B_k, \Delta\mathcal{C}_k) = (M_{k+1}, \mathcal{C}_{k+1}, \rho_{k+1}, \mu_{k+1})$, where $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \Delta\mathcal{C}_k$. New tests link to relevant classes and functions through ρ_{k+1} and are assigned to exactly one class through μ_{k+1}. Inheritance edges must still satisfy the rooted-tree constraint in Section 2.2.1. For test c, let $S_{k+1}(c) = \{u \in V_{c,k+1} \mid c \in \rho_{k+1}(u)\}$ be its linked class set. Its unique assignment is the lowest common ancestor (LCA) of these classes in the inheritance tree: $\mu_{k+1}(c) = \mathrm{LCA}_{\pi_{k+1}}(S_{k+1}(c))$. Thus, a test linked to one class is assigned to that class; a test linked to multiple classes is assigned to their LCA.

After each update, compute class u's load $\mathrm{load}_{k+1}(u)$ as in B. If $\mathrm{load}_{k+1}(u) > N_c$, the Specification Modeling Agent splits class u by behavioral differences in its tests. Tests for one behavior group move to a suitable existing or newly created subclass; tests spanning groups stay in the parent. Direct subclasses are regrouped if their count exceeds N_c. Splitting continues until all classes either meet the load limit or qualify for a recorded exception: an internal node may exceed N_c only because cross-subclass tests must remain at their LCA. Such nodes are not split again unless their structure or evidence changes. Classes, functions, and relations may change, but existing test inputs and outputs remain unchanged.

E. Iteration and Termination. After initialization, C and D alternate until $|\mathcal{C}_k| \geq N_{\max}$ or all leaf classes reach depth L. With equal batch budgets and matched sampling assumptions, fixed per-class

exploration achieves at least the expected coverage of random exploration. Structural evolution preserves existing constraints when tests and their execution semantics remain unchanged, while added tests further constrain candidate implementations (see supplementary Sections 1–2).

2.3 PLAN-GUIDED COVERAGE-EXPANDING CONSTRUCTION (PCC)

The final specification produced by ECC, $B_\star = (M_\star, \mathcal{C}_\star, \rho_\star, \mu_\star)$, guides iterative student construction in PCC. The Builder Agent can access task documentation, teacher $\mathcal{T}$, model $M_\star$, and the test suite $\widetilde{\mathcal{C}}_\star$, which retains input scripts and operations from $\mathcal{C}_\star$ but omits expected outputs and associated assertions. The fixed suite $\mathcal{C}_\star$, including expected outputs, remains in a validation environment and can only be executed for feedback. The agent also maintains a fully inspectable self-generated suite $\mathcal{C}_t^+$ with expected outputs from teacher executions, initialized as $\mathcal{C}_0^+ = \varnothing$.

Using task documentation, $M_\star$, and $\widetilde{\mathcal{C}}_\star$, the agent drafts an initial natural-language plan P_0 specifying the implementation language, module decomposition, and implementation order. At iteration t, the state $(\mathcal{S}_t, P_t, \mathcal{C}_t^+, F_t)$ records the student software, dynamic plan, self-generated suite, and test feedback. As shown in Figure 1(b), each iteration (1) builds or repairs $\mathcal{S}_t$ using P_t, $M_\star$, $\widetilde{\mathcal{C}}_\star$, and $\mathcal{C}_t^+$, yielding $\mathcal{S}_{t+1}$; (2) probes $\mathcal{T}$ based on the implementation, plan, and feedback to generate new tests $\Delta\mathcal{C}_t^+$ that expose discrepancies, updating $\mathcal{C}_{t+1}^+ = \mathcal{C}_t^+ \cup \Delta\mathcal{C}_t^+$; (3) runs $\mathcal{C}_\star$ and $\mathcal{C}_{t+1}^+$ for per-test outcomes and failure feedback; and (4) revises implementation order and adds repair tasks based on the feedback, producing P_{t+1}. PCC ends when the agent outputs `finish`, returning the current student software.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate BEHAVIORDISTILL on ProgramBench and MirrorCode (Yang et al., 2026; Adamczewski et al., 2026), replicating teacher behavior from scratch using only documentation and black-box execution observations. **Metrics.** (1) Test coverage: mean teacher first-party line coverage across tasks, following ProgramBench. (2) Replication: Raw Pass Rate (mean per-task pass rate) and Almost Resolved (fraction of tasks passing $\geq 95\%$), evaluated on official hidden tests. **Baselines.** (1) Coverage: Gongent's tests (qucheng-gong, 2026), ProgramBench's official hidden tests generated using source code, and MirrorCode's full official test suite (visible and hidden), drawn from mixed sources and revised by humans. (2) Replication: Vanilla (direct implementation without generated specifications), Vanilla + Official Tests (guided by official visible tests; MirrorCode only), SpecFirst (Chen et al., 2026), and Gongent. **Settings.** All methods use `GPT-5.6 Luna` and `DeepSeek-V4-Flash-0731` with high reasoning effort and separate six-hour limits for specification generation (if applicable) and construction. **Implementation.** Vanilla and all BEHAVIORDISTILL agents use mini-SWE-agent. ECC uses $N_0 = 80$ initial tests, a default per-class load limit of $N_c = 80$, a stopping threshold of $N_{\max} = 300$ tests, and a maximum depth of $L = 4$; PCC tests are counted separately. See Appendix C for details.

3.2 MAIN RESULTS

BEHAVIORDISTILL achieves broader source-code coverage than black-box baselines and human-revised tests, approaching that of tests generated using source code. As shown in Table 1, with `GPT-5.6 Luna`/`DeepSeek-V4-Flash-0731`, BEHAVIORDISTILL improves coverage over Gongent by 25.5/17.4 percentage points on MirrorCode and 38.4/37.4 on ProgramBench. On MirrorCode, coverage reaches 51.1%/49.2%, exceeding the 47.1% of the human-revised full official suite (visible and hidden). On ProgramBench, black-box exploration achieves up to 71.3%, approaching the 76.0% of official tests generated using source code. These results support structured exploration and continued test generation: whereas Gongent generates a fixed suite from predefined categories and templates, ECC evolves its class structure to broaden feature discovery and probe behavioral edge cases. PCC further adds targeted tests using implementation state and test feedback, extending coverage throughout exploration and construction.

