THE SPECIFICATIONS NOT SAID: CAN LLMS TURN NOVICE DEVELOPERS' WISHES INTO WORKING PRODUCTS?

Anonymous authors
Paper under double-blind review

ABSTRACT

LLMs increasingly enable novice developers to build applications from scratch with natural language specifications, yet these developers often leave critical requirements unstated. Consequently, turning their immature specifications into working products not only requires implementing the application as instructed but recovering latent specifications as well. This gap is common in practice: in a deployed service where novice developers build mini-programs (a popular type of web application used by over one billion users) from natural-language specifications, only 3.94% of applications satisfy the developer after a single conversation. To evaluate this specification-recovery challenge with real-world developer data, we propose MINIPROGRAMBENCH, a benchmark of 200 tasks across 10 application domains, built from developers' actual application creation and modification experiences and covering both frontend and backend functionality. MINIPROGRAMBENCH pairs feedback from simulated developers with a frozen verification-point rubric, enabling evaluation of both specification recovery and product completion under a constrained feedback budget. Across 11 LLMs, verification-point pass rates reach at most 41.14% initially and 56.97% after up to two feedback-driven revisions. These revisions reveal that models struggle to incorporate feedback while preserving existing functionality. Controlled experiments further show that models perform substantially better with reference specifications than with self-inferred ones, underscoring specification recovery as a key bottleneck.

1 INTRODUCTION

Large language models (LLMs) are making it possible for novice developers to build applications directly from natural-language specifications. These developers often have substantial domain knowledge but limited software engineering experience, and they use LLMs to create personalized tools that support their workflows (Lovable, 2026; Anthropic, 2025). Recent coding systems can modify large codebases and implement new features (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025), while multi-agent frameworks coordinate design, coding, and testing to generate executable applications from natural-language requirements with limited human intervention (Qian et al., 2024; Hong et al., 2024). These advances reduce both the implementation effort and the programming expertise required to build applications. However, they assume that the input specification is sufficiently complete. In practice, novice developers often begin with practical goals and rough ideas, leaving critical requirements unstated.

These initial ideas rarely specify everything needed to build a working product. Developers may understand the problems they want to solve without knowing how to translate their goals into complete software specifications (Brooks, 1987). Their initial requests to LLMs are therefore often brief and underspecified. Consider the request "Build an outdoor signup app." After signing up and reopening an activity, a developer may report, "I signed up, but I'm not listed." This feedback makes a missing requirement concrete: signups must persist and remain visible when the activity is reopened. The reconstructed latent specification also requires separate rosters for different activities (Figure 1). An application generated from such a request may be executable while still failing to meet the developer's practical needs. Reliable application generation requires not only synthesizing code but also recovering the developer's *latent specification*. This specification comprises requirements that

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.

can be inferred from the initial request and others that become identifiable only through interaction with an intermediate implementation and subsequent feedback (Zamfirescu-Pereira et al., 2025; Chou et al., 2026).

We use sustained application use as a behavioral proxy for developers' satisfaction. In data collected from our deployed service, 3.94% of applications overall and only 2.85% of those with no recorded revisions meet our sustained-use criterion. Among trajectories containing developer-requested modifications, 84.1% involve at most two revisions after initial generation (Section 4.1). These observations motivate application-generation systems that infer requirements and use limited feedback effectively to clarify and address unmet needs.

We introduce MINIPROGRAMBENCH[1], a benchmark for generating mini programs that reconstructs latent specifications from real developers' application creation and modification histories (Figure 1). The benchmark uses Tencent's Weixin (WeChat) Mini Program platform, within the broader Weixin and WeChat ecosystem of over 1 billion monthly active users (Tencent, 2026d). The platform provides consistent runtime conditions for building, executing, and evaluating generated applications, while allowing users to access them without installing separate native applications. Unlike frontend-focused benchmarks (Wang et al., 2026; Liu et al., 2026a), MINIPROGRAMBENCH includes applications whose generated frontend and backend code must work together to maintain persistent state and support workflows across multiple user roles.

Each task in MINIPROGRAMBENCH pairs an underspecified developer request with a private *latent specification* defining the requirements to be evaluated. Starting from the request, an application-generation system produces an initial implementation. Guided by this specification, a simulated developer explores the application and provides feedback on unmet requirements to support iterative revision.

