HARNESS HANDBOOK: MAKING EVOLVING AGENT HARNESSES READABLE, NAVIGABLE, AND EDITABLE

Anonymous authors
Paper under double-blind review

ABSTRACT

The capability of a modern AI agent depends on both its foundation model and its harness, which constructs prompts, manages state, invokes tools, and coordinates execution. As models, APIs, execution environments, and application requirements evolve, the harness must be modified to add capabilities and adapt existing behaviors. Reliable modification requires human developers or coding agents to first identify all implementation sites associated with the requested behavior, a task we call *behavior localization*. In production harnesses, this task is challenging because behavior is distributed across files, functions, execution stages, and state transitions, while modification requests describe intended behavior rather than code structure. Code search, repository indexing, and long-context processing facilitate source inspection, but still leave developers and agents to reconstruct how dispersed implementation sites jointly realize a behavior. We introduce **Harness Handbook**, a behavior-centric representation constructed automatically through static program analysis and LLM-assisted behavioral organization, which maps system behaviors to their source implementations. We also develop **Behavior-Guided Progressive Disclosure (BGPD)**, a workflow that guides coding agents from high-level behavior descriptions to relevant implementation details and verifies candidate locations against the current source. Evaluation on diverse modification requests from two open-source agent harnesses shows that planning with Handbook access improves behavior localization and edit-plan quality while using fewer planner tokens, with gains across request types and localization difficulty levels. These findings highlight behavior localization as a key component of harness evolution and demonstrate the value of explicitly connecting behavioral requirements to implementation sites before generating edits.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed as agents that interact with tools, APIs, and complex execution environments (Wang et al., 2024; Yao et al., 2022; Huang et al., 2025; Schick et al., 2023; Zhou et al., 2024; Xie et al., 2024; Yang et al., 2024). These interactions depend on the *harness*, which constructs prompts, manages state, invokes tools, and coordinates execution across components, translating model capabilities into system behavior. As models, APIs, and application requirements change, adapting this software becomes a recurring engineering challenge (Lin et al., 2026; Lopopolo, 2026; Zhong & Zhu, 2026).

Harness evolution involves adding capabilities, adapting existing behaviors, and refining execution workflows (Wang et al., 2025b; 2026). Human developers traditionally perform this work by tracing behavior through the repository, while coding agents are increasingly expected to do so from natural-language requests (Yang et al., 2024; Wang et al., 2025c; Zhong & Zhu, 2026; Zhu et al., 2026). For both human developers and coding agents, the first challenge is to find the code responsible for the behavior they need to change. We call the task of identifying all relevant implementation sites **behavior localization**. In production-scale harnesses, a single behavior may span nonadjacent functions, execution stages, and shared states (Ning et al., 2026), requiring developers to reconstruct their relationships before planning a complete edit. Coding agents must also work within limited context, making iterative exploration necessary and leaving scattered or rarely executed paths vulnerable to omission (Zhang et al., 2026; Tamoyan et al., 2026).

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.

For example, requiring an agent to confirm task completion three times affects more than the branch that returns a final result: the relevant state must also be initialized, updated, and reset consistently. A search that finds the obvious completion check may therefore produce an incomplete plan, even when the proposed edit to that check is correct. Behavior localization requires tracing the full set of related responsibilities, including code reached only under particular conditions. To determine the full scope of a change, developers and coding agents must connect each function's local operations to its execution conditions and shared state. A useful representation should make these relationships easier to trace, particularly when related logic spans modules or alternative execution paths.

Repository maps, code search, summarization, repository memory, and long-context editing help coding agents locate, inspect, and retain relevant code (Zhang et al., 2026; Bhola et al., 2026; Wang et al., 2025a; Tamoyan et al., 2026; Cao et al., 2026). However, these methods do not explicitly capture how code across files and execution paths collectively implements a behavior, or which locations a requested change may affect. Addressing this gap requires a representation that links behavioral requirements to their distributed implementation sites.

Harness Handbook provides this mapping by organizing repository knowledge around runtime behavior and linking each behavior to its source. Built automatically through static analysis and LLM-assisted organization, the Handbook lets developers and coding agents identify relevant behavior before examining its implementation. We pair it with **Behavior-Guided Progressive Disclosure (BGPD)**, which narrows the search through a hierarchy of behavior descriptions and verifies candidate sites against current source. On the same modification requests from two open-source harnesses, Handbook-Assisted planning improves localization and edit-plan quality while using fewer planner tokens. Our contributions are:

- We define **behavior localization** as finding all code locations that implement the behavior described in a modification request. This step is necessary before a complete edit can be planned.
- We introduce **Harness Handbook**, which organizes implementation knowledge by what the harness does and links each behavior directly to its source code. We also develop an automated framework that builds Handbook instances from existing codebases using static program analysis and LLM-assisted behavioral structuring.
- We introduce **Behavior-Guided Progressive Disclosure (BGPD)**, a workflow that guides coding agents from high-level behavior descriptions to relevant implementation details in stages.
- We evaluate **Harness Handbook** on diverse modification requests from two open-source harnesses. We compare Handbook-Assisted BGPD planning with planning without Handbook access on the same tasks. Handbook assistance improves behavior localization and edit-plan quality while using fewer planner tokens.

2 Related Work

Our work connects agent harness design, harness evolution, and repository representations through the problem of behavior localization.

Agent harness. Recent work treats the harness as a first-class runtime. *Code as Agent Harness* frames code as an executable, stateful substrate for reasoning, action, feedback, and coordination (Ning et al., 2026), while surveys describe prompt and context construction, state and memory management, tool use, and execution control (Meng et al., 2026; He et al., 2026). AutoGen and OpenHands provide modular orchestration and environment interaction (Wu et al., 2023; Wang et al., 2025c); Claude Code and Codex emphasize harness design for long-running coding agents (Rajasekaran, 2026; OpenAI, 2025). These works specify harness responsibilities and composition, whereas we study which components and state transitions in an existing harness jointly implement a behavior.

Harness evolution. Recent work makes the harness itself an object of adaptation (Yang et al., 2024; Wang et al., 2025c). HarnessX adapts harness primitives from execution traces, HarnessForge jointly evolves the harness and model policy, and HarnessFix repairs trace-diagnosed flaws (Chen et al., 2026c;b;a); related efforts study harness engineering and observability-driven optimization (Zhong & Zhu, 2026; Zhu et al., 2026; Lin et al., 2026). These methods construct, optimize, or repair a harness. Harness Handbook addresses the preceding localization problem: finding the sites affected by a requested behavior before either human-led or automated evolution.

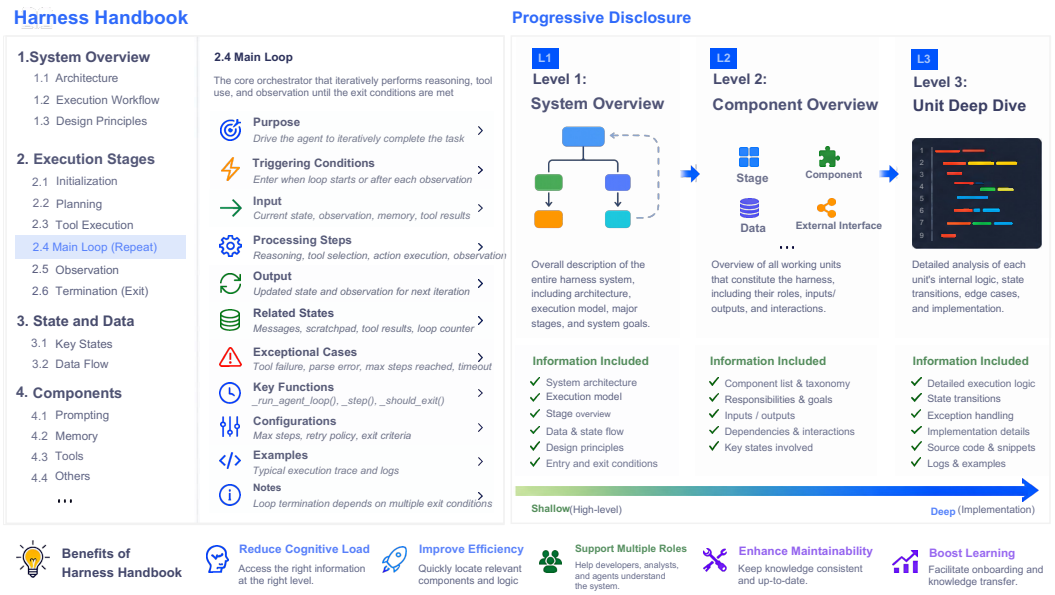

Figure 1: Overview of the Harness Handbook representation. Its three-level hierarchy progresses from a system-level overview to stage-level component overviews and source-backed unit details. The navigation pane provides component and state indexes for direct access and cross-stage tracing.

Behavior representations. Repository-understanding methods include neural search and agentic exploration (Gandhi et al., 2025; Zhang et al., 2026; Tamoyan et al., 2026), hierarchical and project-context summaries (Oskooei et al., 2025; Lomshakov et al., 2024), structural maps and indexes (Cherny-Shahar & Yehudai, 2026; Bhola et al., 2026), and repository memory, natural-language artifacts, and long-context processing (Wang et al., 2025a; Pan et al., 2026; Cao et al., 2026). They improve retrieval and diagnosis but remain organized around files, symbols, regions, or stored artifacts. A behavioral request may instead connect individually correct sites through execution order or shared state. Harness Handbook encodes these relationships through execution stages and state dependencies, while BGPD verifies candidate locations against current source. It therefore complements repository tools by exposing why dispersed sites share one modification scope and helping planners assess coverage.

3 Harness Handbook

Harness Handbook has three parts for understanding and modifying an agent harness. The *representation* organizes source around runtime behavior (Section 3.1). The *construction* pipeline builds this representation from a repository (Section 3.2). The *modification* workflow uses the handbook to guide code changes and automatically resynchronizes it after every non-empty repository diff (Section 3.3).

3.1 Harness Handbook Representation

Source repositories show where code is stored, but they do not directly show how a runtime behavior unfolds. One behavior may cross several files, execution stages, and shared states. Harness Handbook reorganizes this information around behavior while preserving links to the source. As Figure 1 shows, it contains an L1–L3 document tree $\mathcal{D}$ and a complementary state-register view $\mathcal{Z}$.

A reader normally starts at **L1 (system overview)**, which summarizes the architecture, execution model, major stages, and global data flow. The reader then moves to **L2 (component overview)** for the responsibilities, inputs, outputs, dependencies, and local state of a selected stage. Finally, **L3 (unit deep-dive)** links that stage to source-grounded implementation entries. The complementary view $\mathcal{Z}$ records state relationships that cross stage boundaries.