Table 1: Source-code coverage and downstream replication performance on ProgramBench and MirrorCode with two models. Cov. denotes task-averaged first-party line coverage of the teacher; Raw (Raw Pass Rate) and AR (Almost Resolved) denote the student's task-averaged hidden-test pass rate and the percentage of tasks passing at least 95% of hidden tests, respectively. BEHAVIORDISTILL denotes the full ECC + PCC pipeline, with coverage measured over tests from both stages. In the Official Tests rows, coverage is model-independent: ProgramBench uses source-assisted hidden tests as a coverage reference, while MirrorCode uses all official tests (visible and hidden). MirrorCode Raw/AR results in these rows come from Vanilla + Official Tests, which receives only visible tests and is evaluated on hidden tests. All values are percentages (higher is better); bold marks the best non-official results within each model, leaving all-zero columns unmarked, and "–" denotes not applicable.

Method	ProgramBench												MirrorCode											
	Overall			Easy			Medium			Hard			**Overall**			Small			Medium			Large		
	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR
`GPT-5.6 Luna`																								
Official Tests	76.0	–	–	88.7	–	–	76.8	–	–	59.8	–	–	47.1	60.1	40.9	46.2	96.9	77.8	48.5	44.6	20.0	45.4	1.45	0
Vanilla	–	51.4	0.50	–	68.7	3.57	–	52.7	0	–	28.1	0	–	45.5	13.6	–	84.3	33.3	–	23.8	0	–	1.08	0
SpecFirst	–	59.0	3.00	–	74.4	10.7	–	61.2	2.10	–	33.6	0	–	46.9	18.2	–	89.4	44.4	–	22.6	0	–	0	0
Gongent	25.7	56.0	0.50	38.2	71.7	3.57	24.7	57.7	0	18.2	32.6	0	25.6	45.0	4.55	36.2	84.7	11.1	19.1	22.8	0	15.2	0	0
BEHAVIORDISTILL	**64.1**	**67.2**	**9.50**	**85.3**	**79.4**	**21.4**	**64.8**	**69.4**	**9.09**	**39.6**	**44.6**	0	**51.1**	**60.8**	**31.8**	**52.3**	**96.2**	**77.8**	**52.3**	**45.8**	0	**43.4**	**4.36**	0
`DeepSeek-V4-Flash-0731`																								
Official Tests	76.0	–	–	88.7	–	–	76.8	–	–	59.8	–	–	47.1	77.7	59.1	46.2	98.9	88.9	48.5	76.4	50.0	45.4	18.0	0
Vanilla	–	69.3	18.5	–	85.8	32.1	–	71.0	19.6	–	45.3	0	–	59.3	31.8	–	96.9	77.8	–	40.0	0	–	11.0	0
SpecFirst	–	71.5	25.0	–	89.2	46.4	–	73.2	25.9	–	45.9	0	–	61.1	36.4	–	97.6	77.8	–	41.0	10.0	–	18.7	0
Gongent	33.9	70.6	21.5	51.3	87.7	42.9	33.5	72.3	21.7	18.9	45.9	0	31.8	67.8	45.5	39.5	98.8	**100**	28.7	55.0	10.0	19.3	17.1	0
BEHAVIORDISTILL	**71.3**	**77.5**	**32.0**	**87.8**	**91.2**	**50.0**	**72.9**	**79.0**	**33.6**	**47.4**	**56.7**	**6.90**	**49.2**	**71.1**	**50.0**	**52.7**	**99.0**	**100**	**50.1**	**57.9**	**20.0**	**35.4**	**31.7**	0

BEHAVIORDISTILL improves replication across both benchmarks and models, raising Almost Resolved by 13.5 percentage points over Vanilla on ProgramBench. As shown in Table 1, with `GPT-5.6 Luna`, it improves Raw Pass Rate by 15.8 percentage points on ProgramBench and Almost Resolved by 18.2 points on MirrorCode. On ProgramBench with `DeepSeek-V4-Flash-0731`, it improves Almost Resolved over Vanilla, SpecFirst, and Gongent by 13.5, 7.0, and 10.5 percentage points, respectively. MirrorCode's visible tests define the replication scope, while hidden duals evaluate corresponding functionality with different inputs (Adamczewski et al., 2026). These examples give Vanilla + Official Tests explicit guidance on the evaluated behaviors. Despite lacking this scope prior, BEHAVIORDISTILL achieves a Raw Pass Rate comparable to that of Vanilla + Official Tests with `GPT-5.6 Luna` (60.8% versus 60.1%). These results show that BEHAVIORDISTILL effectively uses automatically acquired and continually refined behavioral specifications to improve student–teacher behavioral consistency and help more tasks achieve near-complete replication across benchmarks and models.

3.3 SCALING WITH TEST-SET SIZE AND CONSTRUCTION TIME

ECC matches the source-code coverage of source-assisted official tests through black-box exploration alone. We use `DeepSeek-V4-Flash-0731`, remove ECC's time and depth limits, and set $N_0 = N_c = 200$ and $N_{\max} = 5{,}000$. After generation, we randomly subsample each task's test pool at target sizes doubling from 64 to 4,096, using all tests when fewer are available. As shown in Figure 2 and Table 2, overall coverage on ProgramBench rises from 60.3% to 77.7%, a gain of 17.4 percentage points. Random subsets of 64 tests already achieve higher coverage than Gongent's full suite (33.9%), showing that small samples retain broad coverage and supporting ECC's use of class structure to guide broad and fine-grained exploration. Coverage reaches 75.6% at 1,024 tests, approaching the official suite's 76.0%, and surpasses it at 2,048. From 1,024 to 4,096 tests, Hard gains 4.80 percentage points in coverage, compared with 1.30 for Easy. These results show diminishing overall coverage gains as test-set size grows, while hard tasks continue to benefit more from additional tests at larger scales.