Building such a benchmark presents two key challenges. First, evaluating applications generated from underspecified requests and providing feedback on them require a fuller account of developers' intended needs. We address this challenge through a mini-program generation service deployed on the WeChat Mini Program platform, where developers describe their desired applications and request modifications through feedback. We collect anonymized trajectories comprising initial requests, subsequent modification requests, and application usage records. We retain trajectories whose modification feedback reveals unstated requirements and whose final applications exhibit sustained use, providing a behavioral signal of practical value. From these trajectories, we reconstruct *latent specifications*, with each requirement linked to supporting evidence and reviewed by experienced mini-program developers. This process preserves the underspecification of the initial requests while grounding evaluation in requirements revealed through subsequent developer feedback.

Second, reliable evaluation requires explicit behavioral criteria and evidence of whether generated applications satisfy them. GUI-based evaluators may leave relevant conditions untested or misattribute agent interaction failures to application defects (Li et al., 2026). We mine specification-grounded verification points (VPs) from recurring defects in model-generated implementations and freeze them as a common rubric across revisions. A code-based LLM judge combines source-code inspection with GUI traces from the same application version to assess these points, including conditions not exercised during exploration. While mini programs serve as our testbed, the construction and evaluation pipelines can be adapted to other application-generation settings.

Our experiments show that current models struggle to turn developers' rough ideas into products that satisfy their needs. Given only the initial underspecified request, the best-performing model achieves a verification-point pass rate of 41.14%. Motivated by the median of two revisions among trajectories with developer-requested modifications in our deployed service, we allow up to two revisions guided by simulated-developer feedback. Within this budget, the highest pass rate across models reaches 56.97%. The two evaluated agentic harnesses improve the pass rate by up to 9.70 percentage points over direct generation on a 50-task subset, but substantial gaps in product completion remain. Across revisions, some failed VPs associated with feedback-targeted requirements remain failed, while some previously passed VPs receive failure verdicts. This pattern suggests that current systems struggle to incorporate feedback while preserving previously satisfied behavior. Controlled experiments further show that the same models achieve substantially higher product completion with reference latent

[1]We release the evaluation code and dataset in an anonymous repository: `https://anonymous.4open.science/r/miniProgramBench-0527`.

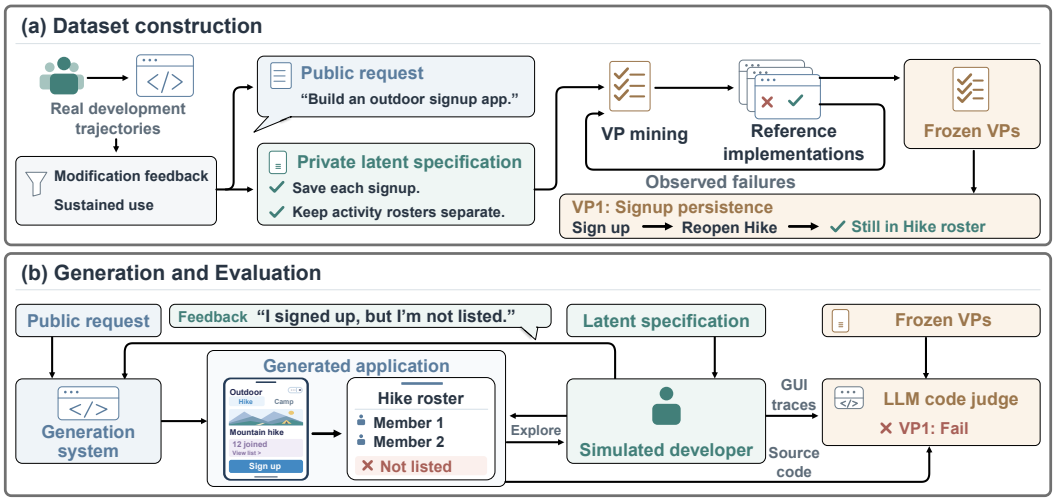


Figure 1: **Overview of MINIPROGRAMBENCH. (a)** Selected developer trajectories yield public requests and private latent specifications. Verification points (VPs) are refined using the training subset of reference implementations, selected using the validation subset, then reviewed and frozen. Each check specifies a precondition, an action, and an expected outcome. **(b)** A simulated developer uses the latent specification to explore generated applications and provide revision feedback. The LLM code judge assesses frozen VPs using source code and GUI traces from the same application version.

specifications than with self-inferred specifications. These results identify intent recovery as a key bottleneck in translating models' implementation capabilities into working products.

