SOFTWARE WORLD MODELS: FROM CONSEQUENCE PREDICTION TO DECISION VALUE

Anonymous authors
Paper under double-blind review

ABSTRACT

A coding agent may safely modify one repository while silently breaking downstream services, libraries, or datastores that depend on it. Exhaustively running integration tests after every agent action is impractical, so the agent must predict these failures before executing them. Existing software world models predict the agent's own observations, while static change-impact analysis only identifies where a change may propagate. We instead introduce the *Software World Model* (SWM), which models the broader system affected by a code change and predicts its *blast set*: the components that the change will break. SWM follows three stages: explore, learn, and act. *Explore* executes candidate changes from restored system states, prioritizing regions where observed failures contradict the dependency graph. *Learn* fine-tunes a language model on these execution outcomes to predict downstream breakage. *Act* converts sampled predictions into per-consumer break probabilities for change ranking, proactive migration, and deciding when another execution is worth its cost. On held-out synthetic systems, SWM improves blast-set F1 from 0.431 for static reachability to 0.571 ± 0.037, more than halves ranking regret, and improves migration return at all nine evaluation checkpoints. The two methods are complementary: reachability is stronger on dependencies represented in the graph, while SWM recovers failures caused by couplings the graph misses. Experiments on held-out real libraries further show that predicting structured failure outcomes, rather than only scalar risk, is important for downstream decision quality.

1 INTRODUCTION

A real software system comprises interdependent repositories, services, libraries, and datastores. These *components* are coupled through APIs, schemas, shared data, and configuration. A coding agent, however, typically works inside one repository and verifies its changes with that repository's tests (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025). A change can therefore be locally correct while silently breaking a downstream *consumer*, a component that depends on the changed interface. Every local check can pass while that consumer fails when the system runs. Before landing a change, the agent needs to anticipate its *blast set*, the components that the change breaks.

Executing every potentially affected component would reveal the blast set, but does not scale (Memon et al., 2017; Machalica et al., 2019). Classical change-impact analysis instead identifies what a change may affect from an extracted dependency graph (Bohner & Arnold, 1996; Rothermel & Harrold, 1997), but not whether a particular consumer actually breaks. Our static baseline, typed reachability (reachability below), marks consumers connected to the changed interface by dependency paths. It cannot distinguish a consumer that tolerates a change from one that does not, and misses failures through couplings the extractor did not recover. We call couplings missing from the extracted graph *hidden channels*. A *silent break* is a downstream failure missed by repository tests. More fundamentally, any predictor computed from the extracted graph and change alone carries an error floor given by variation in outcomes among systems sharing that observation (Remark 1).

Existing software world models address the complementary problem: they learn what the agent itself will observe. CWM learns from execution traces and agent interactions (Copet et al., 2025), SWE-World predicts execution and test feedback (Sun et al., 2026), and Qwen-AgentWorld predicts tool-call observations across agent domains (Zuo et al., 2026). Their state is the agent-visible envi-

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.

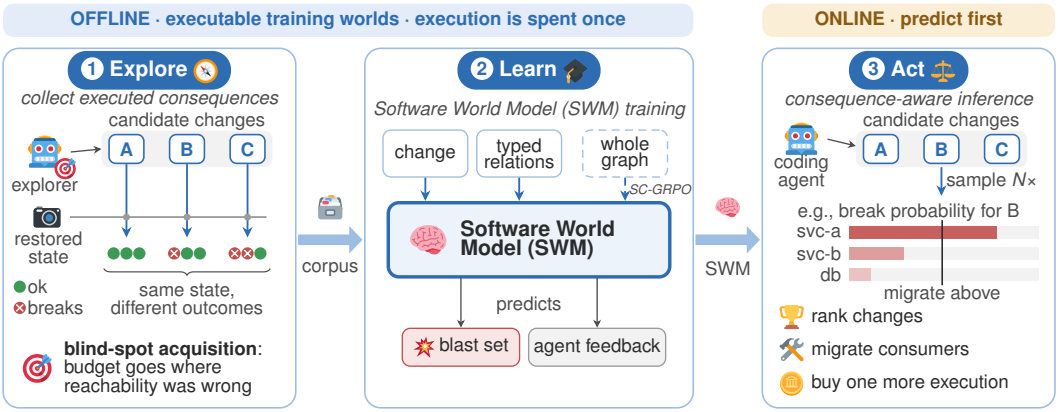


Figure 1: **SWM closes the loop from exploring executable software worlds, to learning their change-conditioned consequences, to improving the coding agent that acts in them.**

ronment, their action is a tool invocation, and their target is the returned observation. For a locally consistent change, however, that observation can be correct while a downstream service outside the workspace breaks. We instead model the *software system the agent changes*, rather than only the environment it observes.

This shift raises three coupled challenges. *Experience*: downstream consequences are revealed only by executing affected components, and ordinary agent trajectories observe only one change from each state. *Representation*: consequences depend on both what the change does and how other components use the changed interface, under incomplete structural observability. *Decision use*: the learned knowledge must improve the agent's decision before the costly execution that would otherwise reveal the failure.

We introduce the *Software World Model* (SWM), a learned consequence predictor, and an *Explore, Learn, Act* framework to build and use it (Figure 1). *Explore* executes counterfactual changes from restored software states to collect their downstream consequences, concentrating limited execution on informative interventions. *Learn* trains the coding model on these executed consequences, using the proposed change and typed software relations to learn change-conditioned system dynamics. *Act* returns this learned world knowledge to the same coding agent before future changes, allowing it to anticipate affected consumers rather than discover them only after execution. The learned consequences are reused for later decisions (Section 3).

Different exploration policies produce different training experience, with targeted exploration most useful at small budgets. Training on executed consequences improves the same coding agent under a fixed inference interface: it identifies more affected consumers and ships fewer downstream failures on held-out software worlds. Further analysis shows that explicit structure and learned dynamics are complementary. Reachability is strongest on recorded dependency channels, while learned experience helps where structure is incomplete. Downstream decision value also depends on *which* errors a world model makes, not forecast accuracy alone (Section 4).

Our contributions are:

- **Software world modeling.** We formulate system-level consequence prediction for coding agents and characterize why partial software structure alone cannot determine downstream behavior (Remark 1).
- **Explore, Learn, Act.** We introduce a framework that actively collects counterfactual software experience, trains SWM on change-conditioned consequences, and uses the learned dynamics to improve the agent's decisions before execution (Section 3).
- **Agent improvement.** We show that learning from executed consequences improves the same coding agent on held-out software worlds, and analyze how exploration, observability, and model errors determine downstream decision value (Section 4).

2 PROBLEM FORMULATION

2.1 THE SOFTWARE WORLD

A software world is an executable system with checks and available actions. Its environment $\mathcal{E}$ evolves a true state $s_t = (G_t^\star, x_t)$. Here $G_t^\star$ is the true System-Surface Graph, a typed, directed multigraph $G = (V, E, \rho, \omega)$ with nodes V, edges E, and two layers. The system layer holds components and shared resources; the surface layer holds their interfaces, such as endpoints, symbols, and schemas. The map ω assigns each surface node its owner, and $\rho : E \to \mathcal{R}$ labels each edge with its dependency channel, the kind of coupling it represents (Appendix D.3). The behavioural part is a latent execution state $x_t : V \to \{\texttt{ok}, \texttt{broken}\}$, revealed only by running something.

2.2 AGENT, ACTIONS AND OBSERVATIONS

An agent follows $\pi(a_t \mid h_t)$ with history $h_t = (o_0, a_0, \ldots, o_t)$ and actions $\mathcal{A} = \mathcal{A}_{\text{int}} \cup \mathcal{A}_{\text{probe}} \cup \mathcal{A}_{\text{rep}}$: repository changes (interventions), execution checks (probes), and repairs. The agent observes

$$o_t = \big(G_t,\ o_t^{\text{cons}}\big), \qquad G_t = \text{ex}(s_t), \tag{1}$$

where G_t extracts dependencies across the system but omits component code and some couplings (the hidden channels), and o_t^{cons} records feedback from checks already run. Changes keep the edited repository's code and tests consistent, so local checks pass by construction; downstream checks do not run by default. Training worlds can be reset, with each executed intervention and its checks consuming one unit of budget b. Deployment has no reset: local checks cost nothing extra, while each additional check costs c_{chk} (Appendix D.9).

2.3 SOFTWARE WORLD MODELING

The problem is to anticipate a candidate change's system-wide consequences before executing it. Define its *blast set* as the components whose execution state changes,

$$B(s_t, a_t) \ = \ \{\, v \in V \ : \ x_{t+1}(v) \neq x_t(v) \text{ when } a_t \text{ is executed in } s_t \,\}. \tag{2}$$

Given h_t and a_t, a software world model estimates

$$\boxed{p\big(\, B(s_t, a_t),\ o_{t+1}^{\text{cons}} \,\big|\, h_t,\ a_t \,\big)} \tag{3}$$

over the blast set and check feedback. At the *forecast step*, a change lands; its prediction is made beforehand and scored against all executed downstream effects. The label $\Delta s_t = B(s_t, a_t)$ denotes this blast set rather than the full state difference.

The predictive objective is to assign high likelihood to executed consequences,

$$\max_{p} \ \ \mathbb{E}_{(h_t,\, a_t) \sim \mathcal{D}_{\text{f}}} \Big[\, \log p\big(\, B(s_t, a_t),\ o_{t+1}^{\text{cons}} \,\big|\, h_t,\ a_t \,\big) \Big], \tag{4}$$

where $\mathcal{D}_{\text{f}}$ contains forecast-step records from the executed corpus $\mathcal{D}$. Its practical value is whether these forecasts improve candidate selection and consumer protection before a change lands; Appendix D.7 specifies the evaluation criteria.

Learning and using this distribution raises three unique challenges:

- **C1. Experience.** Executed consequences are costly to acquire. A limited budget must expose failures missed by reachability and compare alternative changes from the same state, beyond one observed trajectory.
- **C2. Representation.** Predicting breakage requires relating changes to consumer behavior despite missing couplings. The graph alone leaves an irreducible error floor (Remark 1); learning must use executed outcomes and represent residual uncertainty.
- **C3. Decision use.** Before execution reveals failures, comparable consumer risks must support change selection, preventive migration, and decisions about paid checks.

Algorithm 1 The SWM framework. Explore and Learn run once, offline; Act runs online before each change.

Input: training worlds, budget b, explorer π_{e}, pretrained language model M, initial executed corpus $\mathcal{D}$, sample count N, costs $c_{\mathrm{act}}, c_{\mathrm{dmg}}, c_{\mathrm{chk}}$.

Explore (offline; Section 3.1)

while the explorer has spent fewer than b executions **do**
  choose states whose pending interventions have the highest $\hat{d}$ on $\mathcal{D}$ (7)
  at each chosen state s_t, π_{e} proposes A_t; execute every $a \in A_t$ from the restored s_t and add $\mathcal{B}(s_t)$ to $\mathcal{D}$ (5)
end while

Learn (offline; Section 3.2)

fine-tune M on $\mathcal{D}$ into W_ϕ by supervision or group relative policy optimization (GRPO; 10)

Act (online, before each change; Section 3.3)

the agent proposes candidate changes A_t in its repository
sample W_ϕ N times per candidate; select $a_t^\star$ by predicted consequence (12)
compute β_u for $a_t^\star$ (11); while a check pays (Proposition 1), check the consumer with the largest expected saving and condition β on the answer
migrate each u with $\beta_u > c_{\mathrm{act}}/c_{\mathrm{dmg}}$; land $a_t^\star$

3 Explore, Learn, Act: The SWM Framework

To address these challenges, we propose an *Explore, Learn, Act* framework that builds and uses SWM (Figure 1; Algorithm 1). For the experience challenge, *Explore* executes counterfactual changes from restored training states, concentrating limited execution on informative downstream consequences (Section 3.1). For the representation challenge, *Learn* trains a consequence predictor that combines each change with typed dependencies and observed health to anticipate system-wide effects (Section 3.2). For decision use, *Act* brings these learned consequences back to the coding agent to guide candidate selection, preventive migration, and additional checking before a change is committed (Section 3.3). Explore and Learn run offline; their executed experience supports later agent decisions. The primary study uses the trained weights directly as the agent (Section 4.1); the decision rules below separately test how sampled predictions guide action.

3.1 Explore: branch from restored states where reachability fails

Explore answers the experience challenge by building the corpus that Learn trains on. An explorer π_{e} is a policy that proposes changes in a training world; we evaluate the executed records it collects within budget b. Section 4 compares exploration policies at matched budgets; Appendix D.5 describes exploration on real software. The exploration stage combines two operations. Blind-spot acquisition, which selects executions where previous outcomes contradicted reachability, picks the states to branch from, and at each, counterfactual branching has the explorer propose candidate changes that the environment executes in full. Counterfactual branching supplies same-state comparisons; the real-library acquisition comparison appears in Figure 6.

Counterfactual branching. Given a state, the explorer proposes $A_t = \{a_t^1, \ldots, a_t^K\}$ spanning distinct roles (the action it prefers, a plausible alternative, a risky one, a subtly wrong one, a recovery), and the environment executes *all* of them from the same restored state,

$$\mathcal{B}(s_t) \;=\; \left\{\left(s_t,\; a_t^k,\; \Delta s_t^k,\; o_{t+1}^{\mathrm{cons},k}\right)\right\}_{k=1}^{K}. \tag{5}$$

We call $\mathcal{B}(s_t)$ a *branch set*. Branch sets use the explorer as a sampler of candidates and make observable the quantity that ranking compares. Because the environment executes every a_t^k from the same restored s_t, the corpus records, for any two candidates,

$$\left|\Delta s_t^i\right| - \left|\Delta s_t^j\right| \;=\; \left|B(s_t, a_t^i)\right| - \left|B(s_t, a_t^j)\right| \tag{6}$$

with the state held fixed. No confound between what an action does and the state a policy happened to reach enters the comparison. A single-trajectory corpus cannot support Equation 6: it observes one action per state, so a difference in outcomes conflates the action with the state.

Blind-spot acquisition. The structural limit in Remark 1 motivates collecting outcomes that the graph alone does not determine. Let $R(G, a)$ denote typed reachability (Equation 15). Writing $\mathcal{P}$ for the pending interventions, the explorer selects

$$\mathcal{A}^\star \in \arg\max_{\mathcal{A}\subseteq\mathcal{P},\,|\mathcal{A}|\le b} \sum_{a\in\mathcal{A}} \hat{d}\big(\mathrm{tgt}(a)\big), \qquad \hat{d}(v) \;=\; \widehat{\Pr}\Big[\, B(s,a) \neq R(G,a) \;\Big|\; \mathrm{tgt}(a) = v \,\Big], \quad (7)$$

where the explorer estimates $\hat{d}$ from the executions already in $\mathcal{D}$. When Explore branches from the states of the selected interventions, each branch set spends K units of b, so the selection admits at most b/K states. Components with high $\hat{d}$ are reachability blind spots: past executions frequently disagreed with the graph-based prediction. The criterion needs no label for the candidates it selects: past disagreement between execution and reachability serves as a proxy for where the next label is informative. This rule differs from the decision-aware exploration policy in the primary experiments, which also uses model uncertainty and novelty (Appendix A).

3.2 Learn: fine-tune a language model on executed consequences

Consequence predictor. Learn fine-tunes a language model W_ϕ on executed outcomes. Given the history and a candidate change, it predicts the blast set and check feedback,

$$W_\phi : (h_t, a_t) \longmapsto \big(\widehat{\Delta s_t},\ \hat{o}^{\mathrm{cons}}_{t+1}\big), \quad (8)$$

where hats denote predictions. The forecast-step blast set is the primary target, and check and repair steps supply auxiliary labels (Appendix D.11). W_ϕ predicts one step ahead; recursive rollout is a separate extension (Appendix D.12).

Text input. The default predictor reads a text representation S^{sem}_t, formed by serialisation τ of the typed neighbourhood within r hops of the change (at most m edges, the retrieval budget), together with the action and the observed health η_t (component statuses revealed by earlier checks),

$$S^{\mathrm{sem}}_t = \tau\big(\mathcal{N}^{r,m}_t(a_t), a_t, \eta_t\big). \quad (9)$$

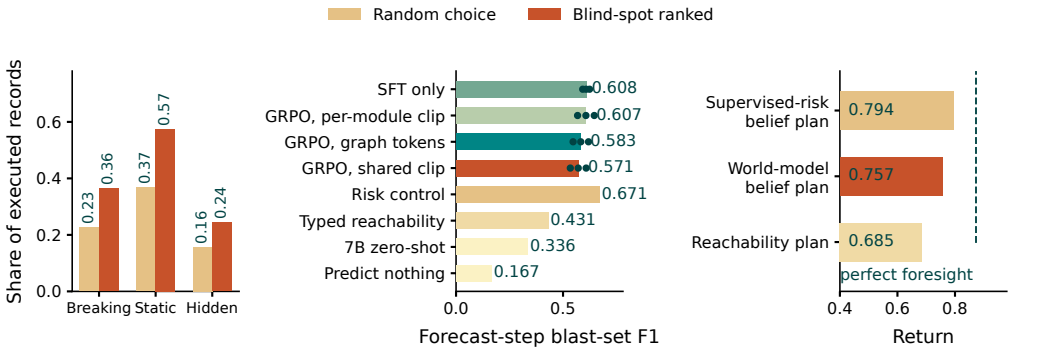
Typed relations distinguish calls, imports, reads, and subscriptions, which reachability reduces to reached or not; they also retain weaker traces of hidden channels. On real libraries, adding each consumer's usage of the changed symbol raises hidden-channel F1 from 0.142 to 0.198 (Figure 6a). The input ablation in Figure 8c adds dependency relations to the same observed graph to measure which information helps prediction. The model emits one structured object holding the predicted blast set, outcome class (no break, silent break, or a break detected by repository tests), check feedback, and the dependency channel explaining the failure, which puts candidate actions on a common scale (Qian et al., 2026).

Training on executed outcomes. A short supervised warmup teaches the output format. Its loss combines language modeling with auxiliary losses on the blast set, the outcome class and the fraction of healthy services; the decision rules do not use the last of these outputs (Appendix D.10). The model then undergoes group relative policy optimization (GRPO) (Shao et al., 2024) to a reward that a program scores against the executed transition,

$$R_{\mathrm{WM}}(\hat{y}) = w_f R_{\mathrm{format}} + w_s F_1\big(\widehat{\Delta s}, \Delta s\big) + w_o \mathbf{1}[\hat{c} = c] + w_m \tfrac{1}{2}\big(F_1^{\mathrm{tok}}(\hat{o}, o) + \mathrm{chan}(\hat{y})\big), \quad (10)$$

with format, blast-set, outcome-class and observation/channel terms (Appendix D.13); no learned reward model enters. Figure 8a compares this recipe with matched supervised training on the same corpus.

Graph-conditioned variant. This variant conditions the same language model on the whole extracted graph: a relational graph convolutional encoder (Schlichtkrull et al., 2018) reads the graph, and a few learned queries over its node embeddings produce *graph tokens*, continuous embeddings that precede the text input. Structure-coupled GRPO (SC-GRPO), our extension of GRPO, trains the encoder and the language model jointly on the reward of Equation 10, with one gradient-norm clip per module (Appendix D.13; Appendix D.10). The companion decision and mechanism analyses pool this variant with the default model (nine checkpoints); Table 8 compares the two head-to-head, and Appendix D.13 isolates the effect of per-module clipping.



(a) Explore (real libraries) (b) Learn (held-out synthetic) (c) Act (migration tasks)

Figure 2: Companion evidence across Explore, Learn, and Act. (a) Fractions of locally passing real-library perturbations that break a consumer, contradict static prediction, or expose a hidden channel. (b) Blast-set F1 on 90 held-out synthetic forecasts; dots mark the mean and mean ± three-seed half-range. (c) Explicit-belief planning on 40 migration tasks, with perfect foresight as a reference. The panels use separate evaluation populations.

3.3 Act: Consult the Predictor Before a Change Lands

Before landing a change, the agent proposes candidates A_t and samples W_ϕ for each candidate to rank changes, decide whether to pay for another check, and migrate consumers at risk. A preventive migration updates a consumer to remain compatible with the pending change; a repair restores a consumer after it has broken. Only the checks the agent buys cost execution.

Break probabilities. Over the consumer set C_t, sampling N predicted blast sets estimates each consumer's break probability β_u. We call these probabilities the *belief* and use the factorized approximation

$$\hat{p}\big(B \mid h_t, a_t\big) = \prod_{u \in C_t} \beta_u^{\mathbf{1}[u \in B]} \, (1-\beta_u)^{1-\mathbf{1}[u \in B]}, \qquad \beta_u = \tfrac{1}{N} \textstyle\sum_{n=1}^{N} \mathbf{1}\big[u \in \widehat{\Delta s}^{(n)}\big], \tag{11}$$

where $\mathbf{1}[\cdot]$ is an indicator. A structural predictor such as R supplies only $\beta_u \in \{0, 1\}$, which is the formal sense in which reachability cannot express a preference between two candidates it marks alike.

Ranking. W_ϕ imagines each candidate, and the agent lands only the best,

$$a_t^\star \;=\; \arg\max_{a \in A_t} U\big(W_\phi(h_t, a)\big), \tag{12}$$

with U the negative predicted blast size, or the predicted return for migration plans.

Preventive migrations and paid checks. The belief Equation 11 converts preventive action into a decision under uncertainty rather than a traversal of a set.

Proposition 1 (One-step decision rules under the belief)**.** *Let c_{act} be the cost of migrating a consumer, c_{dmg} the cost of leaving it broken, and c_{chk} the cost of checking it. Under the factorisation Equation 11 the expected-cost-minimising one-step choices are*

$$\textit{migrate } u \iff \beta_u > \frac{c_{\text{act}}}{c_{\text{dmg}}}, \qquad \textit{check } u \iff c_{\text{chk}} + \beta_u c_{\text{act}} < \min\big(c_{\text{act}},\, \beta_u c_{\text{dmg}}\big), \tag{13}$$

and the agent checks the consumer with the largest expected saving until no check pays.

After each check, the agent conditions the belief on the answer by reweighting the sampled blast sets that agree with it (Appendix C.3).

Both rules use the same break probabilities for preventive migration and paid checks. A set-valued predictor restricts them to $\beta_u \in \{0, 1\}$; probabilistic predictions instead support decisions at the cost thresholds in Equation 13 (Appendix C.3).

4 Experiments

4.1 Experimental Setup

Models and comparisons. All backbones in the main agent comparison have 3B parameters. We compare the base Qwen2.5-Coder-Instruct agent, its consequence-trained SWM counterpart, and our CWM and Qwen-AgentWorld implementations on synthetic worlds and held-out real libraries. Baseline adaptation is described in Appendix D.6. The synthetic ablations hold the base checkpoint, training recipe, and acting interface fixed while varying the training corpus. Random samples eligible targets; Coverage cycles through change types; Expert uses true-graph consumer information; Decision-aware combines uncertainty, disagreement with reachability, and novelty.

Matched synthetic evaluation. A training record pairs one executed change with its consequences. Each 360-record corpus has three training seeds. Random and decision-aware exploration also use 100- and 200-record budgets. These 24 trained arms and the base checkpoint run on the same 40 tasks from seven held-out worlds, yielding 1,000 episodes. Each agent first lists candidate consumers for a fixed change, then selects consumers to migrate. Decoding is greedy, and execution accounting is identical. The trained checkpoint acts directly, without external predictions or additional policy fine-tuning.

Metrics and uncertainty. Return is $J = 1 - 0.25n_{\text{broken}} - 0.05(n_{\text{fixes}} + 1)$: the counts are services left broken and preventive migrations, and the extra action ships the change. Candidate recall is the fraction of truly affected consumers in the first call's list. The main table reports means; standard deviations are retained in Table 5. Synthetic ablations average three seeds within each task and bootstrap paired return differences by world, using 10,000 resamples. These intervals apply to the synthetic ablations, not the real-library comparison. Full task accounting and seeds appear in Appendix A.2; implementation and metrics are in Appendix D.7. Companion prediction and explicit-belief studies use separate populations and the configuration in Table 13.

4.2 Main Results: Does Learning a Software World Improve the Agent?

Exploration determines the experience available for learning. Different policies expose the learner to different software consequences. Targeted exploration concentrates execution on breaking and locally invisible transitions (Figure 3a). These corpus differences also change prediction at matched budgets: with 100 records, decision-aware acquisition yields blast-set F1 of 0.529, versus 0.399 for random acquisition. The complete curves show that prediction quality need not grow monotonically with the record count (Appendix A.1).

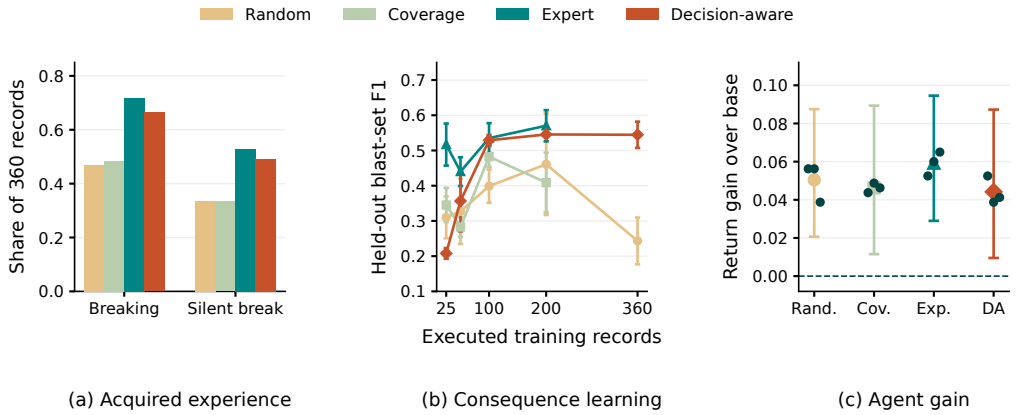
Learning executed consequences improves the same coding agent. Consequence training raises synthetic mean return from 0.721 to 0.773 (Table 1). All four full-budget corpora improve return: each paired interval is positive, as are the point gains for all twelve training seeds (Appendix A.2).

Table 1: Mean agent performance on synthetic and real-library worlds. Return subscripts give gains over the base agent. Standard deviations appear in Table 5.

Agent	Synthetic			Real libraries		
	Return ↑	Recall ↑	Broken ↓	Return ↑	Recall ↑	Broken ↓
Base agent	0.721	0.444	0.600	0.453	0.276	0.387
CWM	$0.740_{+0.019}$	0.533	0.499	$0.548_{+0.095}$	0.381	0.351
Qwen-AgentWorld	$0.754_{+0.033}$	0.559	0.447	$0.571_{+0.118}$	0.446	0.360
SWM (Ours)	$0.773_{+0.052}$	0.586	0.363	$0.613_{+0.160}$	0.499	0.309

Differences are computed before rounding the displayed means. CWM (Copet et al., 2025) and Qwen-AgentWorld (Zuo et al., 2026) are adapted as described in Appendix D.6.

The trained agent anticipates more failures



(a) Acquired experience (b) Consequence learning (c) Agent gain

Figure 3: Exploration, consequence learning, and agent improvement. The prediction study in (b) reports three-seed half-ranges; the agent study in (c) reports paired world-bootstrap 95% intervals and individual seed means.

before acting. On both datasets, the trained agent identifies more affected consumers and leaves fewer services broken. Return also charges for migrations, so its gain reflects the balance between prevention cost and avoided damage. These measurements support improved anticipation and action without isolating their individual contributions (Appendix A.3).

The same performance pattern holds on real libraries. SWM has higher mean return and candidate recall, with fewer broken services, than the base agent and both world-model baselines on each dataset (Table 1). On held-out real libraries, mean return rises from 0.453 to 0.613. This extends the observed benefit beyond procedural systems. Real-library prediction and exploration diagnostics appear in Appendix A.4.

4.3 Ablation: What Makes Software-World Learning Work?

Targeted exploration helps most at the smaller tested budgets. Decision-aware acquisition has a larger point gain than random acquisition at 100 and 200 records. At 100 records, only its interval against the base agent excludes zero. The gap narrows with more experience, and the ordering reverses at 360 records (Figure 7). The pattern suggests a sample-efficiency benefit. Intervals compare each policy with the base agent, rather than with the other policy. Full results are in Appendix B.1.

Table 2: Consequence training improves the base agent on 40 matched tasks. Trained rows average three seeds at 360 records; brackets give paired world-bootstrap 95% intervals for return differences, computed before rounding.

Training corpus	Return ↑	Δ return [95% interval]
Base agent	0.721	n/a
Random	0.772	+0.050 [+0.021, +0.088]
Coverage	0.768	+0.046 [+0.012, +0.089]
Expert	0.780	+0.059 [+0.029, +0.095]
Decision-aware	0.765	+0.044 [+0.010, +0.087]

Several exploration policies supply useful training data. All four full-budget corpora improve return under the same training recipe (Table 2). Expert exploration gives the largest gain but uses privileged graph information. Random acquisition also helps, so the observed benefit is not confined to the targeted acquisition rule. This tests the consequence-training pipeline; an ordinary fine-tuning control would be needed to isolate its objective.

Richer state helps only when relevant information remains accessible. Adding runtime relations lowers learned F1 at the default input budget. Doubling that budget raises runtime-input F1 from 0.469 to 0.592, above the smaller static input. The pattern is consistent with competition for limited context (Appendix B.2). Training-recipe comparisons, including graph tokens and gradient clipping, are reported separately in Appendix B.3.

4.4 Analysis: What Do We Learn About Software World Modeling?

Structure and learned experience resolve complementary uncertainty. Reachability has higher F1 on direct propagation, while learned predictions recover more hidden-channel failures in both evaluated hidden-channel populations. This contrast supports combining explicit dependencies with executed experience to predict failures the graph misses (Appendix C.1).

Prediction accuracy alone does not determine decision value. At matched blast-set F1, learned beliefs produce better migration plans than a noise-perturbed oracle, but incur higher ranking regret. Pairwise ranking accuracy also correlates more strongly with regret than aggregate set F1. The reversal shows why decision evaluation must account for where prediction errors occur (Appendix C.2).

The value of prediction depends on the alternatives to execution. The main agent comparison favors SWM, but companion studies expose its boundaries. A supervised risk scorer has higher in-distribution forecast F1 and belief-plan return. Cheap checks of every consumer can also outperform the learned belief plan. These comparisons use different interfaces from the main agent evaluation. Their mixed outcomes bound the benefit by the available structure, execution cost, and decision rule (Appendix C.3).

5 Related Work

Agent World Models. Learned world models let agents train inside or consult imagined futures (Richens et al., 2025; Ha & Schmidhuber, 2018; LeCun et al., 2022; Hafner et al., 2025a;b; Gu et al., 2024; Chae et al., 2025). Software world models predict execution or tool feedback (Copet et al., 2025; Sun et al., 2026; Zuo et al., 2026; Guo et al., 2025; Zhang et al., 2026), while executable environments supply experience for agent training (Jimenez et al., 2024; Yang et al., 2025; Pan et al., 2025; Badertdinov et al., 2025). Our target is the downstream system affected by a change, including failures outside the agent's workspace. We learn these consequences from execution and evaluate their value for subsequent decisions.

Software Change Modeling. Change-impact analysis and regression-test selection use dependencies to identify potentially affected components (Bohner & Arnold, 1996; Rothermel & Harrold, 1997; Memon et al., 2017; Salis et al., 2021; Reyes et al., 2024). Graph-conditioned language models provide structural context through graph representations (Perozzi et al., 2024; He et al., 2024; Hu et al., 2025; Chen et al., 2024; Li et al., 2023). SWM uses this structure to predict whether a particular change actually breaks a consumer. Its evaluation connects to decision-focused prediction, where quality depends on the decisions a forecast induces (Elmachtoub & Grigas, 2022).

6 Conclusion

Coding agents should reason beyond the repository they edit. They should also anticipate how their changes affect the broader software system. We introduced Software World Models and an *Explore–Learn–Act* framework for this goal. Our experiments show that learning executed consequences improves the same coding agent under a fixed inference interface. The trained agent identifies more affected consumers and leaves fewer downstream services broken. We also find that structure and experience provide complementary information. Explicit dependencies are reliable when the relevant structure is observed, while learned dynamics help when that structure is incomplete. Finally, forecast accuracy alone does not determine decision value. The location and type of prediction errors also matter. These results point toward coding agents that do more than react to failures. By exploring and learning their software worlds, agents can anticipate the consequences of future changes before acting.

AI USE STATEMENT

Generative AI tools were used to assist with code implementation, debugging, experiment orchestration, and language editing. All generated code was reviewed by the authors and validated through the project's test and execution pipelines. AI tools were not used to generate experimental labels or reported results: software-world consequences were obtained from executed environments, and all reported numbers were computed from saved experimental outputs. The authors take full responsibility for the methodology, analysis, claims, and manuscript.

ETHICS STATEMENT

This work studies coding agents that predict the downstream consequences of software changes. Our experiments use procedurally generated systems and permissively licensed open-source software; they involve no human subjects, personal data, or user-generated content. The proposed techniques may help identify cross-component failures before software changes are deployed, but similar capabilities could also reveal structural weaknesses in software systems. Our experiments are restricted to systems and environments under the experimenter's control. We encourage deployment only with appropriate authorization, access controls, and human oversight for consequential software changes.

REPRODUCIBILITY STATEMENT

Training and evaluation worlds are disjoint, corpus generation is seeded, and each reported training condition retains its data split, configuration, random seeds, and model checkpoint. Executed transitions—rather than model-generated judgments—serve as supervision for downstream consequences. For the main agent experiments, we retain per-task outputs and report results across training seeds with world-clustered uncertainty estimates. The appendices document the software worlds and graph schema, exploration and evaluation protocols, model training details, decision accounting, and complete experimental results.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 26420–26466, 2025. doi: 10.52202/085713-0788. URL https://arxiv.org/abs/2505.20411.

Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996. ISBN 9780818673849.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a00548031e4647b13042c97c922fadf1-Abstract-Conference.html.

Runjin Chen, Tong Zhao, Ajay Kumar Jaiswal, Neil Shah, and Zhangyang Wang. LLaGA: Large language and graph assistant. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 7809–7823. PMLR, 2024. URL https://proceedings.mlr.press/v235/chen24bh.html.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Adam N. Elmachtoub and Paul Grigas. Smart predict, then optimize. *Management Science*, 68(1): 9–26, 2022. doi: 10.1287/mnsc.2020.3922.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, et al. Is your llm secretly a world model of the internet? model-based planning for web agents. *arXiv preprint arXiv:2411.06559*, 2024.

Shangmin Guo, Omar Darwiche Domingues, Raphaël Avalos, Aaron Courville, and Florian Strub. World modelling improves language model agents. *arXiv preprint arXiv:2506.02918*, 2025.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018. URL https://arxiv.org/abs/1803.10122.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640(8059):647–653, 2025a. doi: 10.1038/s41586-025-08744-2. URL https://www.nature.com/articles/s41586-025-08744-2.

Danijar Hafner, Wilson Yan, and Timothy Lillicrap. Training agents inside of scalable world models. *arXiv preprint arXiv:2509.24527*, 2025b.

Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-Retriever: Retrieval-augmented generation for textual graph understanding and question answering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 132876–132907, 2024. doi: 10.52202/079017-4224.

Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. GRAG: Graph retrieval-augmented generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 4145–4157. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.232. URL https://aclanthology.org/2025.findings-naacl.232/.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Yann LeCun et al. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review*, 62(1):1–62, 2022.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19730–19742. PMLR, 2023. URL https://proceedings.mlr.press/v202/li23q.html.

Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 91–100, 2019. doi: 10.1109/ICSE-SEIP.2019.00018.

Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 233–242, 2017. doi: 10.1109/ICSE-SEIP.2017.16.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Bryan Perozzi, Bahare Fatemi, Dustin Zelle, Anton Tsitsulin, Mehran Kazemi, Rami Al-Rfou, and Jonathan Halcrow. Let your graph do the talking: Encoding structured data for llms. *arXiv preprint arXiv:2402.05862*, 2024. URL https://arxiv.org/abs/2402.05862.

Cheng Qian, Emre Can Acikgoz, Bingxuan Li, Xiusi Chen, Yuji Zhang, Bingxiang He, Qinyu Luo, Gokhan Tur, Dilek Hakkani-Tür, Yunzhu Li, and Heng Ji. Current agents fail to leverage world model as tool for foresight. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13686–13723. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.623. URL https://aclanthology.org/2026.acl-long.623/.

Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. BUMP: A benchmark of reproducible breaking dependency updates. In *Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2024. doi: 10.1109/SANER60148.2024.00024. URL https://arxiv.org/abs/2401.09906.

Jonathan Richens, David Abel, Alexis Bellot, and Tom Everitt. General agents contain world models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, 2025. URL https://arxiv.org/abs/2506.01622.

Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997. doi: 10.1145/248233.248262.

Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical call graph generation in python. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*, 2021. URL https://arxiv.org/abs/2103.00587.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *The Semantic Web – 15th International Conference, ESWC 2018*, volume 10843 of *Lecture Notes in Computer Science*, pp. 593–607. Springer, 2018. doi: 10.1007/978-3-319-93417-4_38.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, et al. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, et al. The openhands software agent sdk: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690*, 2025.

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

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 107402–107452, 2025. doi: 10.52202/085713-3239. URL https://arxiv.org/abs/2504.21798.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, Jian Xie, Yuxuan Sun, Boyu Gou, Qi Qi, Zihang Meng, Jianwei Yang, Ning Zhang, Xian Li, Ashish Shah, Dat Huynh, Hengduo Li, Zi Yang, Xuefei Cao, Lawrence Keunho Jang, Shuyan Zhou, Jiacheng Zhu, Huan Sun, Jason E. Weston, Yu Su, and Yifan Wu. Agent learning via early experience. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=N3dXUHY5dD.

Yuxin Zuo, Zikai Xiao, Li Sheng, Fei Huang, Jianhong Tu, Yuxuan Liu, Tianyi Tang, Xiaomeng Hu, Yang Su, Qingfeng Lan, Yantao Liu, Qin Zhu, Yinger Zhang, Bowen Yu, Haiquan Zhao, Haiyang Xu, Jianxin Yang, Jiayang Cheng, Junyang Wang, Lianghao Deng, Mingfeng Xue, Tianyi Bai, Yang Fan, Yubo Ma, Yucheng Li, Zeyu Cui, Zhihai Wang, Zhihui Xie, Zhuorui Ye, An Yang, Dayiheng Liu, Jingren Zhou, and Ning Ding. Qwen-agentworld: Language world models for general agents. *arXiv preprint arXiv:2606.24597*, 2026. URL https://arxiv.org/abs/2606.24597.

A Complete Evidence for Main Results

The main comparison is reported in Table 1. This appendix separates acquisition, controlled agent improvement, behavioral measurements, and real-library diagnostics. Companion prediction and explicit-belief studies use their own populations; Table 3 summarizes those results without pooling them with the main agent comparison.

Table 3: Exploration, forecast, and explicit-decision results. Each row uses its stated population and comparator.

Component	Evaluation	Score	Comparator or context	Gain or evidence
Explore	Real hidden-channel yield	0.244	random: 0.157	+55%
Learn	Synthetic blast-set F1 (supervised recipe)	0.608 ± 0.016	reachability: 0.431	+0.177
Learn	Synthetic hidden-channel F1	0.304	reachability: 0.189	+61%
Learn	Real-library hidden-channel F1	0.198 ± 0.038	reachability: 0.137	+45%
Act	Candidate-ranking regret ↓	0.593	reachability: 1.430	−59%
Act	Belief-plan return ↑	0.757	reachability: 0.685	+0.072
Analysis	Pairwise accuracy and regret correlation	$\rho = -0.93$	blast-set F1: −0.56	larger correlation magnitude

A.1 Exploration and Consequence Learning

Acquisition protocol. The decision-aware corpus uses the product acquisition rule

$$a^\star = \arg\max_{a \in \mathcal{P}} u_{\text{ep}}(a)\, u_{\text{dec}}(a)\, u_{\text{nov}}(a), \tag{14}$$

where u_{ep}, u_{dec}, and u_{nov} measure variation across trained models, disagreement with reachability, and lack of previously executed changes of the same type, respectively.

The decision-aware corpus starts with 60 random records, one per training world. Each of three rounds scores 960 candidate transitions (16 per world) and executes the top 100. For each consumer, model uncertainty is the standard deviation of predicted break probabilities across three model seeds (16 samples per model); the score averages these deviations. Reachability disagreement sums the absolute difference between mean break probability and the binary reachability indicator over consumers predicted to break by any model or marked by reachability. Novelty is $1/(1+n)$, where n counts prior executions with the same action kind, target kind, and fan-in bucket (the number of incoming observed edges, capped at three). Their product implements Equation 14. The 25- and 50-record prefixes contain only initial random data. Random, coverage, and expert corpora are separate exploration baselines; all four full-budget corpora contain 360 records. Random samples eligible targets without using true consumer information. Coverage additionally cycles through change types. Expert selects targets using the true graph, so it is a privileged synthetic-data reference, not a policy available under partial observability.

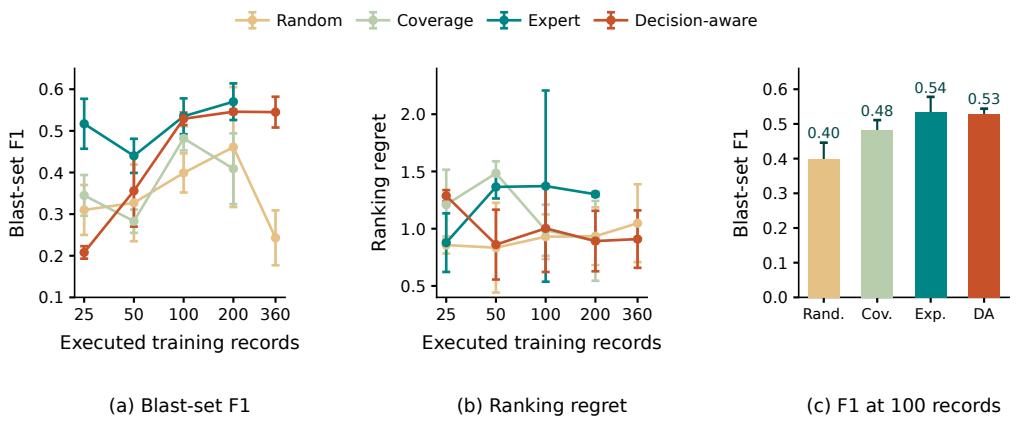
Companion prediction study. SWM-15 and SWM-19 evaluate held-out synthetic forecasts under the configuration in Table 13. Figure 4 includes every available exploration/budget cell; absent cells are unevaluated, not interpolated. The random 360-record endpoint uses the matched-recipe rerun in SWM-15.

A.2 Matched Agent Improvement

The controlled synthetic campaign evaluates 24 trained arms and the same base checkpoint on 40 tasks. Table 4 retains every training seed and its paired interval. Appendix D.8 specifies aggregation and the world-cluster bootstrap; Appendix B.1 reports means by corpus and budget.

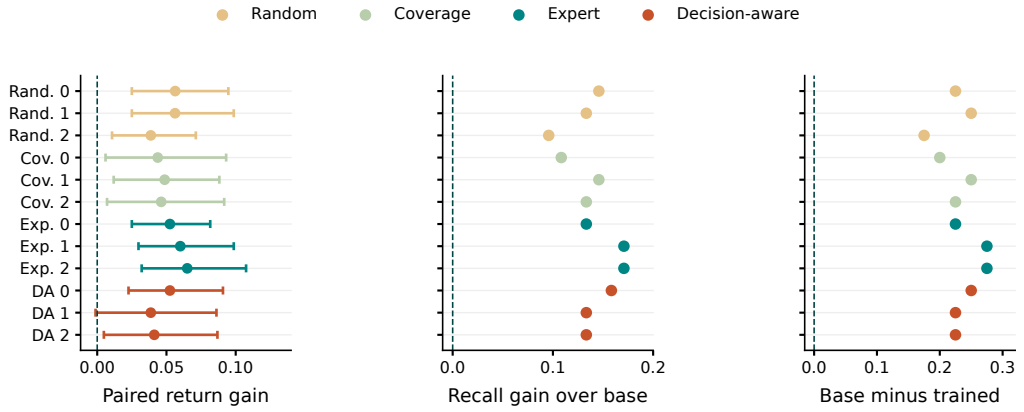
Table 4: All 24 trained arms on the same 40 tasks; base-agent return is 0.721. Intervals are the source campaign's per-seed paired world-cluster 95% intervals.

Corpus	Records	Seed	Return	Δ return	95% interval
Random	100	0	0.749	$+0.028$	$[-0.009, +0.072]$
Random	100	1	0.754	$+0.033$	$[+0.005, +0.067]$
Random	100	2	0.731	$+0.010$	$[-0.023, +0.052]$
Random	200	0	0.755	$+0.034$	$[+0.011, +0.065]$
Random	200	1	0.769	$+0.048$	$[+0.018, +0.086]$
Random	200	2	0.778	$+0.056$	$[+0.025, +0.094]$
Random	360	0	0.778	$+0.056$	$[+0.025, +0.095]$
Random	360	1	0.777	$+0.056$	$[+0.025, +0.099]$
Random	360	2	0.760	$+0.039$	$[+0.011, +0.071]$
Coverage	360	0	0.765	$+0.044$	$[+0.006, +0.093]$
Coverage	360	1	0.770	$+0.049$	$[+0.012, +0.088]$
Coverage	360	2	0.767	$+0.046$	$[+0.007, +0.092]$
Expert	360	0	0.774	$+0.052$	$[+0.025, +0.082]$
Expert	360	1	0.781	$+0.060$	$[+0.030, +0.099]$
Expert	360	2	0.786	$+0.065$	$[+0.032, +0.107]$
Decision-aware	100	0	0.768	$+0.046$	$[+0.014, +0.084]$
Decision-aware	100	1	0.762	$+0.041$	$[+0.009, +0.080]$
Decision-aware	100	2	0.764	$+0.043$	$[+0.009, +0.086]$
Decision-aware	200	0	0.768	$+0.046$	$[+0.007, +0.094]$
Decision-aware	200	1	0.782	$+0.061$	$[+0.031, +0.096]$
Decision-aware	200	2	0.779	$+0.058$	$[+0.029, +0.096]$
Decision-aware	360	0	0.774	$+0.052$	$[+0.023, +0.091]$
Decision-aware	360	1	0.760	$+0.039$	$[-0.001, +0.086]$
Decision-aware	360	2	0.762	$+0.041$	$[+0.005, +0.087]$



(a) Blast-set F1 (b) Ranking regret (c) F1 at 100 records

Figure 4: Exploration affects prediction and ranking differently. (a) Blast-set F1 and (b) ranking regret across execution budgets; points are three-seed means and error bars are half-ranges. (c) The 100-record comparison. Coverage and expert exploration were not evaluated at 360 records; the first 60 decision-aware records are random. Higher F1 and lower regret are better.



(a) Return gain (b) Recall gain (c) Failure reduction

Figure 5: All twelve full-budget training seeds improve agent return, candidate recall, and services left broken relative to the base checkpoint. Panel (a) includes paired world-cluster 95% intervals; (b–c) show per-seed mean changes on the same tasks.

A.3 Anticipation and post-change failures

Candidate recall is measured before preventive migrations. Broken-service counts measure the outcome after the change lands. All twelve full-budget seeds improve both measures relative to the base checkpoint, as well as return. These observations locate behavioral changes but do not decompose causal contributions of prediction, proposal, and action selection.

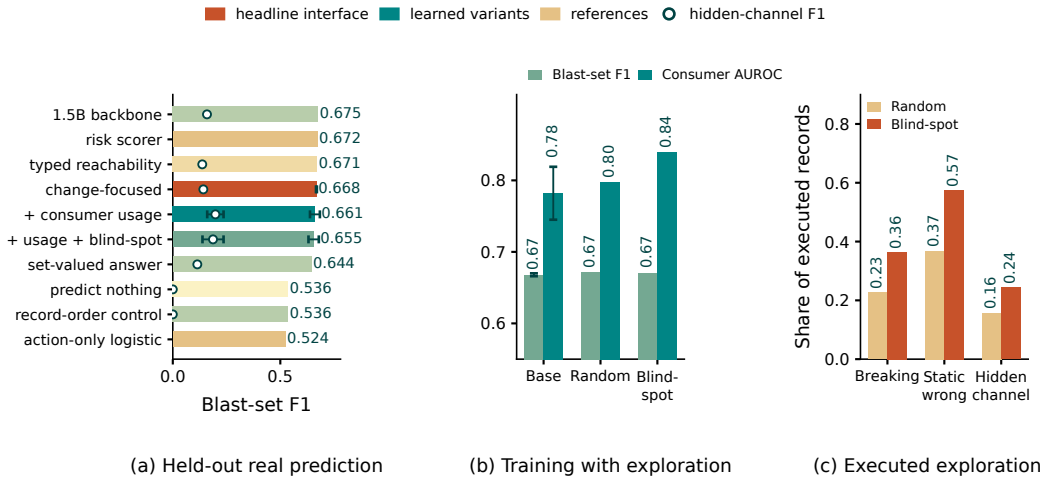
A.4 Real-library results

A set-valued prediction lists all affected consumers in one response; a per-consumer prediction answers separately for each consumer. Record-order context retains the input record's ordering, while change-focused context prioritizes information around the changed interface. Consumer-usage vari-

Table 5: Complete agent means and standard deviations for the main comparison. Values are means, with standard deviations in superscripts where available. Return subscripts show the mean difference from the base agent within each dataset. Recall measures affected-consumer coverage; broken services are counted per task.

Agent	Synthetic worlds			Real libraries		
	Return ↑	Recall ↑	Broken ↓	Return ↑	Recall ↑	Broken ↓
Base agent	0.721	0.444	0.600	0.453	0.276	0.387
CWM (Copet et al., 2025)	$0.740^{\pm 0.006}_{+0.019}$	$0.533^{\pm 0.007}$	$0.499^{\pm 0.020}$	$0.548^{\pm 0.005}_{+0.095}$	$0.381^{\pm 0.005}$	$0.351^{\pm 0.014}$
Qwen-AgentWorld (Zuo et al., 2026)	$0.754^{\pm 0.006}_{+0.033}$	$0.559^{\pm 0.022}$	$0.447^{\pm 0.027}$	$0.571^{\pm 0.005}_{+0.118}$	$0.446^{\pm 0.018}$	$0.360^{\pm 0.022}$
SWM (Ours)	$0.773^{\pm 0.008}_{+0.052}$	$0.586^{\pm 0.017}$	$0.363^{\pm 0.021}$	$0.613^{\pm 0.006}_{+0.160}$	$0.499^{\pm 0.015}$	$0.309^{\pm 0.018}$

Differences are computed before rounding the displayed means. Baseline training is described in Appendix D.6.



(a) Held-out real prediction (b) Training with exploration (c) Executed exploration

Figure 6: Real-library prediction and executed exploration. (a) Blast-set F1 on 984 held-out records; hollow markers show F1 on the 137 graph-missed breaking records. Error bars retain the reported three-seed spreads; unmarked learned variants are single runs. (b) Prediction after adding random or blind-spot exploration, with base-corpus seed spreads. (c) Acquisition yields from 450 candidate perturbations per policy on three training worlds, conditional on passing local checks. Consumer AUROC measures discrimination among broken and unbroken consumers.

ants additionally include how the dependent code uses the changed symbol. Consumer AUROC measures ranking of broken versus unbroken consumers; the Brier score measures squared error of their break probabilities.

Figure 6 connects prediction to executed exploration. Consumer-usage context improves hidden-channel F1 even though aggregate F1 changes little. The all-negative control also attains substantial aggregate F1, making the breaking-record and consumer-level diagnostics necessary for interpreting the overall score.

Table 7: All corpus budgets in the agent comparison, averaged over three seeds. Intervals are paired world-bootstrap 95% intervals against base-agent return 0.721.

Corpus	Records	Return	Δ return [95% interval]	Candidate recall	Broken services
Random	100	0.745	+0.023 [−0.008, +0.062]	0.573	0.342
Random	200	0.767	+0.046 [+0.018, +0.082]	0.594	0.342
Random	360	0.772	+0.050 [+0.021, +0.088]	0.569	0.383
Coverage	360	0.768	+0.046 [+0.012, +0.089]	0.573	0.375
Expert	360	0.780	+0.059 [+0.029, +0.095]	0.603	0.342
Decision-aware	100	0.765	+0.043 [+0.013, +0.081]	0.607	0.342
Decision-aware	200	0.776	+0.055 [+0.023, +0.094]	0.598	0.350
Decision-aware	360	0.765	+0.044 [+0.010, +0.087]	0.586	0.367

Exploration yields and probability quality. Local checks pass for 337 random and 332 blind-spot perturbations; these are the denominators for panel (c). Blind-spot selection increases all three yields, but its downstream benefit is metric-dependent: it gives the highest consumer AUROC, whereas the random round gives the lowest Brier score. Brier scores for the base, random-augmented, and blind-spot-augmented corpora are 0.177, 0.143, and 0.164; their hidden-channel F1 values are 0.142, 0.161, and 0.145. These differences prevent interpreting the AUROC gain as an improvement in every aspect of probability quality.

Table 6: Additional interface diagnostics on the same 984 held-out records. Columns retain the recorded "Extracted breaks" and "Silent" measures; blast-set and hidden-channel F1 appear in Figure 6a.

Interface	Extracted breaks	Silent
Set-valued, record-order context	0.417	0.841
Per-consumer, record-order context	0.000	1.000
Per-consumer, change-focused context	0.586	0.808
+ consumer usage	0.575	0.780
+ usage and blind-spot round	0.576	0.805
Alternative 1.5B backbone	0.689	0.799
Typed reachability	0.675	0.808

B Complete Ablations

B.1 Exploration policy and execution budget

The paired synthetic comparison changes the training corpus and budget under a fixed agent interface. The three training seeds are averaged within each task before world-cluster resampling. Prediction learning curves are in Appendix A.1; this subsection reports agent decisions.

B.2 State information and input budget

The input configurations in Figure 8c are nested: `tree` includes ownership relations; `imports` adds package and symbol dependencies; `calls` adds API calls and schema relations; `semantic` includes all extracted static relations; and `runtime` adds simulated deployment settings and calls missed by static extraction. These are graph-input configurations, not different prediction targets.

Coverage and input budget. For the five input configurations in panel (c), mean edge counts are 27.3, 42.2, 46.9, 70.0, and 96.7; traversal precision is 1.000, 1.000, 1.000, 0.778, and 0.803, while recall is 0.167, 0.250, 0.495, 0.574, and 0.691. The richer runtime input improves traversal coverage but reduces learned F1 at the default retrieval budget; increasing that budget recovers performance.

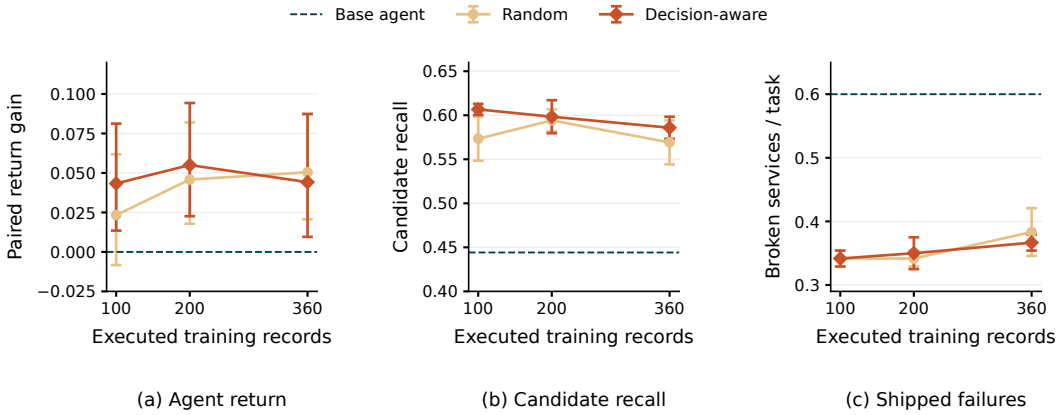


Figure 7: Training budget and agent behavior on the same 40 tasks. Return gains use paired world-bootstrap 95% intervals; candidate recall and broken-service counts show three-seed half-ranges, with the base agent as the dashed reference.

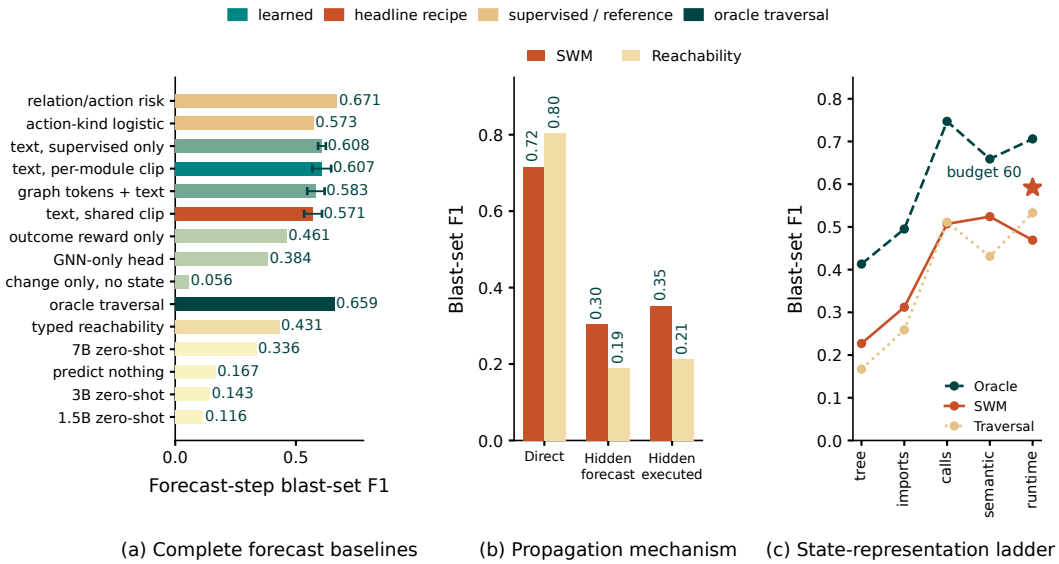


Figure 8: Forecast accuracy depends on both propagation mechanism and input information. (a) Baselines on 90 held-out synthetic forecast steps; error bars show three-seed half-ranges, and unmarked learned variants are single diagnostic runs. (b) Direct and hidden-channel F1 across nine checkpoints, with 360, 315, and 819 seed-record pairs in the three groups, respectively. (c) Nested graph inputs under identical labels; the star increases the runtime input budget from 30 to 60. Oracle traversal restores hidden graph channels.

B.3 Training recipe and representation

Figure 8 compares prediction recipes, propagation mechanisms, and input configurations. Relation/action logistic regression has the highest aggregate F1 in this companion study. Learned predictors improve on reachability for hidden channels but trail it on direct propagation, so their value depends on the mechanism being predicted.

Table 8: Per-module against shared gradient clipping for the companion recipe, three paired training seeds, forecast-step blast-set F1 on the 90 held-out synthetic forecast steps. The graph-token arm uses K_g=8 learned attention queries and per-module clipping.

Configuration	Seed 0	Seed 1	Seed 2	Mean ± half-range
Learned configurations				
text input, per-module clip	0.642	0.614	0.565	0.607 ± 0.039
graph tokens + text input, per-module clip	0.621	0.549	0.579	0.583 ± 0.036
text input, shared clip (preregistered)	0.565	0.537	0.611	$\mathbf{0.571 \pm 0.037}$
Structure				
typed reachability	0.431	0.431	0.431	0.431

Three-seed consequence learning. Table 8 reports all seeds for the text-input and graph-token recipes. Their mean blast-set F1 values range from 0.571 to 0.607, above typed reachability's 0.431. Seed identifiers denote training repetitions, not bit-reproducible statistical pairs; the table retains the full variation. Per-module clipping controls the influence of auxiliary gradient norms and is an optimizer design, rather than a bound on policy divergence.

C Analysis and Failure Modes

C.1 Direct and hidden propagation

Figure 8b compares the same learned checkpoints with reachability within each propagation population. Reachability performs better on direct channels, while learned predictions perform better on hidden channels. The counts repeat each record across nine checkpoints; they are not counts of independent worlds.

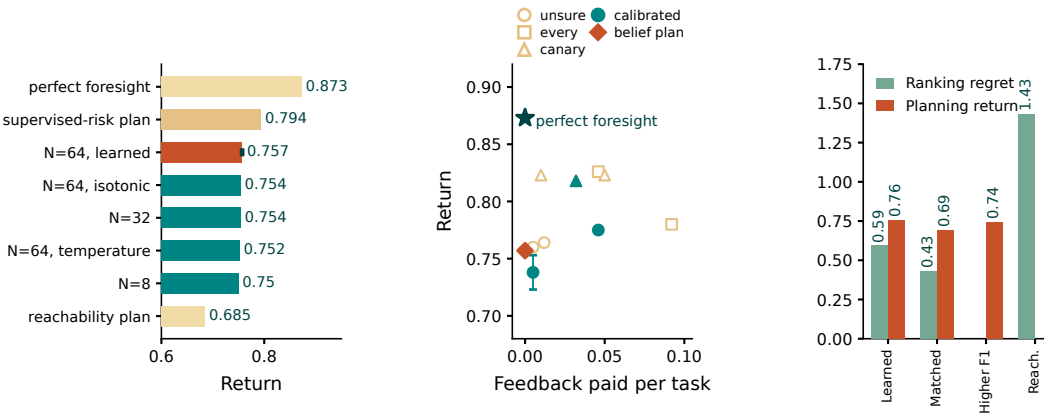
Across the nine checkpoints, 482 of 565 missed positives (85%) and 353 of 366 false positives (96%) lie outside typed reachability. Even when the blast set is correct, outcome classification is correct on only 243 of 297 cases (82%). These diagnostics distinguish missing structural coverage from errors in predicting consequences.

C.2 Forecast errors and decision value

The correlation in Table 3 is Spearman correlation between pairwise ranking accuracy and ranking regret across checkpoints and controls; pairwise accuracy excludes equal-damage pairs and gives half credit to predicted ties.

Figure 9c compares learned predictions with a noise-perturbed oracle at matched blast-set F1. Ranking and migration depend on where errors occur, which motivates directly measuring agent decisions in the primary experiment rather than using blast-set F1 as a substitute.

At matched ranking F1 of 0.525, the perturbed oracle incurs less regret than the learned checkpoints. In planning, the direction reverses at matched F1 of 0.546: learned beliefs yield higher return, including relative to an oracle with F1 0.593. A low-fidelity oracle at F1 0.213 still has ranking regret 1.343, below reachability's 1.430. The supervised controls change less under the clean-to-full-noise intervention: ranking regret rises from 0.165 to 0.258, and planning return falls from 0.794 to 0.764. Thus matching set accuracy does not match the decision consequences of errors.



(a) Belief planning (b) Paid feedback (c) Fidelity and decisions

Figure 9: Explicit-belief decisions and prediction-error diagnostics. (a) Mean return for preventive-migration plans on 40 held-out tasks; N is the number of sampled blast sets. The learned $N = 64$ error bar is the three-seed standard error. (b) Return versus mean expenditure on paid checks: shape identifies the check strategy and filled markers denote calibrated beliefs; the displayed error bar is a standard error. (c) Noise-perturbed oracles matched to learned F1 have lower ranking regret but lower planning return. Ranking and planning use separate populations; lower regret and higher return are better.

C.3 Execution costs and explicit-belief decisions

Use consequence probabilities to choose preventive actions. We apply the rules in Proposition 1 to sampled break probabilities. A paid check reveals one consumer's outcome; a system-wide trial execution (called a canary in Figure 9) reveals the complete blast set. Each observation conditions the sampled belief before the remaining migrations are chosen.

Measured consequences support better plans than reachability. Figure 9a–b compares planning and feedback on 40 held-out tasks. Sampled learned beliefs achieve return 0.757 at $N = 64$, versus reachability's 0.685. The supervised-risk and perfect-foresight references are included alongside all calibration and paid-feedback arms, so additional information is evaluated together with its execution cost.

Evaluation protocol. A perfect-foresight plan uses the true blast set before the change. The learned rows average three training seeds except the marked temperature scaling run; $\pm$ is their standard error. Calibration adjusts predicted break probabilities using training worlds only: isotonic calibration fits a monotone mapping, whereas temperature scaling rescales prediction logits. Costs follow Appendix D.7; per-consumer checks cost either 0.01 or 0.02. In that order, uncalibrated "check where unsure" gives (return, mean expenditure) of (0.764, 0.012) and (0.760, 0.005); calibrated checks give (0.775, 0.046) and $(0.738 \pm 0.015, 0.005)$. Checking every consumer gives (0.826, 0.046) and (0.780, 0.092). Always running a canary gives (0.823, 0.050); uncalibrated and calibrated belief-triggered canaries give (0.823, 0.010) and (0.818, 0.032), respectively.

C.4 Structural limits of consequence prediction

The typed-reachability baseline follows consumer-dependency paths backwards from the nodes modified by the change, $\mathrm{tgt}(a_t)$:

$$R(G_t, a_t) = \{u \in V : u \text{ has a consumer-dependency path to } \mathrm{tgt}(a_t)\}, \tag{15}$$

It is a deterministic function of the extracted graph and the change; Appendix D.3 defines the dependency relations it follows.

To locate where accuracy beyond reachability must come from, we call a predictor *structural* if it uses only the extracted graph and the action, that is, measurable with respect to (G_t, a_t); reachability is one, and so is any learned model restricted to these inputs.

Remark 1 (Structural predictors carry irreducible error)**.** *Let f be any predictor with values in $[0,1]^V$ that is measurable with respect to (G_t, a_t), as the indicator vector of R is. Then f is constant on states sharing an extracted graph, and for the squared loss $\ell(B, f) = \|\mathbf{1}_B - f\|^2$ on the indicator vector of B, whose expectation the conditional mean minimises,*

$$\mathbb{E}\big[\ell\big(B(s_t, a_t),\ f(G_t, a_t)\big)\big] \ \geq\ \mathbb{E}\big[\,\mathrm{Var}\,\big(B(s_t, a_t)\,\big|\,G_t, a_t\big)\big]. \tag{16}$$

where Var *sums the per-component variances. The right-hand side is nonzero for two reasons, both properties of the environment rather than of the estimator. Consumers differ in whether they tolerate a change on a dependency they share, and the extractor drops channels that carry* 34.8% *of the broken consumers at our forecast steps (Appendix D.4).*

The bound applies to learned models too: the expected loss is the floor plus the expected squared distance between f and the conditional mean of $\mathbf{1}_B$ given (G_t, a_t). No predictor restricted to these inputs can remove the floor; it can only represent it, as a graded break probability per consumer. A predictor can narrow the distance, and reachability leaves it wide by marking every reachable consumer as broken, tolerant or not, and every consumer reached only through a hidden channel as safe. Narrowing it requires executed supervision where reachability is wrong; going below the floor requires information outside the graph, such as executed outcomes in h_t. The evaluated SWM also reads the observed health η_t, the component health that earlier checks revealed (Section 3.2). When η_t carries no further information about B, SWM's floor equals the structural one, and SWM can only narrow the distance and represent the floor.

D Worlds, Protocol, and Reproducibility

This appendix first defines the software world and its extracted graph. It then specifies corpus generation, evaluation populations, metrics, and pre-registration. Implementation and model variants follow in Appendix D.9, decision protocols in Appendix C.3, and complete result tables in Appendix A.

D.1 Software worlds

We use four complementary families of executable software worlds (Table 9). The procedural systems provide disjoint training and evaluation populations with controlled hidden channels. Acme exercises the same interfaces in a compact multi-repository service system. Online Boutique provides a polyrepository application built from real services and contracts. The real-library worlds pair an upstream project with selected downstream packages that depend on it; their own test suites measure consequences.

D.2 World construction and evolution

Each world is frozen at an identified baseline and exposes a declarative action catalog. An action contains its target surface, materialized patch, expected local checks, and the checks that can observe downstream effects. Before every transition, the harness restores the baseline, verifies its health, applies exactly one selected patch, and reruns the declared checks. The resulting transition stores the pre-change observation, action identity, patch hash, local outcome, downstream blast set, check feedback, and restoration receipt. Repeating a transition starts from the same restored state.

Evolution occurs only through verified transitions. During counterfactual exploration, several candidates branch from one shared state and only the selected branch extends the agent history. During sequential exploration, the next request receives the outcomes of earlier selected actions but never sees validation or test outcomes. A trained world model may then forecast the remaining candidates. Its frozen forecasts are returned to the coding agent, which selects the next action before the harness executes it.

Table 9: Software worlds used for exploration, training, and evaluation. Each change must pass the changed repository's own checks before downstream outcomes are measured.

World	Composition	Changed interface	Executed signal
Procedural systems	4–7 services, shared library, datastore, events, configuration and access control	Nine change families over typed relations	Repository suites, service health, checks, and true blast sets
Acme	Four repositories with three Flask services and one shared library	API, dependency, database, event, configuration, flag and permission changes	Local suites plus end-to-end checks
Online Boutique	Ten repositories containing nine Go, Node and Python microservices	gRPC contracts, service wiring, data and latency changes	Repository tests, service health, direct and web journeys
Real-library worlds	Marshmallow with four dependents; OpenTelemetry semantic conventions with four instrumentation packages	Locally consistent upstream API and contract changes	Frozen upstream and downstream repository test suites

D.3 System-Surface Graph schema

Node kinds and edge relations are tabulated in Table 10 and Table 11.

Table 10: Node kinds. Every surface-layer node has exactly one owning system-layer node, which identifies the component that owns each interface.

Layer	Kind	Meaning
System	`repo`	a version-controlled repository
	`service`	a deployable component
	`library`	a shared importable component
	`database`	a datastore instance
	`table`	a table or collection
	`topic`	an event stream
	`config`	a configuration value a component depends on
	`flag`	a feature flag
	`external`	a third-party dependency
Surface	`endpoint`	an HTTP/gRPC route a service exposes
	`symbol`	an exported function or class
	`schema`	a data contract
	`column`	a table column
	`scope`	an authorisation permission

D.4 Corpus generation

Systems. Each system contains 4–7 services and one shared-library repository with three exported symbols. Services have 2–3 endpoints, a feature flag, and a timeout configuration. A datastore with two columns is attached to 75% of services. Synchronous calls form a directed acyclic graph over the service order, with edge probability 0.35. Event topics are published by 40% of services and consumed by 30% of the others. Cross-service column reads occur with probability 0.18, modeling the shared-database antipattern. Half the services have authorisation scopes.

Across the held-out corpus, 34.8% of broken consumer references in the 90 forecast-step blast sets are reached only through a dropped channel.

Table 11: Edge relations. The consumer relations are those along which breakage propagates backwards, and are the ones the reachability baseline traverses.

Relation	src → dst	Recovered from	Consumer
`contains`	repo → service/library	repository layout	
`depends_on`	repo/service → library	manifest, lockfile	✓
`imports_symbol`	service → symbol	import statements	✓
`provides`	service → endpoint	route declarations	
`calls_api`	service → endpoint	client call sites	✓
`publishes`	service → topic	bus client calls	
`consumes`	service → topic	bus client calls	✓
`conforms_to`	endpoint/topic → schema	schema registry	✓
`reads/writes`	service → table/column	SQL, migrations	✓
`guarded_by`	service → flag/config	configuration references	✓
`requires_scope`	endpoint → scope	auth middleware	✓
`field_of`	column → table	DDL	

Extractor model. Channels are dropped from the observed graph at per-relation rates reflecting real extraction failure modes: `reads` 0.60 (class-level data attributes), `calls_api` 0.35 (attribute-chain resolution on a client object), `consumes` 0.30 (topics bound as wire literals), `guarded_by` and `requires_scope` 0.10, `imports_symbol` 0.05, `depends_on` 0.00 (manifests parse reliably). Every dropped edge leaves a weaker dependency trace in the observed graph: a repository-level `depends_on` edge to the provider, plus a base-URL or topic-name configuration node for API and event channels. These traces can help predict failures without restoring the missing dependency path (Section 2.3).

Propagation. Breakage is computed deterministically on the true graph, according to the change type. field renames break callers whose recorded `fields_read` includes the renamed field. Symbol renames and breaking dependency upgrades break importers. Column renames break cross-service readers. Event-key renames break consumers. Route moves and new authorisation requirements break callers. Flag flips break callers of the owning service with probability 0.5 through a *behavioural* channel with no edge; timeout tightenings break callers whose fan-out is at least 2 through a *latency* channel, likewise edgeless. Breakage cascades one further hop with probability 0.40. A broken consumer's own suite catches it with probability 0.12, which is what makes most breaks silent.

Rollouts. Executed-path rollouts have four steps: apply the change (the forecast step, whose target is the latent blast set), run the repository suites, run the end-to-end checks, repair one consumer. Branched corpora execute K explorer-proposed candidates from the same state instead, labelled `chosen`, `alternative`, `risky`, `subtly_wrong`, `recovery`.

D.5 Real-world corpora

Sources and construction. The real-application families in Table 9 follow the restore, execute, and verify procedure in Appendix D.2. An upstream change must pass its own test suite before the frozen downstream suites measure its effects.

Real-library corpus and exploration. The real-library study trains on three training worlds built from upstream projects and their dependent packages and evaluates prediction on 984 held-out records drawn from libraries disjoint from those worlds. Within the 984 records, 137 breaking records are missed by the extracted graph, which gives a real-data hidden-channel population. Executed exploration follows the acquisition protocol of the synthetic study: each policy scores 450 candidate perturbations per round, and the executed round is added to the training corpus (Figure 6).

Because the downstream suites are the projects' own tests, a break recorded here is a real test failure rather than a simulated propagation event.

Relation to the synthetic population. The synthetic agent comparison uses procedural systems with controlled propagation and inexpensive counterfactual execution. The real-library comparison in Table 1 extends the agent evaluation. Companion prediction and exploration studies use organic dependencies and repository suites (Figure 6a); Appendix D.7 describes the feasibility study for agent use.

D.6 Comparison methods

For the main comparison, we trained 3B adaptations of CWM and Qwen-AgentWorld following the training procedures described in their papers (Copet et al., 2025; Zuo et al., 2026). These are our implementations at a common model size: the original CWM has 32B parameters, while Qwen-AgentWorld releases 35B-A3B and 397B-A17B mixture-of-experts models (the suffix gives active parameters).

Table 12 separates structural, supervised, generative, and oracle references. All repository-level methods receive the same observed state and action whenever their input contract permits it.

Table 12: Comparison methods and their roles. Learned controls are fitted only on training worlds. Oracle rows use information unavailable at decision time.

Family	Method	Input and output	Role
Structural	Typed reachability	Extracted graph and changed target; reachable consumers	Reachability baseline
Supervised risk	Action-only logistic	Change kind and target count; consumer probabilities	Action-prior control
Supervised risk	Relation/action logistic	Action features, typed degrees, coverage and hops; consumer probabilities	Strong tabular control
Neural structure	Graph neural network (GNN) only	System-Surface Graph; blast set	Graph encoder without language generation
Generative	Zero-shot code models	Serialized state and change; parsed blast set	Task training against model scale
Bounds	Empty, true-graph traversal, and oracle consequence	No failure, inaccessible structure, or executed outcome	Lower and upper references

D.7 Evaluation protocol and pre-registration

Exploration yields. A yield is the fraction of executed records satisfying the stated condition: breaking yield counts changes that break at least one consumer, and hidden-channel yield counts breaking records missed by reachability. Static-prediction error counts any mismatch between the reachability set and the executed blast set. Real-library yields use records whose upstream checks pass; corpus size and denominators accompany each table.

Blast-set F1. Each evaluation record carries the change, the recovered graph, and the blast set observed when the change was executed in the true system. Blast-set F1 compares the predicted and executed blast set, computed per record and averaged over records (record-mean); the companion forecast population is the 90 step-0 forecast steps of the held-out evaluation systems. The direct-channel group contains failures on recovered dependency paths; the hidden-channel group contains at least one missing dependency channel. Behavioral/latency records have no hidden channel but depend on response behavior or timing; no-break records contain no downstream failure. These groups concern graph coverage, whereas silent/visible labels concern detection by repository tests.

Decision accounting. A migration episode scores return $= 1 - 0.25\, n_{\text{broken}} - 0.05\,(n_{\text{fixes}} + 1) -$ feedback. Here n_{broken} counts services broken when the change ships, and n_{fixes} counts consumers migrated before shipping. Feedback is the total cost of paid checks at the stated price (e.g. 0.02 for a consumer check). Belief plans (Figure 9) threshold sampled break probabilities at the cost ratio and execute the result; hard-selector rows execute a fixed prediction as the plan.

Ranking. Each of the 90 choice sets presents the executed branches of one counterfactual as candidates. Top-1 is the fraction of sets whose safest executed candidate is ranked first; pairwise accuracy is the fraction of candidate pairs ordered correctly; regret is the expected number of broken services above the best candidate under uniform tie-averaging.

Agent exploration. The action catalog and split are frozen before the first request. At each step the tool-using agent may inspect only eligible training candidates, selects one action, and receives only that action's executed downstream observations. Acquisition is measured by verified transitions, repository test suites executed, breaking yield, hidden-channel yield, and the utility of the resulting training corpus. Validation and test outcomes are never returned to the explorer.

Training from acquired experience. Agent-selected and matched control corpora use the same optimizer, update count, model initialization, and evaluation actions. Training seeds measure optimization variation; exploration seeds measure acquisition variation and are not conflated. A checkpoint is evaluated only after its training receipt, configuration, corpus hash, and output parser have been archived.

Companion decision protocols. Besides the direct agent evaluation in Appendix A, we distinguish three protocols. Ranking chooses among candidates whose outcomes are later executed. Belief planning converts sampled consequence frequencies into migrations and paid checks. The real-world agent protocol freezes forecasts before returning them to the coding agent, records whether the response consulted them, saves the selected patch before execution, and then verifies patch identity, restoration, and downstream outcomes. The companion real-world coding-agent protocol tests end-to-end feasibility. It is distinct from the comparative agent results in Table 1.

Prediction-noise diagnostic. We perturb true blast sets to compare error patterns at matched blast-set F1. Each prediction-noise axis is swept over eleven noise levels (Figure 9c). The induced blast-set F1 is not evenly spaced, so a comparison "at matched blast-set F1" interpolates the noise-perturbed oracle's curve at the learned model's own mean F1 rather than reading the two curves at the same noise level. The same nine checkpoints have mean blast-set F1 0.587 on the 90 forecast steps, 0.525 on the 340 executed ranking candidates and 0.546 on the 40 planning tasks. Both decision evaluations use this interpolation: the ranking population at F1 0.525 and the planning population at F1 0.546. On the ranking axis a grid level also falls within one step (0.503), so that side does not depend on the interpolation; on the planning axis no level falls between F1 0.484 and 0.603, so the interpolation is load-bearing there.

D.8 Matched agent accounting and training protocol

Evidence source and accounting. The controlled synthetic campaign is SWM-20, frozen at source revision `3fadee6b67ec` of the research repository. In paper revision `2e32829`, `evidence/loop_source.json` stores the 25-arm summary, all 1,000 per-task records, source paths and checksums, companion summaries, and the training configuration. `scripts/build_loop_evidence.py` verifies identical task IDs, episode counts, and return accounting, then recomputes Table 2 and Table 7. The source summaries round some means to four decimals; paper values are computed from episodes before rounding. Seven distinct world IDs occur in the 40-task synthetic sample (a migration task specifies a system state and one pending change). We sample those seven clusters with replacement 10,000 times (random seed 20260926), retain every task in each sampled cluster, and recompute the task-weighted paired mean. Training seeds are averaged within each task before this bootstrap; individual seed estimates remain visible in Table 4.

Training and acting protocol. All exploration conditions share the update counts, input budgets, adapters, and reward weights in Table 13. Low-rank adaptation (LoRA) uses dropout 0.05. The

backbone and two-call acting interface are specified in Section 4.1. Candidate proposals in this interface are consumers, whereas A_t in the ranking protocol contains alternative changes.

D.9 Implementation scope

Training worlds reset; deployment does not. The reset and execution-cost assumptions are defined in Section 2. SWM targets the regime in which checking every affected consumer costs more than the breaks that per-consumer break probabilities let the agent avoid. At the prices of Appendix C.3 ($c_{\text{chk}} \leq 0.02$ against 0.25 per broken consumer), checking every consumer still beats migrating by predicted break probability with the evaluated predictor. We assume that deployment systems and changes follow the training distribution, and generalization beyond that distribution is not assumed.

This appendix specifies the consequence-predictor configuration and graph-conditioned variants; Appendix A gives the agent-training protocol.

D.10 Training configuration and hyperparameters

Table 13: Consequence-predictor configuration. The default serialises retrieved state as text; graph-conditioned variants are specified in Appendix D.13.

Predictor configuration	
input	text input (2 hops; ≤ 30 edges)
graph mode	none (no graph tokens)
base	Qwen2.5-Coder, 0.5B Instruct
adapters	LoRA r=16, α=32
target modules	q, k, v, o projections
precision	bfloat16
prompt budget	768 tokens
answer budget	192 tokens
Reward weights	
w_f format	0.2
w_s blast set	1.0
w_o outcome	0.0
w_m observation/channel	0.0

GRPO	
steps	250
learning rate	1×10^{-5}
group size G	6
prompts per step	2
micro-batch size	1
clip range	0.2
KL coefficient	0.02
KL estimator	low-variance k_3
loss aggregation	token-mean
graph encoder	not used
gradient clip	1.0 shared/global norm
Supervised warmup	
steps	200
learning rate	2×10^{-4}
batch size	4
λ_s affected nodes	1.0
λ_o outcome	0.5
λ_v healthy fraction	0.5

Node features. Concatenation of a node-kind one-hot (14), an observed-health one-hot (3), an action-target indicator (1), an action-kind one-hot (12), and a length-64 hashed bag of the identifier's alphanumeric tokens, using a stable hash so features are reproducible across processes. Identifiers such as `col:users.users.email` carry real signal about what a node is and who owns it.

R-GCN update rule. Starting from input embeddings $h_v^{(0)}$, the encoder aggregates relation-specific messages (Schlichtkrull et al., 2018):

$$u_v^{(l)} = W_{\text{self}}^{(l)} h_v^{(l)} + \frac{1}{d_v} \sum_{r \in \mathcal{R}_{\text{enc}}} \sum_{u \in \mathcal{N}_v^r} W_r^{(l)} h_u^{(l)},$$

$$h_v^{(l+1)} = \text{LayerNorm}\big(h_v^{(l)} + \text{ReLU}(u_v^{(l)})\big).$$

Here $\mathcal{R}_{\text{enc}}$ includes forward, reverse and self-loop relation types; $\mathcal{N}_v^r$ lists incoming neighbours of type r, counting edges, and $d_v = \max(1, \sum_r |\mathcal{N}_v^r|)$ is their total count. The W matrices are learned transformations; biases and training-time dropout are omitted. Normalisation is per node, across all relations, as in the implementation.

Affected-node head. A per-node multilayer perceptron scores whether each node is affected,

$$m_v = \text{MLP}_\psi\big(h_v^{(L)}\big), \tag{17}$$

where $h_v^{(L)}$ is node v's embedding after L graph layers. Attention over all node embeddings produces graph tokens; this head supplies an auxiliary prediction of whether each node is affected.

Graph-token scaling. The projector output is layer-normalised and scaled to the root-mean-square magnitude of the token embedding matrix. This keeps graph tokens on the scale of the language model's text embeddings during joint training.

These runs use one A100-80GB GPU. Gradient accumulation provides the effective batch sizes listed in Table 13.

D.11 Prompt and output format

The user turn contains the retrieved subgraph around the change (nodes grouped by kind, edges as typed triples, and the observed health), followed by the action, its target, and a natural-language description of the change. The system turn fixes the output contract: a single JSON object with fields `delta` (the service identifiers that break), `outcome` (one of `no_break`, `silent_break`, `visible_break`), `observation` (what the checks will report), and `mechanism` (which channel carries the breakage and why repository-level checks miss it). At check steps, `delta` instead lists newly revealed failures; at repair steps it lists repaired services. These auxiliary targets describe check and repair outcomes, not the forecast-step blast set. The blast-prediction metric uses forecast steps only. Prompts exceeding the budget are truncated from the middle of the edge list, so the contract at the head and the action at the tail are always preserved.

D.12 Prediction horizon

The evaluated model predicts one transition from the observed state and refreshes its input after execution. Candidate ranking compares these one-step forecasts. Recursive latent dynamics are outside the evaluated model family.

D.13 Factorization and the SC-GRPO update rule

The variant of Section 3.2 couples a graph encoder to the language model. It factorises the transition,

$$\underbrace{T_t = g_\psi(S_t^{\text{struct}})}_{\text{graph tokens}} \qquad \underbrace{p_\theta\big(\Delta s_t, o_{t+1}^{\text{cons}} \mid S_t^{\text{sem}}, T_t\big)}_{\text{behavioural consequence}}, \tag{18}$$

where S_t^{struct} encodes the extracted graph with the node features in Appendix D.10, and $\phi = (\psi, \theta)$: the graph encoder g_ψ is a four-layer relational graph convolutional network (Schlichtkrull et al., 2018) with relation-specific messages and auxiliary outcome and value heads, and the behavioural factor p_θ is the same language model as the text-input model. The graph representation reaches the language model as graph tokens: K_g learned queries $Q \in \mathbb{R}^{K_g \times d}$ cross-attend over the node embeddings,

$$T_t = P_\psi\big(\text{Attn}(Q, H^{(L)}, H^{(L)})\big), \qquad p_\theta(\cdot) = \text{LM}_\theta\big(\cdot \mid [T_t;\ \text{emb}(S_t^{\text{sem}})]\big), \tag{19}$$

where d is the node-embedding width, $H^{(L)}$ stacks those embeddings, and P_ψ projects into the language model's token space (Li et al., 2023). The token budget is fixed and no node is excluded by a selection step, so the tokens summarise the whole system while the text input carries the detail near the change.

Vanilla GRPO optimises a language model alone, and the coupled policy is not one: Equation 18 joins a graph encoder to the language model through the graph tokens, and the encoder carries auxiliary structural losses of its own during supervised warmup. We therefore train the variant with

structure-coupled GRPO (SC-GRPO), which changes the update in the two places the hybrid policy forces. First, the reward gradient trains both factors: each micro-batch of the surrogate backpropagates into a detached copy of the graph tokens T, the token gradients are accumulated, and a single backward pass carries their sum through the encoder, so the structural factor is shaped by the same verifiable reward as the language model at the cost of one encoder traversal per step. Second, gradient-norm clipping is per module during both supervised warmup and GRPO: let $\mathcal{J}$ be the text loss: negative log-likelihood during supervised warmup, or the negative clipped GRPO surrogate plus its KL penalty during GRPO. Let $\mathcal{L}_{\text{aux}}$ be the weighted auxiliary losses during supervised warmup and zero during GRPO. Writing Π_γ for clipping a gradient's Euclidean norm to at most γ,

$$\begin{aligned} \nabla_\psi \mathcal{J} &= \left(\frac{\partial T}{\partial \psi}\right)^\top \sum_\mu \nabla_T \mathcal{J}_\mu, \\ \tilde{g}_\theta = \Pi_\gamma(\nabla_\theta \mathcal{J}), &\qquad \tilde{g}_\psi = \Pi_\gamma\big(\nabla_\psi(\mathcal{J} + \mathcal{L}_{\text{aux}})\big). \end{aligned} \tag{20}$$

Here T is the graph-token tensor, μ indexes token-weighted micro-batch losses, and $\tilde{g}_\theta, \tilde{g}_\psi$ are the gradients passed to the optimiser. Clipping applies separately to the parameter groups defined in Equation 18, rather than to their combined norm. Training-seed results appear in Table 8; single-run differences need not exceed seed variation.