BEHAVIORDISTILL achieves better replication in less construction time, with a clear completeness advantage on hard tasks. To study scaling beyond the standard six-hour budget, we eval-

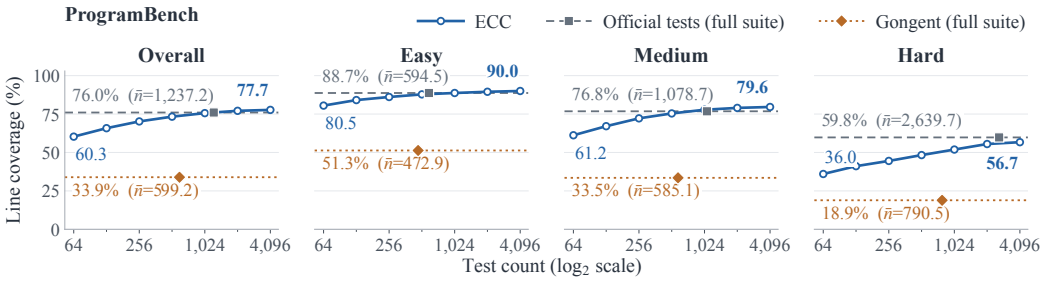

Figure 2: ECC coverage scaling on ProgramBench. Panels show task-averaged teacher first-party line coverage overall and by difficulty. After ECC completes test generation, we randomly sample subsets from each task's final test pool at target sizes doubling from 64 to 4,096. If a task's pool contains fewer tests than the target size, we use the entire pool. Horizontal lines give the full-suite coverage of official tests and Gongent; squares and diamonds mark their mean test counts per task ($\bar{n}$), respectively. The horizontal axis is logarithmic (base 2).

uate construction over 6–48 h on ProgramBench with `DeepSeek-V4-Flash-0731`, keeping the six-hour budget for preliminary exploration unchanged where applicable. For all methods, whenever a Builder Agent finishes early, we preserve its artifacts and launch an identically configured agent to continue within the remaining budget, repeating until the budget is exhausted. As shown in Figure 3 and Table 3, BEHAVIORDISTILL's overall Raw/Almost scores rise from 78.5%/33.5% at 6 h to 82.8%/48.5% at 48 h, leading all baselines at every checkpoint. Its Almost reaches 41.0% at just 12 h, exceeding the best baseline's 40.5% at 48 h. From 6 to 48 h, BEHAVIORDISTILL's Raw improves by 2.4, 3.1, and 12.1 percentage points on Easy, Medium, and Hard, respectively, with hard tasks gaining the most from additional construction time. On Hard, its Raw reaches 67.9% at 24 h, surpassing the best baseline's 61.5% at 48 h; its Almost at 48 h reaches 13.8%, versus the best baseline's 3.45%. On Medium and Hard tasks, Gongent has the lowest final Raw scores and gains the least in Almost Resolved from additional construction time. PCC continually expands tests and uses dynamic plans to turn newly identified behavioral gaps into implementation and repair tasks; its higher scores support this combination for improving replication under extended construction budgets.

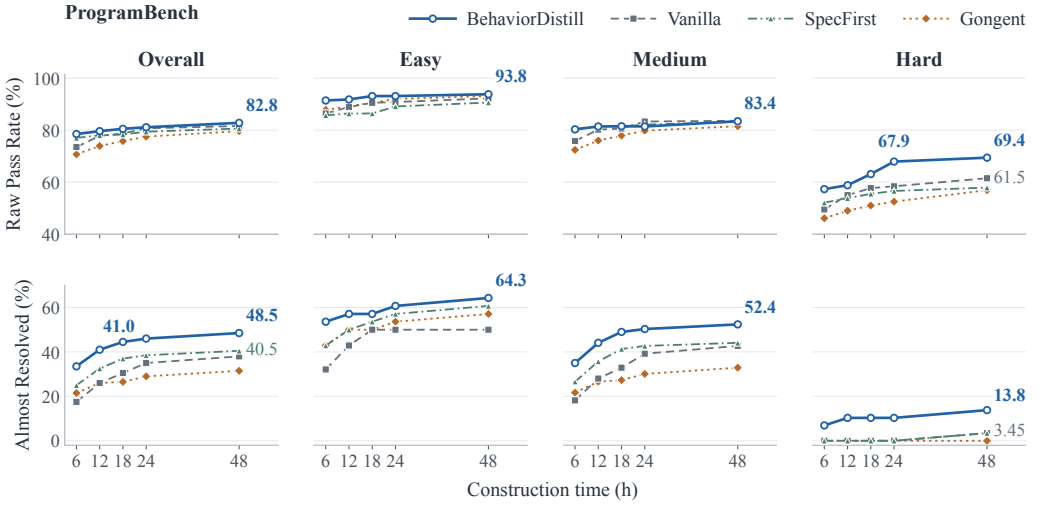

Figure 3: Construction-time scaling on ProgramBench. Rows show Raw Pass Rate and Almost Resolved; columns show overall results and difficulty groups. All methods use `DeepSeek-V4-Flash-0731`. Construction time excludes preliminary specification generation; agents that finish early continue from preserved artifacts.

3.4 Ablation Study

We conduct three ablations on ProgramBench with `DeepSeek-V4-Flash-0731`, using the same reasoning effort and evaluation protocol as the main experiments. Preliminary exploration and specification generation (if applicable) share a six-hour budget; construction receives a separate six hours.

ECC and PCC jointly improve behavioral coverage and replication. We compare Vanilla (construction from documentation and black-box teacher access), ECC + Vanilla (Vanilla guided by ECC specifications), PCC only (planning, test expansion, validation, and repair without ECC), and the full BehaviorDistill pipeline (ECC + PCC). As shown in Table 4, the full pipeline achieves overall Raw Pass Rate/Almost Resolved (AR) of 77.5%/32.0%, versus 69.4%/25.0% for ECC + Vanilla and 71.6%/28.0% for PCC only. Coverage reaches 71.3%, exceeding the two variants by 1.5 and 8.7 percentage points, respectively. The gains of 8.1 points in Raw and 7.0 in AR over ECC + Vanilla reflect the stages' complementary roles: ECC supplies initial behavioral constraints, while PCC uses dynamic plans to organize implementation and repair and expands tests through construction feedback, translating exploration knowledge into better replication.

Class structure improves exploration coverage, with further gains from reference and call relations. We compare Flat, Tree-only, and Full ECC with $N_{\max} = 300$. Flat repeats warmup, using accumulated tests to guide teacher exploration until the threshold is reached, without class modeling, hierarchical exploration, or structural evolution. Tree-only retains these ECC stages but allows only inheritance between classes (`inherit`) and ownership of functions by classes (`own`), omitting reference (`ref`) and call (`call`) relations. Full ECC retains all four relation types. As shown in Table 5, overall coverage is 58.5%, 63.9%, and 69.8%, respectively. Full ECC improves on the two variants by 11.3 and 5.9 percentage points and leads across all difficulty groups. These gains support organizing tests in class structures to guide further exploration; reference and call relations provide additional structural cues for exploring dependencies and interactions beyond the hierarchy.