We summarize our main contributions as follows:

- We provide a behavioral study of application generation for novice developers in a deployed service. We show that few applications meet our sustained-use criterion without revision and that most trajectories with developer-requested modifications contain at most two revisions, motivating evaluation of specification recovery under a limited feedback budget.
- We introduce MINIPROGRAMBENCH, comprising 200 tasks across 10 application domains grounded in real developers' creation and modification trajectories, together with a developer simulation framework for iterative feedback and a verification-point rubric for product-level evaluation.
- We systematically evaluate 11 language models and 2 agentic harnesses, analyzing product completion and the ability to incorporate feedback while preserving previously satisfied behavior. Controlled experiments identify intent recovery as a key bottleneck in realizing models' implementation capabilities.

2 RELATED WORK

2.1 CODE GENERATION AND SOFTWARE DEVELOPMENT AGENTS

Coding agents support tasks ranging from modifying existing codebases (Yang et al., 2024; Zhang et al., 2024; Chen et al., 2024) to generating general-purpose repositories from scratch (Luo et al., 2026). For end-to-end application development, WebGen-Agent (Lu et al., 2025) uses screenshot analysis and GUI-agent feedback to iteratively improve generated websites, while TDDev (Wan et al., 2025) follows a test-driven workflow to generate full-stack web applications. Beyond the web, EvoDev (Liu et al., 2026b) organizes Android development around a feature-dependency graph. app.build (Kniazev et al., 2025) provides a production-oriented framework for end-to-end application generation.

Complementing these generation systems, WebCoderBench (Liu et al., 2026a) evaluates web application generation from real user requirements using fine-grained quality metrics. Our work examines whether generated applications satisfy requirements that remain implicit in developers' initial requests and are revealed through subsequent interaction.

2.2 Benchmarks for Underspecified Software Requirements

Users may leave important requirements unstated and articulate them only after inspecting or interacting with an implementation (Zamfirescu-Pereira et al., 2025; Chou et al., 2026). Recent benchmarks examine different aspects of this setting. SpecBench (Hamblin et al., 2026) tests whether models can identify deficiencies in design specifications, while SR-Eval (Zhan et al., 2026) evaluates iterative code generation as requirements are progressively refined. SWE-INTERACT (Raghavendra et al., 2026) and Dialogue SWE-Bench (King & Flanigan, 2026) introduce multi-turn user interaction into development tasks within existing codebases.

Asuka-Bench (Wang et al., 2026) studies application generation from underspecified requests with iterative feedback. It derives requests and evaluation criteria from LLM-formalized product requirements documents, whereas MiniProgramBench reconstructs latent specifications from real developers' application creation and modification histories. Asuka-Bench evaluates self-contained frontend applications through GUI interaction. Our environment supports backend-dependent applications and multi-role workflows, and we assess implementations against frozen verification points using source-code inspection and execution evidence.

3 Benchmark Design

We construct MiniProgramBench from real developers' application creation and modification trajectories to evaluate whether generation systems can turn under-specified requests into working products. As illustrated in Figure 1, we select trajectories with modification feedback and sustained application use, then reconstruct private latent specifications grounded in these records. The specifications guide simulated-developer feedback for iterative refinement and ground the verification points used by a code-based judge to evaluate generated applications using source code and GUI interaction traces.

3.1 Developer Trajectory Collection and Selection

Service deployment and data collection. We deploy a mini-program generation service powered by DeepSeek-V4-Flash (DeepSeek, 2026) on the WeChat Mini Program platform (Tencent, 2026c). Developers describe their desired applications in natural language, interact with the generated implementations, and request changes through feedback. These interactions capture how developers progressively articulate requirements left unstated in their initial requests. Over three months, we collect more than 76,000 generation trajectories, comprising initial requests, intermediate application versions, subsequent modification requests, and application usage records. We screen and anonymize the collected developer requests and user identity information. Appendix A.1 provides further details of the deployed service and data collection.

