The Empire, Long Divided, Must Unite: Architectural Convergence in Three LLM Agent Harnesses*

Jiahong Dai
jiahong001@e.ntu.edu.sg

25 August 2026

Abstract

An *agent harness* is what turns a language model into an autonomous agent: the surrounding code that builds the model's context, mediates its tools, runs the loop, and persists state across a long-horizon run. This layer, not the model it wraps, is increasingly the binding constraint on agent behaviour. We present a source-level, multi-case study of three open coding-agent harnesses built from deliberately opposing philosophies: LangChain's DEEPAGENTS (batteries-included), Earendil's PI (radical minimalism), and DeepSeek's DSH (everything-is-a-plugin). Reading each at a pinned commit and following its commit history, we find that the two mature harnesses have travelled in *opposite* directions (DEEPAGENTS subtracting authored scaffolding, PI accreting durable infrastructure), yet converged toward one architectural middle form of five recurring elements: a commoditised loop, an append-only replayable session record, model quirks kept as data, progressive disclosure of context, and explicit extension seams. A third harness, read afterward as a held-out check, exhibits all five, and in one seam reuses another's implementation outright. We therefore do not claim independent invention, and decompose the convergence into parallel discovery, diffusion, and literal reuse. Finally, one load-bearing dimension shows no convergence, and indeed no presence: *external verifiability*, a tamper-evident record an outside party can check without trusting the runtime. We read this absence not as an oversight but as a predictive gap, the next axis on which harnesses for provenance-sensitive domains will differ.

Keywords: LLM agents; agent harness; software architecture; multi-case study; architectural convergence; auditability.

1 Introduction

Autonomous coding agents have advanced quickly on long-horizon software tasks [1, 2, 3], and the progress is usually credited to the model. A growing body of evidence points elsewhere. Among models of comparable frontier capability, the *harness* that surrounds the model, not the model itself, often governs the larger share of performance variance on long-horizon tasks. Recent work formalises this as a "binding constraint": single-harness changes move Terminal-Bench 2 pass@1 by several points and SWE-bench Verified by up to fifteen, with the model held fixed [4]. If the harness is the binding constraint, then where harness architectures are heading is a first-order question, not an implementation detail.

Yet prior work maps this layer only partially. The nearest source-level taxonomy [5] reads thirteen coding-agent scaffolds at pinned commits, but excludes DEEPAGENTS and analyses static

*The title adapts the opening line of Luo Guanzhong's *Romance of the Three Kingdoms* (trans. M. Roberts): "The empire, long divided, must unite; long united, must divide." Here three harnesses, from opposing philosophies, unite on one middle form, and on one dimension (Section 7) they remain, for now, divided.

snapshots only. A longitudinal study of five command-line harnesses [6] relates release velocity to quality, but not to the architectural form that evolution produces. And the survey and reading list that treat the harness as a research object [7, 8] do not read implementations at all.

Three gaps therefore remain in our understanding of the harness layer. First, **the harness layer itself is unread** (G1), since existing source-level studies cover coding-agent applications rather than the general harness layer beneath them, and neither PI nor DSH appears in any architectural study. Second, **architectural trajectories are uncharacterised** (G2), since prior readings analyse static snapshots and the one longitudinal study quantifies quality rather than form. Third, **convergence is neither claimed nor tested** (G3), since the only study to observe converging dimensions reports the observation as incidental and asks nothing about where the field's architectures are heading.

To address these gaps, we present a source-level, multiple-case study of three open coding-agent harnesses chosen for maximal philosophical spread:

- DEEPAGENTS (LangChain): batteries-included, with a middleware stack, pluggable storage backends, first-class subagents, and four layers of automatic context management, wired before the developer writes a line.
- PI (Earendil Works; Mario Zechner and Armin Ronacher): the "smallest useful harness," with four tools, a ∼300-token base prompt, and the credo that everything else is an extension the user can see.
- DSH (DeepSeek): everything-is-a-plugin, a service-locator runtime in which even the agent loop is a swappable configuration row.

Reading all three does not end in taking a side. Close reading, close enough to reproduce defects and file issues in two of the trackers, surfaces a different picture. We organise the study around three research questions (RQs), each answered by a tagged section (Sections 4, 5, and 7):

RQ1 (Divergence). From what architectural positions did the harnesses start, and along what trajectories have they evolved?

RQ2 (Convergence). Is there a common form they are arriving at; what is it; and *why* are they converging?

RQ3 (Boundary). Is there a dimension on which they have *not* converged, and what explains the gap?

Figure 1 charts how the questions narrow: from three origins, to one convergent form, to the one dimension that remains open.

Our central finding is the following:

> *Consistent with a shared and intensifying selection pressure (long-horizon autonomous operation), three harnesses that began from opposing philosophies have converged on one architectural middle form; the convergence is real but not fully independent, and its boundary, the one dimension where no convergence has occurred, is external verifiability.*

Contributions. Our main contributions are fivefold:

- **Source-Level Coverage** (G1): the first source-level architectural study of the harness layer represented by DEEPAGENTS, PI, and DSH; the nearest prior taxonomy explicitly excludes DEEPAGENTS, and PI and DSH appear in no prior architectural study.