Explicit planning and test expansion improve replication, with larger losses from removing expansion. During construction after ECC, w/o planning removes both initial plan generation and subsequent updates while retaining test expansion, validation, and repair. The w/o test expansion variant disables new test generation, retaining the fixed ECC suite for validation, plan generation and updates, and implementation repair. Full PCC retains all components and denotes the construction stage of the full ECC + PCC pipeline. As shown in Table 6, removing planning reduces overall Raw/AR to 75.5%/28.0%, down 2.0/4.0 percentage points from Full PCC; removing test expansion reduces them to 72.4%/21.0%, down 5.1/11.0 points. These results highlight PCC's coupling of behavioral discovery and implementation repair: the current implementation guides targeted test generation, and newly exposed discrepancies drive plan revisions and subsequent repairs, revealing behaviors missed by the initial specification and incorporating them into the code. The AR gains from test expansion further suggest that active exploration during construction helps more tasks achieve near-complete replication.

Additionally, we compare methods with a cumulative budget of 200M input and output tokens per task and no time limit. As in construction-time scaling, agents that finish early are restarted from preserved artifacts to continue construction until the budget is exhausted. BehaviorDistill achieves 78.2% Raw Pass Rate and 32.0% Almost Resolved (Table 7), outperforming the strongest baseline, SpecFirst, by 4.9 and 4.5 percentage points, respectively. We further compare BehaviorDistill with the strongest baseline over three runs under the main experimental setting, showing consistent and substantial gains (Table 8).

4 Conclusion

BehaviorDistill reconstructs black-box software through evolving, verifiable behavioral specifications. ECC co-evolves class structures and tests; PCC couples dynamic planning, test expansion, and repair. Across benchmarks and models, gains over Vanilla reach 15.8 and 18.2 percentage points in Raw Pass Rate and Almost Resolved, respectively, illuminating a path toward autonomous software reconstruction.

AI USE STATEMENT

We used generative AI tools to assist with manuscript editing, code debugging, and the organization and formatting of mathematical notation and equations. LLM use within our proposed method is described in Sections 2 and 3. The authors take responsibility for the final manuscript, including its mathematical content, code, and reported results.

REFERENCES

Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In *International Symposium on Formal Methods*, pp. 10–27. Springer, 2012.

Tom Adamczewski, David Owen, David Rein, Florian Brand, Giles Edkins, Allen Hart, and Daniel O'Connell. Mirrorcode: Ai can rebuild entire programs from behavior alone. *arXiv preprint arXiv:2606.30182*, 2026.

Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 317–329, 2007.

Yihao Chen, Shi Chang, Feng Lin, Khaled Chawa, Boyuan Chen, Shaowei Wang, and Ahmed E Hassan. Specfirst: Behavioral specification elicitation as a first-class step in agent-based program synthesis from scratch. *arXiv preprint arXiv:2607.27167*, 2026.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

Alessio Ferrari, Giuseppe Lipari, Stefania Gnesi, and Giorgio O Spagnolo. Pragmatic ambiguity detection in natural language requirements. In *2014 IEEE 1st International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, pp. 1–8. IEEE, 2014.

Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software testing, verification and reliability*, 3(2):63–82, 1993.

hdresearch. veldt. GitHub repository, 2026. URL https://github.com/hdresearch/veldt. Accessed 2026-09-16.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. Prompting large language models to tackle the full software development lifecycle: A case study. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7511–7531, 2025.

Junwei Liu, Chen Xu, Chong Wang, Tong Bai, Weitong Chen, Kaseng Wong, Yiling Lou, and Xin Peng. Towards iterative end-to-end software development: A feature-driven multi-agent framework. *arXiv preprint arXiv:2511.02399*, 2025.

Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pp. 35–46, 2000.

Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl Barr, and Sergey Mechtaev. The fact selection problem in llm-based program repair. *arXiv preprint arXiv:2404.05520*, 2024.

qucheng-gong. Gongent: a Builder/Adversary Agent Loop for Reverse-Engineering Black-Box CLIs. GitHub repository, 2026. URL https://github.com/qucheng-gong/gongent. Accessed 2026-09-16.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yifei Wang, Ruiyin Li, Peng Liang, Qiong Feng, Zengyang Li, Mojtaba Shahin, and Arif Ali Khan. Codeteam: An llm-powered multi-agent framework for repository-level code generation. *arXiv preprint arXiv:2606.22082*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

zbl1998-sdjn. ReBuilder Framework. GitHub repository, 2026. URL https://github.com/zbl1998-sdjn/rebuilder. Accessed 2026-09-16.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. Repozero: Can llms generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026.

Qianhui Zhao, Li Zhang, Fang Liu, Junhang Cheng, Chengru Wu, Junchen Ai, Qiaoyuanhe Meng, Lichen Zhang, Xiaoli Lian, Shubin Song, et al. Towards realistic project-level code generation via multi-agent collaboration and semantic architecture modeling. *ACM Transactions on Software Engineering and Methodology*, 2025a.

Wenting Zhao, Nan Jiang, Celine Lee, Justin Chiu, Claire Cardie, Matthias Gallé, and Alexander Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, volume 2025, pp. 12061–12076, 2025b.

A Related Work

Benchmarks for Repository Generation and Black-Box Software Replication. Commit0 provides a code skeleton, so it does not assess construction from an empty workspace (Zhao et al., 2025b). DevEval evaluates implementation from supplied requirements and designs without teacher exploration (Li et al., 2025), while NL2Repo-Bench starts from an empty workspace but provides only natural-language specifications, with no teacher execution interface for active probing (Ding et al., 2025). RepoZero likewise studies repository generation from scratch, primarily through cross-language reconstruction from API specifications (Zhang et al., 2026). Our focus, however, is to discover teacher behavior through active black-box exploration and use execution observations to guide construction. We therefore evaluate exploration and construction on ProgramBench and MirrorCode, which provide documentation and black-box teacher access, require replication from scratch, and assess behavioral consistency with hidden tests (Yang et al., 2026; Adamczewski et al., 2026). ProgramBench withholds evaluation tests, whereas MirrorCode additionally provides visible tests to define the replication scope.