Trajectory selection. We identify eligible trajectories using two criteria. First, a trajectory must contain subsequent developer-requested modifications, providing evidence of requirements absent from or insufficiently specified in the initial request. Second, the resulting application must exhibit sustained use, defined as at least 10 application invocations after subtracting recorded edits, with at least seven days between the first and last invocation, providing a behavioral indication that the application satisfies the developer's wishes to some extent. Together, these criteria yield 984 eligible trajectories. From this pool, we select 200 trajectories to cover diverse application domains, core functionalities, and usage workflows, forming the tasks in MiniProgramBench. These 200 selected cases undergo additional manual review and anonymization to remove identifying information while preserving the requirements needed for task construction and evaluation.

3.2 TRAJECTORY-GROUNDED TASK CONSTRUCTION

From each selected trajectory, we construct a task $\mathcal{T}_i = (x_i, z_i, vp_i)$ comprising a public request, a private latent specification, and verification points. The public request x_i is derived from the developer's initial request, preserving its under-specification without incorporating requirements revealed in later feedback. To reconstruct the latent specification z_i, we use GLM-5.3 (Z.ai, 2026) to extract requirements from the complete trajectory, group them by feature, and express them as independently verifiable statements. Each requirement retains an `original_from` field linking it to supporting evidence in the trajectory. Ten experienced mini-program developers review the extracted requirements and remove those unsupported by the trajectory or overly specific to an individual developer. The resulting specification guides simulated-developer interaction and grounds the verification points vp_i constructed in Section 3.4. During evaluation, only x_i is provided for initial application generation. Additional requirements are communicated through subsequent simulated-developer feedback. Listings 1 and 2 in Appendix A.7 illustrate the task and verification-point formats.

3.3 SIMULATED DEVELOPER

To support iterative refinement, we instantiate a simulated developer that receives the public request x_i and the private latent specification z_i and interacts with the current application through its GUI. For applications involving shared data and multiple roles, assessing a requirement may require coordinated interactions across participants. A controller guides GUI exploration using the specification, maintaining separate sessions and identities and arranging prerequisite actions by GUI agents acting as other participants when needed. It reviews the resulting observations and execution evidence with the help of source-code inspection, requests further exploration when necessary, and provides concise feedback on an unmet requirement without disclosing the complete latent specification. Compilation failures are returned as compiler-error feedback. After `accept`, no further feedback is issued, and the accepted application version is retained for evaluation in all remaining rounds. The resulting interaction traces are retained as complementary evidence for the subsequent code-based evaluation. Appendix C provides the detailed interaction protocol.

3.4 VERIFICATION POINT-BASED CODE JUDGE

GLM-5.3-Flash serves as the primary code judge for generated applications, using a frozen rubric of specification-grounded VPs mined from recurring implementation defects. Drawing on established UX frameworks (Morville, 2004; Hassenzahl & Tractinsky, 2006), we organize our verification rubric into three dimensions. *Useful* captures the functionality needed to accomplish the developer's goal. *Usable* covers interaction correctness, navigation, state management, and reliable use. *Desirable* captures experience and presentation quality. We operationalize each dimension through features assessable from the implementation and GUI traces, rather than direct measurements of subjective preferences or emotional responses. Each VP describes expected behavior through one or more atomic checks, each specifying a precondition, an action, and an expected outcome. For each VP, we retain its source requirement(s) in the latent specification, enabling subsequent analysis to link feedback-targeted requirements to their associated verification checks.

Reference implementations. We first use 10 models to generate 10 reference applications per task, yielding 2,000 implementations across 200 tasks. Appendix D.2 lists the reference models. For each task, we partition reference implementations into mutually exclusive Train, Validation, and Test sets containing 6, 2 and 2 applications, respectively. Train implementations expose defects that guide candidate generation, while Validation implementations indicate whether these defects recur beyond the examples used to propose them. Test implementations remain inaccessible throughout mining and are evaluated only after the VPs have been frozen, as reported in Section 4.7.