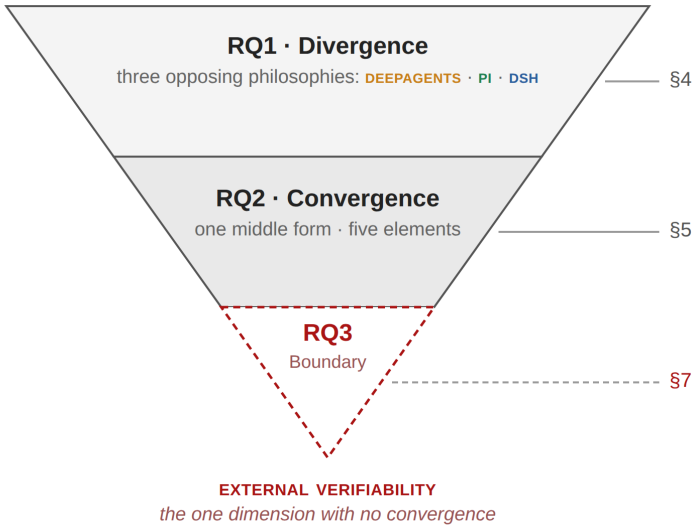

Figure 1: The three research questions. Each band narrows to the next: three origins (RQ1, Section 4), one convergent form (RQ2, Section 5), one open dimension (RQ3, Section 7). Dashed = open, as in every figure.

- **Trajectory Evidence** (G2): the first account of harness *trajectories* from commit archaeology (subtraction, accretion, and reuse), where prior work is either a static snapshot or a quality-focused longitudinal study.
- **The Convergent Middle Form** (G3): a five-element middle form grounded in two cases, checked against a third read subsequently (not fully independent of the second), and decomposed into parallel discovery, diffusion, and literal reuse.
- **Convergent Fault Lines**: an illustrative taxonomy of four architectural seams on which reproduced defects recur across opposing designs.
- **The Verifiability Boundary**: evidence that *external verifiability* is a dimension on which all three harnesses are absent, and the argument that its absence is a predictive gap rather than an oversight.

To the best of our knowledge, this is the first study to make architectural convergence the central, evidenced claim about the LLM agent harness layer: five recurring elements, three convergence mechanisms, and one dimension on which convergence has not begun.

Three scope caveats bound these claims. We do not claim the three converged independently: they are mutually aware, and one literally reuses another (Section 5.3). We do not claim a systematic longitudinal analysis; commit trajectories are directional corroboration, not a controlled study. And we do not use this paper to advance any particular solution to the verifiability gap: we characterise the gap, and stop.

As Table 1 shows, the comparison separates a convergent core from a divergent gap. The five middle-form capabilities are present across all three production harnesses (with the append-only record only partially realised in DEEPAGENTS, Section 5), yet the prescriptive protocols specify them only partially, the seam row excepted, which protocols exist to standardise. The middle band holds the one capability that remains genuinely divergent: an OS-enforced sandbox ships in DSH, partially in DEEPAGENTS, and deliberately not at all in PI ("containerise yourself"), the audience split of Section 5. The verifiability rows invert this: absent from every production harness, but placed at the centre of prescriptive proposals such as Autogenesis, which specifies

Table 1: Capability matrix: three production harnesses vs. the prescriptive protocol layer. ✓: present, ▲: partial, ×: absent, —: not applicable. *Protocols* scores what prescriptive proposals (Autogenesis [9], MCP, A2A) *specify*, not what any shipped harness implements.

	Built bottom-up			Proposed top-down
Capability	**deepagents**	**pi**	**dsh**	**Protocols**
Convergent middle form (§5)				
Commoditised loop (small, readable, or delegated)	✓	✓	✓	—
Append-only, replayable session record	▲	✓	✓	▲
Model quirks externalized as data	✓	✓	✓	▲
Progressive disclosure of context	✓	✓	✓	▲
Orthogonal extension seams	✓	✓	✓	✓
Residual divergence (§5)				
OS-enforced sandbox, per-call policy	▲	×	✓	×
The verifiability gap (§7)				
Externally verifiable, tamper-evident record	×	×	×	▲
Domain provenance (as-of, entitlement, cost)	×	×	×	▲

runtime-internal lineage, itself short of outsider-checkable, hence a partial mark. That asymmetry is the paper's boundary claim: what practitioners converged on bottom-up, protocols only partly specify; and the verifiable record protocols prescribe top-down, no shipped harness has built.

2 Background and Related Work

Harness engineering as an object of study. The harness has recently been recognised as a distinct research object, with a survey formalising it as a tuple of execution loop, tool registry, context manager, state store, lifecycle hooks, and evaluation interface [7], a curated reading list mapping the area [8], and a subfield emerging around automatic harness optimisation [4]. These works establish the vocabulary but do not read implementations comparatively; a comparative, source-level reading of the harness layer is precisely what this paper supplies.

The nearest neighbour. Closest to our work is a source-code taxonomy of thirteen coding-agent scaffolds at pinned commits, characterising each across twelve dimensions in three layers, grounded in file and line references [5]. We inherit its methodological discipline (pinned commits, line-level evidence) and differ in three ways that define our contribution. First, *objects*: that study explicitly excludes DEEPAGENTS as "general-purpose rather than coding-specific," and neither PI nor DSH appears in it; we study the general harness layer beneath coding-agent applications. Second, *time*: it analyses static snapshots and disavows evolution tracking; our central evidence is the trajectory each harness has travelled. Third, *framing*: it reports convergence as an incidental observation ("dimensions converge where external constraints dominate") and treats persistence as a detail within state management; we make convergence the central thesis and treat auditability as a first-class dimension. A complementary longitudinal study tracks five command-line harnesses over twelve months and finds that rapid release velocity does not yield proportional quality gains [6]; it quantifies *evolution-to-quality* but does not characterise the *architectural form* that evolution is producing, which is our subject. In short, prior work draws a static map of the design space; this paper traces three trajectories through it and identifies the point at which they meet.

Why the destination matters. That the harness can dominate the model motivates studying its architecture directly [4]. The insight that the agent-computer interface shapes capability originates with tool-centric agents [1, 2]; how tools are organised and exposed further shapes agent behaviour [10], and scaffolding can determine performance more than model choice [11]; and multi-agent frameworks established the harness as reusable infrastructure [3]. This line establishes that the interface shapes capability; it does not ask where harness architectures are heading, which is the question this paper answers.

The prescriptive counterpart. A complementary line approaches the harness top-down, by specifying what it *should* provide: MCP standardises tool invocation and A2A agent-to-agent messaging [12, 13], and Autogenesis [9] (part of a broader self-evolving-agents programme [14]) proposes a protocol above both that registers prompts, agents, tools, environments, and memory as versioned, lifecycle-managed resources, so that every self-modification is traceable and reversible. We work in the opposite direction: rather than prescribing what harnesses should look like, we document at the source level what three production harnesses have in fact become, and the auditable lineage such proposals place at their centre is precisely the dimension on which all three show no convergence, and no presence (Section 7).

Method lineage. We follow established guidance for multiple-case study design and reporting in software engineering [15, 16], and treat source-level architectural narrative in the tradition of open-source architecture studies [17]. These sources supply the method; none applies it to the agent harness, the object this study reads.

3 Study Design

Design. This is an explanatory, theory-building multiple-case study following a *literal replication* logic across maximum-variation cases: the first two cases (DEEPAGENTS, PI) ground a candidate model of the convergent form, and the third (DSH) is read subsequently as a held-out check that, because DSH reuses PI's provider catalogue, is confirmatory only for the elements where its instantiation is independent (Section 5.3). Sections 4, 5, and 7 answer RQ1, RQ2, and RQ3 respectively; Section 6 adds an illustrative second evidence line for RQ2.

Case selection. Cases were chosen for maximal spread, not convenience. They span three points of the design space (batteries-included / minimal / everything-is-a-plugin), three organisational forms (a venture-backed framework company / two independent engineers / a frontier lab), and two languages (Python / TypeScript). Each is open-source and readable at a pinned revision. The candidate pool was open-source, actively developed, source-readable coding-agent harnesses; from it we selected for maximal philosophical spread. Prominent harnesses we did not read at source level (OpenHands, Aider, SWE-agent's harness, Gemini CLI, Codex CLI, Claude Code) are not counted as evidence; a documentation-level scan of two of them for the five elements is future work (Section 9), and where Table 1 and the text say "the field," the claim is scoped to these three cases. We deliberately *disclose an independence limitation*: DSH's generic provider adapter depends on PI's published package (Section 5.3); the three are therefore not fully independent lineages, which we handle analytically rather than assume away.

Pinned revisions. All file and line references are to DEEPAGENTS 0.7.8 at commit `2c8015378` (trajectory evidence also cites `23b83ad50`), PI `main` @ `a470b121b`, and DSH at release tag

`dsh-v0.1.1-rc.2`, commit `b150a551b`.[1] Line counts are whole-file, blank and comment lines included.

Data sources. Four kinds, all re-checkable: (i) *source* at the pinned revisions, with claims anchored to file:line; (ii) *commit, PR, and issue archaeology* for trajectories; (iii) *hands-on reproduction*, comprising two plugins we wrote against DSH's runtime, and two defects (the DEEPAGENTS path-normalisation bypass and the PI throttle-misclassification) independently re-derived in a sandbox, with further seam instances located by source reading; and (iv) *upstream confirmation of specific claims*. On (iv): we submitted issues and a pull request; one documentation fix was accepted, assigned, and merged, and one defect was filed (issue #5640). These confirm individual point claims (not the five-element model or the seam taxonomy), and we scope their evidential weight accordingly (Section 9).

A priori dimensions. The comparison dimensions (Table 2) were fixed before close reading, to avoid fitting a framework to observations. Where they overlap the nearest neighbour's twelve dimensions we adopt its discipline; we add two axes it lacks (*auditability* and *evolution*) and refine two (session-record *strength*, recovery *semantics*) from binary presence to graded form.

Reading with AI assistance. The source sweep used parallel AI agents to locate and excerpt code; every claim was then verified by hand against the source. We disclose this because it bears on reliability, and because the reproduction artefacts (below) are the durable check on it. Reading depth is asymmetric and we report it as such: DEEPAGENTS and PI were read line-by-line; DSH received one thorough pass plus a hands-on plugin experiment.

4 Divergent Origins (RQ1)

This section answers RQ1. As shown in Figure 2, the three harnesses occupy a single spectrum (how much the harness automates versus how much it leaves observable) and travel it in three ways: i) DEEPAGENTS subtracts authored scaffolding from the batteries-included end; ii) PI accretes durable infrastructure from the minimal end; and iii) DSH enters near the centre and, in one seam, literally reuses PI.

4.1 deepagents: the subtracting maximalist

Position. DEEPAGENTS is an assembly layer over LangChain's runtime: planning, filesystem, subagents, summarisation, memory, skills, permissions, and human-in-the-loop are *all* middleware, composed in a three-phase stack (`graph.py:816-893`).

Architecture. Storage is a backend protocol whose methods default to `NotImplementedError`, so capabilities are optional and probed by identity comparison. Context is defended in four automatic layers: a delta-channel reducer (replacing LangGraph's quadratic whole-list checkpointing), LLM summarisation near 85% of the window, eviction of oversized tool results to storage behind a head/tail preview and a pointer, and cheap truncation of stale tool arguments first. Subagent isolation is careful: the `task` tool copies parent state minus messages and private fields inward and returns only a final message outward, with private fields late-bound after assembly.

[1] Repositories: https://github.com/langchain-ai/deepagents, https://github.com/earendil-works/pi, https://github.com/deepseek-ai/deepseek-harness.