Agentic Software Construction. SWE-agent and OpenHands iteratively modify existing code through tool use and environmental feedback (Yang et al., 2024; Wang et al., 2025). For full-project generation, ProjectGen organizes architecture and implementation with a semantic software architecture tree (Zhao et al., 2025a), CodeTeam coordinates development through role-based collaboration (Wang et al., 2026), and EvoDev uses a Feature Map to represent feature dependencies and guide iterative construction (Liu et al., 2025). These methods structure long-horizon tasks through stages, roles, or intermediate representations; AFlow further automates workflow search, showing that effectiveness varies across tasks and base models (Zhang et al., 2025). For repository generation from API specifications, RepoZero's ACE repairs implementations using a small set of generated tests and error feedback (Zhang et al., 2026), but limited test coverage may leave key behavioral discrepancies undetected. How agents use information also matters: research on fact selection in program repair shows that adding prompt information does not always improve repair outcomes (Parasaram et al., 2024).

Software Behavior Modeling and Specification Mining. Beyond construction workflows, exploration and verification depend on how behavioral knowledge is represented. Natural-language specifications offer a convenient way to describe requirements (Nuseibeh & Easterbrook, 2000) but can be ambiguous (Ferrari et al., 2014) and cannot themselves be executed. Tests enable verification through concrete input–output pairs, while structural models organize the behavioral relations these tests reflect. Prior work uses automata, dynamic invariants, input classification trees, and protocol message formats to capture specific behavioral structures (Ammons et al., 2002; Ernst et al., 2007; Grochtmann & Grimm, 1993; Caballero et al., 2007). These representations, however, cover different aspects of behavior; some require internal execution information, and model forms and abstraction rules often remain manually specified. Automatic abstraction refinement has been developed for restricted classes of state machines (Aarts et al., 2012). Beyond automation, unifying diverse behavioral structures still requires preserving objects, operations, and relations through abstraction and supporting composition and refinement in a common representation.

Black-Box Software Replication Methods. Veldt extends a supplied implementation scaffold (hdresearch, 2026), while ReBuilder organizes reconstruction into stages using documentation and teacher execution observations (zbl1998-sdjn, 2026). To guide implementation, SpecFirst converts black-box probing results into natural-language specifications (Chen et al., 2026); Gongent instead uses an Adversary Agent to generate tests and guides its Builder Agent in repairing the implementation through feedback from differential testing of the teacher and student (qucheng-gong, 2026). BehaviorDistill's ECC organizes verifiable tests in an evolving class-structured behavior model to locate coverage gaps and coordinate exploration breadth and depth. PCC then uses dynamic plans to organize test constraints into staged implementation and repair tasks, while construction feedback drives further test generation and implementation improvements.

B Additional Experimental Results

B.1 Scaling Experiments

Table 2 reports the numerical results plotted in Figure 2. ECC runs without time or depth limits, with $N_0 = N_c = 200$ and $N_{\max} = 5{,}000$.

Table 2: ECC source-code coverage on ProgramBench at increasing test-set sizes. Values are task-averaged teacher first-party line coverage in percent, overall and by difficulty. After ECC completes test generation, we randomly sample subsets from each task's final test pool at the target sizes shown; if a task has fewer tests than the target size, we use its entire pool. Gongent and Official Tests report coverage for their complete test suites, with the mean number of tests per task in parentheses.

Group	ECC (target tests per task)							Gongent	Official Tests
	64	128	256	512	1,024	2,048	4,096		
Overall	60.3	65.8	70.2	73.3	75.6	77.1	77.7	33.9 (599.2)	76.0 (1,237.2)
Easy	80.5	84.1	86.1	87.8	88.7	89.5	90.0	51.3 (472.9)	88.7 (594.5)
Medium	61.2	67.1	72.2	75.4	77.8	79.0	79.6	33.5 (585.1)	76.8 (1,078.7)
Hard	36.0	41.1	44.5	48.3	51.9	55.5	56.7	18.9 (790.5)	59.8 (2,639.7)

Table 3 reports the numerical results plotted in Figure 3.

Table 3: Replication performance on ProgramBench across construction budgets. All methods use `DeepSeek-V4-Flash-0731`. Raw is the task-averaged hidden-test pass rate; AR (Almost Resolved) is the percentage of tasks passing at least 95% of hidden tests. Time excludes preliminary specification generation, and agents continue from preserved artifacts when they finish early. All scores are percentages (higher is better); bold numbers mark column maxima at each budget, including ties.

Method	Time (h)	**Overall**		**Easy**		**Medium**		**Hard**	
		Raw	AR	Raw	AR	Raw	AR	Raw	AR
Vanilla	6	73.5	17.5	86.4	32.1	75.8	18.2	49.5	0.0
	12	77.8	26.0	88.9	42.9	80.2	28.0	55.0	0.0
	18	78.7	30.5	90.5	50.0	80.6	32.9	57.7	0.0
	24	80.7	35.0	90.8	50.0	**83.3**	39.2	58.4	0.0
	48	81.6	38.0	92.2	50.0	**83.6**	42.7	61.5	3.45
SpecFirst	6	77.0	25.0	85.7	42.9	**80.3**	26.6	52.1	0.0
	12	78.0	32.5	86.4	50.0	81.3	35.7	53.8	0.0
	18	78.3	37.0	86.4	53.6	81.4	41.3	55.5	0.0
	24	79.4	38.5	89.1	57.1	82.1	42.7	56.6	0.0
	48	80.6	40.5	90.6	60.7	83.2	44.1	57.9	3.45
Gongent	6	70.7	21.5	87.8	42.9	72.4	21.7	46.1	0.0
	12	73.9	26.0	89.2	50.0	76.0	26.6	49.0	0.0
	18	75.8	26.5	90.5	50.0	77.9	27.3	51.0	0.0
	24	77.5	29.0	92.0	53.6	79.8	30.1	52.5	0.0
	48	79.6	31.5	93.1	57.1	81.5	32.9	56.9	0.0
BehaviorDistill	6	**78.5**	**33.5**	**91.4**	**53.6**	**80.3**	**35.0**	**57.3**	**6.9**
	12	**79.6**	**41.0**	**91.8**	**57.1**	**81.4**	**44.1**	**58.8**	**10.3**
	18	**80.5**	**44.5**	**93.1**	**57.1**	**81.5**	**49.0**	**63.1**	**10.3**
	24	**81.1**	**46.0**	**93.1**	**60.7**	81.4	**50.3**	**67.9**	**10.3**
	48	**82.8**	**48.5**	**93.8**	**64.3**	83.4	**52.4**	**69.4**	**13.8**

B.2 Ablation Study

We conduct three ablation studies on ProgramBench using `DeepSeek-V4-Flash-0731` with high reasoning effort. Preliminary exploration and specification generation, where applicable, have a combined six-hour budget; construction has a separate six-hour budget that includes test expansion, validation, and plan updates. Evaluation follows the main experiments.

Table 4: ECC and PCC component ablations on ProgramBench. Vanilla uses documentation and black-box teacher access without generated specifications; ECC + Vanilla adds ECC-generated specifications; PCC only skips ECC; and ECC + PCC is the full BehaviorDistill pipeline. Cov. is task-averaged teacher first-party line coverage over each variant's generated tests, including both stages for ECC + PCC. Raw and AR denote the task-averaged hidden-test pass rate and the percentage of tasks passing at least 95% of hidden tests, respectively. All values are percentages (higher is better); bold marks column maxima, and "–" denotes not applicable.

Method	**Overall**			Easy			Medium			Hard		
	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR	Cov.	Raw	AR
Vanilla	–	69.3	18.5	–	85.8	32.1	–	71.0	19.6	–	45.3	0
ECC + Vanilla	69.8	69.4	25.0	84.1	88.6	42.9	71.8	71.3	25.9	46.1	41.8	3.45
PCC only	62.6	71.6	28.0	74.5	90.7	46.4	63.7	73.6	29.4	45.8	43.2	3.45
ECC + PCC	**71.3**	**77.5**	**32.0**	**87.8**	**91.2**	**50.0**	**72.9**	**79.0**	**33.6**	**47.4**	**56.7**	**6.90**

Table 5: ECC structure ablations on ProgramBench, with a stopping threshold of $N_{\max} = 300$ for all variants. Flat repeats warmup until this threshold is reached, using accumulated tests to guide exploration without class modeling. Tree-only retains modeling, hierarchical exploration, and structural evolution but allows only `inherit` and `own` relations, omitting `ref` and `call`. Full ECC uses all four relation types. Values are task-averaged teacher first-party line coverage in percent (higher is better); bold marks column maxima.

ECC structure	**Overall**	Easy	Medium	Hard
Flat	58.5	76.2	59.8	35.3
Tree-only	63.9	78.6	65.7	40.6
Full ECC	**69.8**	**84.1**	**71.8**	**46.1**

Table 6: PCC planning and test-expansion ablations on ProgramBench. Both ablations retain ECC specifications: w/o planning removes initial plan generation and subsequent updates while retaining test expansion; w/o test expansion disables test generation during construction while retaining the fixed ECC suite and dynamic planning. Validation and implementation repair remain enabled in both. Full PCC is the construction stage of the full ECC + PCC pipeline. Raw and AR denote the task-averaged hidden-test pass rate and the percentage of tasks passing at least 95% of hidden tests. All values are percentages (higher is better); bold marks column maxima.

Method	**Overall**		Easy		Medium		Hard	
	Raw	AR	Raw	AR	Raw	AR	Raw	AR
w/o planning	75.5	28.0	83.3	46.4	78.4	30.1	53.5	0.00
w/o test expansion	72.4	21.0	87.1	42.9	73.8	21.0	51.4	0.00
Full PCC	**77.5**	**32.0**	**91.2**	**50.0**	**79.0**	**33.6**	**56.7**	**6.90**

Table 7: Supplementary comparison on ProgramBench with `DeepSeek-V4-Flash-0731`. This experiment uses a maximum combined input and output token budget of 200M per task with no time limit. All methods follow the same continuation protocol as construction-time scaling, with token usage accumulated across restarts until the budget is exhausted. Raw and AR denote Raw Pass Rate and Almost Resolved, respectively. Scores are percentages (higher is better); bold marks the best results.

Method	Max. tokens per task	Raw	AR
Vanilla	200M	71.8	24.0
SpecFirst	200M	73.3	27.5
Gongent	200M	69.7	23.5
BEHAVIORDISTILL	200M	**78.2**	**32.0**

B.3 RESULTS ACROSS MULTIPLE RUNS

We compare BEHAVIORDISTILL and SpecFirst over three runs on ProgramBench with `DeepSeek-V4-Flash-0731`, using the same configuration and evaluation protocol as the main experiments.

Table 8: Overall replication results across three runs under the main experimental setting on ProgramBench with `DeepSeek-V4-Flash-0731`. Values are mean $\pm$ standard deviation, in percent. Raw and AR denote Raw Pass Rate and Almost Resolved, respectively. Bold marks the higher mean for each metric.

Method	Raw (%)	AR (%)
SpecFirst	70.9 ± 2.63	25.2 ± 3.25
BEHAVIORDISTILL	$\mathbf{77.8 \pm 2.87}$	$\mathbf{32.2 \pm 2.75}$

B.4 CASE STUDIES

These cases illustrate how BEHAVIORDISTILL connects behavioral discovery with implementation refinement. In DuckDB, ECC organizes broad capabilities into finer behavioral subspaces, such as CSV exchange, Parquet exchange, and I/O recovery. This structure provides explicit scopes for broad exploration and more focused checks, including whether failed imports preserve existing table contents. In the bitwise calculator, existing tests covered implicit multiplication and `BIT` expressions separately, yet their combination remained incorrect. The added tests exposed two distinct gaps: `2 BIT(3)` returned 8 instead of 16, while the invalid form `2BIT(3)` was accepted. Comparing these inputs clarified both the required computation and the boundary between valid and invalid expressions. The findings then became explicit testing tasks and coordinated changes to the lexer and parser, after which existing and added tests passed. This progression demonstrates how executable behavioral specifications turn previously uncovered interactions into concrete implementation requirements, extending replication beyond the behaviors constrained by the original test suite.

B.4.1 CLASS-STRUCTURED BEHAVIORAL SPECIFICATIONS IN DUCKDB

DuckDB is an analytical database with a command-line interface for executing SQL and managing data. Figure 4 shows the final class-structured behavior model from ECC exploration. High-level classes organize capabilities such as SQL execution, file I/O, and storage lifecycle, while subclasses refine their behavioral scope. For example, File I/O is divided into CSV exchange, Parquet exchange, and I/O recovery, among other classes. Function declarations describe operations within each class. This structure organizes behavioral knowledge at different levels of detail to guide broad feature discovery and focused exploration.

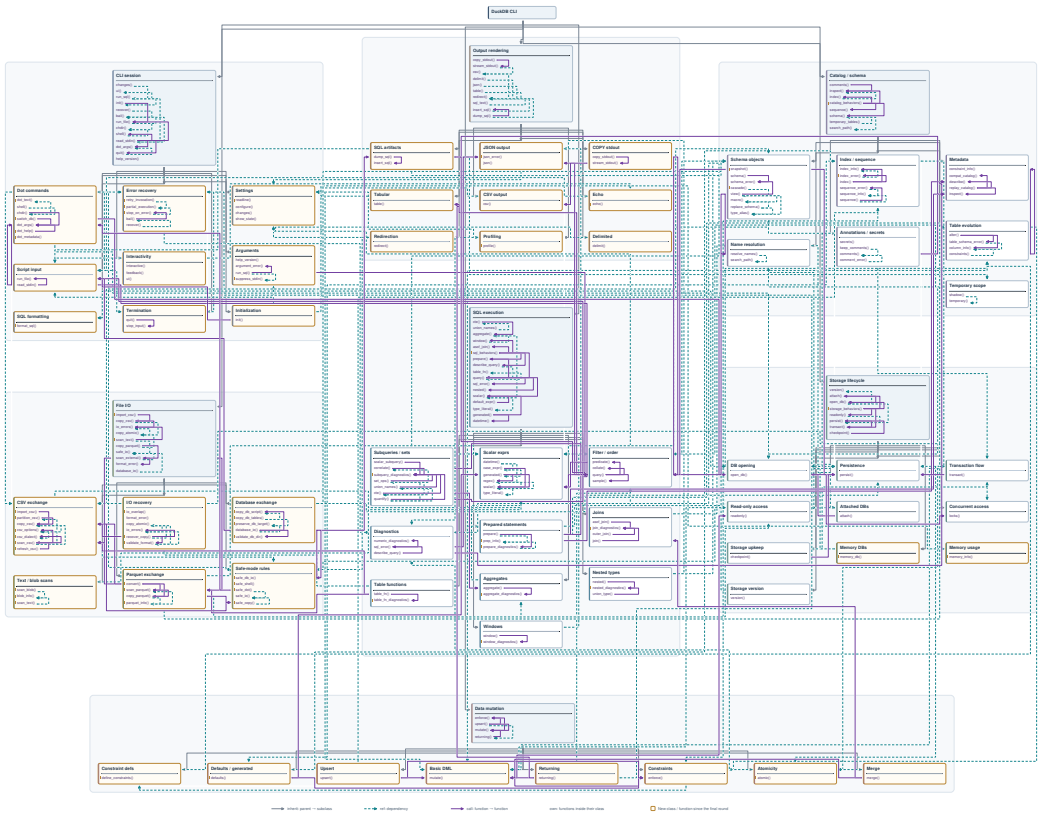

Figure 4: Class-structured behavioral specifications for DuckDB. Class boxes group related behaviors, and function declarations describe operations. Inheritance represents hierarchical refinement, and containment denotes function ownership. Reference and call edges illustrate relationships drawn from behavioral descriptions.

B.4.2 TEST EXPANSION AND IMPLEMENTATION REPAIR IN BITWISE

We use the bitwise task from MirrorCode as a case study. `BIT(n)` denotes a bit mask with bit n set; for example, `BIT(3)` equals 8. The implementation already supported ordinary implicit multiplication and `BIT` expressions separately, but additional tests exposed incorrect semantics when the two were combined. Table 9 summarizes the test expansion, plan update, and repair.

Table 9: Test expansion and repair in bitwise. Additional tests reveal an uncovered interaction between implicit multiplication and `BIT` expressions, prompting updates to the testing plan and parser.

Stage	**Details**
Existing tests	Existing tests covered `2(3+4)` and `BIT(2)BIT(3)`, but not a number followed by a space and a `BIT` call.
Added tests	Tests were added for `2 BIT(3)` and `2 BIT(3) BIT(4)`, with the unspaced form `2BIT(3)` as a control.
Discrepancy	The teacher evaluates `2 BIT(3)` as implicit multiplication, yielding $2 \times 8 = 16$, whereas the student returns 8. The teacher also rejects `2BIT(3)`, which the student incorrectly accepts. The student thus fails to distinguish a number followed by `BIT` from consecutive `BIT` selectors.
Plan update	The agent assigned the issue to `expression-semantics`, explicitly adding checks for adjacent `BIT` calls, implicit multiplication, operator precedence, and lexical boundaries, and added the three cases to the test suite.
Repair and validation	The lexer was updated to reject the unspaced form `2BIT(3)`. The parser now tracks whether the left operand is an ordinary number or a `BIT` selector, allowing the spaced form `2 BIT(3)` to use implicit multiplication. After repair, `2 BIT(3)` evaluates to 16, and all existing and added tests pass.

The final implementation passes all **989 official tests**, including **479 hidden tests**, achieving an official pass rate of **100.0%**. This case shows how additional tests uncover feature interactions missed by the original suite and directly drive changes to the testing plan and expression parser.

C EXPERIMENTAL DETAILS

C.1 BENCHMARK TASKS

We evaluate BEHAVIORDISTILL on ProgramBench and MirrorCode (Yang et al., 2026; Adamczewski et al., 2026). Both benchmarks require agents to build software from scratch that reproduces a teacher program's behavior using documentation and black-box access to the teacher, without access to its source code. We evaluate student implementations on each benchmark's official hidden test set.

ProgramBench. ProgramBench covers command-line tools, interpreters, databases, and multimedia software. Each task provides usage documentation, an execute-only teacher program, and any necessary input files. Agents receive neither source code nor evaluation tests. The official end-to-end tests assess behavioral consistency through standard output, standard error, exit status, and filesystem side effects.

MirrorCode. MirrorCode covers Unix utilities, data processing tools, interpreters, static analyzers, and compression software. Its official setup provides usage documentation, an execute-only teacher program, and visible tests. The visible tests define the replication scope and guide implementation, while their hidden counterparts evaluate the corresponding functionality with different inputs. In our experiments, replication performance is evaluated on official hidden tests, while the full official suite, including visible and hidden tests, serves as the coverage reference. Only Vanilla + Official Tests receives official visible tests for construction; BEHAVIORDISTILL uses no official tests during exploration or construction.