Verification point mining. Starting from an empty candidate pool for each task and dimension, GLM-5.3 iteratively updates the VP candidates over three rounds using the latent specification, historical VPs, and previous evaluations on the Train implementations. Each round proposes 16 candidates, either introducing new points or refining existing ones based on the failures observed in previous rounds. GLM-5.3-Flash then reviews the Train reference implementations against these

candidates. Let $y_{ca} = 1$ if implementation a receives a failure verdict for candidate c, and $y_{ca} = 0$ otherwise. For an implementation split A, we define the failure count as $F_A(c) = \sum_{a \in A} y_{ca}$, where $A \in \{\mathcal{I}_{\text{tr}}, \mathcal{I}_{\text{val}}\}$ denotes the Train or Validation set. Throughout the 3 rounds, we preserve historical candidates and their Train evaluations to guide further generation and refinement. After 3 rounds, we rank the candidates with at least two Train failures, $F_{\mathcal{I}_{\text{tr}}}(c) \geq 2$, by decreasing Validation failure count and retain at most five points per task and dimension, requiring each to have at least one Validation failure, $F_{\mathcal{I}_{\text{val}}}(c) > 0$, and excluding semantically equivalent points or pairs in which one point covers the other. Three expert reviewers then examine cases flagged by an automated quality review and revise or remove problematic VPs (Appendix D.4). After this human review, we freeze the resulting VPs as the full Top-5 rubric used for benchmark evaluation.

Code-based evaluation. In MINIPROGRAMBENCH's evaluation process, after each generation round, the judge evaluates the full frozen rubric using source code and GUI traces from the same application version. The source-based review also covers VPs whose triggering conditions were not reached during GUI exploration. Atomic-check verdicts are aggregated conservatively and a VP passes only when all its checks pass.

4 EXPERIMENTS

4.1 DEVELOPER USAGE AND BENCHMARK COMPOSITION

Sustained use remains uncommon. We analyze more than 76,000 anonymized generation trajectories collected over three months, approximately 7.1% of which contain developer-requested modifications. Using the sustained-use criterion defined in Section 3.1 as a behavioral proxy for satisfaction, we find that 3.94% of applications overall meet this criterion. The proportion is 2.85% among applications with no recorded edits and 11.27% among those with one or two edits. These observational differences do not establish a causal effect of revision. Sustained use remains uncommon even among applications that receive one or two rounds of developer feedback.

Observed revision patterns motivate a limited feedback budget. Among trajectories containing modifications, 84.1% involve only one or two revisions, with a median of two. These observed interaction patterns motivate evaluating both initial generation and refinement with limited feedback. We therefore allow at most two feedback-driven revisions, yielding up to three generation rounds. This protocol examines both the ability to infer unstated requirements from an initial request and the ability to incorporate requirements communicated through subsequent feedback.

Benchmark selection and coverage. Across all trajectories containing modifications, 984 meet the sustained-use criterion. These trajectories combine feedback that helps reconstruct latent requirements with behavioral evidence of the resulting application's practical value. Following the procedures in Sections 3.1 and 3.2, we select 200 tasks from this pool to cover diverse application domains, core functionalities, and usage workflows. The resulting benchmark contains 1,408 *latent requirements* across 10 application domains and seven task types. The benchmark also covers workflows that require frontend interactions, persistent state, and backend execution. Personal stateful and collaborative/service applications account for 81.0% of tasks. Archived traces further show generated cloud-function execution in 66.0% of tasks and multi-account GUI interactions in 16.0%.

4.2 EVALUATION SETUP

Evaluated systems. We evaluate 11 language models from seven families under the reasoning configurations reported in Table 1. These configurations follow model-specific interfaces and do not imply equal inference budgets. On a randomly selected 50-task subset, we additionally compare direct generation with 2 agentic harnesses, pairing GPT-5.6 Sol (OpenAI, 2026b) with Codex (OpenAI, 2026a) and Claude Sonnet 5 (Anthropic, 2026c) with Claude Code (Anthropic, 2026a). Both harnesses use `high` reasoning effort and a 30-minute generation timeout per round. Appendix E.1 provides further configuration details.

Metrics and aggregation. Let $P_{i\ell}$ denote the frozen VPs for task i and dimension $\ell \in \mathcal{L} = \{\text{Useful}, \text{Usable}, \text{Desirable}\}$, and let v_{mirp} denote the verdict for system m, task i, round r, and VP

Table 1: Results on MINIPROGRAMBENCH for initial generation (Round 1) and after up to two feedback-driven revisions (Round 3). Scores are task-macro averages under the frozen VP rubric. All values are percentages.

#	Model	Reasoning	Round 1 (%)				Round 3 (%)			
			Product	Useful	Usable	Desirable	**Product**	Useful	Usable	Desirable
1	GPT-5.6 Sol	high	32.17	34.46	35.04	25.70	50.77	57.61	56.14	38.89
2	Claude Opus 5 (Anthropic, 2026b)	high	31.05	34.50	35.20	23.37	**56.97**	**64.73**	**60.97**	**45.58**
3	Claude Sonnet 5	high	21.10	23.03	22.83	16.50	40.67	47.30	44.89	30.05
4	Gemini 3.1 Pro Preview (Google, 2026)	high	12.89	14.62	14.00	9.53	24.16	28.92	25.99	17.36
5	Gemini 3.8 Flash (Google DeepMind, 2026)	high	25.86	27.98	27.03	21.80	47.00	54.61	48.50	37.68
6	DeepSeek-V4-Pro (DeepSeek, 2026)	high	16.84	19.72	18.89	11.53	35.11	42.03	38.08	25.14
7	DeepSeek-V4-Flash	high	12.50	13.71	14.51	9.22	23.56	27.94	24.91	17.43
8	Kimi K3 (Kimi Team, 2026)	max	<u>38.71</u>	<u>40.52</u>	<u>43.18</u>	**32.32**	54.05	58.95	<u>59.34</u>	43.38
9	GLM-5.3 (Z.ai, 2026)	max	**41.14**	**44.33**	**46.56**	<u>32.23</u>	<u>55.53</u>	<u>62.52</u>	59.18	<u>45.08</u>
10	Hy3 (Tencent, 2026a)	high	11.84	14.35	13.49	7.12	27.87	33.66	30.65	19.30
11	Hy4-preview (Tencent, 2026b)	high	19.63	24.29	20.14	14.88	36.78	45.89	37.98	27.30

Bold and underlined values denote the best and second-best results in each column, respectively. Full three-round results are reported in Appendix E.2.

p. For each task, dimension scores are the percentage of passed VPs among those with pass or fail verdicts. Product pools VPs across the three dimensions before computing this percentage:

$$s_{mir,\text{Product}} = \frac{\sum_{\ell \in \mathcal{L}} \sum_{p \in P_{i\ell}} \mathbb{1}[v_{mirp} = \text{pass}]}{\sum_{\ell \in \mathcal{L}} \sum_{p \in P_{i\ell}} \mathbb{1}[v_{mirp} \in \{\text{pass}, \text{fail}\}]} \times 100\% \tag{1}$$

We macro-average task-level scores, assigning equal weight to each task. For trajectories accepted early, the accepted application version is evaluated in all remaining rounds. Compilation failures receive zero for Product and each applicable dimension, whereas evaluations affected by judge or infrastructure failures are excluded from aggregation.

4.3 MAIN RESULTS

Product completion remains limited. Table 1 shows that current generation systems struggle to satisfy the frozen verification rubric. Given only the initial under-specified request, GLM-5.3 achieves the highest Product score at 41.14%. After up to two feedback-driven revisions, Claude Opus 5 obtains the highest final score at 56.97%. Despite these gains, substantial gaps in product completion remain under the benchmark's evaluation criteria.

Revision improves scores with generally smaller later gains. Product increases at both revision steps for all 11 models. The second revision yields a smaller gain for most of the 11 models. Claude Opus 5 gains 17.24 percentage points in the first revision and 8.68 in the second. Gemini 3.1 Pro Preview is the only exception to this pattern, gaining 5.33 and then 5.94 points. Additional revision continues to improve the observed scores, although its marginal gains are generally smaller within the evaluated interaction budget. Appendix E.2 reports the full trajectories and dimension scores for all 11 models.

4.4 FEEDBACK AND REVISION OUTCOMES

To examine whether feedback translates into requirement satisfaction, we link feedback-targeted requirements to their associated VPs using the mapping retained in Section 3.4. We measure *targeted repair* as the fraction of previously failed target-linked VPs that pass after revision, and *preservation* as the fraction of all previously passed VPs that remain passed. Both metrics use adjacent-round pass/fail verdicts on revisions with confirmed feedback delivery, with the same judge configuration across rounds. Table 2 reports both metrics by revision step.

Table 2: Targeted repair and preservation rates on revisions with confirmed feedback delivery. Rates are computed from paired pass/fail VP verdicts.

Model	Targeted repair (%)		Preservation (%)	
	R1→R2	R2→R3	R1→R2	R2→R3
GPT-5.6 Sol	46.5	41.9	88.5	88.0
Claude Opus 5	53.0	62.9	89.0	94.5
Claude Sonnet 5	41.8	40.6	87.3	84.9
Gemini 3.1 Pro Preview	24.3	27.1	79.9	79.7
Gemini 3.8 Flash	44.3	42.6	87.0	90.9
DeepSeek-V4-Pro	34.6	37.0	85.2	85.6
DeepSeek-V4-Flash	23.1	19.6	76.4	77.9
Kimi K3	51.9	46.4	90.9	87.8
GLM-5.3	46.0	48.7	90.7	90.3
Hy3	31.4	27.3	82.8	80.5
Hy4-preview	34.4	36.4	89.1	89.7

Table 3: Product scores for direct and harness-assisted generation on a 50-task subset. Scores are macro-averaged over matched evaluable tasks within each model and round.

	GPT-5.6 Sol / Codex			Claude Sonnet 5 / Claude Code		
Generation	R1	R2	R3	R1	R2	R3
Direct	30.93	41.58	45.11	22.01	33.28	42.09
Agentic	**33.66**	**46.40**	**47.75**	**27.02**	**40.40**	**51.79**
Δ Product	+2.73	+4.82	+2.64	+5.01	+7.12	+9.70

Targeted repair remains incomplete. Pooling eligible VP transitions across both revision steps, Claude Opus 5 achieves the highest observed targeted repair rate at 57.1%. Even for this model, many previously failed VPs associated with the feedback target remain failed after revision. This rate measures repair of associated checks, which may cover conditions beyond the specific complaint.

Previously satisfied behavior is not always preserved. Previously passed VPs receive failure verdicts after revision for every evaluated model. Pooling both revision steps, 22.7% of previously passed VPs with binary paired verdicts turn into failures for DeepSeek-V4-Flash. Thus, aggregate Product gains can coexist with losses on previously satisfied checks, highlighting the need to preserve existing behavior when implementing clarified requirements. These results describe model-specific feedback trajectories and do not constitute controlled comparisons on identical repair requests. Appendix E.5 provides the analysis details and implementation case studies.

4.5 Evaluation of Agentic Harnesses

We examine whether agentic harnesses improve product completion over direct generation under the protocol in Section 4.2.

Harness support improves performance but remains insufficient. Both evaluated model–harness pairings improve Product over direct generation in all three rounds (Table 3). In the final round, Codex improves GPT-5.6 Sol by 2.64 percentage points, while Claude Code improves Claude Sonnet 5 by 9.70 points. Even the strongest pairing, Claude Sonnet 5 with Claude Code, reaches only 51.79 Product, indicating persistent gaps under the benchmark's verification criteria.

4.6 Understanding the Intent Recovery Bottleneck

We compare three input conditions for Hy3, DeepSeek-V4-Flash, and Kimi K3 on all 200 benchmark tasks. Each model generates applications from the initial under-specified request alone, the request

Table 4: Application generation under three specification conditions on all 200 tasks. Results use initial generation (Round 1) without developer feedback. All scores are percentages.

Model	Input condition	Product	Useful	Usable	Desirable
Hy3	Direct	11.84	14.35	13.49	7.12
	Self-inferred spec	18.85	23.98	20.25	12.31
	Reference spec	**44.08**	**54.23**	**47.14**	**31.53**
DeepSeek-V4-Flash	Direct	12.50	13.71	14.51	9.22
	Self-inferred spec	14.52	14.91	16.17	12.20
	Reference spec	**28.35**	**32.89**	**31.39**	**21.58**
Kimi K3	Direct	38.71	40.52	43.18	32.32
	Self-inferred spec	43.20	48.32	48.49	33.46
	Reference spec	**64.70**	**72.70**	**70.19**	**51.09**

augmented with a self-inferred specification, or the request augmented with the reference latent specification.

Intent recovery constrains product completion. Across all three models, reference specifications yield higher Product scores than both direct generation and generation with self-inferred specifications (Table 4). For Kimi K3, self-inference raises Product from 38.71 to 43.20, whereas the reference specification yields 64.70, a further gain of 21.50 points. With the generating model and evaluation protocol held fixed within each comparison, these gains show that the models can implement substantially more of the intended functionality when its requirements are explicit. The smaller gains from self-inference identify intent recovery as a key bottleneck in realizing models' implementation capabilities.

4.7 Ablation of Verification Point Mining

Mining improves failure detection on held-out implementations. We compare the final mined rubric with a one-shot rubric generated directly from the latent specifications. With VP counts matched per task and dimension, we evaluate both rubrics on the same 400 held-out Test implementations across 200 tasks after the VP sets are frozen. As shown in Table 5, mining increases task-macro VP failure rates across all three dimensions, with gains ranging from 11.93 to 19.08 percentage points. These results show that implementation-conditioned mining exposes more judge-identified failures on held-out implementations.

Table 5: Held-out VP failure rates for the final mined and specification-only rubrics over 200 tasks. Δ denotes Mined minus Spec-only in percentage points.

Facet	Mined (%)	Spec-only (%)	Δ
Useful	80.95	61.87	+19.08
Usable	82.18	65.28	+16.89
Desirable	87.73	75.80	+11.93

5 Conclusion

We introduce MiniProgramBench to evaluate whether application-generation systems can turn developers' under-specified requests into working products. Grounded in real developer trajectories, the benchmark combines latent specifications, simulated-developer feedback, and a frozen verification-point rubric. Across 11 language models and 2 agentic harnesses, feedback-driven revision and harness support improve product completion, but substantial gaps remain. Revisions leave some feedback-targeted requirements unresolved and can disrupt previously satisfied behavior. Controlled experiments identify intent recovery as a key bottleneck in realizing models' implementation capabilities. These findings motivate systems that proactively identify missing requirements, clarify unresolved needs through interaction, and incorporate feedback while preserving working behavior, helping novice developers translate their practical goals into applications that meet their needs.

Reproducibility Statement

Task construction and evaluation are described in Sections 3.2–3.4. Appendices C and D.6 provide the interaction protocol and judge validation. We provide privacy-reviewed task specifications, fixed task IDs, redacted prompts, configuration templates, evaluator code, task-level outcomes, and analysis scripts in our anonymous repository: https://anonymous.4open.science/r/miniProgramBench-0527.

Ethics Statement

This work derives benchmark tasks from product-development traces. Raw traces must remain restricted. Any released task should undergo privacy and policy review and contain only the minimum product requirements required for research. Generated Mini-Programs may concern health, finance, education, or personnel workflows; they are benchmark artifacts and should not be represented as safe production systems.

AI Use Statement

Generative AI was used in both the research workflow and manuscript preparation. In the research workflow, language models powered the application-generation systems under evaluation and assisted with task-specification extraction, simulated-user interaction and feedback, verification-point generation and mining, code-based judging, reference-label construction, and feedback-content analysis. Model outputs therefore contributed to benchmark artifacts, feedback, labels, and reported evaluation measurements; the interaction protocol, human review of benchmark artifacts, and judge validation are documented in Sections 3.2–3.4 and Appendices C and D.6. In manuscript preparation, generative AI assisted with restructuring and editing author-written text, formatting mathematical notation, checking consistency between repository artifacts and reported results, and drafting and formatting figures. The authors verified cited works against primary sources and checked the reported measurements, figures, and conclusions against saved experiment artifacts. Generative AI did not independently formulate the research questions or methodology, design the experiments, retrieve or select related work, or determine the paper's conclusions. The authors reviewed all AI-assisted work and take responsibility for the final content and reported results.

References

Anthropic. How Anthropic teams use Claude Code, 2025. URL https://www-cdn.anthropic.com/58284b19e702b49db9302d5b6f135ad8871e7658.pdf. Official report; accessed September 18, 2026.

Anthropic. Claude Code: Overview, 2026a. URL https://code.claude.com/docs/en/overview. Official documentation; accessed September 21, 2026.

Anthropic. Introducing Claude Opus 5, July 2026b. URL https://www.anthropic.com/news/claude-opus-5. Published July 24, 2026; accessed September 21, 2026.

Anthropic. What's new in Claude Sonnet 5, 2026c. URL https://platform.claude.com/docs/en/models/sonnet-5/whats-new-sonnet-5. Official model documentation; accessed September 21, 2026.

Frederick P. Brooks, