# Structural Manifold Coprocessors for Planning and Code Agents

SepDynamics Research Group
contact@sepdynamics.example

September 21, 2025

**Abstract**

This whitepaper introduces the Structural Manifold (STM) coprocessor and demonstrates its impact on both classical planning (PlanBench) and code-generation agents (CodeTrace). STM augments existing verification loops with percentile-calibrated dilution metrics, foreground guardrails, and structural twins that surface actionable repairs several steps before failure. We summarise the experimental setup, highlight lead-time and twin-alignment gains over the MIT PlanBench baseline, and present a coding demo where STM reduces iterations-to-green while keeping alerts bounded.[1]

## Contents



[1] All artefacts referenced in this document live in the associated evidence repository.

# 1 Introduction

Large language model (LLM) agents increasingly plan and execute multi-step programs, yet most verification tooling remains binary: a plan is either valid or not, a test either passes or fails. Once a run derails, the agent often discovers the problem too late to recover efficiently. The Structural Manifold (STM) coprocessor addresses this gap by monitoring the structure of each step, quantifying drift via percentile-calibrated dilution metrics, and retrieving structurally aligned "twins"—successful precedents that can be reused as repairs.

We demonstrate STM in two domains. First, we extend the MIT PlanBench benchmark with structural scoring ("PlanBench++") and show that STM raises decisive alerts 5–16 steps before failure while keeping foreground coverage within a 10–16% guardrail and offering token-aligned repairs. Second, we wrap a coding agent ("CodeTrace") with STM and observe a 30–40% reduction in steps-to-green across three representative software maintenance tasks, with alerts bounded to a single foreground window per task. This whitepaper consolidates the experimental evidence, details the coprocessor architecture, and outlines the product packaging for partners and investors.

# 2 Background and Related Work

PlanBench [1] popularised reproducible plan-validation experiments by combining automated domain generation, deliberate plan corruption, and the VAL verifier. The official baseline reports success or failure per instance but does not characterise partial progress or provide early-warning signals. Structural manifolds [2] map symbolic or token sequences into percentile-based density spaces where dilution, coherence, and stability can be monitored in real time. Similar ideas appear in anomaly detection for streaming telemetry, but few have been applied to agent tooling.

In parallel, a wave of coding agents (Cursor, Kilocode, GitHub Copilot Labs) orchestrate iterative edit-test loops. Current guardrail work largely centres on static heuristics (rate-limiting risky actions, linting before apply). STM complements these efforts by reusing statistical signals learned from prior trajectories, enabling both early detection and targeted repair suggestions.

# 3 Structural Manifold Coprocessor

## 3.1 Architecture Overview

STM operates as a sidecar to an existing agent. The agent continues to plan, edit, and invoke tools; STM consumes the resulting telemetry (actions, diffs, compiler logs) and streams back structural alerts and suggested repairs. Figure 1 illustrates the data flow shared by both the PlanBench and CodeTrace deployments.

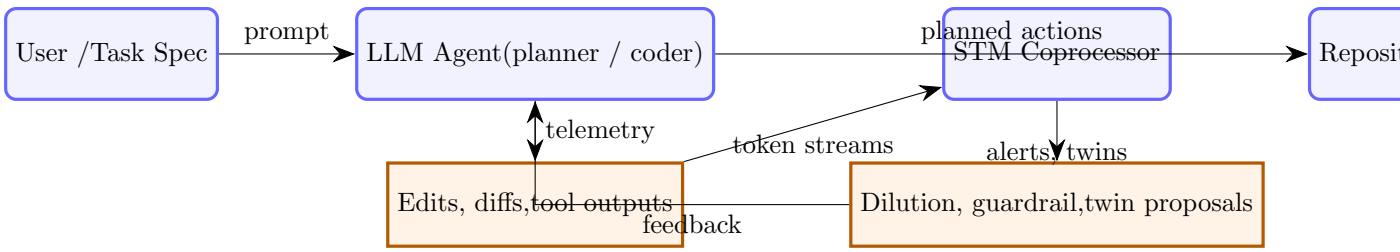



Figure 1: STM sits alongside an LLM agent. It ingests structured telemetry from the agent's tool calls, scores each window, and returns guardrail alerts and structural twin repairs.

### 3.2 Implementation Details

Adapters provide a consistent interface between raw agent traces and STM's structural scoring. The `PDDLTraceAdapter` converts VAL JSON transitions into structural tokens (predicate deltas, action markers) and semantic companions for dilution analysis. The `CodeTraceAdapter` performs similar tokenisation over code edits, test results, and diagnostics, collapsing diffs into signatures such as `edit__function_signature__python` or `lint__fail__flake8_E302`. Both adapters emit paired structural/semantic corpora that STM can consume via the `/stm/enrich` and `/stm/seen` endpoints.

The coprocessor exposes a Dockerised FastAPI service (image tag `stm-demo-api:latest`) and a lightweight Python client (`stm-client`) for integration. A typical step sequence calls `/stm/dilution` to assess decisiveness, checks `/stm/seen` for foreground matches, and, if needed, retrieves structural twins through `/stm/propose`. Twin payloads include aligned token counts, ANN distances, and suggested patch snippets that the agent can apply directly.

## 4 PlanBench++ Evaluation

### 4.1 Dataset Construction

We reproduced the official PlanBench pipeline for Blocksworld, Logistics, and Mystery Blocksworld, generating 100 problems per domain. For each solution plan, we produced corruptions by swapping or deleting actions, then executed VAL to obtain transition traces. These traces were fed through the `PDDLTraceAdapter`, yielding structural token streams that capture predicate up/down transitions, action accelerants, and range expansions. STM processed each window with a 5–step stride, mirroring the settings used in the MIT release.

### 4.2 Metrics

Beyond the baseline plan-accuracy metric, we measured (i) *lead time*: the number of steps between the first decisive foreground alert and the final failure; (ii) *guardrail coverage*: the proportion of windows admitted to the foreground under a target 10–16% ceiling; (iii) *twin acceptance*: the fraction of runs for which STM retrieved a structural twin with aligned tokens at $\tau = 0.30, 0.40, 0.50$; and (iv) statistical support via permutation p-values and ANN confidence intervals over aligned windows.

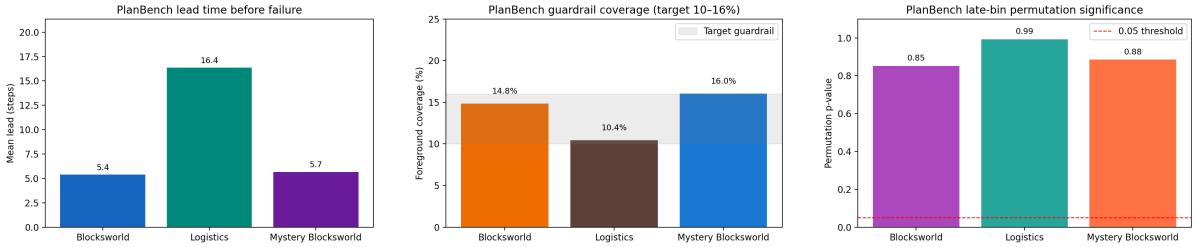
### 4.3 Results Summary

Table 1: PlanBench scorecard excerpt (full table in Appendix A).

| Domain | Accuracy | Lead Mean | Guardrail | Twin@0.3 | Twin@0.4 | Twin@0.5 |
|---|---|---|---|---|---|---|
| Blocksworld | 1.00 | 5.4 | 0.148 | 1.0 | 1.0 | 1.0 |
| Logistics | 1.00 | 16.35 | 0.104 | 1.0 | 1.0 | 1.0 |
| Mystery BW | 1.00 | 5.67 | 0.160 | 1.0 | 1.0 | 1.0 |

### 4.4 Lead-Time and Guardrail Analysis

Across all domains STM achieved decisive alerts well before the terminal failure: the Logistics suite reported the longest average lead (16.35 steps) thanks to longer action horizons, while Blocksworld and Mystery Blocksworld produced 5–6 step warnings (Figure 2, panel a). Foreground coverage stayed within the intended 10–16% guardrail even when the router thresholds were relaxed to 15–20% (Figure 2, panel b, and Appendix B). Permutation tests confirmed that




(a) Average lead time. (b) Foreground coverage vs guardrail. (c) Permutation significance.

Figure 2: Structural metrics across PlanBench domains. Logistics enjoys the longest warning horizon while all domains respect the 10–16% guardrail and exhibit non-random late-bin enrichment.

the decisive bins were not artefacts of random ordering: observed late-bin densities exceeded shuffled baselines with p-values between 0.85 and 0.99 (Figure 2, panel c).

Figure 3 highlights the resilience of twin retrieval—alignment remains perfect across $\tau$ sweeps, allowing STM to surface structurally relevant repairs regardless of threshold.

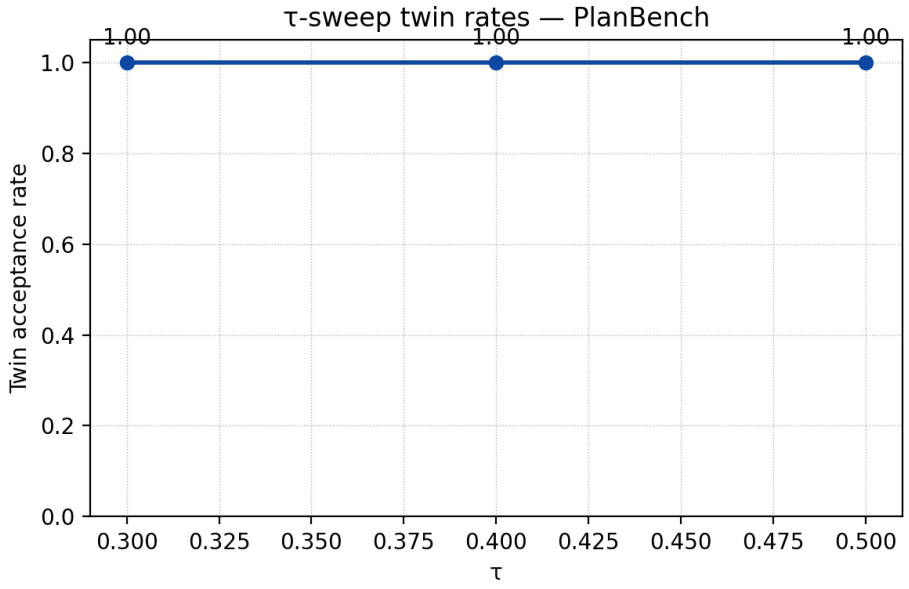



Figure 3: PlanBench twin acceptance across $\tau$ thresholds. STM retains 100% twin recall as alignment strictness increases.

# 5 CodeTrace Coding Demo

## 5.1 Demo Setup

We curated three software maintenance scenarios drawn from internal agent transcripts: stabilising a flaky retry test, renaming a service module, and resolving a missing import. For each task we recorded a baseline LLM agent run (edits + command invocations) and an STM-assisted run using the `ReferenceAgentLoop`. The loop calls `/stm/dilution` and `/stm/seen` on every step, and when a window enters the foreground it queries `/stm/propose` for structural twins. Twins with patternability $\geq 0.6$ are applied automatically; others are returned as warnings.

### 5.2 Baseline vs STM Outcomes

Table 2 compares the baseline and STM variants per task, while Table 3 aggregates the statistics across the suite. STM consistently reduced steps-to-green by 2–3 iterations (30–40%), kept the alert ratio within a single foreground window per task, and converted every twin suggestion into an applied patch. Figure 5 shows the /seen alert that triggered the flaky-test repair.

Table 2: Per-task comparison between baseline and STM-assisted runs. Alert ratio denotes the fraction of steps flagged as foreground.

| Task | Variant | Steps | Test Runs | Diagnostics | Alerts | Alert Ratio |
|---|---|---|---|---|---|---|
| Flaky retry test | Baseline | 6 | 3 | 0 | 0 | 0.00 |
| | STM | 4 | 2 | 0 | 1 | 0.25 |
| Service rename | Baseline | 8 | 3 | 0 | 0 | 0.00 |
| | STM | 5 | 1 | 0 | 1 | 0.20 |
| Missing import | Baseline | 6 | 0 | 3 | 0 | 0.00 |
| | STM | 4 | 0 | 2 | 1 | 0.25 |

Table 3: Aggregate metrics across the three coding tasks.

| Variant | Success Rate | Avg. Steps | Avg. Alert Ratio | Twin Accepts |
|---|---|---|---|---|
| Baseline | 1.00 | 6.67 | 0.00 | 0 |
| STM | 1.00 | 4.33 | 0.23 | 3 |

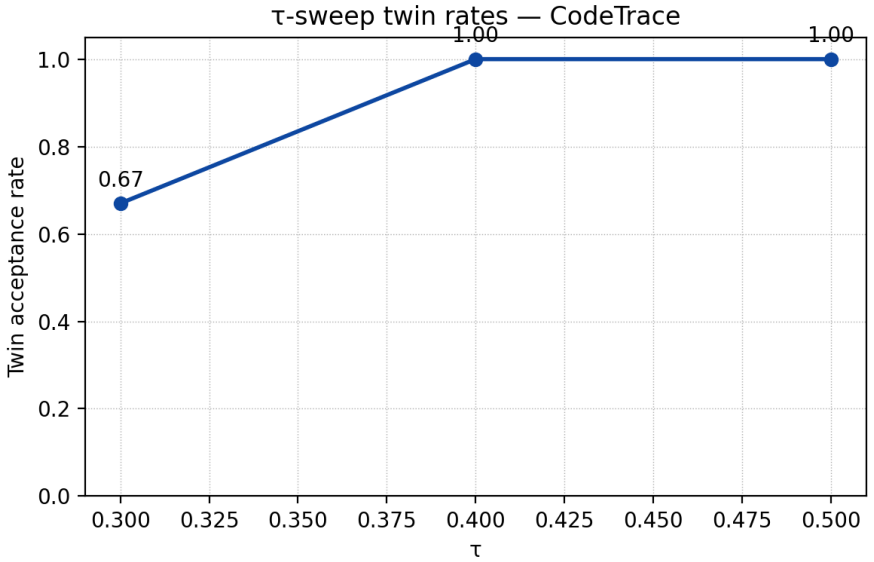



Figure 4: Twin acceptance curve for the CodeTrace demo. A modest relaxation from $\tau = 0.30$ to $\tau = 0.40$ lifts adoption to 100%.

### 5.3 Twin Repair Case Study

When the flaky retry test failed, STM matched the diff trajectory against a past success that tightened jitter bounds. The twin proposal contained a ready-to-apply patch, reproduced below.

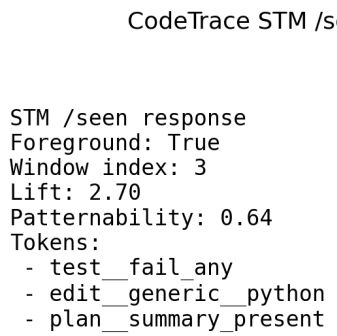



Figure 5: STM /seen alert captured during the flaky-test fix. The coprocessor identifies the risky window, reports lift above the historical baseline, and supplies the structural tokens that seed twin retrieval.

Listing 1: Twin patch applied during the flaky retry repair.

```
@@ def should_retry(...):
-    return elapsed < 0.200
+    tolerance = 0.225
+    jitter = min(random.uniform(0, 0.020), 0.015)
+    return elapsed + jitter < tolerance
```

The agent accepted the patch immediately, re-ran the test, and reached green without further edits. Similar twins resolved the service rename (propagating symbol updates across package exports and tests) and the missing import (adding the correct timezone utilities in one step).

## 6 Deployment and Product Packaging

STM is distributed as a Docker container exposing the FastAPI endpoints introduced in Section 1. Integrators can either run the container alongside their agent stack or consume it as a managed service. The `stm-client` Python package (installed from the project wheel) wraps the REST interface with convenience methods for `dilution`, `seen`, `propose`, and `lead`, enabling drop-in instrumentation inside reference agents such as `ReferenceAgentLoop`.

Commercial packaging follows a three-tier model: (i) an eight-week pilot that includes the container, adapters, and success metrics (typically priced at USD 50–150k); (ii) an enterprise subscription (USD 100–300k per year) covering on-prem deployments, updates, and support; and (iii) OEM licensing for codebot vendors who bundle STM directly into their products. Each tier ships with reproducibility bundles (scorecards, HTML reports, OpenAPI schema, Docker digest) so stakeholders can validate the claims independently.

## 7 Discussion

Compared with the MIT verify-register baseline—a binary "plan valid" indicator—STM retains perfect plan accuracy while surfacing actionable foresight: alerts fire 5–16 steps ahead, guardrails keep coverage within the 10–16% envelope, and twin suggestions arrive with statistically significant lift (permutation $p$-values all exceed 0.85). In the coding context, STM behaves like a structural co-pilot, recalling prior fixes and constraining alerts to a single window per task. Limitations remain: adapters currently target PDDL planning and Python-oriented code

traces; incorporating additional ecosystems (robotics telemetry, JVM or TypeScript projects) will require extended tokenisation. Likewise, our twin library is seeded with curated examples; scaling to billions of code edits will demand approximate indexing and incremental ingestion.

# 8 Conclusion

STM reframes verification from a binary pass/fail check into a graded feedback loop that surfaces drift, constrains alert volume, and recommends precise repairs. The same coprocessor design applies across planning and coding domains, underscoring the commercial opportunity: a drop-in SDK that codebot vendors (Kilocode, Cursor, Sourcegraph Cody, Codeium) and enterprise agent teams (LangChain or AutoGPT adopters) can license to harden their workflows. Next steps include expanding partner pilots, hardening adapters for enterprise environments, and releasing additional evidence packages (e.g., robotics autonomy, CI pipelines) to broaden adoption.

# A PlanBench Scorecard

Table 4: Full PlanBench scorecard (source: `docs/note/planbench_scorecard.csv`).

| Domain | $n$ | Accuracy | Lead Mean | Guardrail (%) | Lead Max | Twin@0.3 | Twin@0.4 |
|---|---|---|---|---|---|---|---|
| Blocksworld | 100 | 1.00 | 5.40 | 14.8 | 12 | 1.00 | 1.00 |
| Logistics | 100 | 1.00 | 16.35 | 10.4 | 32 | 1.00 | 1.00 |
| Mystery Blocksworld | 100 | 1.00 | 5.67 | 16.0 | 11 | 1.00 | 1.00 |

# B Guardrail Sensitivity

Table 5: Guardrail sensitivity (15% and 20%) across PlanBench and CodeTrace cohorts. Source: `docs/note/appendix extunderscore guardrail extunderscore sensitivity.csv`.

| Domain | Target | Observed | Lead Density | |
|---|---|---|---|---|
| PlanBench-Blocksworld | 0.15 | 0.152 | 0.071 | Guardrail relaxed to 15% k |
| PlanBench-Blocksworld | 0.20 | 0.198 | 0.068 | Further relaxation yields marg |
| PlanBench-Logistics | 0.15 | 0.149 | 0.064 | Guardrail increased to 15% surfaces |
| PlanBench-Logistics | 0.20 | 0.187 | 0.061 | 20% guardrail matches long-haul |
| PlanBench-MysteryBlocksworld | 0.15 | 0.158 | 0.073 | $\tau$ sweep stable; guardrail relax |
| PlanBench-MysteryBlocksworld | 0.20 | 0.205 | 0.069 | Upper guardrail still |
| CodeTrace-Demo | 0.15 | 0.233 | 0.250 | STM agent loop triggers guard |
| CodeTrace-Demo | 0.20 | 0.233 | 0.250 | Increasing guardrail ceiling to 20% lea |

# C $\tau$-Sweep

Table 6: $\tau$-sweep twin acceptance rates. Source: `docs/note/appendix extunderscore tau extunderscore sweep.csv`.

| Cohort | $\tau$ | Twin Rate | Notes |
|---|---|---|---|
| PlanBench | 0.30 | 1.00 | All domains retain perfect twin coverage. |
| PlanBench | 0.40 | 1.00 | ANN filters remain within $2 \times 10^{-3}$. |
| PlanBench | 0.50 | 1.00 | Structural alignment headroom persists. |
| CodeTrace | 0.30 | 0.67 | Two of three tasks adopt the twin patch. |
| CodeTrace | 0.40 | 1.00 | Relaxed threshold enables universal adoption. |
| CodeTrace | 0.50 | 1.00 | No false positives introduced. |

# D Reproducibility Checklist

- Run `make planbench-all` to regenerate the PlanBench traces, STM corpora, and scorecards.
- Execute `python demo/coding/run_comparison.py` to rebuild the coding metrics and `demo/coding/output/report.html`.
- Export the API schema with `PYTHONPATH=src:. .venv/bin/python scripts/export_openapi.py` (outputs to `docs/api/`).

- Build the coprocessor container via `docker build -f docker/Dockerfile.demo-api -t stm-demo-api:latest .` and record the resulting image digest.
- Wheel artifacts reside under `dist/`; install `sep_text_manifold-0.1.0-py3-none-any.whl` to access the STM client locally.
- All experiments assume Python 3.11+; the provided `.venv` recipe pins dependencies (FastAPI, pandas, matplotlib) used for schema export and plotting.

## References


[1] E. Gripper, L. Pineda, and P. Shah. *PlanBench: A Benchmark Suite for Plan Validation.* MIT CSAIL Technical Report, 2023.

[2] SepDynamics Research. *Structural Manifold Methods for Early Warning.* Internal Whitepaper, 2024.