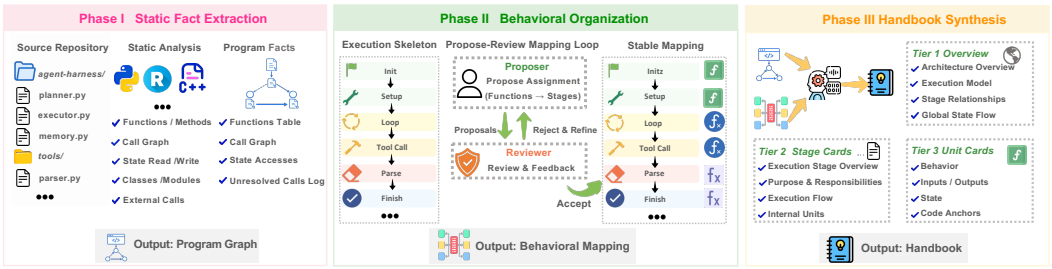

Figure 2: Construction pipeline for Harness Handbook. Static analysis extracts source-linked facts, behavioral organization maps source units to execution stages, and hierarchical synthesis builds the L1–L3 handbook.

The two views answer complementary navigation questions. The hierarchy $\mathcal{D}$ identifies where a behavior occurs in the execution lifecycle and narrows the search from the system to stages and implementation units. $\mathcal{Z}$ instead links units that access shared state across stage boundaries. For a completion-policy change, for example, the hierarchy routes the planner to termination behavior, while the register view exposes related initialization, update, and reset sites that must be considered together.

Two rules keep the representation useful. ***Progressive disclosure*** means that readers move from L1 to L3 only when a task requires more detail. ***Behavior–implementation alignment*** means that every active L3 locator must still resolve to the current repository. If a locator cannot be revalidated, its entry is frozen and excluded from localization until it is refreshed. The repository therefore remains the authority for implementation details.

3.2 Harness Handbook Construction

Construction builds a handbook from repository $\mathcal{R}$ under a leaf mode $g \in \{\text{function}, \text{file}\}$ fixed for its lifetime. The mode sets the granularity of L3 entries: in function-as-leaf mode, each entry covers a whole function or contiguous regions; in file-as-leaf mode, each represents a file.

The modes also obtain the stage skeleton differently. function-as-leaf starts from a trusted seed $\mathcal{S}_0$ defining stages and state registers, and is used when function-level entries fit the budget. file-as-leaf infers the skeleton when no suitable seed is available or function-level organization would exceed the budget. Both produce the representation in Section 3.1 through the branches of Algorithm 2. Construction then proceeds in three phases, illustrated for the function-as-leaf branch in Figure 2.

Phase I. Static fact extraction. Language-specific adapters parse the repository and extract functions, named external boundaries, source locations, signatures, and call edges. The program graph $\mathcal{G}$ keeps only calls that resolve to an internal function or a named boundary. Unresolved calls are logged rather than assigned guessed targets. This phase is deterministic and makes no LLM calls.

Phase II. Behavioral organization. The pipeline organizes source units into the execution-stage skeleton $\mathcal{S}$ according to the selected leaf mode.

- ***Function-as-leaf.*** The pipeline uses source and call-graph context to propose function-to-stage assignments, which are refined through iterative review. A function can be assigned to one or more execution stages, either as a whole or as contiguous regions when it serves multiple behavioral roles. Accepted assignments and revisions update the stage skeleton $\mathcal{S}$.
- ***File-as-leaf.*** The pipeline summarizes the scanned files as cards and combines them with the program graph to infer the stage skeleton $\mathcal{S}$. It then organizes the files by execution stage. An optional proposal-and-review process iteratively refines this structure, while uncovered files and unresolved organizational issues remain explicitly recorded in $\mathcal{Y}$.

Appendix A provides the full organization procedures and validation checks for both modes.

Phase III. Hierarchical synthesis and packaging. This phase converts the stage skeleton and source organization from Phase II into the L1–L3 document tree and cross-stage state-register view. Each L3 entry is linked to a statically identified source location and validated against the current

repository. This keeps the handbook traceable to the source. Finally, the pipeline renders $\mathcal{V}$ and packages the structured data needed for source localization and future resynchronization.

Across all three phases, statically extracted facts remain authoritative. The model may organize source units and explain their behavioral roles, but it cannot alter file paths, function identities, signatures, source ranges, or resolved calls. Unresolved calls and unplaced units remain explicit rather than being assigned guessed targets. Before packaging, the pipeline resolves every active L3 locator against the repository; entries that cannot be revalidated are frozen and excluded from localization. Generated descriptions are cached so that later resynchronization can reuse unaffected content while refreshing only the portions touched by a change.

Coverage remains explicit in both leaf modes. In function-as-leaf mode, assignment review stops only when the skeleton and memberships stabilize or the organization budget is exhausted, with unresolved functions retained as unmapped. In file-as-leaf mode, omitted or failed file cards are retried, and files that still cannot be assigned remain visible in the coverage record $\mathcal{Y}$. Neither branch forces uncertain units into a plausible stage, which exposes construction gaps to later validation and resynchronization.

3.3 Handbook-Guided Modification and Resynchronization

After construction, the handbook guides repository modification. Given a request q, the workflow uses handbook $\mathcal{H}$ with its corresponding repository $\mathcal{R}$ while keeping leaf mode g fixed.

The workflow consists of four steps. First, ***Behavior-Guided Progressive Disclosure (BGPD)*** navigates from the requested behavior to candidate source locations and verifies them against the current repository, producing source-grounded evidence. Second, the planner converts this evidence into an edit plan $\mathcal{P}$ and action declarations Γ. Third, a separate executor applies $\mathcal{P}$. Finally, any non-empty diff triggers handbook resynchronization. Algorithm 1 summarizes the complete workflow.

Algorithm 1 Handbook-Guided Modification with Automatic Resynchronization

Require: request q, handbook package $\mathcal{H}$, corresponding repository snapshot $\mathcal{R}$
Ensure: plan $\mathcal{P}$ and declarations Γ; updated repository $\mathcal{R}'$, diff Δ, and handbook package $\mathcal{H}'$

$g \leftarrow \mathrm{LeafMode}(\mathcal{H})$
$\widehat{\mathcal{E}}_q \leftarrow \mathrm{BGPD}(q, \mathcal{H}, \mathcal{R})$
$(\mathcal{P}, \Gamma) \leftarrow \mathrm{Plan}(q, \widehat{\mathcal{E}}_q)$
$\mathcal{R}' \leftarrow \mathrm{Execute}(\mathcal{R}, \mathcal{P})$
$\Delta \leftarrow \mathrm{Diff}(\mathcal{R}, \mathcal{R}')$
if $\Delta \neq \emptyset$ **then**
    $\mathcal{H}' \leftarrow \mathrm{Resync}_g(\mathcal{H}, \mathcal{R}, \mathcal{R}', \Delta, \Gamma)$
else
    $\mathcal{H}' \leftarrow \mathcal{H}$
end if
return $(\mathcal{P}, \Gamma, \mathcal{R}', \Delta, \mathcal{H}')$

Behavior localization with BGPD. BGPD identifies candidate sites from coarse to fine and verifies them against the current repository. It first uses L1 and L2 of $\mathcal{D}$ to select stages directly relevant to the request, then follows $\mathcal{Z}$ to include stages coupled through shared state. Within this scope, it selects relevant L3 entries and retrieves their source locators.

BGPD next expands the candidates along the function-call graph in function-as-leaf mode or the induced file-call graph in file-as-leaf mode; external boundary nodes provide context but are never edit sites.

Finally, it resolves the candidate locators against $\mathcal{R}$ and retains those relevant to q, yielding evidence $\widehat{\mathcal{E}}_q$ with a file path, an optional function or region anchor, and a current source excerpt. The handbook guides the search, while the repository remains the basis for the edit plan.

The walkthrough in Appendix E illustrates this sequence for a request that raises the number of required completion confirmations. Behavioral navigation first identifies the Completion Gate and its pending-completion register. Following that register expands the scope beyond the return condition

to initialization and per-run reset logic. Source verification then confirms the relevant locations in the live repository before the planner proposes edits. The example shows how stage and state relationships recover nonadjacent responsibilities that a search for the final completion branch alone could miss.

Edit planning and execution. The planner converts $\widehat{\mathcal{E}}_q$ into an edit plan $\mathcal{P}$. Each edit block records a target file, an optional function or region anchor, a current source excerpt, and the intended change. The planner also records one action type—*modify*, *add*, or *remove*—for each target in $\Gamma = (\Gamma_{\text{modify}}, \Gamma_{\text{add}}, \Gamma_{\text{remove}})$. A rename is represented as one removal and one addition.

The executor applies $\mathcal{P}$ to $\mathcal{R}$ and produces the updated repository $\mathcal{R}'$. The resulting diff is $\Delta = \text{Diff}(\mathcal{R}, \mathcal{R}')$. As the factual record of changes, Δ drives handbook invalidation, while Γ only checks whether execution followed the plan.

Handbook resynchronization. Every non-empty diff Δ triggers Resync_g, which updates affected content when possible. First, it reparses the source, refreshes the program graph, and aligns repository versions to identify added, removed, and modified units: function-as-leaf uses line-independent function fingerprints, while file-as-leaf uses file-set differences and content hashes.

Second, if stage skeleton $\mathcal{S}$ remains valid, it refreshes only affected entries and their enclosing structure; otherwise, it reruns Algorithm 2 on $\mathcal{R}'$ with the same g and Θ.

Finally, unparsable or unclassifiable content is frozen or recorded rather than guessed. The pair $(\mathcal{R}', \mathcal{H}')$ starts the next request. Only classification, file assignment, within-stage organization, and description revision use model calls; all other operations are deterministic.

4 Experiments

We evaluate Harness Handbook on two open-source agent harnesses around three research questions, describing the experiment setup in Section 4.1 and reporting results in Section 4.2.

- **RQ1:** Does handbook-guided localization improve plan quality while reducing planning cost?
- **RQ2:** Does the handbook improve a weaker planner's localization agreement with reference plans from more capable models?
- **RQ3:** Do these gains persist across request types and localization difficulty levels?

4.1 Experimental Setup

Evaluation setting. We evaluate localization and edit planning using a read-only NexAU planner (Nex-AGI Team et al., 2025) powered by **DeepSeek-V4-Pro**, scoring its edit plan $\mathcal{P}$ for each natural-language request rather than the executed modification. The *Baseline* explores the repository directly, while the *Handbook-Assisted* arm follows a BGPD-consistent navigation policy with a handbook built from the same source. Both arms share requests, model, repository snapshot, tool permissions, and decoding settings, differing only in handbook access and navigation instructions. **GPT-5.5**, **Opus 4.8**, and **DeepSeek-V4-Pro** independently judge plan quality.

Evaluating the plan rather than the executed patch isolates localization and planning from executor behavior, test availability, and patch-application failures. Fixed repository snapshots prevent code drift between arms, while three independent judges reduce dependence on a single evaluator. The comparison therefore tests whether handbook-guided exploration produces a better-supported edit plan under matched conditions.

Benchmarks and modification requests. We evaluate **Terminus-2** (Merrill et al., 2026) and **Codex** (OpenAI, 2025), using the leaf-mode selection rule in Section 3.2: Terminus-2 has a reliable seed skeleton and uses function-as-leaf, while Codex uses file-as-leaf to infer a hierarchy at repository scale.

Each harness contributes 30 behavior-driven requests grouped into **Query (Q)**, changes to existing behavior without target-location hints; **Cross-file (CF)**, capabilities spanning files or modules; and

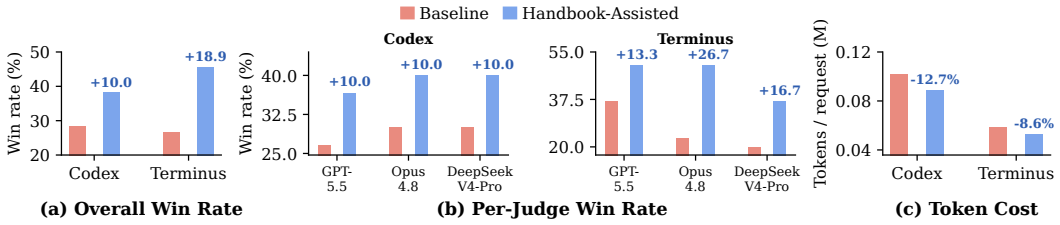

Figure 3: Plan quality and planner token usage on Codex and Terminus-2. (a) Overall win rates aggregated across the three judges. (b) Win rates reported separately by GPT-5.5, Opus 4.8, and DeepSeek-V4-Pro. (c) Average number of planner tokens per request; lower values indicate greater efficiency.

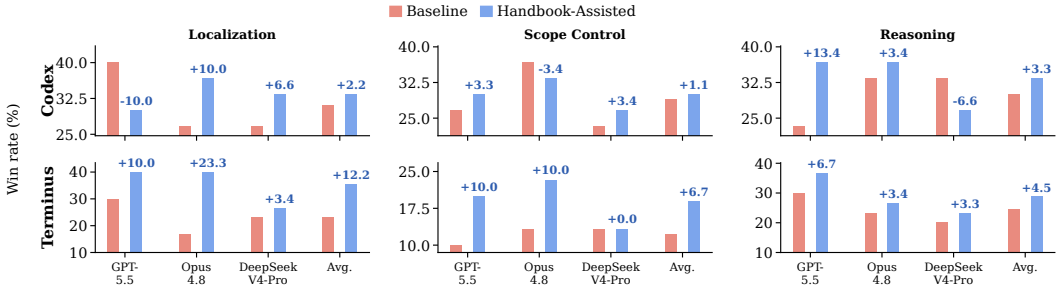

Figure 4: Per-judge win rates for three evaluation dimensions. Rows show results for Codex and Terminus-2, while columns show Localization, Scope Control, and Reasoning.

Search-hostile (SH), changes whose implementation sites are difficult to recover through keyword search. Requests are also labeled **Easy**, **Medium**, or **Hard** by localization difficulty.

The suites are evenly split across the three request types, and all requests omit target-location hints. The snapshots also expose different representation scales: Terminus-2 contains six Python source files and yields a function-level handbook with 20 stages and 106 L3 entries, whereas Codex spans 2,267 Rust files and yields a file-level handbook with 140 stages and 2,267 L3 entries. This contrast tests the same navigation abstraction at both fine-grained and repository-scale organization; Appendix C reports the complete program-graph and handbook statistics.

Plan quality. Before seeing either plan, each judge independently builds an answer key from the request and pristine repository. It then scores each arm against that key on three dimensions. *Localization* measures whether the plan identifies all required implementation sites, including mirrored, fallback, cold-path, state-reset, and template locations. *Scope Control* rewards a focused plan that avoids unrelated files or overly broad edits, while *Reasoning* measures whether the proposed scope is justified by the request and current source. The dimensions are combined as

$$S = 0.5S_{\text{Loc}} + 0.25S_{\text{Scope}} + 0.25S_{\text{Reason}}, \tag{1}$$

and reported on a 0–100 scale. Localization receives half of the total weight because identifying the correct edit sites is the primary objective. For each judge–request pair, the higher-scoring arm wins when its advantage is at least $\delta = 3\%$; smaller differences are ties. For any reported slice, the **win rate** divides wins by valid judge–request comparisons, so every request can contribute one comparison per judge.

Localization accuracy. As a complementary, location-level measure, we compare predicted edit sites with independent reference plans from Opus 4.8 and GPT-5.5. File-level matching tests whether the planner reaches the correct repository regions, while symbol-level matching tests whether it identifies the relevant functions or anchored implementation units within them. Recall measures reference-site coverage, Precision measures how much of the predicted scope is supported by the reference, and F1 balances the two. These metrics are macro-averaged across valid requests; missing outputs and planner errors are excluded. We also report *Wrong*, the share of valid requests

Table 1: Reference-plan localization metrics (%). Predicted implementation sites are compared against independent reference plans generated by Opus 4.8 and GPT-5.5 at file and symbol granularity. Metrics are macro-averaged across valid requests; missing outputs and planner errors are excluded. *Wrong* indicates zero overlap with the reference plan at the reported granularity, so lower values are better.

			Opus 4.8			GPT-5.5		
Harness	Level	Metric	Baseline	Handbook	Gap	Baseline	Handbook	Gap
Codex	File	Recall	55.4	69.7	+14.3	49.4	49.8	+0.4
		Precision	46.1	60.2	+14.1	53.0	62.2	+9.2
		F1	46.6	61.8	+15.2	47.3	52.3	+5.0
		Wrong ↓	37.0	14.8	-22.2	21.4	21.4	0.0
	Symbol	Recall	47.1	65.4	+18.3	46.4	49.1	+2.7
		Precision	38.0	55.3	+17.3	48.4	60.4	+12.0
		F1	38.3	57.1	+18.8	43.8	51.2	+7.4
		Wrong ↓	44.4	18.5	-25.9	28.6	21.4	-7.2
Terminus-2	File	Recall	74.7	83.9	+9.2	76.1	87.5	+11.4
		Precision	74.1	86.2	+12.1	78.3	93.3	+15.0
		F1	74.1	84.7	+10.6	76.5	89.3	+12.8
		Wrong ↓	24.1	13.8	-10.3	20.0	6.7	-13.3
	Symbol	Recall	64.7	76.3	+11.6	73.0	87.5	+14.5
		Precision	65.1	78.4	+13.3	73.9	93.3	+19.4
		F1	64.8	77.1	+12.3	73.0	89.3	+16.3
		Wrong ↓	24.1	13.8	-10.3	20.0	6.7	-13.3

with zero reference overlap at the reported granularity, where lower values indicate fewer complete localization failures.

Planning cost. We report the average number of planner tokens per request over localization and plan construction, including handbook and source context consumed during repository exploration. The metric excludes subsequent execution and judging, which are shared evaluation stages rather than part of the planner's search. Because both arms use the same model and decoding settings, the comparison captures differences in the context and navigation required to produce an edit plan.

4.2 Experiment Results

The results show three consistent patterns. Handbook guidance improves plan quality without increasing planner token use. It also helps a weaker planner produce file- and symbol-level predictions that more closely align with reference plans from stronger models. These gains persist across request types and localization difficulty levels. The following analyses examine each finding in turn.

Better plans at lower token cost. Figure 3 summarizes plan-quality win rates and average planner token use. The Handbook-Assisted arm achieves a higher overall win rate on both harnesses: 38.3% versus 28.3% on Codex and 45.6% versus 26.7% on Terminus-2. The improvement is consistent across all three judges: the gap is 10.0% for every judge on Codex and ranges from 13.3% to 26.7% on Terminus-2. Thus, the direction of the result does not depend on any single judge. Figure 4 decomposes these judgments by dimension. Averaged across the three judges, the Handbook-Assisted arm has higher win rates for Localization, Scope Control, and Reasoning on both harnesses. The respective gains are 12.2%, 6.7%, and 4.5% on Terminus-2, and 2.2%, 1.1%, and 3.3% on Codex.

The quality gains are accompanied by lower planner token use. Average use falls from 0.102M to 0.089M tokens per Codex request (12.7%) and from 0.058M to 0.053M tokens per Terminus-2 request (8.6%). The simultaneous increase in win rate and decrease in token use shows that the improvement is not explained by a larger planning-token budget. Taken together, the gains across all three quality dimensions and the lower token use are consistent with more directed repository exploration.

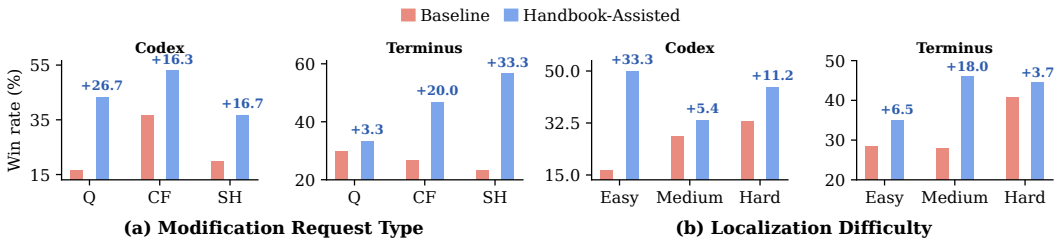

Figure 5: Win rates by (a) modification request type and (b) localization difficulty. Q, CF, and SH denote Query, Cross-file, and Search-hostile requests.

A weaker planner matches stronger models. To assess whether handbook guidance helps a weaker planner approach stronger models in implementation-site localization, we compare its predicted edit sites with independent reference plans from Opus 4.8 and GPT-5.5 (Table 1).

Across both harnesses, reference models, and granularities, all 24 Recall, Precision, and F1 comparisons favor the Handbook-Assisted arm, with F1 gains of 5.0%–18.8%. Recall and precision increase together, indicating broader coverage without reduced focus. Agreement is highest on Terminus-2, and the consistent direction against Opus 4.8 and GPT-5.5 shows that the improvement is not tied to one reference model.

Codex shows lower absolute agreement and greater variation across references, consistent with harder localization in the larger repository and some dependence on the reference plan. File-level overlap measures whether the planner reaches the correct repository regions, while symbol-level overlap tests the relevant implementation units within them. Handbook guidance also reduces complete failures: Wrong never increases and falls by as much as 25.9%, so the gains include fewer complete misses.

Consistent gains across requests and difficulty. Figure 5 stratifies the win-rate results by request type and localization difficulty. All six harness-by-type comparisons favor the Handbook-Assisted arm, with gains ranging from 16.3% to 33.3%. Codex improves most on Query requests (26.7%), whereas Terminus-2 improves most on Search-hostile requests (33.3%). Cross-file requests improve by 16.3% and 20.0%, respectively. The gains therefore span existing-behavior changes, cross-file capabilities, and requests that are difficult to localize through keyword search.

