

Contract Before Code: Failure-Compiled Contracts for Long-Horizon Coding Agents

Jiangnan YU

ECE

The Hong Kong University of
Science and Technology
jyucr@connect.ust.hk

Kisson Songqi Lin

Artificial Intelligence

Super Intelligence Inc.

Abstract

Long-horizon coding agents can make many edits before testing a task-wide assumption about an API, authentication mode, environment, or persistent state. A late verifier then exposes that the implementation rests on a false premise. We call the missing step contract acquisition and propose Contract Before Code (CBC), harness-level middleware that compiles historical failures into executable pre-flight gates. CBC normalizes trajectories, extracts recurring failure signatures, and compiles contracts containing an applicability condition, a probe template, and a planning consequence. At inference time, it retrieves applicable contracts and runs instantiated probes before broad implementation work. Among 1,421 parsed failed SWE-Marathon trajectories, weak labels identify candidate rates of 2.6% overall and 17.5% on stripe-clone; these are not root-cause prevalence estimates. A 120-trajectory human audit reports 76.3% contract precision and 71.4% executable-probe synthesis. Across four service/API-clone tasks and 32 runs per method, CBC achieves 31.3% success (95% CI 18.0–48.6), an observed 21.9-percentage-point increase over vanilla, with 55.6% fewer pre-verification edits at $1.11\times$ normalized token cost. The small end-to-end study supports preliminary within-family benefits; broader transfer and the stability of these gains remain unestablished.

1 Introduction

Coding agents are increasingly expected to perform repository-scale software work rather than isolated code completion. Modern harnesses let them inspect files, edit multiple modules, run shell commands, call package managers, and iterate against verifiers (Yao et al., 2023; Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025). This shift makes planning more than a question of which file to patch next. Before implementation becomes

useful, the agent must decide which task-wide assumptions are worth validating: an API boundary, authentication mode, database state, or hidden acceptance criterion. When one such assumption is wrong, a long implementation branch can be coherent locally but irrelevant globally.

Existing benchmarks and execution frameworks make these failures visible. Repository benchmarks, tool sandboxes, trajectory evaluators, and test-based verifiers expose where an agent eventually fails (Jimenez et al., 2024; Deng et al., 2025; Zhou et al., 2024; Xie et al., 2024; Yao et al., 2024; Li et al., 2026; Kim et al., 2025; Chen et al., 2023; Ni et al., 2023). Reflection and memory systems can also store natural-language lessons from past attempts (Shinn et al., 2023; Park et al., 2023; Packer et al., 2023; Wang et al., 2023). However, these mechanisms usually intervene after the agent has already acted, or they preserve failure knowledge as advice rather than as an executable gate. They do not specify how a past task-wide contract failure should constrain the next agent before it spends its editing budget.

We identify this missing step as *contract acquisition*. A *task-wide contract* is an assumption whose violation invalidates broad downstream implementation work, while contract acquisition is the act of checking that assumption early enough to change the plan. For example, in a payment-service clone, endpoint work on customers, payment intents, idempotency, and webhooks is wasted if the local Stripe-compatible SDK rejects the benchmark authentication mode. Figure 1 contrasts this late-checking path with a contract-gated path that probes the root assumption first. The goal is not to run every possible test before coding, but to ask a small number of questions whose answers determine whether a large implementation branch is meaningful.

In this paper, we present Contract Before Code, a harness-level framework that compiles failed tra-

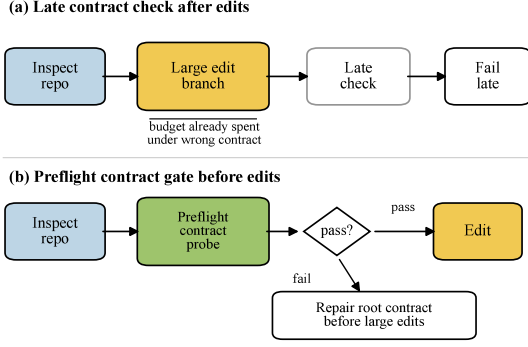


Figure 1: A common long-horizon coding-agent failure: implementation proceeds before task-wide contracts are validated.

jectories into reusable task-wide contracts and executable preflight probes. The unit of reuse is not a patch or a natural-language lesson, but a contract family with an applicability condition, a probe template, and a planning consequence. At inference time, the harness retrieves contracts whose family and applicability match the new task, instantiates their probes in the current repository, and runs them before allowing large implementation edits.