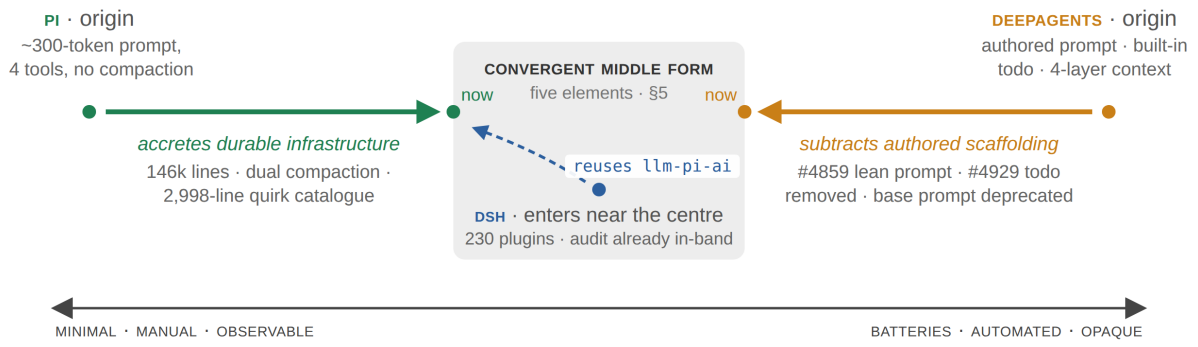

Figure 2: Divergent origins, converging trajectories. Positions are interpretive, not measured; arrows denote the direction of each project's commit history, not speed. DEEPAGENTS subtracts authored scaffolding; PI accretes durable infrastructure; DSH enters near the middle and, in one seam, literally reuses PI.

Trajectory. DEEPAGENTS subtracts. The commit history records a trajectory the project's public positioning does not: the authored base prompt is deprecated with a note that the harness "no longer provides an authored base prompt." PR #4859 stripped built-in tool-usage prose ("lean system prompt by default"). PR #4929 removed the todo/planning middleware from the default stack, leaving it only as an opt-in within one model's profile: planning reclassified from a harness default to a per-model need. Each step moves toward PI's stated positions.

4.2 pi: the accreting minimalist

Position. PI's public identity is minimalism [18], and the loop honours it: its agent package is 2,368 lines (the loop file, `agent-loop.ts`, is 796), readable end-to-end, with no `try/catch` in the loop at all (Section 5).

Architecture. Beneath the minimal loop, PI has built serious infrastructure: a 2,941-line formal specification defining three durable forms (immutable entries, mutable registers, append-only usage rows) plus a durable "program counter" for crash recovery that overwrites a single register per operation, so recovery reads the register and resumes without replaying the log.

Trajectory. PI accretes. The repository around the loop is 146,170 lines across ten packages. The harness that rejected subagents now reserves "lanes" in its session model naming subagents as a design target; the one that rejected compaction runs two implementations mid-migration; the one that dismissed per-model coaching maintains a 2,998-line generator producing one of the most detailed model-quirk catalogues in the open, with roughly fifteen compatibility flags for a single wire format. What it rejected in prose returned as data.

4.3 dsh: the plugin absolutist

Position. DSH is a TypeScript monorepo of ∼230 plugins over a vendored service-locator runtime.

Architecture. A plugin is any object implementing a service; consumers resolve a service by a stable context key and never import an implementation; load order is expressed declaratively via `inject`; registration is a reversible side effect that unwinds on reload. Even the agent loop is a configuration row (`ctx.agentLoop`), replaceable from config: "everything is a plugin" holds literally. Its session model is the strongest audit story of the three (Section 5), and its sandbox throws `SANDBOX_UNAVAILABLE` rather than run unconfined, with policy travelling per call rather

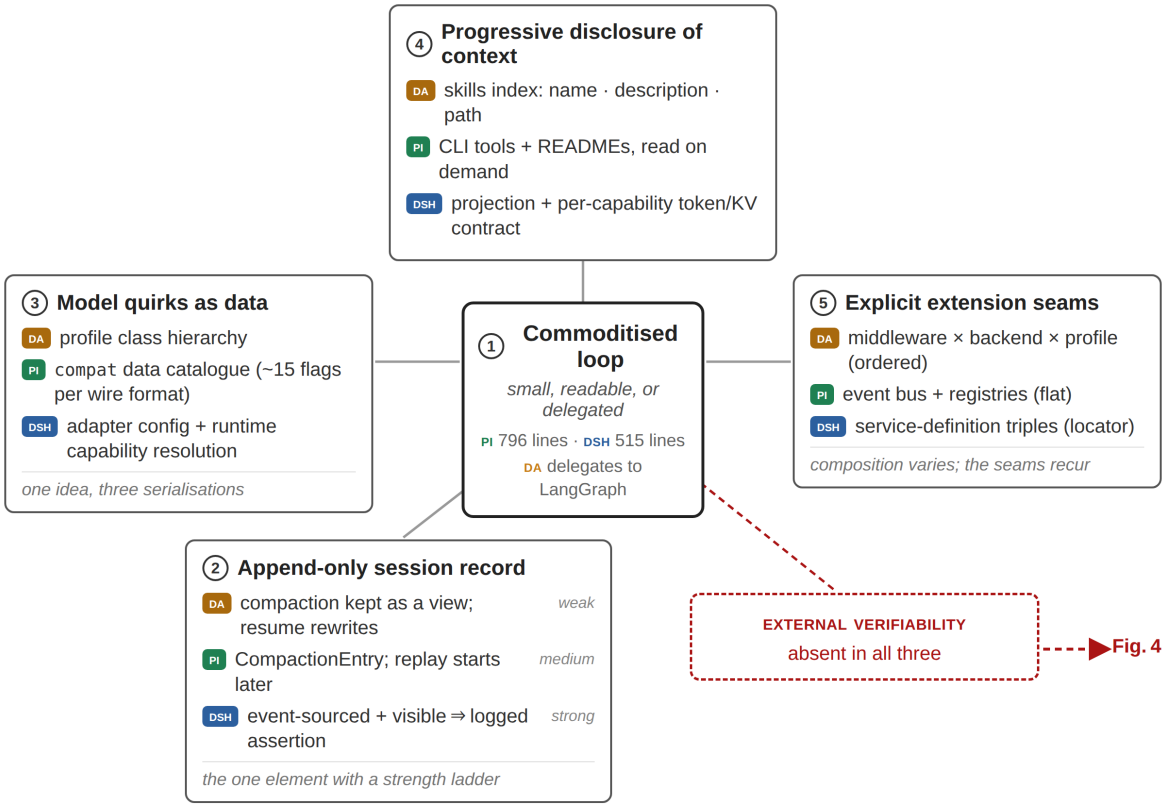

Figure 3: The convergent middle form. One idea, three serialisations: each element lists its DEEPAGENTS (da), PI (pi), and DSH instantiation. Solid boxes are converged elements; the dashed slot is the non-converged dimension of Section 7 (Figure 4).

than per provider.

Trajectory. DSH reuses. It did not travel far to reach the middle; it entered near it. Its generic provider adapter, `llm-pi-ai`, depends directly on PI's published `@earendil-works/pi-ai` package and is described in-tree as a "design-verification twin," a case we return to in Section 5.3.

5 The Convergent Form (RQ2)

This section answers the first half of RQ2: the destination. As Figure 3 shows, the form is a commoditised loop with the remaining four elements at its seams, each instantiated three different ways; the dashed slot marks the one dimension where no harness appears (Section 7). Table 2 gives the per-dimension evidence for the five elements.

5.1 Five recurring elements

(1) A commoditised loop. PI's agent-loop file is 796 lines; DSH's driver is 515; DEEPAGENTS does not own a loop at all, delegating to LangGraph. The element is not loop size but that none of the three competes on the loop: it is small, readable, or delegated wholesale, never a locus of differentiation, a claim a harness built around a large proprietary loop would falsify. A PI detail worth adopting is the never-throw stream contract; provider failures become a terminal assistant message with `stopReason` of `error` or `aborted`, and on `stopReason == length` *every* tool call in the message is failed, since any may carry truncated JSON.

Table 2: Nine dimensions across three harnesses. The five convergent elements of Section 5 are typeset in **bold**; the final row is the non-converged dimension of Section 7. Abbrev.: da = DEEPAGENTS.

Dimension	deepagents	pi	dsh
Philosophy	batteries-included middleware	minimal loop, extend outward	everything-is-a-plugin
Loop (commoditised)	none owned (delegates to LangGraph)	796-line loop file, no try/catch	515-line driver, *itself* a swappable config row
Append-only session	delta-channel; compaction kept as a *view* (weak)	append-only entries + registers (medium)	event-sourced + runtime "visible⇒logged" assertion + replace-op (strong)
Crash recovery	checkpoint; resume *rewrites* history	durable register; read-and-resume	*closes* an interrupted turn; refuses mid-stream corruption
Model quirks as data	profile class hierarchy	`compat` data catalogue (defaults inferred)	adapter config + runtime capability resolution
Progressive disclosure	skills index (name/desc/path)	CLI tools + READMEs, read on demand	projection + per-capability token/KV contract
Explicit seams	middleware / backend / profile (ordered)	event bus + registry (enumerated)	service-definition triples (locator)
Sandbox	permissions + HITL + paid remote backends	none ("YOLO"; containerise yourself)	bwrap/Landlock; throws rather than degrade
External verifiability	none (outsourced telemetry)	none (mutable private files)	operational only (not third-party checkable)

(2) An append-only, replayable session record, and here the three form a strength ordering. DEEPAGENTS keeps compaction as a *view* over an untouched history ("for replay, evals, and shared state," says the source), which is weak because recovery still rewrites messages. PI appends a `CompactionEntry` carrying `firstKeptEntryId`; nothing is destroyed and replay simply starts later, a medium form (this is PI's newer harness path; an older interactive compaction still overwrites history, the defect of Section 6). DSH is strongest: an event-sourced log with a runtime-asserted invariant that anything a model saw must be reconstructable from the log, history a projection rather than storage, and a `replace` surface-op that masks a span while preserving the original events. All three learned that mutating history is a trap; they differ in how far they enforce it.

(3) Model quirks as data. PI's `compat` catalogue, DEEPAGENTS's registered profiles, and DSH's adapter config with runtime capability resolution are one idea in three serialisations: a class hierarchy, a data directory, and a runtime resolver. DSH's is the most dynamic and, correspondingly, the least inspectable at rest.

(4) Progressive disclosure of context. All three put only a name, description, and path in the prompt and read the body on demand (DEEPAGENTS's skills index, PI's CLI-tools-with-READMEs, DSH's runtime projection), and both DEEPAGENTS and PI adopted the `AGENTS.md` convention and the filesystem as memory. DSH goes furthest, attaching a written per-capability contract stating each plugin's token and KV-cache cost.

(5) Explicit seams instead of a monolith. DEEPAGENTS exposes three orthogonal

axes (middleware, backend, profile); PI a flatter event bus plus registries; DSH named service-definition triples where one provider swap moves an entire capability world. The disagreement is composition style (ordered middleware versus event bus versus locator), not whether seams should exist.

Three findings emerge from Table 2. First, all five elements recur in all three columns despite opposing philosophies, consistent with the middle form being a property of the problem rather than of any single lineage. Second, no element recurs as the same code (a class hierarchy, a data directory, and a runtime resolver serialise one idea three ways), suggesting convergence of idea rather than of implementation. Third, only the session record forms a strength ladder (view $<$ append-only $<$ event-sourced with a runtime assertion) rather than a cluster, and that ladder points directly at the non-converged dimension of Section 7. The practical implication is direct: treat the five shapes as settled, adopt the strongest available form of each, and spend novelty on the dimensions that remain unsettled.

5.2 The residual disagreement is audience, not architecture

Where the three still differ is *how much to automate versus leave observable*. DEEPAGENTS automates context management because its users embed agents in products; PI keeps it manual because its user is a person at a terminal who wants to see everything. That is a difference in audience, not in architecture, and it predicts the shape of each project's remaining roadmap without contradicting the convergence.

5.3 Mechanisms of convergence

This subsection answers the second half of RQ2: the mechanism. Two teams arriving at the same design is a weak signal: they could have read each other. We therefore decompose the convergence into three mechanisms rather than assert independent invention.

Parallel discovery is visible where the same shape is reached from demonstrably different starting prose and different pressures: the append-only record reached by PI from crash pressure and by DEEPAGENTS from checkpoint-cost pressure; progressive disclosure reached by both before either shipped subagents. *Diffusion* we concede openly: PI and DEEPAGENTS are both public and mutually known, and shared conventions like `AGENTS.md` spread by imitation, not rediscovery. *Literal reuse* is the strongest and most concrete: DSH mounts PI's provider catalogue wholesale through `llm-pi-ai`, one team adopting another's solved problem rather than defending its own. This is convergence in its most concrete and verifiable form, and it is exactly the "adopt it, don't defend your own version" move the convergent form recommends, occurring between the projects themselves. The thesis does not require independence; it requires that the same shapes recur under shared pressure, which all three mechanisms support and none undercuts.

6 Convergent Fault Lines

Architectures are not the only thing shaped by a shared problem; *defects* may be too. The defects we reproduced or identified across DEEPAGENTS and PI (two re-derived in a sandbox, the rest located by source reading) each fall on one of four recurring seams (Table 3). We present these as illustrative case reports, not as an independent test: the four seams were induced from a small, analyst-directed sample, some are language-specific (S1's sync/async drift cannot arise in PI's single-threaded model), and S4 is a cross-cutting property rather than a fourth bin. They *suggest*, rather than establish, that these fault lines track the problem shape.

S1: Sync/async drift. In DEEPAGENTS, a hand-written synchronous tool guards

Table 3: The four seams on which every reproduced defect landed. Loss column: • = irreversible, ○ = recoverable, — = n/a. da = DEEPAGENTS.

Seam	Specimen	Mechanism → consequence	Loss
S1 sync/async drift	da, `async_subagents`	guard inside `try` in the sync twin, outside it in the async twin → unhandled `KeyError` on recovery	○
S2 normalisation trust gap	da, `utils.py:691`	`//secrets` ≠ `/secrets` by path component but = by filesystem → delete-deny, approval, and routing checks fail open	•
	PI, `overflow.ts:75`	throttle rethrown as a bare object, JSON-stringified past a caret-anchored regex → destructive compaction on a transient throttle	•
S3 string-matched semantics	PI, `overflow.ts:60`	error meaning decided by ∼25 regexes → a reworded rate-limit misread as context overflow	•
S4 silent vs. loud failure	all four specimens	an unrecognised shape defaults and runs on instead of stopping (cf. da `graph.py`, which raises)	—

a client lookup inside a `try`; its asynchronous twin performs the same lookup outside it (`async_subagents.py:445` and `:625`, against the guarded sync halves at `:421`/`:594`). A checkpointed task that references a since-renamed subagent therefore crashes the async caller with an unhandled `KeyError` while the sync caller degrades to a clean error; a third instance survives in the update tool's two branches (`:496`/`:535`). Paired async written by hand is two copies of one intent that the compiler never checks for agreement. *Rule.* Derive one twin from the other or share a single guarded core; where both are hand-written, diff their guard clauses and test both halves against the same failure. (Filed as issue #5640.)

S2: Normalisation trust gaps. A value crosses a boundary and the two sides disagree on its shape, with no enforced canonicalisation between. In DEEPAGENTS, `os.path.normpath` preserves exactly two leading slashes (POSIX-reserved), so `//secrets` compares unequal to `/secrets` by path component (`validate_path`, `utils.py:691`; `_paths_overlap`, `:607-616`) while the filesystem strips the redundant slash and resolves both to one file. Three component-based checks fail open: a delete deny-rule (`filesystem.py:558`), a human-approval interrupt (`_fs_interrupt.py:126`), and composite-backend routing (`composite.py:166-177`), while the adjacent `wcmatch`-based read/write/edit checks (`filesystem.py:420-430`), which normalise, are immune. In PI, a mid-stream Bedrock throttling error is rethrown as a bare object (`bedrock-converse-stream.ts:306`), fails an `instanceof` guard (`:385-390`), is JSON-stringified (`error-body.ts:38`), and slips past a caret-anchored negative regex written to catch exactly it (`overflow.ts:75`)—because the string now begins with a brace. Same disease, two languages. *Rule.* One mandatory canonicalisation point per trust boundary; downstream accepts only the canonical form.

S3: Semantics by string-matching. PI classifies provider errors with ∼25 regexes plus a negative list (`overflow.ts:60-142`); the Bedrock case is its failure mode. Deciding meaning from text means a reworded or re-wrapped message flips the decision, and here the cost of being wrong is a transient rate-limit routed into destructive compaction, a paid summarisation that permanently replaces conversation history via PI's older interactive compaction path (`agent-session.ts:2079`), not the append-only `CompactionEntry` of Section 5; the same path also fails to mark the error retryable (`retry.ts:26-44`), so the throttle is neither retried nor survived. *Rule.* Errors carry a typed kind; regex is a last-resort fallback that must fail toward the safe side: here, "do not compact" when unsure, since compacting a non-overflow is unrecoverable while declining to compact a real overflow is not.

S4: Silent degradation versus loud failure. Every defect above keeps running after the mistake: the bare object becomes a string and flows on, the `//` path routes to the wrong backend, the misclassified throttle proceeds to compaction. Contrast DEEPAGENTS's own configuration layer, which *raises* when an exclusion rule matches nothing rather than silently no-op'ing. A harness is long-running and autonomous; a wrong value that surfaces ten steps downstream costs far more to diagnose than a hard stop at the seam. *Rule.* At every boundary, an unrecognised shape is an error, not a default.

Three observations follow. First, the seams cut across philosophy: fault lines of the same kinds appear in both a batteries-included Python framework and a minimalist TypeScript harness (though not every seam in both, since S1 is Python-specific), suggesting they track the problem rather than either design. Second, the damage concentrates where the seam is irreversible: three of the four specimens end in unrecoverable loss (Table 3, last column), a recursive delete slipping a deny-rule, and conversation history overwritten by compaction, while the one recoverable specimen (S1) merely crashes. Third, the lesson generalises: the expensive seams are exactly the ones an autonomous system cannot walk back, so every automated destructive action (compaction, eviction, deletion) should be recorded and, where possible, reversible. That requirement is precisely what none of the three harnesses provides, and it carries us to the dimension of the next section.

7 The Boundary: A Dimension Without Convergence (RQ3)

This section answers RQ3. Across the dimensions on which the three converge, one load-bearing dimension shows no convergence and, more tellingly, no presence: a record an outside party can verify *without trusting the runtime that produced it*. As shown in Figure 4, the possibilities form a ladder of increasing verifiability, and every harness stops below the external-verifiability line.

The ladder ascends from a baseline with no durable record. DEEPAGENTS occupies the rung above it: its checkpoints exist for resumption, not evidence, and its resume path rewrites message history to patch dangling calls. PI climbs one higher: its sessions are append-only on the newer path, but they remain files the process owns, mutable and unverifiable by a third party. DSH climbs highest and is the instructive exception, demonstrating that the gap is substantive: durable approval records, the visible-implies-logged invariant, recovery that closes rather than truncates. Yet even DSH stops below the line, aiming at *operational* reconstructability (can the runtime rebuild what happened) rather than *external* verifiability, and its own documentation is candid at the boundary: `ctx.tools.restrict()` is described as "a visibility composition, not a permission boundary." The two rungs above the line are empty in two distinct ways: no harness produces a tamper-evident record an outsider can check without trusting the runtime (rung 5), and none treats "what was this data as of," "may this result leave the building," or "prove the cost ceiling held" as first-class, provable questions (rung 6). The distinction matters because the second rung is the narrower, domain-shaped gap that remains even once generic verifiability exists.

We read this absence as a *predictive gap*, not an oversight. Nor is it absent for lack of proposals: protocol designs such as Autogenesis [9] make versioned lineage and auditable rollback their centrepiece, so auditability is actively prescribed top-down in the research literature while absent bottom-up in every harness we read: a selection-pressure asymmetry, not a failure of imagination. The three converged on everything a developer at a terminal needs; verifiability is precisely what that audience does not need, and so precisely where convergence stops. That makes it the natural next axis of competition for harnesses deployed where provenance is not optional. Consistent with our stated discipline, we characterise the gap and do not, here, advance a way to fill it.

Figure 4: The verifiability ladder. DSH climbs highest but stops below the external-verifiability line; the two rungs above it are unreached. Solid = occupied; dashed = open. The figure charts the gap only; it does not propose how to fill it.

8 Discussion and Implications

Three implications follow. First, **for harness builders**, the five elements are a checklist paid for three times under one selection pressure: if you are hand-rolling a loop, mutating session history, coating model quirks in conditionals, front-loading context, or fusing your extensions into a monolith, you are re-deriving a lesson already paid for. We scope that advice to the audience all three serve, a developer at a terminal in an attended, long-horizon session: a scaffold that runs unattended to a benchmark score pays for durability and disclosure it may never draw on, and we would not press the same checklist on it. Within that audience, adopt the settled shapes; do not defend your own version, advice the projects themselves follow, one reusing another's provider catalogue outright. The pull of the form is visible even outside our sample: the reference system accompanying the Autogenesis protocol [9], a research prototype built to demonstrate self-evolution rather than to serve developers, likewise pairs a planner that only plans with sub-agents behind explicit seams, generates capability contracts progressively to spare the prompt, coordinates through a structured `plan.md` artefact, and normalises provider quirks in a model manager, a further sighting of the middle form, subject to the same diffusion caveat as any other (Section 5.3).

Second, **for the "less harness" thesis**, the trajectories qualify it. DEEPAGENTS is indeed subtracting, which fits "stronger models need less scaffolding." But PI is simultaneously accreting durable infrastructure (session records, recovery, quirk catalogues) that has nothing to do with coaching the model and everything to do with surviving long autonomous runs. The lesson is

not “thinner everywhere” but “thinner in coaching, thicker in durability,” and how thin is set by audience.

Third, **for evaluation**, if the harness is the binding constraint [4] and harnesses are converging in form, then the residual performance differences increasingly live in the un-converged dimensions (recovery semantics, quirk coverage, and the verifiability others have not built) rather than in the loop everyone now shares. For future harness work, the implication is to adopt the settled shapes, differentiate on the unsettled dimensions, and treat the verifiable record as the likely next requirement where deployment domains demand provenance.

9 Threats to Validity

External validity. $N = 3$; all are coding-agent harnesses read at one point in time (August 2026), so generalisation to other agent shapes and to future revisions is limited. We mitigate by maximal-spread selection and by pinning revisions so claims are re-checkable.

Internal validity. The three are not fully independent (DSH depends on PI), so part of the convergence is diffusion, not rediscovery; Section 5.3 handles this by decomposing the mechanism rather than assuming independence. Case selection favours well-known, source-available projects, a survivorship bias we state plainly.

Construct validity. Two constructs carry different risk. The *dimensions* (Table 2 rows) were fixed a priori (Section 3) and aligned with a prior taxonomy [5], limiting fitting. The *five elements* (the central construct) were instead induced during reading of the two grounding cases and checked against one more, and we mark this as the primary construct risk; the inclusion rule was line-level evidence in both grounding cases. We also disclose that the verifiability dimension aligns with the first author’s own research interest, stated here so the reader can weight it. The upstream-merged fix confirms one point claim, not the framework; we scope it accordingly (Section 3).

Reliability. A single analyst read the source with AI assistance, which introduces both single-coder and instrument risk; DSH was read less deeply than the other two, and we mark DSH claims as documentation-level where we did not reach line evidence. Against this, every file:line is re-checkable at the pinned revisions and the reproduction artefacts are executable. A second-coder re-scoring of a sample of table cells, and inter-run agreement for the AI-assisted pass, are the natural next mitigations and are deferred to the scaled study. That study is underway as a companion paper, which re-expresses the dimensions of Table 2 as blind detectors bound by a mandatory file:line evidence contract, applies them in independent replicated runs across a stated population of harnesses, and so reports inter-run agreement and detector-versus-human agreement as measurements rather than as promises. The present paper’s labels are the calibration set that study is checked against; we therefore state them here without borrowing its results.

10 Conclusion

Three harnesses began from opposing philosophies and evolved in opposite directions, yet moved toward one middle form: a commoditised loop, an append-only replayable record, quirks as data, progressive disclosure, and explicit seams. The convergence is real but not independent: we saw parallel discovery, diffusion, and one case of literal reuse. When separate teams under the same pressure keep landing on the same shapes, those shapes are settled, and the engineering move is to adopt them rather than defend one’s own. The same current that reveals the settled parts throws the unsettled one into relief: external verifiability, a record an outsider can check

without trusting the runtime, is a dimension on which all three are not merely divergent but absent. That is where the next harness will differ from the three we read.

Data Availability

Reproduction artefacts (the two DSH plugins and the two sandbox defect reproductions) and the full per-dimension evidence table with file:line references are available in the supplementary material, to be archived with a DOI on release; the defect reproductions are released under coordinated disclosure. All source claims are verifiable at the pinned revisions and repositories of Section 3.

Acknowledgement of AI Assistance

The source sweep underlying this study used parallel AI agents to locate and excerpt code; every claim was subsequently verified by hand against the source by the author, who is solely responsible for all errors.

References

[1] J. Yang et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," arXiv:2405.15793 (NeurIPS), 2024.

[2] X. Wang et al., "Executable Code Actions Elicit Better LLM Agents," arXiv:2402.01030 (ICML), 2024.

[3] Q. Wu et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," COLM, 2024 (arXiv:2308.08155).

[4] Y. Zhang et al., "Stop Comparing LLM Agents Without Disclosing the Harness," arXiv:2605.23950, 2026.

[5] B. Rombaut, "Inside the Scaffold: A Source-Code Taxonomy of Coding Agent Architectures," arXiv:2604.03515, 2026.

[6] O. Ben Sghaier, H. Li, B. Adams, and A. E. Hassan, "Don't Blame the Large Language Model: How Agent Harness Evolution Shapes Coding Agent Quality," arXiv:2607.03691 (to appear, ACM TOSEM), 2026.

[7] Q. Meng et al., "Agent Harness for Large Language Model Agents: A Survey," Preprints 202604.0428, 2026.

[8] RUCAIBox, "Agent Systems with Harness Engineering," 2026. https://github.com/RUCAIBox/awesome-agent-harness

[9] W. Zhang, Z. Zhao, H. Wen, Y. Wu, C. Guo, M. Yin, and B. An, "Autogenesis: A Self-Evolving Agent Protocol," arXiv:2604.15034, 2026.

[10] X. Xu et al., "The Devil Is in the Interface: Evaluating How Tool Architecture Shapes Coding Agent Behavior," arXiv:2608.11386, 2026.

[11] S. Wong et al., "Confucius Code Agent: Scalable Agent Scaffolding for Real-World Codebases," arXiv:2512.10398, 2025.

[12] Anthropic, "Introducing the Model Context Protocol," Nov. 2024. https://www.anthropic.com/news/model-context-protocol

[13] R. Surapaneni, M. Jha, M. Vakoc, and T. Segal, "Announcing the Agent2Agent Protocol (A2A)," Google Developers Blog, Apr. 2025. https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/

[14] H.-a. Gao et al., "A Survey of Self-Evolving Agents: What, When, How, and Where to Evolve on the Path to Artificial Super Intelligence," arXiv:2507.21046, 2025.

[15] R. K. Yin, *Case Study Research and Applications: Design and Methods*, 6th ed. SAGE, 2018.

[16] P. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering*, 14(2):131–164, 2009.

[17] A. Brown and G. Wilson (eds.), *The Architecture of Open Source Applications*, 2011.

[18] M. Zechner, "What I Learned Building an Opinionated and Minimal Coding Agent," 2025. https://mariozechner.at/posts/2025-11-30-pi-coding-agent/