C.2 BASELINES

Test source-code coverage. Following ProgramBench, we measure the teacher's first-party line coverage and average it equally across tasks. In the main experiments on both benchmarks, we compare the coverage of BEHAVIORDISTILL's combined ECC and PCC test suite, Gongent's generated tests, and the corresponding official suite. For ProgramBench, we use the official hidden tests, generated by agents with access to the teacher's source code, as a reference for coverage under source-assisted test generation. For MirrorCode, we use the full official suite of visible and hidden tests, collected from multiple sources and revised and consolidated by humans. Gongent serves as a black-box test-generation baseline: its Adversary Agent probes the teacher using predefined test dimensions and combination rules, then uses execution observations to produce a fixed test suite (qucheng-gong, 2026). This provides a comparison of test coverage without source code access.

Replication performance. We compare the full BEHAVIORDISTILL pipeline with the following baselines:

- **Vanilla.** We use mini-SWE-agent with only task documentation and access to the teacher's execution interface. No pregenerated tests are provided on either benchmark.
- **Vanilla + Official Tests.** We augment Vanilla with MirrorCode's official visible tests to assess their contribution to replication. This baseline applies only to MirrorCode.
- **SpecFirst.** Following its publicly released workflow, we first elicit natural-language behavioral specifications through black-box exploration, then use them to guide student construction (Chen et al., 2026).
- **Gongent.** Following its publicly released workflow, an Adversary Agent generates tests, and feedback from differential execution of the teacher and student guides the Builder Agent in iteratively repairing the implementation (qucheng-gong, 2026).

C.3 IMPLEMENTATION AND CONFIGURATION

Common settings. All methods use `GPT-5.6 Luna` and `DeepSeek-V4-Flash-0731`, with high reasoning effort and the same runtime environment and tool permissions. Preliminary exploration and specification generation, where applicable, share a six-hour budget; subsequent con-

struction receives a separate six-hour budget. All methods run in sandboxes with strictly enforced network isolation.

Agent configuration. BEHAVIORDISTILL's Exploration Agents, Specification Modeling Agent, and Builder Agent are all implemented using mini-SWE-agent. During ECC warm-up, each round runs n_a Exploration Agents in parallel. During hierarchical exploration, each eligible class is assigned n_a Exploration Agents. During PCC, the Builder Agent iteratively implements the student according to a dynamic plan, adding tests and revising the plan based on test feedback.

ECC parameters. We set the initial test target to $N_0 = 80$, the default per-class load limit to $N_c = 80$, the stopping threshold to $N_{\max} = 300$, the maximum exploration depth to $L = 4$, and the number of Exploration Agents per group to $n_a = 3$. The 300-test stopping threshold applies only to ECC; tests generated during PCC are counted separately.

ECC specification updates. The Specification Modeling Agent uses new tests to update classes, functions, and their relations, maintain test links and unique assignments, and check structural validity and type consistency. Classes are split or regrouped under the load policy in ECC step D. For each permitted internal-node exception, we record the node, affected tests, and reason; its status is reconsidered only when evidence or structure changes. ECC terminates when the test count reaches $N_{\max}$ or all leaf classes reach depth L.

C.4 EVALUATION PROTOCOL

Following the benchmarks' task-level aggregation conventions, we report results separately for each benchmark and compute unweighted means across tasks. This prevents tasks with larger codebases or more tests from dominating the overall results. Below, m denotes the number of tasks included in the corresponding metric.

Test source-code coverage. Following ProgramBench (Yang et al., 2026), we measure first-party line coverage on an instrumented build of the teacher. We count only executable first-party source lines, excluding system headers, third-party dependencies, and automatically generated code. All test suites for a given task use the same build and source-line inclusion rules. Let q_j be the fraction of included source lines covered by the test suite for task j. Overall coverage is $\frac{1}{m}\sum_{j=1}^{m} q_j$.

Raw Pass Rate. We follow both benchmarks' official protocols for computing per-task pass rates with equal weight for each test, and evaluate exclusively on official hidden tests. Let $\mathcal{H}_j$ denote the official hidden tests included in scoring for task j. Its pass rate and the overall Raw Pass Rate are

$$p_j = \frac{\text{number of hidden tests passed}}{|\mathcal{H}_j|}, \qquad \text{Raw Pass Rate} = \frac{1}{m}\sum_{j=1}^{m} p_j.$$

For MirrorCode, $\mathcal{H}_j$ contains only tests officially labeled as hidden; visible tests withheld from the agent are not added to this set.

Almost Resolved. We adopt ProgramBench's 95% threshold and apply the same definition to MirrorCode: task j is Almost Resolved when $p_j \geq 0.95$. We report $\frac{1}{m}\sum_{j=1}^{m} \mathbf{1}[p_j \geq 0.95]$ as a percentage.

Execution and evaluation rules. A teacher's nonzero exit code and associated outputs can constitute a valid behavioral observation. We exclude executions that time out, encounter environment failures, or yield incomplete observations from the generated test suite. When construction ends or reaches the six-hour limit, we save the current code, rebuild it in the evaluation environment, and run the official hidden tests. Missing submissions and build failures receive a score of zero. Source-code coverage results and final hidden-test feedback are used only for offline evaluation and are not provided to agents performing black-box replication.

C.5 IMPLEMENTATION DETAILS

Full prompts are provided in three supplementary files: `01_ECC_Exploration.pdf` contains the Exploration Agents' prompt; `02_Specification_Modeling.pdf` contains the Specification Modeling Agent's prompt and the specification validation prompt; and

`03_PCC_Construction.pdf` contains the Builder Agent's prompts for initial planning and iterative construction.

After each exploration round, BEHAVIORDISTILL collects candidate tests from all Exploration Agents. Duplicate test names trigger an error. For tests that pass validation, the top-level function name is normalized to `_test`, and the serialized Python ASTs are compared without source-location attributes. Tests with identical representations are merged. The resulting tests are rerun on the teacher, and only those yielding complete, valid observations enter the ECC test suite.

Initial specification construction and subsequent updates follow the same validation procedure. Independent semantic validation checks behavioral grouping, model relations, and supporting test evidence. Programmatic checks verify the code skeleton's structural and type consistency, test links, unique assignments, and class-load constraints, while ensuring that existing tests remain unchanged. If validation fails, the Specification Modeling Agent revises the specification using the feedback, preserving existing test inputs, expected outputs, and assertions. After local operations are accepted, the updated specification must still pass validation as a whole.