This paper makes the following contributions:

- **Framing.** A contract-acquisition view of long-horizon coding-agent failures, separating task-wide false premises from ordinary local implementation defects.
- **Pipeline.** A failure-to-contract pipeline called Contract Before Code that extracts failure signatures, abstracts contract families for within-family reuse, synthesizes probe templates, and attaches planning consequences.
- **Methodology.** An evaluation methodology that measures contract quality, probe synthesis rate, false-block rate, wasted edits, and end-to-end task success rather than final verifier pass rate alone.
- **Evidence.** A SWE-Marathon study showing a measurable contract-shaped failure class and, in controlled service/API-clone reruns, 31.3% success (95% CI 18.0–48.6) with 55.6% fewer pre-verification edits.

2 Background and Problem

2.1 Harnessed Long-Horizon Coding Agents

We consider a coding agent operating inside a harness that exposes a repository, shell commands, file editing tools, and verifiers (Yao et al., 2023; Yang et al., 2024; Wang et al., 2025). At step t , the agent observes the current workspace and tool output, chooses an action, and updates its plan from the resulting observation. The harness finally scores the task with tests, gates, or benchmark-specific checks. The important feature of this setting is that verification is often delayed. An agent may read files, install dependencies, edit several modules, and only then run the command that exposes the decisive mismatch. This delay is acceptable for local bugs, where partial implementation can still be useful. However, it is costly when the mismatch concerns a task-wide assumption that should have been checked before the implementation branch began.

2.2 Task-Wide Contracts

A task-wide contract is an assumption whose violation makes broad downstream edits irrelevant or misleading. It sits between a local unit invariant and the full task specification: narrower than “solve the task,” but broader than “this function returns this value.” Examples include SDK authentication shape, cross-component dataflow, dependency versions, command-line semantics, database state, and hidden-test output schemas. As its name suggests, a task-wide contract is not the implementation itself; it is the condition under which implementation work becomes meaningful.

Figure 2 summarizes six contract families: interface shape, environment configuration, persistent state, cross-module dataflow, acceptance criteria, and recovery targets. We call a long-horizon task *contract-rich* when many downstream implementation choices depend on such explicit, checkable task-wide assumptions. Service/API clones, schema-dependent backends, and multi-component integration tasks are contract-rich because the same contract family can govern many files and verifier gates. In contrast, an isolated algorithm bug may have a precise specification but little reusable contract structure beyond the local function.

2.3 Failure-Compiled Contracts

A *failure-compiled contract* is a task-wide contract derived from a failed trajectory and paired with a

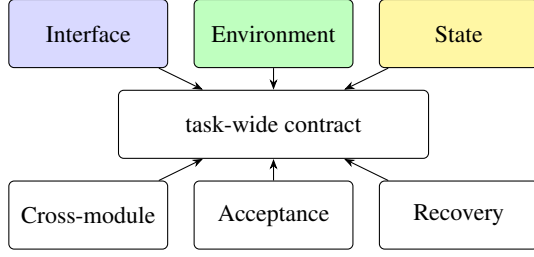


Figure 2: Task-wide contracts capture assumptions whose violation invalidates broad downstream implementation work.

probe template that can be instantiated as an executable preflight probe. We represent it as a tuple (f, s, a, p, g) : a contract family f , a contract statement s , an applicability condition a , a preflight probe template p , and a planning consequence g . The Stripe-clone authentication contract example in Section 4.3 illustrates all five fields. The compiled object is therefore more operational than a natural-language lesson: it says when the lesson applies, how to test it, and what the harness should do if the test fails.

Contract Before Code transfers a contract family and probe template, rather than a concrete patch, within an applicable task family when the new task satisfies the applicability condition; if no contract applies, the method is a no-op.

2.4 Problem: Selecting Preflight Probes

Given historical trajectories $\mathcal{T}$ and a new task x , the problem is to select and instantiate a small set of preflight probes P_x for the task-wide contracts likely to govern x . A good selection maximizes the *protected implementation region*: the work that would be wasted if the contract is false. It must also minimize false blocks, redundant probes, and probe cost. Thus the target is not exhaustive pre-verification, but early execution of high-leverage checks that decide whether the agent should implement, repair a root contract, or ask for missing environment information.

3 Motivation

Final failure often hides where the decisive mistake happened. A trajectory can contain plausible edits but still be doomed because the agent never validated the contract those edits depend on. The useful intervention point is therefore often earlier than the final patch.

