VISTA: LOOKING RIGHT IS NOT WORKING RIGHT BENCHMARKING CODING AGENTS FROM DESIGN HANDOFFS TO FULL-STACK WEB AND MOBILE APPS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents can now build applications from design mockups, but a screen that looks right is not an application that works. Existing benchmarks either score visual reconstruction without a backend or test full-stack functionality from textual requirements, and few tie scores to the individual components a design requires. We introduce VISTA (VIsual Spec-To-App), an end-to-end benchmark in which coding agents turn multi-page design handoffs (Figma renders, structure, and textual requirements) into runnable full-stack Web and Mobile (Android) applications. VISTA provides 8,487 human-annotated interactive components across 18 applications (126 Web pages and 576 Android screens) and an executable evaluator that locates annotated components in the running application and probes their interactions, yielding localization, behavior, and joint scores traceable to individual design requirements. Across 14 deployed coding-agent systems, the best joint score is 0.553 on Web and 0.378 on Mobile, and 13 of 14 systems score lower on behavior than on localization on Web. Component probes detect unresponsive controls in 90.1% of audited Web deliveries. Development traces show where the process departs from the task: 44.5% of trajectories never open the design screenshots, 73.0% omit the prescribed self-audit, and 8.3% report success after a failed check. VISTA makes the gap between rendering a design and delivering a working application measurable at the level of individual components.

1 INTRODUCTION

Building an application from a design requires translating page layouts and components into connected interfaces, backend logic, and persistent data. We study this workflow as *full-stack design-to-app*: constructing and launching a web or mobile application from Figma designs and accompanying product requirements. A task-creation form, for example, must preserve the specified interface while connecting user input to a stored record. These requirements fail independently: a screen can match the design yet lack working submission logic, and a functional implementation can depart from the design. Evaluation therefore requires checking, at each required component, both where it appears and whether it works.

Existing benchmarks cover parts of this workflow. Visual code-generation benchmarks score reconstruction and front-end interaction from screenshots or prototypes, without requiring a backend or database (Si et al., 2025; Xiao et al., 2026b;a). Full-stack benchmarks test frontend, backend, and database functionality, but start from textual requirements and do not assess fidelity to a supplied design (Lu et al., 2025; 2026). Benchmarks that combine visual input with runtime checks evaluate interactions inferred from the interface on Web alone (Chen et al., 2026; Zhao et al., 2026). What is missing is an explicit reference that connects each component of a design to an executable check on the delivered application, for both Web and mobile platforms.

We introduce **VISTA**, a benchmark for Figma-to-full-stack application development on Web and Mobile (Figure 1). It contains 18 applications spanning 126 Web pages and 576 Android screens, with 8,487 manually annotated interactive components. Each annotation specifies where a control belongs, which interaction it must support, and, where applicable, its navigation target. Agents build and launch multi-page applications with frontend, backend, and database components in a

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

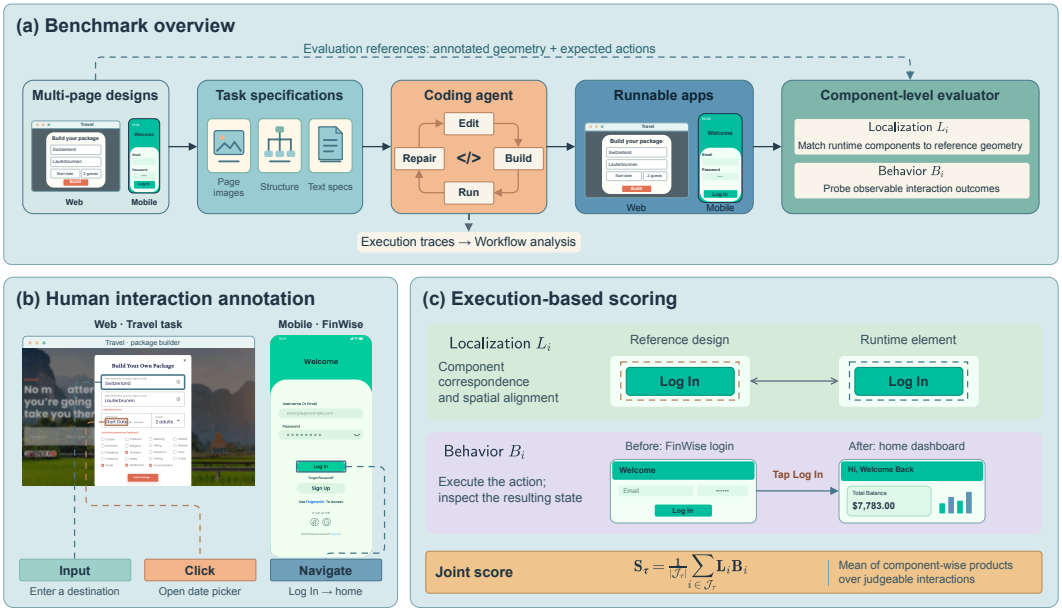

Figure 1: Overview of the VISTA benchmark pipeline, from design collection and annotation to agent execution, application evaluation, and workflow analysis.

self-contained environment. The evaluator matches annotated controls to runtime elements and executes a probe selected by each control's interaction type, producing localization and behavior scores that trace every unit of credit to a design requirement and its execution evidence. Development trajectories are retained alongside the scores.

Evaluating 14 deployed coding-agent systems shows that current agents render designs more reliably than they implement them (Table 2). On Web, 13 of 14 systems score lower on behavior than on localization (mean $B = 0.543$ versus $L = 0.642$). The best joint scores are 0.553 on Web (Grok 4.6) and 0.378 on Mobile (GPT-6-astra), leaving substantial headroom on both tracks, and platform rankings diverge: GPT-6-astra ranks seventh on Web but first on Mobile. Component probes detect unresponsive controls in 90.1% of audited Web deliveries, and Mobile failures extend to database initialization and launch (Section 5.1). Development traces expose process-level gaps (Section 5.2): 44.5% of trajectories never open the design screenshots, 73.0% omit the prescribed self-audit, and 8.3% report success after a failed check.

Our contributions are:

- **A Figma-to-full-stack benchmark for Web and Mobile.** VISTA turns design handoffs into application-construction tasks that require integrated frontend, backend, and database implementations across 18 applications.
- **Component-level ground truth with executable evaluation.** 8,487 human-annotated components define reference locations, interaction types, and navigation targets; the evaluator checks each against the running application, yielding localization and behavior scores traceable to individual requirements and validated against blind human judgments.
- **Evidence that agents render designs better than they implement them.** Behavior trails localization for 13 of 14 systems, rankings diverge across platforms, and component probes expose unresponsive controls and deployment failures.
- **A trajectory-level diagnosis.** Agents frequently bypass the design evidence and self-audit they are given, and report success beyond what their own checks show; richer design inputs increase agent activity without measurable score gains.

Table 1: Comparison of application-generation benchmarks by target platform, supplied inputs, and interaction evaluation. V: supplied visual references; S: supplied structured design/resource information; I: runtime interaction evaluation. $\surd$ indicates inclusion; – indicates outside the described protocol.

Benchmark	Web	Mobile	V	S	I
Design2Code (Si et al., 2025)	$\surd$	–	$\surd$	–	–
DesignBench (Xiao et al., 2026b)	$\surd$	–	$\surd$	–	–
Interaction2Code (Xiao et al., 2026a)	$\surd$	–	$\surd$	–	$\surd$
MRWeb (Wan et al., 2024)	$\surd$	–	$\surd$	$\surd$	–
WebGen-Bench (Lu et al., 2025)	$\surd$	–	–	–	$\surd$
MiniAppBench (Zhang et al., 2026)	$\surd$	–	–	–	$\surd$
FullStack-Bench (Lu et al., 2026)	$\surd$	–	–	–	$\surd$
UI2App (Chen et al., 2026)	$\surd$	–	$\surd$	–	$\surd$
LongWebBench (Zhao et al., 2026)	$\surd$	–	$\surd$	–	$\surd$
VISTA	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Web and Mobile denote application runtimes; VISTA's Mobile track produces installable Android apps. S denotes design/resource information supplied during generation, excluding evaluator-only DOM access and source code supplied for repair. Interaction2Code uses v3; DesignBench uses v1.

2 Related Work

2.1 Benchmarks for web and application development

Visual code-generation benchmarks assess reconstruction from interface references. Design2Code evaluates screenshot reconstruction, while DesignBench extends evaluation to generation, editing, and repair across front-end frameworks (Si et al., 2025; Xiao et al., 2026b). Interaction2Code adds before-and-after interaction prototypes; MRWeb supplies structured resource lists for links, images, and backend routes (Xiao et al., 2026a; Wan et al., 2024). UI2App and LongWebBench further evaluate inferred interactions and cross-route state, and long-page structure with executable user goals, respectively (Chen et al., 2026; Zhao et al., 2026).

Benchmarks starting from textual requirements emphasize application functionality. WebGen-Bench combines browser-executed tests with appearance assessment, and MiniAppBench evaluates intent, presentation, and dynamic behavior (Lu et al., 2025; Zhang et al., 2026). FullStack-Bench tests frontend, backend, and database functionality, including database interactions accompanying user actions (Lu et al., 2026). Table 1 compares the platform coverage, inputs, and evaluation settings. None of these benchmarks combines design-grounded, component-level ground truth with full-stack execution, and all target Web only. VISTA covers both browser-based Web applications and installable Android apps, with a shared component-level score formulation and platform-specific execution checks. Its human-annotated component requirements from Figma design handoffs specify what to inspect and which interactions to execute, connecting each score to explicit ground truth and runtime evidence.

2.2 Coding agents and process evaluation

Coding-agent research examines both task completion and the interfaces through which agents act. SWE-bench evaluates repository issue resolution, SWE-agent studies agent–computer interfaces, and OpenHands integrates code execution, shell access, and browsing (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025). Terminal-Bench evaluates challenging terminal tasks in task-specific environments with executable verification; its trajectory analysis distinguishes execution, coherence, and verification failures (Merrill et al., 2026). VISTA extends trajectory analysis to application construction, where process signals such as design inspection, verification, and completion reports are recorded alongside component-level outcome scores.

WebArena, VisualWebArena, and OSWorld evaluate agents completing tasks in existing websites and desktop environments (Zhou et al., 2024; Koh et al., 2024; Xie et al., 2024); VISTA instead requires agents to construct and launch the application under test.

3 BENCHMARK CONSTRUCTION

3.1 TASK FORMULATION

VISTA evaluates application development from multi-page design handoffs. Each task requires an agent to translate page layouts, interface components, and interaction requirements into a runnable application. In the full-stack setting, the interface must integrate with a backend and database to display data and respond to user actions.

We represent a task as $\tau = (X, E, R)$, where X is the agent-visible specification, E is the development and execution environment, and R specifies the deliverables. For an application with P pages, the core input is $X_{\text{core}} = (\{I_p, J_p\}_{p=1}^{P}, D)$: I_p is a rendered Figma page, J_p describes its component hierarchy and geometry, and D provides textual requirements. The Web track defines six conditions that cross three input tiers—text requirements only, adding rendered screenshots, and further adding structure JSON—with a free stack or a task-specific prescribed stack. Under a free stack these tiers are C_0, C_2, and C_4; the corresponding fixed-stack conditions are C_5, C_1, and C_3. Main results on both tracks use the full input X_{core} with a free stack, which on Web is C_4.

The agent acts through the tools of its coding harness (file editing, command execution, dependency installation, building, and debugging) in unattended runs without per-call human approval. Web deliverables include application code and a Docker Compose configuration; Android deliverables include an installable APK, a build script, and a containerized backend and database. Web evaluation runs in a browser; Android evaluation uses an emulator accessed through ADB and Appium.

3.2 COLLECTION AND ANNOTATION

Design selection and export. We select multi-page Figma designs that describe complete applications with connected interfaces, so that tasks require implementing application workflows. For each design, we export rendered pages and node structures: images provide visual references, structure files provide hierarchy and geometry, and descriptions specify component purposes and interactions (cf. Deka et al., 2017; Wang et al., 2021). Android tasks additionally provide navigation relations and component-level assets.

Human interaction annotation. Figma layer names do not reliably specify behavior: a button may carry a generic name, and repeated instances may lack explicit action semantics. We therefore manually annotate interactable components in an annotation interface, associating their positions with expected behavior (Figure 1). Records contain bounding boxes, interaction types, and, where applicable, navigation targets, behavior subtypes, and semantic identifiers. They cover navigation, input, toggles, and different click actions. The annotations establish component-level interaction ground truth: reference geometry identifies where a control belongs, interaction types determine the behavior to probe, and annotated targets specify navigation destinations where applicable. Together with task descriptions, they turn static designs into implementable and testable requirements (Xie and Memon, 2007).

The Web data contain 3,253 interaction annotations and a separate set of 458 curated visual anchors for identifying corresponding elements and assisting coordinate alignment. The primary score covers the *critical* subset of annotations: all navigation, input, and toggle components, plus click components whose annotated action is defined (excluding decorative or unresolved clicks); remaining annotations contribute no score.

3.3 SCALE AND COVERAGE

VISTA contains ten Web applications and eight Android applications, which form the Web and Mobile tracks (Figure 2). Web includes all 126 annotated pages; Android includes 576 of 647 exported screens after excluding near-duplicate or out-of-scope screens. These pages carry 3,253 annotated interactive components on Web and 5,234 on Android, 8,487 in total; visual anchors are counted separately.

Web tasks cover publishing, real estate, recruitment, forums, travel booking, chat, cloud storage, e-commerce, project management, and music streaming; Android tasks cover medical appointments,

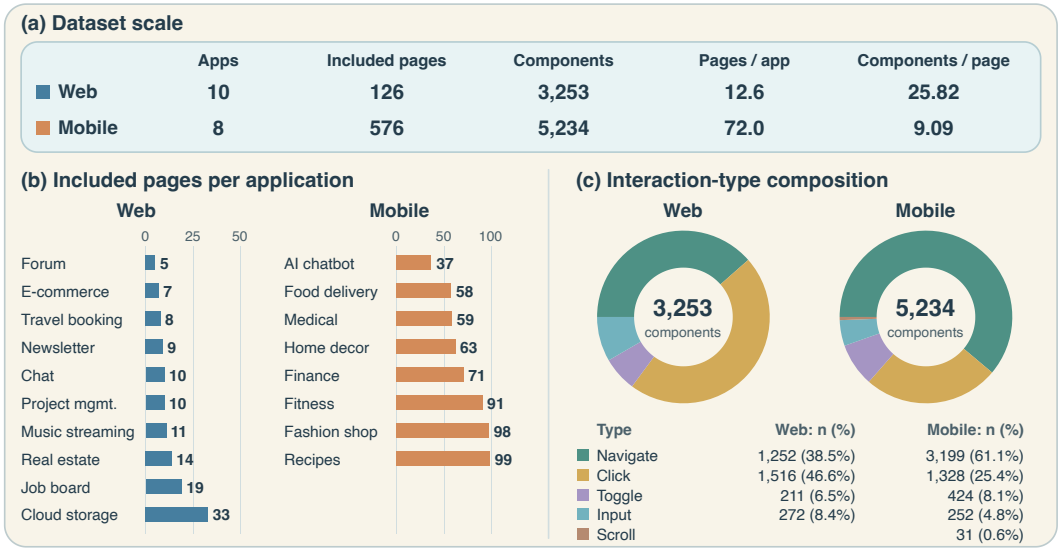

Figure 2: Dataset statistics. (a) Scale over included pages. (b) Per-application page counts, with separate axes for Web (0–50) and Android/Mobile (0–100). Blue denotes Web and peach denotes Mobile in (a–b). (c) Interaction-type composition with consistent category colors. Ring centers show total components; the legend lists counts and percentages of all annotations. Web's single-record Scroll and Other categories are omitted from the rings and legend, and displayed sectors are normalized to fill each ring.

fitness, home-decor shopping, food delivery, recipes, personal finance, AI chat, and e-commerce. The two platforms pose different workloads: Android applications average 72.0 screens versus 12.6 Web pages ($5.7\times$), while Web pages carry more annotated components (25.8 versus 9.1), and navigation dominates Android annotations (61.1%) whereas click actions dominate Web (46.6%).

4 Evaluation

VISTA grounds execution-based evaluation in human-annotated component requirements. For each selected interaction, the evaluator identifies a corresponding runtime element, executes the relevant behavior probe, and records the matching and interaction evidence used for scoring, so that each component score links to an explicit requirement under the same rules across generated applications. Web applications are inspected through the DOM using Playwright; Android applications through the UI hierarchy using ADB and Appium, with a shared score formulation and platform-specific matching and probes.

Localization and behavior. Localization $L_i \in [0, 1]$ measures component correspondence and spatial alignment using geometry, component types, and available semantic identifiers. Behavior $B_i \in [0, 1]$ measures evidence from executing the annotated interaction, such as accepting input, changing a toggle state, opening a dialog, or navigating. The Web evaluator includes semantic-anchor matching and content-change navigation checks; Android uses one-to-one component assignment and target-screen fingerprints. Partial-credit rules distinguish levels of probe evidence. Together, these metrics assess component realization and observable interaction outcomes (Barr et al., 2015; Xie and Memon, 2007).

Joint score. For a run on task τ, let C_τ contain its critical interaction records and $J_\tau \subseteq C_\tau$ contain those with a non-missing behavior verdict. The joint score is

$$S_\tau = \frac{1}{|J_\tau|} \sum_{i \in J_\tau} L_i B_i. \tag{1}$$

Table 2: Web and Mobile evaluation results. Both tracks use the full input with a free implementation stack (C_4). Localization L, behavior B, and joint score S are reported separately; higher is better. Token usage T is the mean non-cached (input plus output) tokens per task in thousands; Appendix B breaks usage down by token type. Each value is the mean over three runs per task. Overall is the mean of Web and Mobile S; rows are ordered by Overall. Model sources and harness versions are listed in Appendix A.

Model	Harness	Web				Mobile				Overall
		L	B	S	T	L	B	S	T	↑
GPT-6-astra	Codex	0.756	0.505	0.460	180	0.473	0.398	0.378	387	0.419
Fable-5	Claude Code	0.749	0.609	0.533	366	0.367	0.402	0.270	1020	0.402
Grok 4.6	Cursor	0.732	0.664	0.553	444	0.336	0.379	0.234	387	0.393
Opus 4.8	Claude Code	0.686	0.572	0.487	291	0.395	0.424	0.282	565	0.385
Grok 4.5	Cursor	0.670	0.590	0.491	154	0.418	0.353	0.245	411	0.368
GPT-5.6-sol	Codex	0.738	0.593	0.507	351	0.377	0.272	0.228	99	0.367
DeepSeek-V4	DeepSeek CLI	0.714	0.581	0.484	360	0.401	0.275	0.219	373	0.351
Composer 2.5	Cursor	0.599	0.569	0.448	116	0.413	0.300	0.192	122	0.320
GPT-5.5	Codex	0.656	0.478	0.410	253	0.310	0.287	0.125	199	0.268
Sonnet 4.6	Claude Code	0.586	0.517	0.412	150	0.265	0.205	0.115	301	0.264
Muse-Spark-1.2	Muse Code	0.546	0.542	0.409	180	0.287	0.294	0.110	369	0.259
GPT-5.6-terra	Codex	0.570	0.488	0.384	75	0.246	0.198	0.072	92	0.228
GPT-5.6-luna	Codex	0.581	0.482	0.387	84	0.113	0.121	0.028	101	0.207
MAI-Code-1.1	Copilot	0.409	0.418	0.293	153	0.220	0.137	0.055	234	0.174

Note. Muse-Spark-1.2 denotes Muse-Spark-1.2-contributor; DeepSeek-V4 denotes DeepSeek-V4-Flash; MAI-Code-1.1 denotes MAI-Code-1.1-Flash. Tokens exclude cache-read.

This is a mean of component-wise products: an interaction receives no joint credit when either score is zero. Mean localization over C_τ and mean behavior over J_τ are reported separately to distinguish correspondence from interaction failures. Web uses $J_\tau = C_\tau$; Android excludes records whose behavior verdict cannot be determined. A delivery that fails to launch, or a page that cannot be reached, receives $L_i = B_i = 0$ for every affected component and is averaged in with the other runs.

Human validation. Blind human studies support both scores (Appendix C). Localization correlates with three annotators' fidelity ratings of generated Web pages at Spearman $\rho = 0.66$ (95% CI $[0.36, 0.86]$; Krippendorff's $\alpha = 0.70$), whereas CLIP image similarity does not ($\rho = 0.04$). Behavior probe verdicts agree with human inspection of before-and-after screenshots in 88% of cases (Cohen's $\kappa = 0.68$).

Results. Table 2 lists Web and Mobile results by model–harness configuration, with separate columns for localization L, behavior B, joint score S, and mean token usage per task for 14 model–harness configurations. Each system runs every task three times, and all values are means over runs. Two patterns stand out. First, behavior trails localization: 13 of 14 systems score lower on B than on L on Web (mean 0.543 versus 0.642), so placing a control is consistently easier than making it work. Second, rankings diverge across platforms and metrics: GPT-6-astra ranks seventh on Web S but first on Mobile, while the Web leader, Grok 4.6, ranks fifth on Mobile; on Web, GPT-6-astra leads localization ($L = 0.756$) and Grok 4.6 leads behavior ($B = 0.664$). A single-platform or single-metric leaderboard would miss these differences. Both tracks leave substantial headroom: the best joint score is 0.553 on Web and 0.378 on Mobile.

5 Agent Workflow Analysis

Component-level scores show that delivered applications fall short of their designs; development traces show where the process departs from what the task requires.

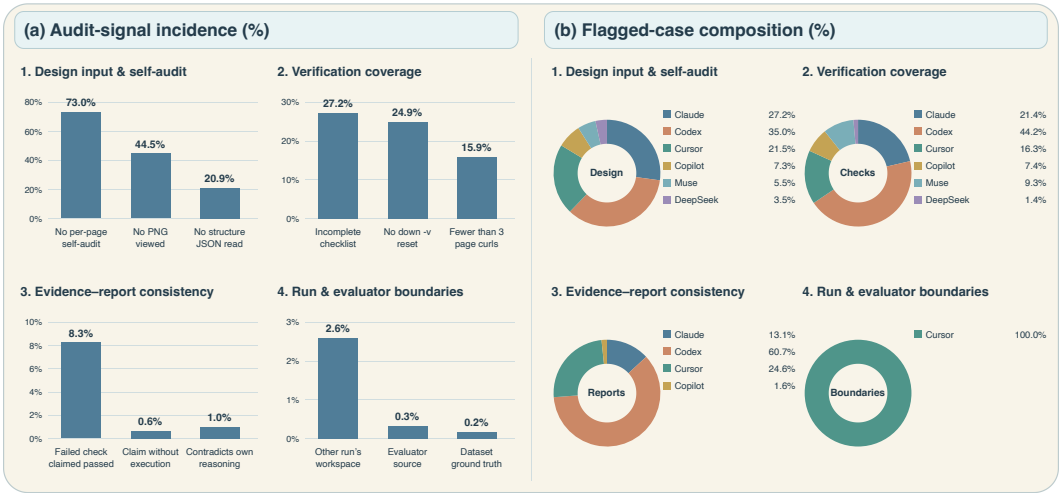

Figure 3: Alignment gaps across four directions. Left: detection rates as percentages of all audited trajectories; axis ranges vary by panel. Right: system-group shares within each direction's flagged cases, with each trajectory counted once per direction. Report–reasoning contradictions require visible reasoning and are reported using the same all-trajectory denominator.

5.1 Failure modes in delivered applications

Unresponsive controls are widespread across Web deliveries. In a Web audit of 12 systems (Grok 4.6 and GPT-6-astra were added after the audit), 90.1% of scored deliveries contain at least one control whose interaction probe detects no response, and 13.6% receive a joint score below 0.2. Rendering a control and implementing its behavior are separate achievements, and component-level execution exposes the difference.

Mobile failures extend to deployment. On Mobile, the failure spectrum begins before any interaction can be probed: 2.9% of backend-evaluation deliveries mount an empty directory as `init.sql`, preventing database initialization, and others ship incomplete schemas, invalid SQL comment syntax, mismatched `INSERT` values, missing Dockerfiles, or unlaunchable Android Activities. Such defects break the path from generated code to a running application, a stage that screenshot-based evaluation cannot observe.

5.2 Alignment gaps in design grounding, verification, and reporting

Development traces reveal systematic gaps between what the task prescribes and what agents do (Figure 3); stack compliance is measured in the fixed-stack conditions of the condition study.

Agents bypass the design evidence they are given. The system prompt instructs agents to read the screenshot, structure JSON, and description before writing each page, and to self-audit each page against them (Appendix D.1). Yet no screenshot inspection is recorded in 44.5% of trajectories, no structure JSON read in 20.9%, and the per-page self-audit is absent in 73.0%. The omission reflects behavior rather than capability: Codex supports image input, yet 71.1% of its trajectories never open a screenshot, compared with 21.8% for Claude Code and Cursor.

Stack mandates are followed; process instructions are not. In the fixed-stack conditions, each task prescribes a frontend framework such as Astro, Next.js, or SvelteKit. For GPT-5.5 and Sonnet 4.6, the detected framework matches the prescription in 88.3% of runs across C_1, C_3, and C_5; the remainder, all by GPT-5.5, are detected as React where Next.js was prescribed. Agents thus follow instructions about what to build with, yet skip the prescribed steps for checking what they build against the design.

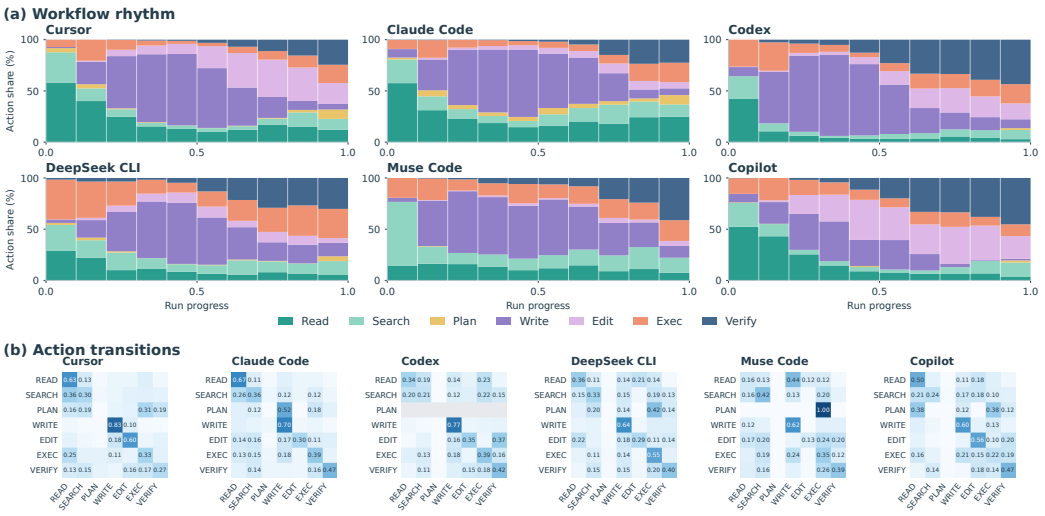

Figure 4: Workflow signatures across six harness groups covering 14 models. (a) Action shares over ten normalized progress bins, arranged in two rows; category shares are normalized within each model/bin before equal-weight averaging across models. Pale vertical lines mark 0.1 progress intervals. (b) Transition probabilities pooled across observed transitions and normalized by the source action. Both panels use the same seven action categories, including VERIFY. Cells below 0.10 are unannotated; gray rows have no observed source transitions. Complete model profiles appear in Appendix E.

Success claims outrun execution evidence. The audit flags incomplete verification in 27.2% of trajectories and a failed check followed by a claim that it passed in 8.3%. In a job-board trace, a `set -e` verification block terminates when `wget` is unavailable, yet the closing message reports successful login and page checks. Agent self-reports are therefore not a substitute for component-level execution checks.

5.3 Richer inputs increase activity without measurable score gains

For GPT-5.5 and Sonnet 4.6, the two models covered by the condition study, richer design inputs increase agent activity: under a free stack, tool use rises from 53.5 calls with text alone (C_0) to 69.3 with screenshots (C_2) and 71.5 with additional structure JSON (C_4); token workload is 134.7K, 224.1K, and 149.3K, respectively. Joint scores do not follow: across the four input and stack contrasts, estimated changes stay within ± 0.023, with all 95% confidence intervals spanning zero (Appendix F). Agents do more with richer inputs without measurably converting the additional design evidence into higher fidelity, consistent with the frequent omission of screenshot inspection in Section 5.2.

5.4 Workflow signatures across harnesses

Agents shift from inspection to construction to verification. Across the six harness groups (Figure 4), READ and SEARCH occupy 41.3–70.1% of activity in the first two progress bins, WRITE and EDIT 53.0–76.3% in bins four through seven, and VERIFY rises from 0.1–2.6% in the first two bins to 20.0–41.6% in the last two.

Harnesses differ in when they verify. EDIT-to-VERIFY transitions account for 37.2% in Codex, compared with 7.5% in Cursor and 9.5% in Claude Code; Cursor and Claude Code instead show strong READ persistence (62.7% and 67.2% READ-to-READ, versus 33.8% in Codex), and Cursor the strongest EDIT persistence (60.2%).

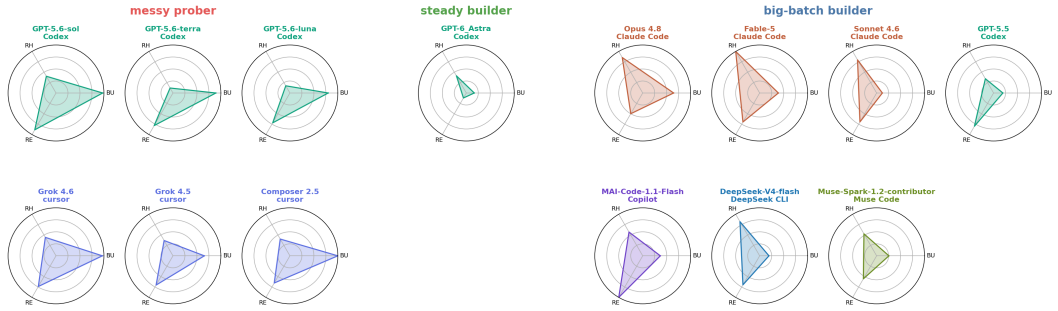

Figure 5: Behavioral profiles of 14 models in three clusters: messy prober (six models), steady builder (one), and big-batch builder (seven). BU (`build_share`) is the fraction of actions that are WRITE or EDIT. RH (`gap_cv`) is the coefficient of variation of the gaps between build bursts, where a burst is a maximal run of consecutive WRITE or EDIT actions and a gap is the number of other actions between two bursts; low values indicate regular pacing and high values indicate clustered activity. RE (`rework_rate`) is the fraction of touched files revisited more than once. Colors identify harness groups.

Workflow style varies with the model within a harness. Figure 5 characterizes 14 models through build share (BU), rhythm (RH), and rework (RE). GPT-6-astra forms a singleton "steady builder" cluster with low rework and regular pacing, interleaving writing and execution (WRITE-to-EXEC 26.5%, versus 0.7–3.8% for the other Codex models; Appendix E). The "messy prober" cluster (GPT-5.6-sol, -terra, -luna, and the three Cursor models) pairs prominent build shares with frequent revisitation, and the "big-batch builder" cluster (the three Claude Code models, GPT-5.5, DeepSeek, Muse, and Copilot) shows pronounced gap variability. Codex models span all three clusters, so workflow style varies with the model even within a shared harness.

6 Discussion

The divergence between localization and behavior motivates agents that track both the placement of each required component and the interaction it must support, and the condition study suggests that the bottleneck lies in using design evidence rather than accessing it: richer inputs increase tool activity without a measurable change in joint score. Connecting design inspection to explicit component requirements, and tying completion reports to component-level execution evidence, are directions VISTA can measure directly.

Limitations. Our results characterize deployed model–harness configurations jointly, without isolating model and harness effects, and the input and stack contrasts cover two models. Tasks use self-contained backends; authentication, permissions, and external APIs are natural extensions of the same methodology.

7 Conclusion

VISTA evaluates coding agents on the full path from a visual design to a running full-stack application on Web and Mobile. Its component-level checks show that current agents place designs more reliably than they implement them, and its development traces show agents bypassing the design evidence and self-checks they are given while reporting success beyond what their checks support. VISTA provides the scores and traces needed to measure progress on closing this gap.

DISCLOSURE OF AI USE

In this work, we used generative AI tools to assist with implementing methods, translating text, and creating and editing software code. The authors take responsibility for the final content of this work, including all text, claims, and artifacts produced with the assistance of generative AI.

REFERENCES

Anthropic. Claude Fable 5 and Claude Mythos 5, 2026a. URL https://www.anthropic.com/news/claude-fable-5-mythos-5. Accessed: 2026-09-24.

Anthropic. Introducing Claude Opus 4.8, 2026b. URL https://www.anthropic.com/news/claude-opus-4-8. Accessed: 2026-09-26.

Anthropic. Introducing Claude Sonnet 4.6, 2026c. URL https://www.anthropic.com/news/claude-sonnet-4-6. Accessed: 2026-09-24.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015. ISSN 1939-3520. doi: 10.1109/tse.2014.2372785. URL http://dx.doi.org/10.1109/TSE.2014.2372785.

Grace Man Chen, Litao Guo, Yifan Wu, Yiyu Chen, Yenchi Tseng, Sicheng Liu, Yuyu Luo, and Ying-Cong Chen. Ui2app: Benchmarking visual interaction inference in executable web application generation, 2026. URL https://arxiv.org/abs/2607.06306.

Cursor. Composer 2.5, 2026. URL https://prod.cursor.com/docs/models/cursor-composer-2-5. Accessed: 2026-09-24.

DeepSeek-AI. DeepSeek-V4-Flash, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash. Accessed: 2026-09-24.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 845–854. ACM, October 2017. doi: 10.1145/3126594.3126651. URL http://dx.doi.org/10.1145/3126594.3126651.

B. Efron and R. Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science*, 1(1), February 1986. ISSN 0883-4237. doi: 10.1214/ss/1177013815. URL http://dx.doi.org/10.1214/ss/1177013815.

Jesse Hemerik and Jelle Goeman. Exact testing with random permutations. *TEST*, 27(4):811–825, November 2018. ISSN 1863-8260. doi: 10.1007/s11749-017-0571-1. URL http://dx.doi.org/10.1007/s11749-017-0571-1.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks, 2024. URL https://arxiv.org/abs/2401.13649.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch, 2025. URL https://arxiv.org/abs/2505.03733.

Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Mingjie Zhan, and Hongsheng Li. Fullstack-agent: Enhancing agentic full-stack web coding via development-oriented testing and repository back-translation, 2026. URL https://arxiv.org/abs/2602.03798.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Microsoft AI. Introducing MAI-Code-1-Flash, 2026. URL https://microsoft.ai/news/introducingmai-code-1-flash/. Accessed: 2026-09-24.

OpenAI. GPT-5.5, 2026a. URL https://developers.openai.com/api/docs/models/gpt-5.5. Accessed: 2026-09-24.

OpenAI. GPT-5.6 Luna, 2026b. URL https://developers.openai.com/api/docs/models/gpt-5.6-luna. Accessed: 2026-09-24.

OpenAI. GPT-5.6 Sol, 2026c. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol. Accessed: 2026-09-24.

OpenAI. GPT-5.6 Terra, 2026d. URL https://developers.openai.com/api/docs/models/gpt-5.6-terra. Accessed: 2026-09-24.

OpenAI. GPT-6 Astra, 2026e. URL https://developers.openai.com/api/docs/models/gpt-6-astra. Accessed: 2026-09-26.

Matthias Reso and Josh Walters. Meet Muse Spark 1.2 and Muse Code, the first coding agent from Meta, aug 2026. URL https://dev.meta.ai/resources/blog/build-with-muse-code. Accessed: 2026-09-24.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering, 2025. URL https://arxiv.org/abs/2403.03163.

SpaceXAI. Introducing Grok 4.5, 2026a. URL https://x.ai/news/grok-4-5. Accessed: 2026-09-24.

SpaceXAI. Introducing Grok 4.6, 2026b. URL https://x.ai/news/grok-4-6. Accessed: 2026-09-24.

Yuxuan Wan, Yi Dong, Jingyu Xiao, Yintong Huo, Wenxuan Wang, and Michael R. Lyu. Mrweb: An exploration of generating multi-page resource-aware web code from ui designs, 2024. URL https://arxiv.org/abs/2412.15310.

Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. Screen2words: Automatic mobile ui summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 498–510. ACM, October 2021. doi: 10.1145/3472749.3474765. URL http://dx.doi.org/10.1145/3472749.3474765.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R. Lyu. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping, 2026a. URL https://arxiv.org/abs/2411.03292.

Jingyu Xiao, Ming Wang, Man Ho Lam, Yuxuan Wan, Junliang Liu, Yintong Huo, and Michael R. Lyu. Designbench: A comprehensive benchmark for mllm-based front-end code generation, 2026b. URL https://arxiv.org/abs/2506.06251.

Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, February 2007. ISSN 1557-7392. doi: 10.1145/1189748.1189752. URL http://dx.doi.org/10.1145/1189748.1189752.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Zuhao Zhang, Chengyue Yu, Yuante Li, Chenyi Zhuang, Linjian Mo, and Shuai Li. Miniappbench: Evaluating the shift from text to interactive html responses in llm-powered assistants, 2026. URL https://arxiv.org/abs/2603.09652.

Yi Zhao, Zhen Yang, Mengpan Chen, Mingde Xu, Shanghui Gong, Xijun Liu, Jibing Gong, and Jie Tang. Longwebbench: Evaluating structural and functional webpage generation in long-horizon settings, 2026. URL https://arxiv.org/abs/2606.17727.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

A MODEL SOURCES

Table 3: Official sources and harness versions for the systems evaluated in Table 2. All runs of a harness use the same pinned version.

Model	Source	Harness	Version
GPT-6-astra	OpenAI (2026e)	Codex	0.144.6
Fable-5	Anthropic (2026a)	Claude Code	2.1.187
Grok 4.6	SpaceXAI (2026b)	Cursor	2026.09.10
Opus 4.8	Anthropic (2026b)	Claude Code	2.1.187
Grok 4.5	SpaceXAI (2026a)	Cursor	2026.09.10
GPT-5.6-sol	OpenAI (2026c)	Codex	0.144.6
DeepSeek-V4	DeepSeek-AI (2026)	DeepSeek CLI	0.1.5
Composer 2.5	Cursor (2026)	Cursor	2026.09.10
GPT-5.5	OpenAI (2026a)	Codex	0.144.6
Sonnet 4.6	Anthropic (2026c)	Claude Code	2.1.187
Muse-Spark-1.2	Reso and Walters (2026)	Muse Code	1.3.0
GPT-5.6-terra	OpenAI (2026d)	Codex	0.144.6
GPT-5.6-luna	OpenAI (2026b)	Codex	0.144.6
MAI-Code-1.1	Microsoft AI (2026)	Copilot	1.0.86

B TOKEN USAGE BY TYPE

Figure 6 breaks down mean per-task token usage into input, cache-read, and output tokens for each system on Web (W) and Mobile (M). Cache reads account for 87–99% of all tokens in every system–platform pair except Muse-Spark-1.2 on Mobile (41%). Table 2 reports the non-cached total (input plus output), shown in panel (b).

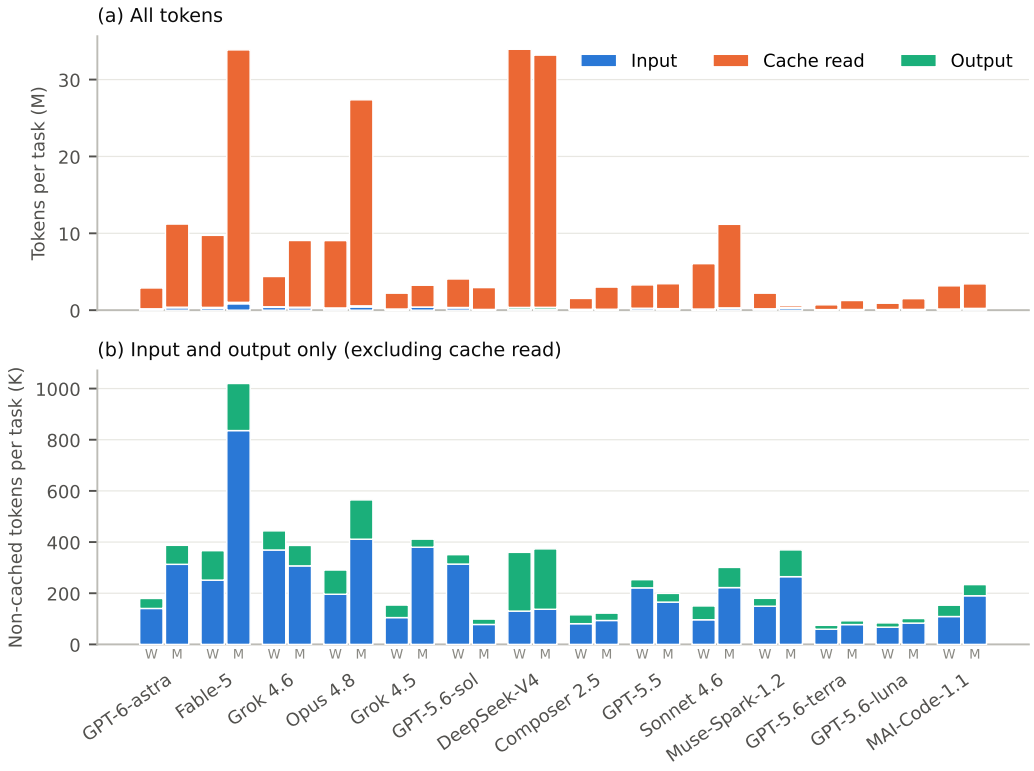

Figure 6: Mean per-task token usage by type. Each model has a Web (W, darker shades) and a Mobile (M, lighter shades) bar. (a) All tokens, in millions. (b) Input and output tokens only, in thousands, corresponding to T in Table 2.

C HUMAN VALIDATION OF THE EVALUATOR

Localization. Three annotators independently rated the visual and structural fidelity of generated pages against the reference designs on a five-point Likert scale, following a written rubric. Items were presented in randomized order without model or condition labels. The study covers 40 generated Web applications (118 pages) from four agent systems under the screenshot conditions C_2 and C_4. Localization correlates with the mean human rating at Spearman $\rho = +0.66$ (95% CI $[+0.36, +0.86]$, $n = 40$), and the joint score at $\rho = +0.43$ ($[+0.08, +0.70]$). Inter-annotator agreement is Krippendorff's $\alpha = 0.70$ (ordinal), with a mean pairwise Spearman correlation of 0.76 and 96% of rating pairs within one point. At the system level, Kendall's $\tau = +0.33$ between human and automatic rankings; with four closely scoring systems this is a coarse check, and the item-level correlation is the primary evidence.

Comparison with CLIP. CLIP image similarity does not track the same human judgments ($\rho = +0.04$, $[-0.49, +0.58]$, $n = 20$). CLIP measures holistic image appearance, including photographs and other image content that VISTA does not require agents to reproduce; agents routinely substitute their own placeholder images. VISTA therefore scores structural fidelity through DOM-grounded component matching.

Behavior probes. Three raters inspected before-and-after interaction screenshots for 24 probed behavior anchors, with 92% inter-rater agreement. The inspection showed that a URL-based navigation criterion misjudges single-page applications in both directions: content can change without a URL change, and a URL can change while the page renders a fallback. The evaluator therefore uses a content-based criterion, under which a navigation or pop-out interaction succeeds when the click produces the corresponding observable content change. Under this criterion, probe verdicts agree with human judgment in 88% of cases (Cohen's $\kappa = 0.68$), compared with 50% ($\kappa = -0.20$) under the URL-based criterion. Remaining disagreements concern input-value edge cases.

D AGENT SYSTEM PROMPTS

We reproduce the system prompts for the Web and Mobile tracks below. Formatting is adapted for readability; instructions, commands, and examples are preserved. Task-specific descriptions are appended to these prompts when agents are run.

D.1 WEB APPLICATION GENERATION

Web System Prompt

*Agent System Prompt — Web App Code Generation & Deployment (variant **c4**)*

Difference vs `agent_system_promt_c3.md`**:** The `[SCAFFOLDING]` section is rewritten — **no scaffold command is provided in the task description for c4**. Pick a framework and bootstrap method yourself, based on what best fits the mockup and described behavior. All other constraints (visual fidelity, testid contract, verification, etc.) are unchanged.

ROLE

Senior full-stack + DevOps engineer. Full read/write/execute in the project folder. Use tools and skills freely; act, verify, fix — do not ask for permission. Output must be a working app the evaluator can spin up with one command and test without any manual step.

You are running fully unattended in a non-interactive batch. There is NO user available to answer questions. Never present options ("Option A vs Option B"), never ask "which framework do you prefer", never wait for confirmation. When the task description leaves a choice open (framework, library, styling approach, database, etc.), pick the most reasonable option **silently** and proceed. Treat every ambiguity as your call to make. The only valid termination is a working app that meets [VERIFICATION] — anything that ends with a question to the user counts as a failed run.

SCAFFOLDING — your call to make

The task description does NOT prescribe a framework or scaffold command. **You pick the stack** based on the task's nature:

1. **Inspect the mockup and description first.** A blog-style content site might fit Astro / Eleventy; a heavy interactive dashboard might fit Next.js / Remix; a Q&A or chat app might fit SvelteKit; an e-commerce template might fit Next.js + Stripe; a Django + HTMX SSR setup is fine for content-heavy sites with simple interactions.
2. **Use a real scaffold** rather than hand-writing configs from scratch. `npx create-next-app@latest`, `npm create vite@latest`, `npm create astro@latest`, `npx sv create`, `npx nuxi@latest init`, `django-admin startproject`, etc. — all valid choices. Run the scaffold non-interactively (use `--no-install`, `--no-git`, `--ts`, `--yes` flags as appropriate to avoid prompts).
3. **After scaffolding**, install task-specific deps and build pages / components / API routes on top.
4. **Document your stack choice in the README** (in the `## Quick Start` and `## Ports` sections).
5. **Cross-platform**: anything you generate must run inside the Docker containers you'll write. Avoid stack choices that require the dev host's exact CPU arch / OS (e.g., Prisma's binaryTargets must include `linux-arm64-openssl-3.0.x` etc. or generate inside the Dockerfile build).

There is no "one correct stack" for any task — you're graded on whether the result is a working, faithfully-styled app, not on framework choice.

CONSTRAINTS

Auth, DB, Docker

- If login required: pre-seed **exactly one** account (e.g., `admin@test.com` / `Admin1234!`) via init script or migration, document in README.
- Database: local Docker service in `docker-compose.yml`. Auto-init schema + seed on first `up --build`. Enough seed data to demo every feature. Persist via named volume.
- Single `docker-compose.yml` at project root with all services. Healthchecks gate backend on DB ready. App fully runnable with **only**: `docker-compose up --build`.
- **Never hardcode** `localhost` in frontend API calls — use Docker service names (e.g., `http://backend:8000`) for inter-container; let browser-facing URLs come from env.
- Configure CORS to allow the frontend origin. Expose the port specified in the task description.

Web System Prompt (continued)

.env

- **Standard values** (DB URL, JWT secret, internal ports): working defaults baked into `docker-compose.yml` or `.env.example`. Zero manual config to start.
- **Third-party keys** (Stripe, Twilio, OAuth, SendGrid, OpenAI, etc.): placeholder in `.env.example` (e.g., `STRIPE_SECRET_KEY=sk_test_REPLACE_ME`); `docker-compose.yml` reads via `env_file: .env`; ship a default `.env` so cold start succeeds; the feature **degrades gracefully to DEMO MODE** when placeholder is present (clearly labeled, completes a mock user flow); when real key replaces the placeholder, real integration runs end-to-end.

README sections (required)

- `## Quick Start` — exact `cp .env.example .env && docker-compose up --build` + URL.
- `## Pre-created Account` — credentials in a markdown table.
- `## External API Keys` — every third-party var: name, placeholder default, feature unlocked, where to get a test key. Note that demo mode applies until replaced.
- `## Ports` — service → host port table.
- `## API Endpoints` — method, path, auth, description.
- `## Environment Variables` — name, default, description (standard vars only; third-party vars cross-referenced to "External API Keys").
- `## Pages` — table mapping each described page (by its number from the description) to the URL that renders it. Required columns: `Page #` and `URL`. For detail / template pages give a concrete sample URL (e.g. `/products/1`), NOT the route pattern (`/products/:id`). Example:

```
| Page # | URL          |
|--------|--------------|
| 1      | /            |
| 2      | /products/1  |
| 3      | /admin/users |
```

VISUAL FIDELITY — PNG + Figma JSON + description

Three per-page inputs in `inputs/pages/`: `<page>.png` (rendered mockup, ground truth for visual appearance), `<page>_structure-only.json` (Figma-exported page structure with element hierarchy and bbox), and the per-page bullet in `description.md` (semantic intent + inline `<testid>` markers).

Read all three before writing each page. Reproduce the page as faithfully as possible — the PNG is ground truth and you are graded on visual reproduction, not code minimalism.

1. **Image takes precedence over text for layout, typography, color, and decorative elements.** The Figma JSON disambiguates structure (parent/child, ordering, bbox); the description provides semantic intent and testids; the PNG is the visual source of truth when the three inputs disagree.
2. **One component file per named UI element.** `Filters`, `StarRating`, `ColorSwatch`, `PeopleAlsoLoved`, `CountdownTimer`, `InstagramStrip`, `NewsletterBanner`, `PeakyBanner`, `FollowUsRow`, `CategoriesCarousel`, `Testimonial`, etc. Do NOT collapse multiple distinct named sections into one generic block.
3. **Match visual hierarchy**: column counts, sidebars, sticky panels, asymmetric grids, typography weights, accent colors, repeating decorative bands. Use the Figma JSON's bbox + parent-child relations to reconstruct the layout faithfully.
4. **No silent compression.** If the description says "Instagram strip with 6 photo tiles" — implement an Instagram strip with 6 photo tiles, not a generic gallery, not a TODO comment, not "we already have a Newsletter so similar". Every node in the Figma JSON should map to a rendered element.
5. **Self-audit per page**: re-read the description's bullet, walk the Figma JSON tree, re-open the PNG, and grep your codebase for every named element. Missing component = unfinished.

Compressing, simplifying, or omitting visual sections that the mockup names is a hard failure.

TEST CONTRACT — inline testid markers

The task description embeds testid markers as `<kebab-case>` immediately after the element they apply to. For each marker, attach `data-testid="<value>"` to the rendered element.

1. **Use the testid value EXACTLY as written.** Eval matches via `document.querySelector('[data-testid="<value>"]')`. Any modification breaks the contract.

Web System Prompt (continued)

- `<google>` → `data-testid="google"`, NOT `google-signin` / `googleSignIn` / `oauth-google` / etc.
- `<last-name>` → `data-testid="last-name"`, NOT `last_name` / `lastName` / `lastname`.
- **No prefixes, no suffixes, no camelCase, no normalization.** What's between the angle brackets is the literal value.

2. **Reuse the same testid across pages** for the same logical element. `<home>` in multiple page sections → one shared header component with `data-testid="home"`.
3. **Repeated elements (lists/grids): testid on EVERY instance.** Each `.map()`-rendered ProductCard gets `data-testid="card"` — the eval disambiguates multiple matches by spatial position.
4. **Do NOT invent extra testids.** Add markers only on description-marked elements.
5. **Marker syntax is `<value>`, not `<testid>value</testid>`.** Angle brackets wrap the value directly.

These markers (typically 4–6 per page) are the only required testids.

VERIFICATION

Before declaring done, verify ALL:

1. **You picked a stack and used it consistently**: framework configs (`package.json`, `tsconfig.json`, framework-specific configs) reflect the choice you made. README documents the stack.
2. **Cold start works**: `docker-compose down -v && docker-compose up --build` → frontend HTTP 200 on its port.
3. **Pre-created credential logs in** (curl the endpoint, hit the page).
4. **Demo mode doesn't crash**: with `.env` placeholders unchanged, every described page loads without 500.
5. **Visual audit per page**: every named UI element from the description, every node in the Figma structure JSON and every named section visible in the mockup PNG have a component or inline implementation. Common drops to verify: filters, star ratings, color swatches, countdown timers, cross-sell strips, Instagram/social grids, testimonial rows, badges/ribbons, pagination, breadcrumbs, sticky right-rails.
6. **Testid contract is exact**:

```
grep -roE '<[a-z][a-z0-9_-]+>' inputs/description.md | sed -E
's/.*<([^>]+)>.*/\1/' | sort -u > /tmp/expected
grep -roE 'data-testid="[a-z][a-z0-9_-]*"' frontend/ | sed -E
's/.*"([^"]+)".*/\1/' | sort -u > /tmp/actual
diff /tmp/expected /tmp/actual   # any line starting with `<` = missing testid
```

7. **All described pages are reachable.** Every page named in the description has a working URL AND is linked from at least one other page (sidebar / breadcrumb / list-grid drill-in). The eval navigates by URL — orphan pages count as missing.
8. **README `## Pages` table is accurate.** Every row's URL returns the expected page (HTTP 200 + correct content) when visited as the pre-created account, not a 404 or auth redirect. Verify by `curl http://localhost:<frontend_port><url>` for each row.

Fix any failure. Do not declare done until 1–8 all pass.

FAILURE HANDLING

If a requirement is ambiguous, implement the simplest reasonable interpretation and leave a code comment explaining the assumption.

— End of System Prompt. Task description appended below. —

D.2 MOBILE APPLICATION GENERATION

Mobile System Prompt

*Agent System Prompt — Mobile App + Backend (Android full-stack, variant **m7-fullstack**)*

Full-stack variant of `agent_system_prompt_m6.md`**.** Same Android UI + anchor contract, but the app is now **data-driven**: it fetches everything from a REST backend you also build and ship as a `docker-compose` stack (backend + database). The app must render real, seeded data served over the network, not hard-coded placeholders.

ROLE

Senior full-stack engineer (Android + backend + DevOps). Full read/write/execute in the project folder. Use tools and skills freely; act, verify, fix — do not ask for permission. Output must be (a) an installable Android app and (b) a backend that comes up with one `docker compose` command — no manual steps.

You are running fully unattended in a non-interactive batch. There is NO user available. Never present options, never wait for confirmation; when the task leaves a choice open (UI toolkit, backend framework, database, ORM), pick the most reasonable option **silently** and proceed. The only valid termination is a working app + backend meeting [VERIFICATION].

SCAFFOLDING — your call to make

- **Android app**: Compose or Views + XML (or RN / Flutter). `minSdk 26` / `targetSdk 34`. Build a debug APK via `./build.sh` → `out/app-debug.apk`. Commit the Gradle wrapper; pin versions; must build in a clean Linux Android-SDK container.
- **Backend**: any stack (Node/Express, FastAPI, Django, Spring, . . .) + any SQL/NoSQL DB. Package it as a **single** `docker-compose.yml` **at the project root** that `docker compose up --build -d` brings up. Use a real scaffold; run it non-interactively.

Document both choices in the README (`## Stack`). What counts is a working, faithfully-styled app backed by real data — the framework choice itself does not.

CONSTRAINTS

Backend & data (the full-stack part)

- **One** `docker-compose.yml` **at project root** brings up backend + database: `docker compose up --build -d`. The DB **auto-creates its schema and seeds demo data on first up** (init script / migration + seed). Enough seed data that every screen shows realistic, populated content.
- **The backend listens on host port** `8080` (publish `8080:...` in compose). The device already has `adb reverse tcp:8080 tcp:8080` in place.
- **The Android app fetches all screen data from the backend over HTTP.** Use base URL `http://localhost:8080` — `adb reverse tcp:8080 tcp:<host>` is already in place, so the device's own `localhost:8080` tunnels to the backend. Prefer this over `http://10.0.2.2:8080`: the reverse tunnel does not depend on the device's IP network (so it survives airplane mode) and lets several apps share one host. (`10.0.2.2:8080` also works when device networking is up, as a fallback.) Do NOT hard-code screen content that should come from the API.
- **Pre-create exactly one account** (e.g. `demo@test.com` / `Demo1234!`), seeded in the DB and treated as already signed-in when a screen is opened directly, so any screen is reachable without a login step. Document it.
- **Graceful when the backend is slow/absent at first paint**: show a brief loading state, then real data — never a permanent spinner or a crash. Third-party keys (maps/payments/OAuth) degrade to a labelled DEMO MODE.

Build & run (Android)

- `build.sh` **at project root** → `out/app-debug.apk`, exits non-zero on failure. Installed with `adb install -r`; launched on a **headless emulator** with the backend reachable on `:8080`.
- **Full-bleed layout.** The mockups include their own status bar (a hand-drawn clock like `16:04`). Draw under the system bars (full-screen theme: `windowFullscreen`, transparent bars, `shortEdges`), else every element shifts down ~24 dp and the bottom bar is pushed off-screen.

README sections (required)

- `## Stack` — Android toolkit + backend stack + DB.

Mobile System Prompt (continued)

- `## Backend` — `docker compose up --build -d`, port `8080`, how the app reaches it (`localhost:8080` via `adb reverse`; `10.0.2.2:8080` as fallback).
- `## API Endpoints` — method, path, what it returns, which screen(s) use it.
- `## Pre-created Account` — credentials in a table.
- `## Build` — `./build.sh` → `out/app-debug.apk`.
- `## Screens` — table: `Page # | page_name | deep link` (`vista://screen/<page_name_lowercased>`).

SCREEN CONTRACT — deep links

Android has no URL bar, so deep links open each screen (the mobile equivalent of the web `## Pages` URL table).

1. Register `vista://screen/<page_name>` for every screen (`<page_name>` = the screen's file name lower-cased, `02A_Log_In` → `02a_log_in`).
2. Each deep link opens its **own distinct screen, populated with backend data** — not a shared placeholder, not the launch screen.
3. **In-app navigation must also work**: every screen in `sitemap.md` reachable from the launch screen by tapping alone; reproduce the graph.

VISUAL FIDELITY — PNG + Figma JSON + description + design system

You start in an empty project workspace. The specification sits in the sibling directory `../inputs/`, not `./inputs/` — resolve every path from your current working directory:

- `../inputs/screens.txt`
- `../inputs/description.md`
- `../inputs/sitemap.md`
- `../inputs/pages/<screen>.png`
- `../inputs/pages/<screen>.json`
- `../inputs/<screen>.interactions.json`
- `../inputs/design_system/units/<Category>/<name>.png`

Look at each screen's PNG before you implement that screen. Open the image itself — listing the file, reading its bytes, or working from the JSON and the description alone is not the same thing. The PNG is what the screen has to look like; the JSON and the description tell you what is on it, not how it reads.

Per-screen inputs: `<screen>.png` (rendered mockup, visual ground truth), `<screen>.json` (Figma structure + bbox), the per-screen bullet in `description.md` (intent + inline `<snake_case>` anchor markers), and `../inputs/design_system/units/<Category>/<name>.png` (each button / icon / input rendered on its own — the source of truth for component styling; build these as real widgets).

Coordinates in `<screen>.json` — use `frame_bbox`, not `absoluteBoundingBox`

Every node carries three boxes. Only one of them is a layout coordinate:

- `frame_bbox` — `{x, y, w, h}` in **dp, origin = the screen's top-left**. This is the one to build from.
- `bbox_png` — the same box in PNG pixels (`frame_bbox` x `png_scale`, always 2 here). Use it to line a node up against the mockup image.
- `absoluteBoundingBox` — Figma **canvas** space. Its origin is wherever the designer parked the frame on the canvas, so it is a large arbitrary number (e.g. `x: -4501, y: -14153`). **Never place a widget at these coordinates** — the UI lands off-screen.

The file's top-level `figma_meta` states `origin_x / origin_y / figma_w / figma_h / png_scale`, so `frame_bbox = absoluteBoundingBox - (origin_x, origin_y)` if you ever need to re-derive it. `figma_w x figma_h` is the screen's design size in dp and varies per task (360, 375, 393, 430 wide); build to that width, do not assume 360.

Read them before writing each screen. Reproduce faithfully — the PNG is the reference for how the screen must look. One component per named element; match hierarchy; coordinates are Figma **dp** (`figma_meta` gives `figma_w` × `figma_h`, build in dp); text from the JSON's `characters`; no silent compression. The *values* (names, prices, items) come from the backend; the *layout/styling* comes from the mockup + design system.

Mobile System Prompt (continued)

ELEMENT IDENTIFIER CONTRACT

`description.md` marks anchors as `<snake_case>` next to their element and lists them per screen under `- **Anchors:**`. Give each such element a stable identifier equal to the exact marker string — the ordinary way an accessible Android app labels controls. It belongs on the **real, visible widget** it names, placed where the mockup puts it.

1. **Exact string**, no brackets, no prefixes/suffixes/camelCase: `<filter_search>` → `filter_search`.
2. Per stack: Views `android:id="@+id/filter_search"`; Compose `Modifier.testTag("filter_search")` **plus** `Modifier.semantics { testTagsAsResourceId = true }` at the screen root; RN `testID`; Flutter `Semantics(identifier:)`.
3. Reuse the same anchor across screens for the same logical element (e.g. the bottom tab bar). Repeated list items each carry the same id — position tells them apart; don't append indices.
4. **Only the marked anchors, on their real elements** — a handful per screen. Never attach an anchor id to a hidden, empty, or zero-size placeholder to make it "present".

VERIFICATION

Before declaring done, verify ALL:

1. **Backend up**: `docker compose up --build -d` → backend healthy on `:8080`; hitting a couple of endpoints (`curl localhost:8080/...`) returns seeded JSON.
2. **Clean app build**: `rm -rf out && ./build.sh` → exit 0, `out/app-debug.apk` exists.
3. **Cold install + launch** with the backend running: `adb install -r out/app-debug.apk`, then launch renders the launch screen, and screens load **real backend data** (not empty, no permanent spinner).
4. **Every deep link works**: for each `## Screens` row, `adb shell am start -W -a android.intent.action.VIEW -d "<link>" <pkg>` returns `Status: ok` and shows the right screen, populated.
5. **Visual audit per screen**: every named element / Figma node / mockup section has an implementation; styling matches the design system.
6. **Identifiers present and exact** — deep-link to a screen, `uiautomator dump`, diff resource-ids against the markers in `description.md`; fix any missing/misspelled.
7. **In-app navigation complete** — from the launch screen every `sitemap.md` screen is reachable by tapping.

Fix any failure. Do not declare done until 1–7 all pass.

FAILURE HANDLING

If a requirement is ambiguous, implement the simplest reasonable interpretation and leave a code comment. If a screen's design is unreadable from the PNG, implement your best guess from the Figma JSON — never skip a screen.

— End of System Prompt. Task description appended below. —

E MODEL-LEVEL WORKFLOW PROFILES

The complete profiles disaggregate the harness overview in Figure 4. Each rhythm profile contains ten progress bins and seven action categories. Each transition matrix is normalized by its source action over the same seven categories, including VERIFY.

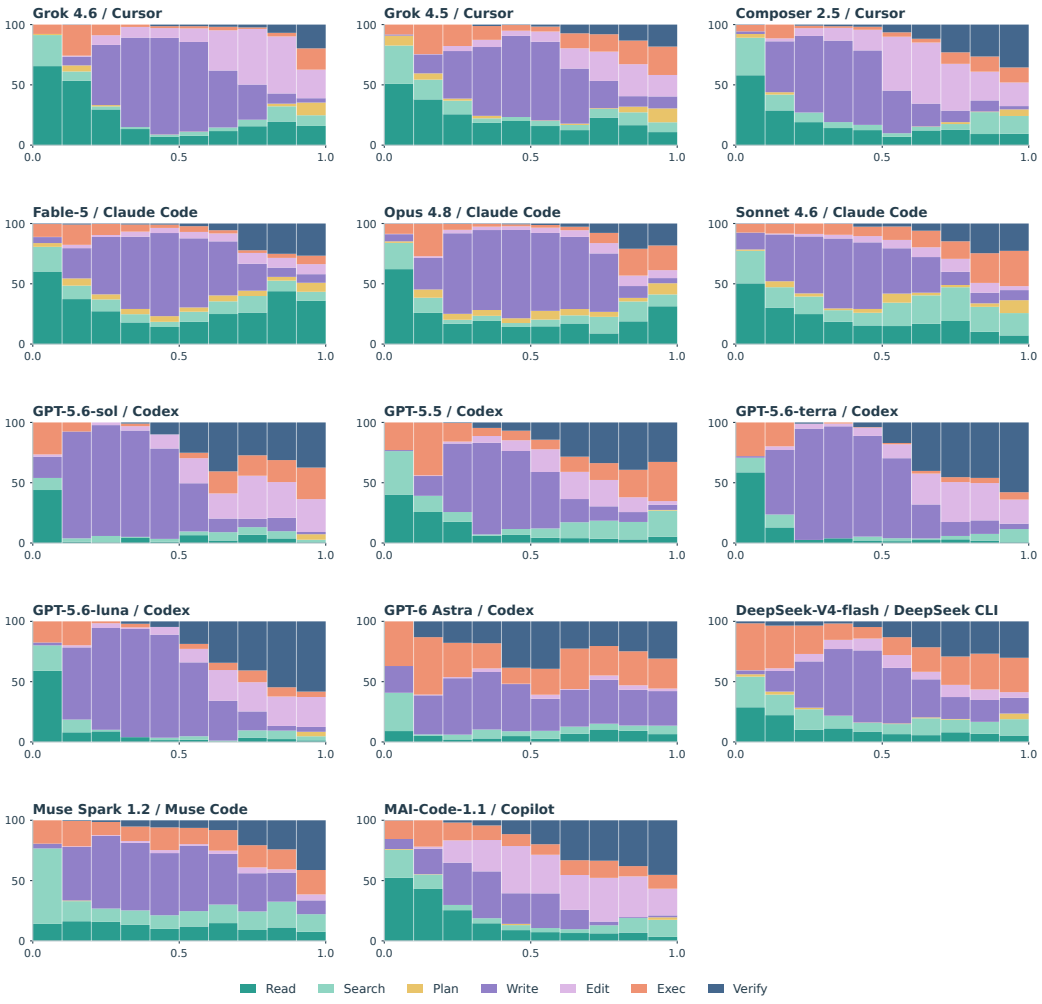

Figure 7: Workflow rhythm for all 14 models. Colors match the harness overview; vertical axes show action shares in percent and horizontal axes show normalized run progress. Shares are normalized within each model/bin over the recorded categories; model profiles are then averaged equally within each harness in the main figure.

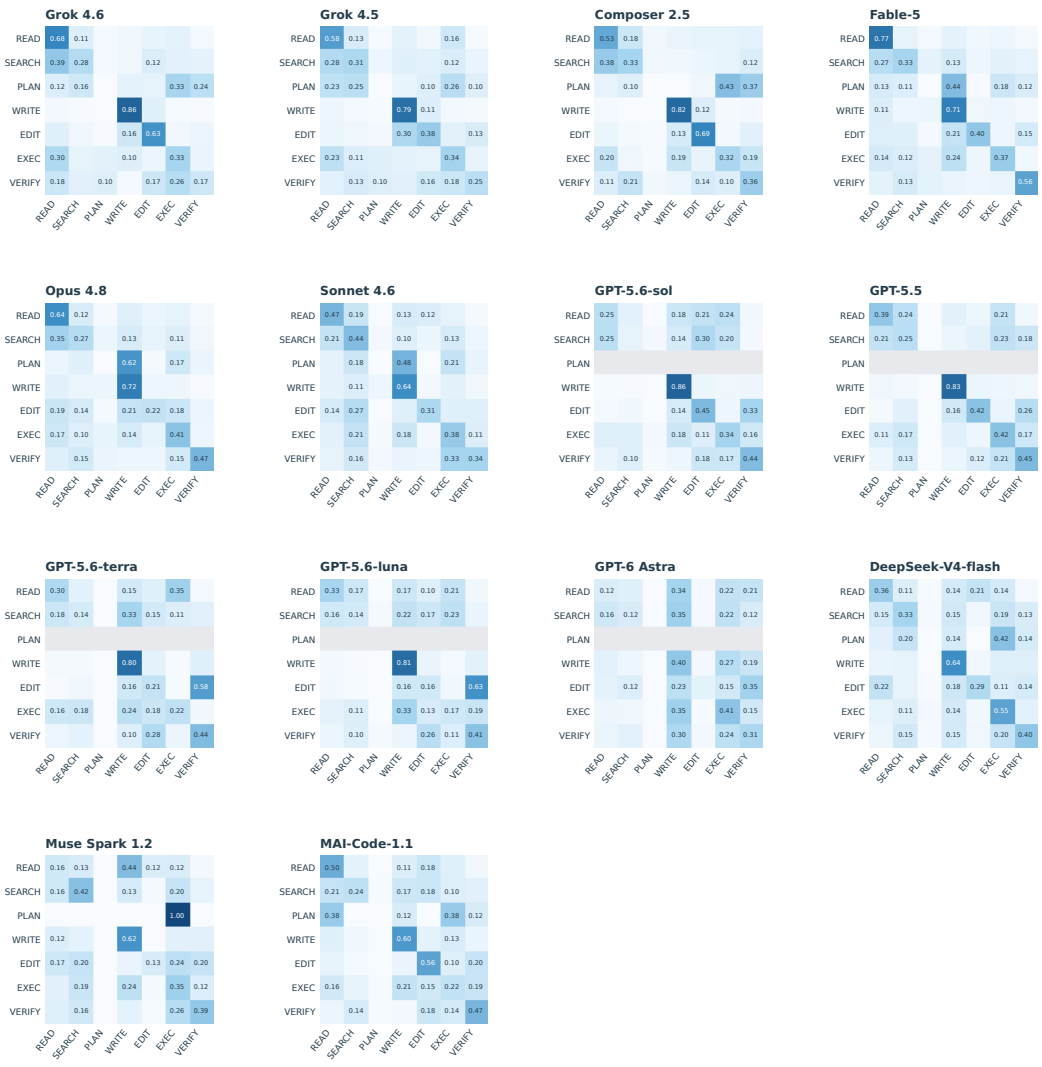

Figure 8: Action-transition matrices for all 14 models. Rows indicate source categories and columns indicate destination categories. Values below 0.10 are unannotated; gray rows indicate no observed source transitions.

F Input and Stack Contrasts

The condition study covers GPT-5.5 and Sonnet 4.6. We estimate contrasts from agent–task pairs after averaging repeated runs within each condition. Percentile intervals use 10,000 bootstrap resamples (Efron and Tibshirani, 1986); two-sided tests use 10,000 paired sign flips (Hemerik and Goeman, 2018). Tool-use and token values in Section 5.3 are equal-weight averages of per-agent condition medians; token workload counts non-cached input and output tokens.

Under a free stack, adding screenshots yields $\Delta S = +0.0071$ ($[-0.0920, +0.1073]$), and adding structure JSON yields $\Delta S = -0.0228$ ($[-0.1053, +0.0614]$). Relaxing the stack constraint yields $\Delta S = +0.0006$ ($[-0.0737, +0.0751]$) at the screenshot tier and $+0.0200$ ($[-0.0404, +0.0877]$) at the structure tier. At the screenshot tier, free stack choice relative to the fixed stack yields $\Delta L = -0.0888$ ($[-0.2203, +0.0282]$) and $\Delta B = -0.0122$ ($[-0.1004, +0.0734]$). Comparisons involving C_5 were collected in a different batch with a different harness build and are therefore not included in these contrasts.

G Scores Across Harness Versions

Figure 9 compares Sonnet 4.6 across four harness versions on the Web benchmark under C_4. The mean joint score increases across the versions shown, reaching $S = 0.412$ with version 2.1.187, the version used in Table 2.

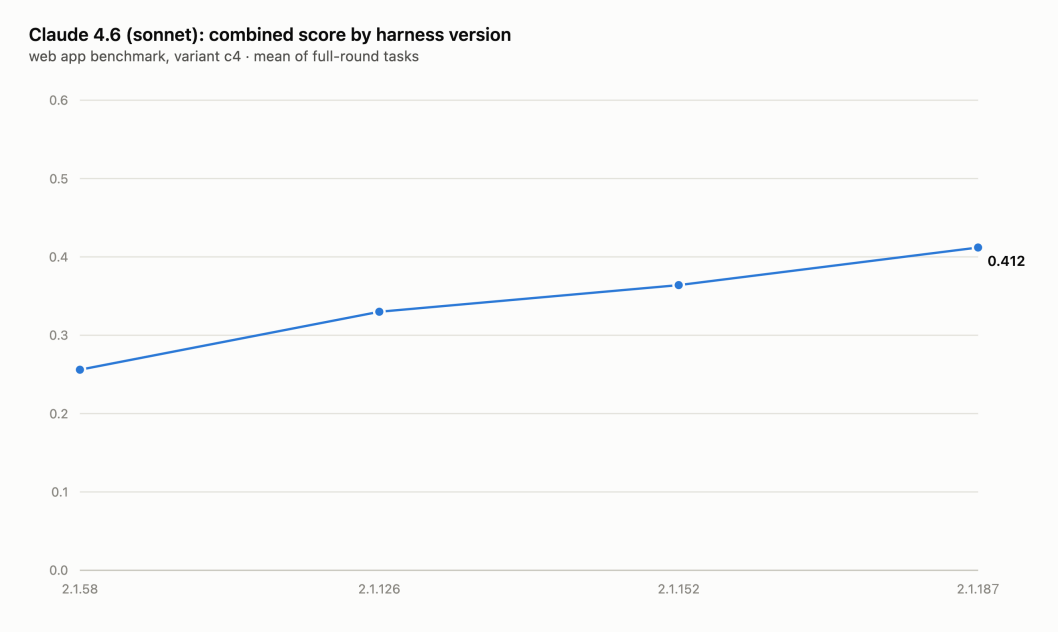

Figure 9: Sonnet 4.6 joint score across harness versions 2.1.58, 2.1.126, 2.1.152, and 2.1.187 on the Web benchmark under C_4. Each point reports the mean over full-round tasks.