All six harness-by-difficulty comparisons are also positive, ranging from 3.7 to 33.3%. Codex improves most on Easy requests, whereas Terminus-2 improves most on Medium requests. Because the gains are not monotonic in labeled difficulty, the difficulty label alone does not explain their variation. Request type and labeled difficulty capture different aspects of localization: cross-file coordination and search-hostile paths can occur at multiple difficulty levels. Positive gains across both stratifications therefore suggest that no single request type or difficulty category accounts for the benefit.

5 Conclusion

Harness Handbook addresses behavior localization by linking runtime behavior to distributed source implementations through an automatically constructed hierarchy and cross-stage state relationships. BGPD uses this representation to guide source inspection and edit planning, while resynchronization keeps the handbook aligned with subsequent code changes. On Codex and Terminus-2, handbook assistance improves judged plan quality and localization agreement with stronger-model reference plans while reducing planner token use, with gains across request types and difficulty levels. These results support an explicit behavior-to-implementation mapping as a useful foundation for harness evolution, although the evaluation concerns planning rather than end-to-end modification success. Future work can assess behavior auditing, regression-impact analysis, and autonomous cycles of localization, planning, execution, and resynchronization as repositories evolve.

AI USE STATEMENT

We used generative AI tools to polish the manuscript's language, visualize the reported experimental data, and create workflow diagrams. The authors take responsibility for the final content of this paper.

Ethics statement. This work adheres to the ICLR Code of Ethics. Our study involves no human subjects, user studies, or collection of personal data. All experiments use publicly available models and established public benchmarks (e.g., Codex, Terminus-2, etc.), used in accordance with their licenses.

Reproducibility statement. We will publicly release our code upon publication to facilitate reproducibility.

REFERENCES

Ishaan Bhola, Adithyan Krishnan, Sravanth Kurmala, and Mukunda NS. Code isn't memory: A structural codebase index inside a coding agent. *arXiv preprint arXiv:2606.22417*, 2026.

Weili Cao, Xunjian Yin, Bhuwan Dhingra, and Shuyan Zhou. Coding agents are effective long-context processors. *arXiv preprint arXiv:2603.20432*, 2026.

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, and Qing Wang. From failed trajectories to reliable llm agents: Diagnosing and repairing harness flaws. *arXiv preprint arXiv:2606.06324*, 2026a.

Mingju Chen, Can Lv, Guibin Zhang, Heng Chang, and Shiji Zhou. Harnessforge: Joint harness and policy evolution for adaptive agent systems. *arXiv preprint arXiv:2606.01779*, 2026b.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, et al. Harnessx: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026c.

Tsvi Cherny-Shahar and Amiram Yehudai. Repository intelligence graph: Deterministic architectural map for llm code assistants. *arXiv preprint arXiv:2601.10112*, 2026.

Siddharth Gandhi, Luyu Gao, and Jamie Callan. Repository-level code search with neural retrieval methods. *arXiv preprint arXiv:2502.07067*, 2025.

Chaoyue He, Xin Zhou, Di Wang, Hong Xu, Wei Liu, and Chunyan Miao. Harness engineering for language agents: The harness layer as control, agency, and runtime. 2026.

Chengkai Huang, Junda Wu, Yu Xia, Zixu Yu, Ruhan Wang, Tong Yu, Ruiyi Zhang, Ryan A Rossi, Branislav Kveton, Dongruo Zhou, et al. Towards agentic recommender systems in the era of multimodal large language models. *arXiv preprint arXiv:2503.16734*, 2025.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. Proconsul: Project context for code summarization with llms. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 866–880, 2024.

Ryan Lopopolo. Harness engineering: Leveraging codex in an agent-first world. `https://openai.com/index/harness-engineering/`, February 2026. OpenAI Engineering Blog.

Qianyu Meng, Yanan Wang, Liyi Chen, Qimeng Wang, Chengqiang Lu, Wei Wu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://arxiv.org/abs/2601.11868`.

Nex-AGI Team, Yuxuan Cai, Lu Chen, Qiaoling Chen, Yuyang Ding, Liwen Fan, Wenjie Fu, Yufei Gao, Honglin Guo, Pinxue Guo, et al. Nex-n1: Agentic models trained via a unified ecosystem for large-scale environment construction. *arXiv preprint arXiv:2512.04987*, 2025. URL `https://arxiv.org/abs/2512.04987`.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

OpenAI. Introducing codex. `https://openai.com/index/introducing-codex/`, May 2025. OpenAI Research Preview.

Amirkia Rafiei Oskooei, Selcan Yukcu, Mehmet Cevheri Bozoglan, and Mehmet S Aktas. Repository-level code understanding by llms via hierarchical summarization: Improving code search and bug localization. In *International Conference on Computational Science and Its Applications*, pp. 88–105. Springer, 2025.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

Prithvi Rajasekaran. Harness design for long-running application development. `https://www.anthropic.com/engineering/harness-design-long-running-apps`, March 2026. Anthropic Engineering Blog.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Hovhannes Tamoyan, Sean Narenthiran, Erik Arakelyan, Mira Mezini, and Boris Ginsburg. Sherloc: Structured diagnostic localization for code repair agents. *arXiv preprint arXiv:2606.24820*, 2026.

Boshi Wang, Weijian Xu, Yunsheng Li, Mei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. *arXiv preprint arXiv:2510.01003*, 2025a.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Ruhan Wang, Zhiyong Wang, Chengkai Huang, Rui Wang, Tong Yu, Lina Yao, John Lui, and Dongruo Zhou. Federated in-context learning: Iterative refinement for improved answer quality. *arXiv preprint arXiv:2506.07440*, 2025b.

Ruhan Wang, Chengkai Huang, Zhiyong Wang, Junda Wu, Rui Wang, Tong Yu, Julian McAuley, Lina Yao, and Dongruo Zhou. Fera: Uncertainty-aware federated reasoning for large language models. *arXiv preprint arXiv:2605.10082*, 2026.

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

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, et al. Swe-explore: Benchmarking how coding agents explore repositories. *arXiv preprint arXiv:2606.07297*, 2026.

Hailin Zhong and Shengxin Zhu. Ai harness engineering: A runtime substrate for foundation-model software agents. *arXiv preprint arXiv:2605.13357*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Ningyan Zhu, Huacan Wang, Jie Zhou, Feiyu Chen, Shuo Zhang, Ge Chen, Chen Liu, Jiarou Wu, Wangyi Chen, Xiaofeng Mou, et al. Semaclaw: A step towards general-purpose personal ai agents through harness engineering. *arXiv preprint arXiv:2604.11548*, 2026.

CONTENTS OF THE APPENDIX

A Harness Handbook Construction Details **15**
A.1 Phase I: Shared Static Fact Extraction 15
A.2 Phase II: Behavioral Organization 15
A.3 Phase III: Hierarchical Synthesis and Packaging 17

B Modification and Resynchronization Details **18**
B.1 Behavior-Guided Progressive Disclosure 18
B.2 Edit Planning 18
B.3 Execution 18
B.4 Automatic Resynchronization 18

C Experimental Details **19**
C.1 Harnesses and Static Facts 19
C.2 Modification Requests 20
C.3 Evaluation Arms 20

D Prompt Templates **22**
D.1 Handbook Generation 22
D.2 BGPD Localization and Planning. 26
D.3 LLM Judging 27

E Illustrative Handbook-Assisted Planning Walkthrough **28**

A HARNESS HANDBOOK CONSTRUCTION DETAILS

This appendix details the three construction phases in Section 3.2, with leaf mode g and configuration Θ fixed throughout construction and subsequent resynchronization. Both modes use the same static facts: function-as-leaf assigns functions within a trusted seed skeleton $\mathcal{S}_0$, whereas file-as-leaf first describes files and then infers the skeleton. They produce the same reader-facing structure—an L1–L3 document tree, a cross-stage state-register view, and L3 source locators—through the shared workflow and mode-specific branches in Algorithm 2.

Algorithm 2 Harness Handbook Construction

Require: repository $\mathcal{R}$; leaf mode $g \in \{\text{function}, \text{file}\}$; construction configuration Θ; seed skeleton $\mathcal{S}_0$ if $g = \text{function}$ (otherwise $\emptyset$)
Ensure: handbook $\mathcal{H}$ containing rendered view $\mathcal{V}$, document tree $\mathcal{D}$, register view $\mathcal{Z}$, fixed leaf mode g, and synchronization state $\mathcal{K}_g$

Phase I: shared static facts.
$\mathcal{G} \leftarrow \textsc{BuildGraph}(\mathcal{R})$ # deterministic; no LLM

Phase II: mode-specific behavioral organization.
if $g = \text{function}$ **then**
  $(\mathcal{S}, \mathcal{U}_g) \leftarrow \textsc{OrganizeFunctions}(\mathcal{R}, \mathcal{G}, \mathcal{S}_0, \Theta)$ # budgeted stage and membership review
  $\mathcal{U}_g \leftarrow \textsc{FinalizeFunctions}(\mathcal{R}, \mathcal{G}, \mathcal{S}, \mathcal{U}_g)$ # regions, ordering, and unmapped functions
else # $g = \text{file}$
  $(\mathcal{S}, \mathcal{U}_g) \leftarrow \textsc{OrganizeFiles}(\mathcal{R}, \mathcal{G}, \Theta)$ # cards, stage inference, and assignments
  if $\textsc{RefinementEnabled}(\Theta)$ **then**
    $(\mathcal{S}, \mathcal{U}_g) \leftarrow \textsc{RefineFileOrganization}(\mathcal{R}, \mathcal{G}, \mathcal{S}, \mathcal{U}_g, \Theta)$
  end if
  $\mathcal{U}_g \leftarrow \textsc{FinalizeFiles}(\mathcal{G}, \mathcal{S}, \mathcal{U}_g)$ # within-stage order and coverage
end if

Phase III: synthesis, grounding, and packaging.
$(\mathcal{D}, \mathcal{Z}, \mathcal{B}) \leftarrow \textsc{Synthesize}(g, \mathcal{R}, \mathcal{G}, \mathcal{S}, \mathcal{U}_g, \Theta)$
$\mathcal{Z} \leftarrow \textsc{ValidateRegisterRefs}(\mathcal{Z}, \mathcal{S})$
$\mathcal{D} \leftarrow \textsc{ValidateAndFreezeLocators}(\mathcal{D}, \mathcal{R}, g)$
$\mathcal{K}_g \leftarrow \textsc{Archive}(g, \mathcal{G}, \mathcal{S}_0, \mathcal{S}, \mathcal{U}_g, \mathcal{B}, \Theta)$ # $\mathcal{S}_0$ is stored only in function mode
$\mathcal{H} \leftarrow \textsc{RenderAndPackage}(\mathcal{D}, \mathcal{Z}, g, \mathcal{K}_g)$
return $\mathcal{H}$

A.1 PHASE I: SHARED STATIC FACT EXTRACTION

Phase I uses language-specific adapters to parse $\mathcal{R}$ and build $\mathcal{G}$ through a deterministic analysis shared by both modes. An internal-function node records the qualified name, source file, signature, line range, enclosing class, and observed state access, while a boundary node represents a named external target such as a library function.

The graph keeps a call edge only when the target resolves to an internal function or a named boundary; unresolved calls are written to an audit log rather than assigned a guessed target. Because this phase uses no LLM, every later organization step builds on the same source-grounded facts.

A.2 PHASE II: BEHAVIORAL ORGANIZATION

Phase II organizes the facts in $\mathcal{G}$ around execution behavior, producing the stage skeleton $\mathcal{S}$ and mode-specific organization state $\mathcal{U}_g$. The leaf modes differ in both their starting point and their unit of organization.

A.2.1 FUNCTION-AS-LEAF

Initial assignment. The function-as-leaf branch ($g = \text{function}$) initializes $\mathcal{S}$ from the stage and state-register declarations in $\mathcal{S}_0$, then attempts to assign every analyzable internal function to one or more stages. Eligible functions are non-synthetic and have usable source locations; their assignments draw on source in $\mathcal{R}$, callers and callees in $\mathcal{G}$, neighboring assignments, and stage descriptions. Functions are normally placed as a whole, but those serving distinct behavioral roles may be divided into contiguous regions and assigned by region.

Review and convergence. A first pass may expose weaknesses in either the assignments or the seed structure, so subsequent rounds review both by proposing stage additions, removals, merges, or splits. A structural change is accepted only if the resulting hierarchy is valid and every state-register declaration refers to an existing stage, after which affected functions are reclassified.

After reclassification, a global audit checks whole-function placements against the surrounding stages, while a separate pass repairs retained region boundaries; proposed changes must pass model-based review before application. Iteration ends when the skeleton and all assignments match the previous round and no function awaits reconsideration, or when the organization budget in Θ is exhausted, with remaining gaps explicitly recorded.

Validation and finalization. Every accepted assignment must identify a valid stage, use a legal source range, and avoid conflicting actions. Region boundaries are aligned to parser-derived statement spans where possible, with the enclosed source hashed for later comparison; unaligned boundaries are marked *needs-review* for Phase III to determine whether their locators remain active or must be frozen.

Deterministic cleanup removes a redundant whole-function entry when retained regions already cover that function in the same stage, without disturbing its assignments to other stages. The pipeline then rebuilds cross-stage references, records all functions that remain unmapped, and orders entries within each stage using their purposes and stage-internal calls. If the proposed order is incomplete or contains duplicates, file and line order provides a stable fallback. Together, the finalized memberships, region anchors and hashes, boundary status, within-stage order, and record of unmapped functions constitute $\mathcal{U}_{\text{function}}$.

A.2.2 FILE-AS-LEAF

The file-as-leaf branch ($g = \text{file}$) operates without a seed skeleton ($\mathcal{S}_0 = \emptyset$), first describing each file, then inferring $\mathcal{S}$ from those descriptions, and finally organizing files within each stage.

Step 1: Build a card for each file. The pipeline attempts to create a card for every scanned file, including those without statically extracted functions, describing its purpose, repository role, and place in the execution lifecycle. Deep card generation additionally includes a static inventory of function identities, signatures, source ranges, and resolved calls, which the model may explain but cannot alter.

Card generation follows a fixed fallback sequence: files omitted from a batch are retried individually, and failed deep cards are retried using source chunks bounded by function ranges. If prose generation still fails, a fallback card retains the graph-derived inventory in deep mode and marks the file as undescribed in $\mathcal{Y}$. The file remains available to later steps, ensuring that no scanned file is silently dropped.

Step 2: Infer stages and assign files. The pipeline groups cards by directory and combines their descriptions with graph-derived entry points and resolved calls to draft an ordered stage skeleton. It then attempts to assign every scanned file to exactly one primary stage, using its card when available; cross-cutting utilities for logging, protocols, types, or configuration may also reference one or two secondary stages. Missing assignments and assignments to unknown stages remain visible in $\mathcal{Y}$ rather than being forced into an unsuitable stage.

The organization variant in Θ determines refinement: the default `oneshot` ends after one stage draft and one assignment pass, whereas `doctor` applies a reviewed refinement loop to that draft. `agent` applies the same loop to an agent-generated draft, falling back to the one-shot draft if generation fails. Each round may add, remove, merge, or split stages and then reconsider only affected or unassigned files. The loop ends when assignments are complete and the skeleton is stable, neither coverage nor stage balance improves for two consecutive rounds, or the organization budget is exhausted; unassigned files remain visible in $\mathcal{Y}$.

Step 3: Organize files within each stage. For each non-empty stage, the file-call graph supplies a caller-before-callee order that file cards refine into thematic groups and a readable sequence. Deterministic validation removes unknown or duplicate paths and requires every assigned file to appear

exactly once, appending omissions to a fallback group and reverting to the flat order if organization fails. This step may reorder files but cannot change the skeleton. The file cards, stage assignments, within-stage grouping and order, and coverage record $\mathcal{Y}$ together constitute $\mathcal{U}_{\text{file}}$.

A.3 Phase III: Hierarchical Synthesis and Packaging

Phase III turns $(\mathcal{S}, \mathcal{U}_g)$ into the document tree $\mathcal{D}$ and state-register view $\mathcal{Z}$, records reusable generation outputs in the cache $\mathcal{B}$, and validates both against the repository before packaging.

Synthesizing the document tree. In both modes, $\mathcal{D}$ contains an L1 system overview, an L2 component overview for each included stage, and source-backed L3 entries. The modes differ in the direction of synthesis:

- **function-as-leaf builds top-down.** The known stage skeleton guides generation from L1 through each L2 component overview to its L3 entries. Each L3 entry represents one function within a stage, combining any retained regions of that function into a single entry.
- **file-as-leaf builds bottom-up.** Assigned file cards form L3, and stages with assigned files or child stages are summarized from the leaves upward, skipping empty placeholders. Child summaries and descriptions of a stage's own files produce L2, while top-level summaries produce L1.

In function-as-leaf mode, each L1, L2, and L3 node undergoes a bounded generate–review–revise process that retains the best draft and stops when it passes the rubric, its score no longer improves, or the budget is exhausted. Independent stages may be processed in parallel, while L3 entries within a stage are generated sequentially so later entries can refer to earlier ones.

In file-as-leaf mode, file cards are rendered directly, while stage and system summaries are generated once with deterministic fallbacks, each parent waiting for its child summaries.

The resulting descriptions and summaries are retained in $\mathcal{B}$ so that unchanged content can be reused during later generation or resynchronization.

Building the state-register view. In function-as-leaf mode, the register identities and stage relationships in $\mathcal{Z}$ come from the declarations in the current stage skeleton; generation turns these declarations into reader-facing descriptions without inventing new registers. In file-as-leaf mode, the pipeline instead proposes cross-stage registers from the top-level stage summaries and the purposes of data-model files. It repeats this search until two consecutive rounds add nothing or the register budget is reached. References to stages absent from the current structure are removed from the candidate mapping.

Grounding and failure handling. The static facts in $\mathcal{R}$ and $\mathcal{G}$ remain authoritative: the model may add behavioral explanations but cannot alter file paths, function identities, signatures, source ranges, or resolved calls. Unplaced functions and files remain explicitly recorded without inferred assignments.

Before rendering, the pipeline resolves every L3 locator in $\mathcal{D}$ against $\mathcal{R}$ and compares the located source with the stored evidence. An entry that cannot be revalidated is marked frozen and excluded from localization until it is refreshed, keeping the repository authoritative even when prose is reused from the cache.

Rendering and saved synchronization state. After validation, the pipeline renders $\mathcal{D}$ and $\mathcal{Z}$ as view $\mathcal{V}$ and packages them with the fixed g and machine-readable resynchronization state $\mathcal{K}_g$ to form $\mathcal{H}$. In both modes, $\mathcal{K}_g$ retains the leaf mode, program graph, current stage skeleton, organization state, construction configuration, and generation cache. The configuration Θ specifies the organization or refinement variant, card-detail level, and applicable budgets for organization, generation, and register extraction. Function mode additionally retains the original seed $\mathcal{S}_0$, while file mode stores none, allowing later changes to refresh affected content while reusing unchanged parts.

B MODIFICATION AND RESYNCHRONIZATION DETAILS

This appendix follows request q through the four-step workflow in Section 3.3—BGPD localization, planning, execution, and resynchronization—using the package contract in Appendix A. Throughout, handbook $\mathcal{H}$ remains paired with its matching repository $\mathcal{R}$ under the fixed leaf mode g.

B.1 BEHAVIOR-GUIDED PROGRESSIVE DISCLOSURE

BGPD guides a read-only agent through progressively finer handbook details, admitting a locator to the evidence set only after it has been resolved against the current repository.

Stage selection. The agent begins with the L1 system overview and L2 stage index in $\mathcal{D}$ to select stages whose behavior matches q, then follows $\mathcal{Z}$ to include stages coupled through shared state. This expansion captures dependencies between structurally distant stages, such as a value written in one stage and consumed several stages later.

Entry selection. Within each selected stage, the agent chooses relevant L3 entries from their summaries and source locators, opening entry bodies only when needed to limit unnecessary context.

Call-relation expansion. The candidate set is expanded along call relations in the program graph $\mathcal{G}$ retained by the synchronization state $\mathcal{K}_g$. In function-as-leaf mode, expansion follows the function-call graph; in file-as-leaf mode, it follows the induced file-call graph. Named external boundary nodes may supply context but are never returned as edit sites.

Source verification. After navigating $\mathcal{H}$, the agent opens $\mathcal{R}$ to resolve candidate locators and retain only sites relevant to q, producing verified evidence $\widehat{\mathcal{E}}_q$. Each record carries a file path, an optional function or region anchor, and a current source excerpt, preserving the repository as the authority for a handbook-guided edit plan.

B.2 EDIT PLANNING

The planner converts $\widehat{\mathcal{E}}_q$ into plan $\mathcal{P}$, with each block specifying a target file, optional anchor, supporting source excerpt, and intended change, expressed as a verbatim old/new pair in our evaluation implementation. Accompanying action declarations are collected as $\Gamma = (\Gamma_{\text{modify}}, \Gamma_{\text{add}}, \Gamma_{\text{remove}})$, with a rename encoded as a removal and an addition.

B.3 EXECUTION

Execution is delegated to a separate agent that applies $\mathcal{P}$ to $\mathcal{R}$, producing $\mathcal{R}'$ and the factual change record $\Delta = \text{Diff}(\mathcal{R}, \mathcal{R}')$.

The executor can read and write files, perform targeted replacements, and search within files, but cannot list directories or access a shell. It attempts each verbatim replacement directly, reading source only on failure, then verifies the affected files and runs a target-specific syntax check when configured. The resulting Δ drives handbook invalidation, while Γ serves only to audit whether execution followed the plan.

B.4 AUTOMATIC RESYNCHRONIZATION

Every non-empty Δ automatically invokes Resync_g, which updates only the affected parts of the handbook whenever possible. If $\Delta = \emptyset$, resynchronization is skipped and $\mathcal{H}' = \mathcal{H}$.

Version alignment. The procedure reparses $\mathcal{R}'$ to build $\mathcal{G}'$ and compares it with the stored $\mathcal{G}$, using Δ to identify added, removed, and modified units.

In function-as-leaf mode, body fingerprints independent of line numbers identify matching functions, so a function that only moves remains unchanged apart from its locator offset; matching the

body below the signature detects renames. In file-as-leaf mode, file-set differences identify additions and removals, while content hashes identify modifications, with a path rename represented as a removal and an addition.

Before the observed actions are compared with Γ, a detected function rename is normalized to one removal and one addition. This audit does not alter the Δ-driven invalidation scope.

Scoped update. If the current skeleton $\mathcal{S}$ accommodates the change, $\mathcal{S}' = \mathcal{S}$ and affected organization state is updated from $\mathcal{U}_g$ to $\mathcal{U}'_g$. In function-as-leaf mode, this removes deleted memberships, carries forward matched functions, reclassifies new or changed functions, and refreshes region anchors, source hashes, order, and unmapped status. In file-as-leaf mode, it removes obsolete cards and assignments, refreshes cards for new or changed files, assigns new files, reorganizes affected stages, and updates coverage from $\mathcal{Y}$ to $\mathcal{Y}'$.

Regenerating affected L3 entries, their L2/L1 ancestors, and dependent state registers produces $\mathcal{D}'$ and $\mathcal{Z}'$. Unaffected outputs are reused from $\mathcal{B}$, while $\mathcal{B}'$ retains those entries and adds or replaces refreshed outputs.

If $\mathcal{S}$ can no longer accommodate the change, Algorithm 2 is rerun on $\mathcal{R}'$ with the same leaf mode g and stored configuration Θ. The function-as-leaf branch also reuses the stored seed skeleton $\mathcal{S}_0$. The file-as-leaf branch instead infers a new stage skeleton from the updated repository. During the full rebuild, Phase I produces $\mathcal{G}'$, Phase II produces $(\mathcal{S}', \mathcal{U}'_g)$, and Phase III synthesis produces $(\mathcal{D}', \mathcal{Z}', \mathcal{B}')$. These components follow the shared validation and packaging contract described below.

Conservative handling. Existing L3 entries whose locators or source hashes fail revalidation are frozen and excluded from localization. In function-as-leaf mode, new or changed functions that cannot be classified remain explicitly unmapped in $\mathcal{U}'_{\text{function}}$, with any previously active entry frozen until classification succeeds. In file-as-leaf mode, files without valid cards or assignments remain in $\mathcal{Y}'$ within $\mathcal{U}'_{\text{file}}$, preserving coverage information when parsing or classification fails.

Parsing, hashing, graph construction, locator updates, and coverage checks are deterministic, with model calls confined to four categories even during a full rebuild: classification of stages and functions, file-to-stage assignment, within-stage file organization, and revision of card, document-node, or register descriptions.

Validation and packaging. Both update paths then follow the packaging contract of Appendix A: the pipeline validates the locators in $\mathcal{D}'$ against $\mathcal{R}'$ and the stage references in $\mathcal{Z}'$ against the updated skeleton, archives $\mathcal{G}'$ and $(\mathcal{S}', \mathcal{U}'_g)$ with $\mathcal{B}'$, the fixed g, and Θ (plus $\mathcal{S}_0$ in function mode) as $\mathcal{K}'_g$, and renders and packages the validated $(\mathcal{D}', \mathcal{Z}')$ with $\mathcal{K}'_g$ as $\mathcal{H}'$. The pair $(\mathcal{R}', \mathcal{H}')$ becomes the starting state for the next request.

C Experimental Details

This appendix expands the setup in Section 4.1, covering the two harnesses and their Phase I static facts, 30 modification requests per harness balanced across type and localization difficulty, and the evaluation arms and `SKILL.md` interface. All factors other than handbook access and the corresponding navigation instructions are shared; evaluation metrics are defined in Section 4.1.

C.1 Harnesses and Static Facts

We evaluate two production-style agent harnesses with distinct scales and implementation languages.

- **Terminus-2** is a Python terminal agent from the Harbor framework that uses a tmux session in an observe–decide–act loop, reading terminal state, querying an LLM, executing commands, and observing their results until completion or a limit. Its 6 source files contain a multi-stage iteration loop, context management, and state carried across iterations.

- **Codex** is the Rust monorepo behind the Codex coding agent, spanning command-line and terminal user interfaces, an application server, configuration, and sandboxing across many crates, with thousands of files and deep call graphs.

Following the selection rule in Section 3.2, Terminus-2 uses the function-as-leaf mode, since a reliable seed skeleton is available and function-level entries stay within budget, whereas Codex uses the file-as-leaf mode to infer the hierarchy at repository scale. Table 2 reports the Phase I static facts extracted from each repository together with the resulting handbook structure.

Table 2: Phase I static facts and resulting handbook structure for the two evaluated harnesses. The upper block reports the program graph $\mathcal{G}$ produced by static analysis (Appendix A); the lower block reports the built handbook. For Terminus-2, an L3 entry covers a whole function or one or more contiguous regions of a function; for Codex, an L3 entry corresponds to one file.

		Terminus-2	**Codex**
Setup	Language	Python	Rust
	Leaf mode	function-as-leaf	file-as-leaf
Program graph (Phase I)	Source files	6	2,267
	Internal-function nodes	103	34,363
	Boundary nodes	41	4,016
	Resolved call edges	257	159,960
Built handbook	Stages (L2)	20	140
	L3 entries	106	2,267
	State registers	10	62

The substantial differences in repository size and call-graph scale illustrate the need for different leaf granularities under the stated selection rule.

C.2 Modification Requests

Each harness contributes 30 behavior-driven modification requests, evenly split across three types and expressed without target-location hints. The suites share behavioral intents but phrase them against each harness's code, so a common goal maps to different implementation sites in Terminus-2 and Codex.

- **Query (Q)** modifies existing behavior, such as a trigger, a timing rule, or a control-flow decision.
- **Cross-file (CF)** adds an end-to-end capability that spans files, schemas, pipeline logic, or interfaces.
- **Search-hostile (SH)** places relevant sites in mirrored implementations, fallback branches, or cold paths that keyword search alone struggles to recover.

Table 3 gives a representative request of each type, chosen separately for each harness.

Requests are additionally labeled Easy, Medium, or Hard by localization difficulty, reflecting whether the change touches one primary behavior, requires coordination across stages and files, or requires discovering dependencies along indirect execution paths. Table 4 gives a representative request at each level for both harnesses.

C.3 Evaluation Arms

Both arms use the same read-only NexAU planner powered by DeepSeek-V4-Pro, with identical requests, repository snapshot, planning contract, and decoding settings. They differ only in handbook access and the corresponding localization instructions, with the assisting handbook built from the evaluated snapshot.

- **Baseline.** Without a handbook, the planner locates edit sites by exploring the repository with read-only tools (file read, in-file search, and directory listing).

Table 3: Representative modification requests, chosen separately for each harness, with the localization challenge each type poses.

	Terminus-2	Codex	Localization challenge
Q	Require marking `task_complete` three times, with an "are you sure?" re-prompt, before grading.	Raise `exec_command`'s wait ceiling to 300 000 ms, keeping schema bound and docs in agreement.	An existing behavior must be found and altered; the difficulty is telling the target apart from similar triggers or rules.
CF	Add a `background` flag that sends keystrokes without capturing output, across the terminal layer and both formats.	Add a per-command `env` override, threaded through the spawn path across both shell tools.	The capability spans coordinated sites (schema, pipeline, interface, docs) that must all be located together.
SH	Mask secrets before any model snapshot and before writing the session recording.	Mask secrets in every capture path and before writing the on-disk rollout.	Relevant sites hide in mirrored, fallback, or cold paths, so keyword search alone misses some of them.

Table 4: Representative modification requests at each localization-difficulty level, chosen separately for each harness, with the localization challenge each level poses.

	Terminus-2	Codex	Localization challenge
Easy	Strip ANSI escape codes from all captured terminal output.	Add an informational `comment` field on both shell tools.	One well-defined site: the change maps to a single behavior that a keyword or one file read can pinpoint.
Medium	Add a per-command `cwd`, threaded through execution, the terminal layer, and both formats.	Add an `expect` output check on both shell tools, annotating the observation on failure.	Coordinated edits: one field or check must be threaded through parsing, execution, and documentation, so several related sites must be found together.
Hard	Keep run counters and step history consistent across a mid-run compaction.	Reconcile overlapping fields across `exec_command`, `shell_command`, and `write_stdin`.	Indirect coupling: the sites sit on cold or mirrored paths (post-compaction bookkeeping; parallel tool schemas) that keyword search rarely surfaces.

- **Handbook-assisted.** The planner is additionally given the handbook as a navigable skill and follows the BGPD policy, proposing source locators from the handbook and then verifying them against the repository.

The handbook is exposed as a skill through the `SKILL.md` manifest in Figure 6, which specifies when and how to consult its reference files. Both leaf modes share this structure, with stage cards representing functions in function-as-leaf mode and files in file-as-leaf mode.

```
---
name: <harness>-handbook
description: >-
  Structural map of the harness, derived from its code. Use it when planning a
  change to find EVERY code site the change must touch -- it maps the code
  stage-by-stage and lists every state register together with all of its
  read/write locations. Consult it before finalizing a plan so that scattered or
  non-obvious sites are not missed.
---

# Harness Handbook: navigation guide

This is a structural map of the harness. Use it to locate where a requested change
must take effect -- especially sites that are not adjacent to the obvious one.

## Reference files
- references/overview.md    -- whole-system orientation: the lifecycle, the main
                               loop, and how the stages fit together. Read first.
- references/index.md       -- the map: every stage (id, title, what it does, the
                               leaves it covers) and every state register.
- references/registers.md   -- for each state register: its purpose and EVERY place
                               it is written and read, across the whole harness.
- references/stages/<id>.md -- one stage's prose plus a card per leaf (function or
                               file): its <summary> line gives the exact code
                               location (path:start-end), and the card holds the
                               interface, behavior, relations, and source.

## How to use it (during planning, before you write your plan)
1. Read references/overview.md first to understand the whole system.
2. Read references/index.md to identify the stages, leaves, AND state registers
   your change involves. Do NOT prematurely narrow to the one obvious stage: a
   single change often touches sites in several stages.
3. For EVERY state register your change touches, read references/registers.md and
   note EVERY write and read site. This read/write registry surfaces scattered,
   non-adjacent sites a top-down code read would miss.
4. Open the relevant references/stages/<id>.md for detail and coupled assumptions.
5. For each site the handbook names, open the real code with read_file /
   search_file_content and locate the precise lines.
6. Your plan must account for every site the handbook surfaced.

The handbook tells you WHERE things live and how they connect. You still decide
what the change should be, and you verify every location against the real code
before planning it.
```

Figure 6: The `SKILL.md` manifest that exposes a handbook to the planner in the Handbook-Assisted arm. The same structure is used in both leaf modes; a leaf is a function in function-as-leaf mode and a file in file-as-leaf mode.

D Prompt Templates

This appendix reproduces the implementation prompts for handbook construction (Appendix D.1), BGPD localization and planning (Appendix D.2), and LLM-based plan evaluation (Appendix D.3). Runtime substitutions appear as `{{...}}` or `{...}` placeholders, with a few non-ASCII characters rendered in ASCII. Qualified names, signatures, source ranges, and call edges are supplied by static analysis and cannot be overwritten by the model.

D.1 Handbook Generation

These prompts organize source units into execution stages and synthesize the L1–L3 document tree through the pipeline in Appendix A, preserving statically extracted identities while adding behavioral descriptions.

D.1.1 Classifying Functions into Stages

In function-as-leaf Phase II, the following rules assign a function to one or more stages, either as a whole or through contiguous regions. At runtime, they are followed by the available stages, function metadata, caller/callee context with current assignments, and line-numbered source.

```
You are analyzing one Python function from the Terminus 2 agent harness, and
classifying it into one or more stages of a hand-authored data-flow skeleton.

GRANULARITY
- "function": the entire function is a single narrative unit. Use for short,
  cohesive functions (<=30 lines or with a single clear purpose).
- "region": the function contains multiple distinct narrative steps and should be
  split into 2-10 regions. Use for large functions with multiple decision points
  or sequential phases.

REGION RULES (when granularity == "region")
- Each region MUST be contiguous and end at a complete statement boundary (not
  mid-statement).
- Provide first_line and last_line as the EXACT text content of the first and last
  lines of the region.
- Each region has its own stage_id.

STAGE ASSIGNMENT
- Pick stage IDs ONLY from the list provided.
- CROSS-CUTTING utility functions (small utilities called from many places -- token
  counting, output length capping, recording markers) go to crosscut-X* ONLY. Do
  NOT add them to consuming stages.
- A function that uses self.logger.X is NOT cross-cutting just because of that --
  assign by its primary identity.
- For api-surface methods (e.g. name(), version()): set function_assignments=[] and
  granularity="function" (will be recorded as unmapped api_surface).
- For subsystem-internal functions (in tmux_session.py / terminus_*_parser.py /
  asciinema_handler.py): assign to the main-flow stage that drives them.

PURPOSE FIELD (60-150 words, structured across 5 aspects)
1. ACTION: what the function does, concretely (no "handles" / "manages")
2. INPUTS / STATE READ: arguments + self._* attrs that determine behavior
3. OUTPUTS / STATE WRITTEN: return value + self._* attrs mutated
4. WHEN INVOKED: who calls it, under what condition
5. NON-OBVIOUS: retry logic, fallback paths, design choices a reader would miss

CONSISTENCY WITH CONTEXT
Look at the caller/callee context provided -- your classification should be
consistent with their stage assignments. A function that's only called from
stage-4.2's region typically belongs near stage-4.2.

OUTPUT
Return ONLY a JSON object inside a ```json fenced block:

{
  "qualname": "<exact qualname>",
  "purpose": "<60-150 word 5-aspect description>",
  "granularity": "function" | "region",
  "function_assignments": ["stage-X", ...],
  "regions": [
    {"line_range":[a,b], "first_line":"...", "last_line":"...",
     "purpose":"<30-80 word>", "stage_id":"..."}, ...
  ] | null,
  "file": "<file>",
  "line_range": [<start>, <end>]
}

When granularity == "function", regions = null.
```

D.1.2 Reviewing Function Assignments

A critic reviews each proposed assignment against source, stage definitions, and caller/callee evidence, with the "engineer" role followed by task context, supporting evidence, the proposal, and output rules. The prompt favors approval of reasonable assignments while reserving revision for specific flaws that materially affect correctness.

```
You are a SENIOR ENGINEER reviewing a proposed change to how a function is
classified or organized in a handbook for an AI agent harness. Your job is to be
skeptical and find real concerns rooted in code behavior.

Focus on:
  - Does the proposed classification reflect what the function ACTUALLY does at the
    code level?
  - Are the line ranges / region boundaries syntactically and semantically sound?
  - Does the proposal misuse cross-cutting categories (e.g. labeling a function
    "crosscut-X3 logging" just because it calls self.logger)?
```

```
  - Are caller/callee relationships consistent with the proposed stage membership?

You may APPROVE, REVISE (with concrete concerns), or REJECT.

DECISION GUIDELINES
- APPROVE: the proposal is reasonable, even if minor wording could be tweaked. A
  correct-enough proposal is APPROVE, not REVISE. Be GENEROUS with APPROVE.
- REVISE: only when there is a SPECIFIC, ACTIONABLE flaw that materially affects
  correctness (wrong stage assignment, wrong region boundary, factual error about
  the code). Do NOT REVISE for style, phrasing, or "could be more thorough" reasons.
- REJECT: only when the proposal is fundamentally wrong and revision can't save it.

By default lean APPROVE. The pipeline must make progress.

OUTPUT FORMAT
Return ONLY a single JSON object wrapped in a ```json fenced block:

{
  "decision": "APPROVE" | "REVISE" | "REJECT",
  "concerns": ["<concern 1>", ...],
  "suggested_revision": { ... } OR null,
  "rationale": "<one sentence explanation>"
}

When decision == "APPROVE", concerns and suggested_revision may be empty.
When decision == "REVISE", concerns must be non-empty AND each concern must point
to a concrete, fixable flaw.
When decision == "REJECT", concerns must be non-empty; suggested_revision is null.
```

D.1.3 GENERATING FILE CARDS

In file-as-leaf Phase II, this prompt summarizes each scanned file. Under deep mode, the model reads the whole file to write its L3 card while treating the graph-derived function inventory, line numbers, and call relations as fixed facts:

```
You are reading SOURCE FILES IN FULL and writing a plain-language, easy-to-follow
description of each, for a system handbook in which the FILE is the smallest unit
(its leaf node). The description you write IS the handbook's content for this file.

WHO YOU ARE WRITING FOR: a curious OUTSIDER -- someone intelligent but new to this
project and possibly not an expert programmer in this area. They should be able to
read your text and come away genuinely understanding what this file does and why it
matters, WITHOUT having to already know the codebase or the jargon.

HOW TO WRITE (this matters as much as the content):
- Use plain, everyday language. Prefer short, clear sentences over dense ones.
- Explain the WHY and the WHAT in human terms before any mechanism: what real
  problem does this file solve, and what would break without it?
- When you must use a technical term or acronym, explain it in plain words the
  first time. Never leave jargon unexplained.
- Use a brief everyday analogy when it genuinely makes something click.
- Do NOT dump implementation trivia. Favor the big picture and intuition. It is
  fine to name key types/functions, but always say what they're FOR in plain words.
- Stay accurate. Simplify, but never say something that is actually wrong.
- Avoid empty filler like "handles"/"manages related logic"; be specific.

Each file also comes with its FUNCTION LIST (qualname + line range), derived from
the call graph. The function inventory, line numbers, and call relations are
FACTS -- do NOT re-list them. Your job is the plain-language prose + an
easy-to-understand note per function (referenced by its exact qualname).

For EACH file return:
- "purpose": 1-2 plain sentences a newcomer can understand: what this file is for.
- "description": an accessible walkthrough (roughly 120-300 words): what problem
  this file solves and why it exists; what it does, step by step, in plain
  language; how its main pieces work together; and any surprising behavior.
- "functions": one entry per function in the file's function list, referenced by
  its exact "qualname", each with:
    - "purpose": in plain words, what this function does and why (1-3 sentences).
    - "data_flow": what goes IN -> what it does -> what comes OUT, told as a simple
      before/after story a non-expert can follow.
    - "relations": how it fits into the bigger flow -- who calls it and when, and
      what it hands off to, grounded in the provided calls/called-by facts.
- "role": EXACTLY one of: entrypoint, orchestration, domain_logic, io_transport,
  data_model, config, util, test, generated, other.
- "lifecycle": SHORT hint of when in the run this file is active (e.g. "startup",
  "config load", "main loop", "request handling", "teardown", "cross-cutting").
```

```
OUTPUT -- ONLY a JSON object in a ```json block:
{
  "purposes": [
    {"file": "<exact path>", "purpose": "...", "description": "...",
      "role": "<role>", "lifecycle": "...",
      "functions": [
        {"qualname": "<fn qualname>", "purpose": "...", "data_flow": "...",
          "relations": "..."}, ...]},
    ...
  ]
}
Return one entry per file given, using the exact file paths provided.
```

D.1.4 SYNTHESIZING THE L1–L3 DOCUMENT TREE

In function-as-leaf mode, each L1, L2, and L3 node is produced through a bounded actor–critic–reflexion loop: an actor drafts from grounded inputs, a critic applies a tier-specific rubric, and revisions continue until the rubric passes, the score stops improving, or the budget is exhausted. L1 uses a guided system overview, while L2 explains the problem, main flow, state flow, and hand-offs. In file-as-leaf mode, validated file cards from Appendix D.1 supply L3 content, with a single-pass rollup producing L2 component overviews and the L1 system overview.

The L3 (function-as-leaf) entry is emitted against the output schema below, grounded in the exact source range(s); a multi-region variant returns one gloss per region with function-level design decisions and relations.

```
## Output format (single)

Return a JSON object inside a ```json fence:

```json
{
  "schema_version": 1,
  "type": "single",
  "locator_role": "<one-line role tag, 6-15 words>",
  "stage_context": "<2-4 sentences: what role this plays in the stage; when it is
                    invoked; how it relates to its siblings>",
  "synopsis": "<2-4 sentences: What + inputs + outputs + side effects>",
  "interface": {
    "signature": "<full signature, e.g. (self, environment) -> None>",
    "params": [{"name": "<param>", "type": "<type, or ? if unclear>",
                "role": "<one-line role / source>"}],
    "reads_state": ["<self._ attributes this function reads>"],
    "returns": "<what it returns; if None, state the real product (side effect)>",
    "side_effects": ["<self._ attributes written / external effects>"]
  },
  "execution_flow": [
    "<step 1: action + which args/state + what's produced>",
    "<step 2: ...>"
  ],
  "design_decisions": [
    "<decision 1: the trade-off point + why this choice + what an alternative costs>",
    "<decision 2: ...>"
  ],
  "relations": {
    "callers": ["<qualname or brief description>"],
    "core_callees": ["..."],
    "config_state_sources": ["..."],
    "results_to": ["..."],
    "siblings": [],
    "register_interactions": [
      {"action": "write|read|clear|reset", "register": "reg-<id>",
       "note": "<5-12 words: when / why>"}
    ]
  }
}
```

Count constraints:
  - interface.signature: required, copied accurately from the source.
  - interface params / reads_state / returns / side_effects: list them all from the
    source; empty array / "none" if there are none.
  - execution_flow: 2-8 steps; design_decisions: 1-5 items.
  - relations.{callers, core_callees, config_state_sources, results_to}: at least 4
```

```
    of these must be non-empty.
  - relations.register_interactions: list every register this function
    reads/writes/resets; empty array [] only if it interacts with none.
```

D.2 BGPD Localization and Planning

The Handbook-Assisted planner uses the following BGPD prompt to navigate the handbook, verify candidate edit sites against current source, and produce a self-contained plan. Harness-specific runtime fields appear as {{...}} placeholders.

```
You are a senior software engineer PLANNING a change to {{PROJECT_INTRO}}, on
behalf of a code reviewer.

You are given ONE natural-language change request. In this phase you produce a
precise, SELF-CONTAINED PLAN of the edits -- you do NOT edit any files. A separate
executor will apply your plan MECHANICALLY: for each edit it will substitute your
exact OLD text with your exact NEW text, WITHOUT re-reading the file. So your
plan's verbatim text must be byte-exact.

## Two artifacts, two distinct roles
- The handbook is a pure LOCATION INDEX of the harness, NOT a description of the
  code. Each function appears as a one-line locator: `<summary><b>Qualified.name</b>
  -- file:start-end . one-line role</summary>`, optionally followed by a Relations
  block (callers/callees/register read-write sites). A card has NO function body and
  NO source code. On top of that, `index.md` lists every stage with its function
  locators and `registers.md` lists every state variable with its exact read/write
  code sites. Use these to decide WHICH files, functions and sites are in scope --
  they surface scattered, non-obvious sites (mirror copies in the other
  parser/template, a register's every read/write, cross-subsystem touch points)
  that a plain text search can miss.
- The real source code is the GROUND TRUTH for WHAT to change. The handbook tells
  you the ADDRESS (file:start-end); the code at that address is the only reliable
  source of the actual structure. You MUST read the real source.

## How to plan -- ROUTE with the handbook, READ the real source, EMIT verbatim edits
1. Understand the request's true intent: the behavior delta, and the
   state/conditions/values it fixes.
2. Route with the handbook. Read its `SKILL.md`, then `index.md`, then only the
   `stages/<id>.md` chapters and `registers.md` entries your intent points to.
   Assemble the candidate set: every file + function + anchor the change must touch.
   Watch for scattered/mirror sites (a parser change usually has a twin in the OTHER
   parser and in both prompt templates; a state change fans out to every read site
   listed under that register).
3. Read the REAL source of every site you intend to edit with `read_file` on the
   actual code files. Confirm the exact body, control flow, conditions, and that the
   site does what the card implied.
4. For EACH edit, produce a self-contained EDIT BLOCK (format below) whose
   `old_string` is copy-pasted verbatim from the `read_file` output you just saw --
   never retyped from memory, never paraphrased. Match whitespace and indentation
   exactly, and include at least 3 lines of context BEFORE and AFTER the changed
   lines so the snippet is UNIQUE in the file.
5. A change can also silently break something the request never mentions; if you
   find such a coupled assumption, add an edit (or note it) accordingly.
6. Only include edits you are confident the request requires.

## EDIT BLOCK format (the executor applies these directly)
For every edit, output exactly:

### EDIT <n>
- file: `<path relative to the working dir, e.g. {{PATH_EXAMPLE}}>`
- where: `<{{WHERE_EXAMPLE}}>` -- why this change
```old
<EXACT current text, copied verbatim from read_file -- whitespace-perfect, unique>
```
```new
<the replacement text -- correct, idiomatic, the smallest change that realizes the intent>
```

Rules for the blocks (the executor trusts them blindly, so precision is on you):
- `old` MUST be byte-exact to the file's current content. If you are not certain it
  is verbatim, `read_file` that region again before writing the block.
- Keep each `old` the SMALLEST span that is still unique (1-8 lines typically); do
  not paste whole functions.
- SAME-FILE edits must NOT overlap. The executor applies your blocks in order
  against text that earlier blocks have ALREADY changed, and it does not re-read
  between them. So no block's `old` (including its context lines) may contain a line
```

```
  that another block in the same file changes -- otherwise the second match will be
  stale. If two changes are close enough that their context would overlap, MERGE
  them into ONE block covering both. Order same-file blocks top-to-bottom.
- For a brand-NEW file, use a single block with an empty ```old``` and the full
  content in ```new```, and say "(new file)" in `where`.
- Anchor on stable lines; never let `old` span a region you are unsure of.

When done, call `complete_task` with: a short prose summary of the edits, then ALL
the EDIT blocks, then the declarations JSON. Do NOT edit any files in this phase.

## Declarations (machine-readable -- the handbook-resync pipeline consumes this)
End with EXACTLY one ```json block declaring the change-set at FUNCTION
granularity, using {{QUALNAME_NOTE}}:

```json
{{DECL_JSON}}
```

- `will_modify` -- every EXISTING function whose implementation your edits change.
- `will_add`   -- every brand-new function introduced.
- `will_remove` -- every function deleted outright. A rename = remove(old)+add(new).
```

D.3 LLM JUDGING

The judge first builds an answer key from the request and pristine source without seeing any plan, then independently scores each plan against that key.

D.3.1 BUILDING THE ANSWER KEY

The answer-key prompt receives the request with hint fields removed and the pristine source before any plan is shown. The version below is for Terminus-2; Codex uses the same structure with `correct_approach` replacing `correct_localization`.

```
[SYSTEM]
You are an expert Python engineer building a grading ANSWER KEY for a code-change
task on the Terminus-2 Python agent harness.

You are NOT looking at any solution yet. Read the pristine source and independently
determine which concrete files, functions, methods, branches, templates, or state
variables a correct solution must localize before editing.

Output ONLY this JSON object:
{"required_sites":["<concrete file/function/branch/template/state site>"],
"correct_localization":"<2-4 sentences: where the planner should look and why>",
"pitfalls":["<easily missed mirror/fallback/cold/state/template sites>"]}

[USER]
## CHANGE REQUEST
{query}

## PRISTINE TERMINUS-2 SOURCE
{pristine}

Build the localization answer key now. Output the JSON object only.
```

D.3.2 SCORING THE PLAN

Each plan is scored independently against the answer key on a 0.0–5.0 scale in 0.5 steps. The three dimensions are combined with weights 0.50 (`localization`), 0.25 (`scope_bloat`), and 0.25 (`reasoning`); `scope_bloat` is reported as Scope Control, where a higher value means a more focused plan.

```
[SYSTEM]
You are an expert Python engineer GRADING the LOCALIZATION quality of one planner's
PLAN for a code-change task on the Terminus-2 Python agent harness.

You judge the PLAN ONLY: which files/functions/state/template sites it decides to
inspect or modify, and whether its reasoning is grounded. Do NOT grade the final
code diff or execution correctness. The executor is shared across arms and is out
of scope.
```

```
Grade these dimensions from 0.0 to 5.0 in 0.5 steps:
- localization : did the plan identify ALL required implementation sites in the
                 answer key, including mirror, fallback, cold-path, state-reset,
                 and template sites?
- scope_bloat  : did the plan avoid unrelated files/functions or overly broad
                 edits? 5 means tightly scoped; low means it wandered into
                 irrelevant areas.
- reasoning    : is the plan's localization reasoning grounded in the real code and
                 request, rather than vague, circular, or guessed?

Output ONLY this JSON object:
{"localization":X,"scope_bloat":X,"reasoning":X,
"justification":"<<=60 words: decisive localization hit/miss -- which sites found or missed>"}

[USER]
## CHANGE REQUEST
{query}

## ANSWER KEY
{answer_key}

## PRISTINE TERMINUS-2 SOURCE
{pristine}

## THIS ARM -- plan.md
{plan}

Grade this arm's PLAN localization against the answer key. Output the JSON object only.
```

E ILLUSTRATIVE HANDBOOK-ASSISTED PLANNING WALKTHROUGH

This appendix reconstructs a recorded Handbook-Assisted run on Terminus-2 (case **Q1**, DeepSeek-V4-Pro planner) to illustrate BGPD's two stages: *routing* through the handbook to identify candidate sites and *verifying* them against current source before planning. All tool calls, source locators, and edits shown below come from the run, with only whitespace trimmed.

Change request (Q1: triple completion confirmation)

Right now the model only has to mark `task_complete` once to reach the "are you sure?" step, then once more to get graded. I want a higher bar: the model must mark completion **three consecutive times** before we actually grade. The first two times just re-show the terminal state and ask "are you sure you want to submit?".

This Query-type request revises an existing control-flow rule without naming a file or function, requiring the planner to locate every site involved in the completion handshake.

Phase 1 — Route through the handbook

Guided by the `SKILL.md` policy, the planner reads the handbook top-down, narrowing the scope before opening any source:

1. `SKILL.md` → `overview.md` (L1) → `index.md`: locate the stage that owns per-iteration continuation and termination.
2. `stages/stage-4.8.md` (*Completion Gate*) and `stages/stage-4.7.md` (*Command Execute*): the handshake lives in the gate.
3. `registers.md`: the gate is backed by the state register `reg-pending-completion`, whose read and write sites span the loop, initialization, and per-run reset.

The behavioral change centers on the Completion Gate and one state register, whose write sites reveal that initialization and per-run reset must be updated alongside the loop's return logic.

Phase 2 — Verify against the live source

In this run, the planner-facing handbook serves as a location index without source bodies, so the planner opens `terminus_2.py` to verify its locators. A `search_file_content` call for `_pending_completion` returns **seven occurrences, all in `terminus_2.py`**, keeping the iden-

tified change within one file. Reading them shows a single Boolean flag that is set on the first completion mark, tested before returning on the second, and cleared on any non-completion turn and at the start of each run. This resolves the handshake to three concrete sites:

- `Terminus2.__init__` (∼L292) — initializes the flag.
- `Terminus2._reset_per_run_state` (∼L1574) — clears it per run.
- `Terminus2._run_agent_loop` (∼L1427–1440, L1552–1559) — the observation branch and the completion gate that read and write it.

Edit plan. With the sites confirmed, the plan replaces the Boolean `_pending_completion` with an integer counter `_completion_confirmations`: the loop increments it on each completion mark, resets it on a non-completion turn, and returns for grading only on the third consecutive mark. The change set is four *modify* actions, all in `terminus_2.py` — initialization, per-run reset, and the loop's two branches. The completion gate itself is the representative edit:

Before — two-mark Boolean handshake:

```
if is_task_complete:
    if was_pending_completion:
        # second task_complete: confirmed, grade now
        return
    else:
        # first attempt: ask to confirm, continue
        prompt = observation
        continue
```

After — three-mark counter:

```
if is_task_complete:
    if was_completion_confirmations >= 2:
        # third consecutive task_complete: confirmed, grade now
        return
    else:
        # first or second attempt: ask to confirm, continue
        prompt = observation
        continue
```