We illustrate the pattern with a stripe-clone failure from the SWE-Marathon corpus (Desai

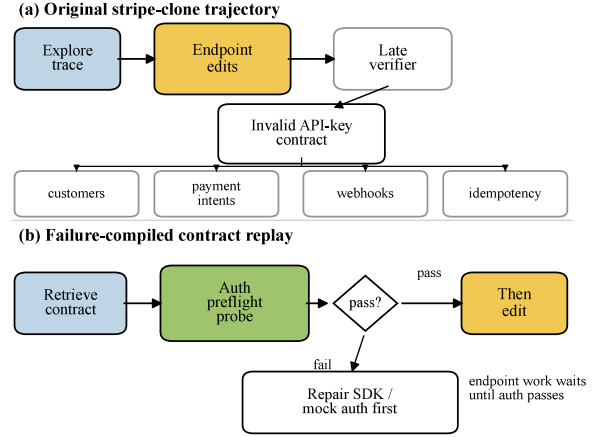


Figure 3: A failure-compiled contract changes the planning point: the agent validates a task-wide contract before spending implementation budget downstream.

et al., 2026) (Figure 3; corpus details in Section 5.1). The agent spent its early budget exploring and then writing across many service components before running a meaningful verifier. The final verifier reported broad failure across authentication, customers, payment intents, webhooks, and idempotency, with many errors sharing an invalid API-key signature. The missing step was an early probe for the Stripe-compatible authentication and mock-server contract.

This motivates a system that distinguishes local defects from global contract violations and makes failure knowledge executable as preflight probes rather than advisory lessons.

4 Contract Before Code

Contract Before Code is a harness-level framework that converts failed trajectories into failure-compiled contracts without changing the base language model. It adds a contract layer between the agent and high-cost implementation actions. Figure 4 gives an overview. The design separates three phases with strict information boundaries: *audit* discovers recurring failure patterns from a corpus of historical trajectories; *compilation* compresses those historical failures into contract objects and executable probe templates for within-family reuse; and *deployment* instantiates compiled contracts in a new, independent run whose task satisfies their applicability conditions. Crucially, the knowledge source is always a past episode’s failure, never the current episode’s future states or outcomes. No information flows backward within a single run.

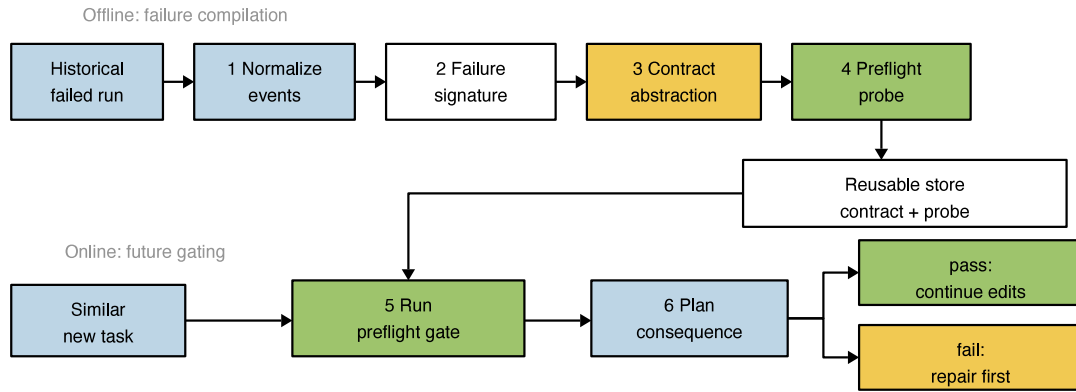


Figure 4: Contract Before Code compiles historical failures into failure-compiled contracts whose preflight probes gate future agent planning.

4.1 Normalizing Trajectories into Events

The pipeline first normalizes a failed trajectory into typed events: file reads, searches, edits, shell commands, tests, verifier outputs, and final benchmark results. For each event, the harness records the step index, touched files, command class, exit status, output signatures, and whether the event mutates the workspace. This model-agnostic representation treats the trajectory as an execution artifact, not as hidden reasoning.

4.2 Extracting Failure Signatures

Next, the system extracts failure signatures from verifier logs and terminal outputs. A signature records the error class, affected components, repeated message fragments, failing tests or gates, and earliest checkable step. Signatures are extracted by a rule-based parser that normalizes verifier logs into (error-class, affected-component, message-hash) triples; no LLM is used at this stage. Rather than summarizing every symptom, the parser asks whether many symptoms share an upstream assumption, such as authentication errors across independent endpoints.

4.3 Compiling Transferable Contracts

The abstraction stage maps concrete failure signatures to transferable task-wide contracts. Abstraction uses a structured LLM prompt with the six contract families as a fixed schema; the model selects a family, writes a contract statement, applicability condition, and probe template in a constrained JSON format. Given a signature, Contract Before Code produces:

1. a contract family, such as interface shape, authentication, schema, persistent state, or ac-

ceptance criteria;

2. a contract statement, such as “the local SDK accepts the benchmark authentication key”;
3. an applicability condition, such as “payment-service clone with a Stripe-compatible SDK surface”;
4. a probe template, such as “create one customer through the SDK using the benchmark key”;
5. a planning consequence, such as “repair auth compatibility before endpoint work.”

The result is a compact failure-compiled contract whose probe template can test whether a future plan rests on the same false premise. The compiled object discards task-specific edits and paths, retaining only the contract family and minimal observable behavior needed for transfer.

4.4 Synthesizing Preflight Probes

Probe synthesis turns each abstract contract into executable code or commands: a smoke test, API call, schema check, or lightweight integration test. Each probe should be small, deterministic when possible, and cheap relative to the implementation branch it protects. For external services, probes use snapshots, mocks, or normalized outputs to avoid brittle transient responses. When deployment occurs on a new task, the probe template is instantiated by substituting repository-specific paths, URLs, and credentials discovered during the normalization phase; the harness validates executability by a dry-run check before gating.

4.5 Gating Agent Execution

At inference time, the harness retrieves candidate task-wide contracts for the new task and runs their probes before allowing large implementation edits. A passing probe permits downstream work; a failing probe redirects the agent to repair the violated contract, narrow the task, or request missing environment information. The gate blocks broad implementation only when the failed probe targets a task-wide contract, not an ordinary local bug.

Retrieval is deliberately conservative: the harness indexes contracts by contract family, repository surface, and failure signatures, then filters through the applicability condition. When multiple contracts match, it prefers those whose probe is executable and whose planning consequence targets a repairable root.

4.6 Cost Model, Operating Point, and Implementation

Because probes add retrieval and execution overhead, the operating point targets a small number of high-leverage checks rather than exhaustive pre-verification; the evaluation measures false-block rate and probe cost alongside success (Section 5.7).

We implement Contract Before Code as a lightweight middleware on OpenHands (Wang et al., 2025), an open-source coding-agent framework. The integration adds four components totaling approximately 700 lines of Python without modifying OpenHands core code: (1) a pre-action hook that intercepts edit actions and triggers contract retrieval; (2) a JSON-backed contract store holding compiled contracts and probe specifications; (3) a probe runner that executes preflight probes inside the OpenHands sandbox runtime; and (4) a retrieval module that matches contracts to the current task by contract family, repository surface, and normalized failure signatures. All experiments use OpenHands v1.8.

5 Experiments

The evaluation asks whether contract-acquisition failures are measurable, whether Contract Before Code improves success and reduces wasted work, and which components explain the gains. We report success rate (95% CI), partial verifier score, pre-verification edits, false-block rate, and normalized token cost. Success is pooled over 32 runs; continuous metrics use task-level means over eight seeds and cross-task standard deviations, as detailed in

Table 1: Weak-label audit of failed SWE-Marathon trajectories. The candidate rate is a heuristic selection rate, not a validated lower bound on root-cause prevalence.

Signal	Count	Rate
Parsed failed trajectories	1,421	100.0%
Delayed implementation (≥ 3 edits)	176	12.4%
Broad verifier failure	319	22.4%
Both signals (candidate)	37	2.6%
Strict repeated-signature subset	14	1.0%
stripe-clone: both signals	14/80	17.5%
Candidate median edits before check		23

Section 5.5. False-block rate matters because a preflight gate can harm an otherwise valid run; normalized cost matters because a method that wins by running exhaustive tests before every action is outside our intended operating point.

5.1 Corpus

We use the live SWE-Marathon trajectory corpus (Desai et al., 2026), shared by the original authors for research use and inspected on August 4, 2026. It contains 1,797 trials across 21 tasks. After removing nop and oracle baselines, 1,422 of 1,587 agent trials fail; our parser reads 1,421 failed trajectories (99.9%). The corpus spans six harnesses and 8+ frontier model families, so the audit targets harness-level failure patterns rather than one model artifact.

5.2 Weak-Label Candidate Rates (E1–E2)

We apply two weak-label classifiers to all 1,421 parsed failed trajectories. A trajectory is marked as *delayed implementation* if it performs at least three mutating edits before the first self-check command. It is marked as *broad verifier failure* if the final verifier reports at least three failing modules/gates or at least ten failed tests. We call the intersection a candidate contract-acquisition failure; a stricter subset also requires the same failure-signature family across at least three modules.

Table 1 and Figure 5 show that 12.4% of parsed failures have delayed implementation, 22.4% have broad verifier failure, and 2.6% have both. The candidates have a median of 23 edits before the first self-check and are concentrated in interface-heavy workloads such as stripe-clone (17.5%). These heuristics identify trajectories for investigation; delayed edits and broad failures alone do not establish a shared upstream root cause. Their false-positive and false-negative rates against human root-cause labels, and sensitivity to the labeling thresholds,

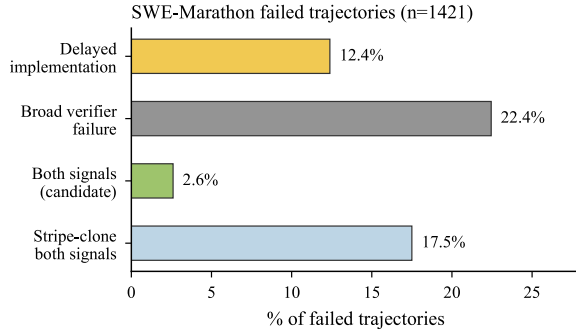


Figure 5: Weak-label signal rates across 1,421 parsed failed SWE-Marathon trajectories. The “both signals” candidate rate is 2.6% overall and 17.5% on stripe-clone.

Workload	Contract precision	Probe synthesis	False contract
Service/API clone	83.1%	82.6%	13.5%
Non-clone tasks	51.7%	57.2%	23.5%
Sample overall	76.3%	71.4%	16.9%

Table 2: Human audit of failure-to-contract compilation. Service/API-clone tasks yield higher precision because their failures often concentrate around explicit interface, authentication, schema, and state contracts.

have not been established. Accordingly, the 2.6% figure is neither an estimate nor a validated lower bound on contract-acquisition failure prevalence.

5.3 Contract Compilation Quality (E3)

We sample 120 failed trajectories from the SWE-Marathon audit: all 37 contract-acquisition candidates and 83 additional failed trajectories stratified by task family, agent harness, and model family. Two annotators label whether each failure admits a task-wide contract, its family, whether the generated contract captures the upstream failure, and whether its probe is executable; disagreements are resolved after computing agreement.

Table 2 shows that 76.3% of generated contracts are correct and 71.4% can be paired with an executable probe. Precision is much higher on service/API-clone tasks (83.1%) than on non-clone tasks (51.7%), and family agreement is substantial ($\kappa = 0.72$), consistent with the intended contract-rich scope. This audit assesses compilation quality on the sampled trajectories; it does not validate the corpus-wide weak-label prevalence estimate. Per-metric denominators and aggregation weights are not available in the reported summary, so the overall percentages cannot be reconstructed from the two workload rows alone.

Metric	Median	P75
Earlier signal (steps)	94	211
Avoided edits	18	52
Avoided tool calls	41	96
Applicable cases	68.2%	

Table 3: Offline replay estimate for confirmed contract failures. Savings count only the protected region between the earliest applicable probe step and the first matching root-failure signal.

5.4 Offline Replay Savings (E4)

For each confirmed contract, we estimate the earliest step at which the probe could run and the first later signal that reveals the same violation. The protected region is the contract-dependent segment between those points; if either endpoint is missing, the case is excluded. This counterfactual estimates wasted budget only, not final task success.

For 68.2% of confirmed contract failures, both endpoints are identifiable. As Table 3 shows, a failing probe would surface the root violation a median of 94 steps earlier and protect a median of 18 edits and 41 tool calls. The estimate is concentrated in service/API-clone tasks, where endpoint work can continue long before a meaningful verifier runs.

5.5 End-to-End Benefit (E5)

We run controlled reruns on four interface-heavy service/API-clone tasks (stripe-clone, s3-clone, slack-clone, mastodon-clone) with eight random seeds per method (32 runs each). All methods use Claude Opus 4 with temperature 0.3, a 500-step budget, 200k-token context window, and a 4-hour wall-clock timeout. The verifier is the SWE-Marathon multi-layer scoring suite; partial verifier score is the fraction of individual gates passed. Normalized cost is total input plus output tokens divided by the vanilla baseline’s mean token count for the same task; it measures token usage rather than a price-weighted dollar cost.

For Contract Before Code, contracts are retrieved by contract family and applicability condition from a contract store compiled from historical failed trajectories. The store excludes: (i) all trajectories from the current run, (ii) all trajectories sharing the same seed, and (iii) all trajectories from the same agent on the same task. It retains only trajectories from other agents on the same task family that terminated before the experiment began; no verifier outcome from the current run is visible to compilation or retrieval. The current

evidence is within-family, though retrieval operates on contract family and repository surface, not task identity. This split does not establish transfer to an unseen task family or repository.

We compare vanilla, reflection memory, post-hoc verification, generated tests, and full Contract Before Code. Success rate and its 95% CI are computed over all 32 runs (pooled). Partial score, pre-verification edits, and cost are first averaged per task over eight seeds, then averaged across four tasks; the reported $\pm$ reflects cross-task standard deviation.

Table 4 shows the main result: Contract Before Code reaches 31.3% success (10/32; 95% CI 18.0–48.6), compared with 9.4% (3/32) for vanilla. The observed difference is 21.9 percentage points over vanilla and 12.5 points over generated tests. It also raises partial verifier score from 0.31 to 0.56, cuts pre-verification edits by 55.6%, and reduces tool calls by 24% at $1.11\times$ normalized token cost. Reflection memory, post-hoc verification, and generated tests show smaller observed gains. The false-block rate is 6.3% (2/32 runs). With only four tasks and 32 runs per method, both the success improvement and false-block rate remain uncertain; these results support a preliminary benefit in the evaluated interface-heavy setting, not a stable effect size across repository-level coding tasks.

5.6 Component Necessity (E6)

We ablate each Contract Before Code component on the same four tasks and eight seeds as E5, yielding 32 runs per variant. Each variant removes or replaces one component and is evaluated on success, false-block rate, and edit reduction.

Table 5 shows that the largest observed gains use contract abstraction, gating, and planning consequence together. Signature-only probes raise false blocks to 21.9%; probe-as-advice yields the weakest edit reduction (28.4%); and removing the planning consequence cuts success to 15.6% despite high edit reduction. Unbounded early tests improve over vanilla; their cost is not reported in this ablation.

5.7 Operating Point (E7)

The sensitivity study reports the same four tasks and eight seeds as E5, with 32 runs per configuration. Each sweep varies one parameter while fixing the others at the reported defaults: $k = 3$, two probes per contract, and a majority-fail gate. The experiment description varies re-

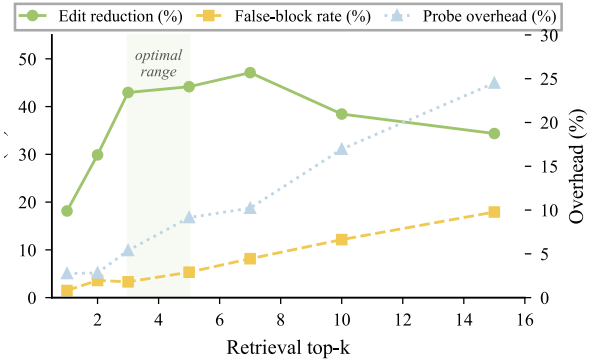


Figure 6: Originally reported retrieval top- k sensitivity plot. The shaded operating range and plotted values are retained from the original experiment; the default-point discrepancy with E5 remains unresolved, so the indicated optimum is not validated.

trieval top- k over $\{1, 2, 3, 5, 8, 10, 15\}$, probe budget over $\{1, 2, 3, 5\}$, and gate strictness over $\{\text{any-fail, majority-fail, all-fail}\}$, reporting edit reduction, false-block rate, and probe overhead averaged across runs.

Figure 6 preserves the reported retrieval sweep. Its default-point edit reduction (approximately 43%) does not reproduce the 55.6% in Tables 4 and 5. Under the stated fixed-default protocol, a single-parameter sweep alone does not explain this difference. The configuration-level provenance has not been reconciled, so this sweep should be treated as exploratory; it does not establish an optimal retrieval setting or independently validate the E5 effect size.

5.8 Failure Modes (E8)

We categorize all 68 Contract Before Code failure cases from controlled reruns (E5) and offline replay (E4). Two annotators assign each case to one of six categories and judge whether it is mitigable by pipeline engineering.

Table 6 shows cold start (32%, no historical analog), residual hidden-criterion failures (16%, the probe checks the right family but the verifier applies a stricter criterion), overly broad contracts or false blocks (18%), and flaky probes (9%). Agent misuse of probe results accounts for 14%, while repairs beyond model capability account for 11%. These are distinct failure mechanisms: improving probe reliability cannot by itself supply missing failure history or strengthen the base model’s repair capability. The results suggest that CBC is most useful in settings with relevant accumulated failure history.

Table 4: Controlled rerun results on service/API-clone tasks (4 tasks $\times$ 8 seeds = 32 runs per method). Success intervals are 95% CIs; other $\pm$ values are cross-task standard deviations of task means.

Method	Success rate [95% CI]	Partial score	Pre-verification edits	Relative token cost
Vanilla	9.4% [3.2, 24.2]	0.31 ± 0.06	48.2 ± 4.6	$1.00 \times$
Reflection	12.5% [5.0, 28.1]	0.36 ± 0.04	43.5 ± 5.4	$(1.02 \pm 0.03) \times$
Post-hoc	15.6% [6.9, 31.8]	0.41 ± 0.05	46.8 ± 6.7	$(1.18 \pm 0.03) \times$
Generated tests	18.8% [8.9, 35.3]	0.44 ± 0.03	38.6 ± 5.9	$(1.24 \pm 0.03) \times$
CBC	31.3% [18.0, 48.6]	0.56 ± 0.03	21.4 ± 3.3	$(1.11 \pm 0.03) \times$

CBC false blocks: 2/32 (6.3%). Tool calls: 142 versus 187 for vanilla (-24%).

Table 5: Ablation study on service/API-clone tasks. Each row removes or replaces one component from the full pipeline. The rerun reporting convention matches Table 4.

Variant	Success	False block	Edit reduction
Full CBC	31.3%	6.3%	55.6%
No retrieval	18.8%	9.4%	32.1%
Signature-only probe	21.9%	21.9%	41.8%
Probe-as-advice	21.9%	—	28.4%
No planning consequence	15.6%	6.3%	48.7%
Unbounded early tests	25.0%	3.1%	51.2%

Table 6: Failure-mode taxonomy for Contract Before Code on controlled reruns and offline replay ($n = 68$ failure cases). Shares pool these two sources rather than describing the E5 failure rate alone.

Failure category	Share	Mitigable?
No historical analog	32%	Partial
Contract too broad / false block	18%	Yes
Probe passes, hidden criterion	16%	Partial
Agent ignores / misuses result	14%	Yes
Repair beyond model capability	11%	No
Flaky probe (environment)	9%	Yes

The table retains the reported aggregate percentages, but underlying category counts and any aggregation weights are unavailable. In particular, 14% and 11% cannot be obtained by ordinary nearest-integer rounding of counts divided by 68. The taxonomy is therefore descriptive; the category shares require reconciliation before they support precise frequency comparisons.

6 Related Work

Agent benchmarks and dynamic tool environments. Recent benchmarks stress agents with realistic tools, stateful environments, long horizons, and tightly coupled constraints (Liu et al., 2023; Qin et al., 2024; Zhou et al., 2024; Xie et al., 2024; Yao et al., 2024). Repository-level software benchmarks further show failures under large ac-

tion spaces, shifting environments, and multi-file integration (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025; Deng et al., 2025; Li et al., 2026). These benchmarks expose failure; Contract Before Code uses their trajectories as raw material for failure-compiled contracts.

Trajectory evaluation. Trajectory-level evaluation makes hidden agent behavior measurable by judging efficiency, hallucination, grounding, and adaptivity across intermediate steps (Yao et al., 2023; Kim et al., 2025). Our output is not a scalar score or critique, but a failure-compiled contract whose probe template can gate a later trajectory.

Verification and tests. Execution-based verification and proactive test generation improve coding agents by finding discriminative cases or exposing hidden bugs (Chen et al., 2023; Ni et al., 2023; Chen et al., 2024). Agentic verifier systems actively search for tests that distinguish candidate programs (Ma et al., 2026). Contract Before Code differs in that its probes are not generated from the current program but retrieved and instantiated from historically compiled contract families, targeting task-wide assumptions before large edits rather than local correctness after them.

Failure-driven workflow refinement and agent memory. Workflow-refinement systems use failure signatures to improve prompts, graphs, or policies, while memory systems store past experiences for later retrieval (Shinn et al., 2023; Park et al., 2023; Packer et al., 2023; Wang et al., 2023). Recurring failure signatures can guide system-level refinement (Zhang et al., 2025), but these approaches typically preserve failure knowledge as advice, learned parameters, or workflow modifications. Contract Before Code preserves it as an inspectable failure-compiled contract with an applicability condition, probe template, and planning consequence.

Safety, oversight, and incident response. Agent safety work studies oversight and guardrails for high-risk actions (Ruan et al., 2024); our narrower focus is coding-agent reliability.

7 Conclusion

Long-horizon coding agents can waste implementation effort before acquiring the task-wide contracts that make that effort meaningful. Contract Before Code compiles historical failures into reusable contract families whose preflight probes turn past failures into executable planning constraints. Controlled service/API-clone reruns show a promising observed improvement, while the audit identifies weaker compilation quality outside clone tasks. The present transfer evidence is within task families. Broader cross-repository evaluation, more runs, and a fully specified gating interface are needed to establish when these benefits extend beyond the evaluated setting.

Limitations

Contract Before Code depends on the quality and representativeness of historical failures. If the corpus lacks a relevant failure mode, the system cannot compile a useful contract; if a failure is misdiagnosed, the probe may block useful work or suggest an irrelevant repair. Evaluation must therefore report probe precision, false-block rate, and failure cases, not only task success.

The end-to-end study uses four related service/API-clone tasks, one model, and one harness. Its 32 runs per method do not establish a stable effect size, and two observed false blocks provide a limited estimate of gate reliability. The contract store retains other agents’ trajectories from the same task family; this supports within-family reuse but does not demonstrate held-out-family or cross-repository transfer. The weak-label candidate rates are not validated root-cause prevalence estimates. The operating-point discrepancy described in Section 5.7 also limits conclusions from the sensitivity study.

The current description specifies the compilation schema and integration components, but does not fully specify the abstraction prompt, retrieval scoring, repository-value binding rules, or the exact threshold for a gated edit. These details remain necessary for exact reproduction. Passing a probe checks one contract, not the full task: a passed gate should not suppress later verifiers, and contracts

should remain defeasible planning constraints.

Ethical Considerations

The approach is best suited to environments where probes can run safely and cheaply. Tasks involving irreversible actions or private data require sandboxed or mock execution (Ruan et al., 2024).

Historical agent trajectories can contain credentials, private URLs, repository structure, and sensitive environment information. Deployment should redact secrets and identifying values before storing logs or sending failure signatures to a compilation model. Reusable templates should retain placeholders rather than historical credentials; repository-specific values should be bound from the current authorized environment, with least-privilege access and controlled retention. These are deployment requirements, not a claim that the benchmark evaluation validates a complete sanitization or privacy mechanism. Probe outputs must be subject to the same protections as their inputs.

References

- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *International Conference on Learning Representations*.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2025. *SWE-Bench Pro: Can ai agents solve long-horizon software engineering tasks?* Preprint, arXiv:2509.16941.
- Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Daniel Wang, Xiangyi Li, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Derek Chen, Pranav Raja, and 7 others. 2026. *SWE-Marathon: Can agents autonomously complete ultra-long-horizon software work?* Preprint, arXiv:2606.07682. Trajectory corpus shared by the authors for research use.
- 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? In *International Conference on Learning Representations*.

- Wonjoong Kim, Sangwu Park, Yeonjun In, Sein Kim, Dongha Lee, and Chanyoung Park. 2025. [Beyond the final answer: Evaluating the reasoning trajectories of tool-augmented agents](#). *Preprint*, arXiv:2510.02837.
- Yuanyang Li, Xue Yang, Longyue Wang, Weihua Luo, and Hongyang Chen. 2026. [ComplexMCP: Evaluation of llm agents in dynamic, interdependent, and large-scale tool sandbox](#). *Preprint*, arXiv:2605.10787.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2023. [AgentBench: Evaluating llms as agents](#). *Preprint*, arXiv:2308.03688.
- Zeyao Ma, Jing Zhang, Xiaokang Zhang, Jiaxi Yang, Zongmeng Zhang, Jiajun Zhang, Yuheng Jing, Lei Zhang, Mingze Li, Wenting Zhao, Junyang Lin, and Binyuan Hui. 2026. [Scaling agentic verifier for competitive coding](#). *Preprint*, arXiv:2602.04254.
- Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. [MemGPT: Towards llms as operating systems](#). *Preprint*, arXiv:2310.08560.
- Joon Sung Park, Joseph C. O’Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *ACM Symposium on User Interface Software and Technology*.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Ruobing Tian, Ruijie Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Tool-LLM: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*.
- Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. 2024. Identifying the risks of llm agents with an llm-emulated sandbox. In *International Conference on Learning Representations*.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. [Voyager: An open-ended embodied agent with large language models](#). *Preprint*, arXiv:2305.16291.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. OpenHands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*.
- Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. [OS-World: Benchmarking multimodal agents for open-ended tasks in real computer environments](#). *Preprint*, arXiv:2404.07972.
- 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. In *Advances in Neural Information Processing Systems*.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. [\$\tau\$ -bench: A benchmark for tool-agent-user interaction in real-world domains](#). *Preprint*, arXiv:2406.12045.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.
- Jusheng Zhang, Jing Yang, Kaitong Cai, Ziliang Chen, Yongsun Zheng, Kwok Yan Lam, Liang Lin, and Keze Wang. 2025. [Failure-driven workflow refinement](#). *Preprint*, arXiv:2510.10035.
- Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*.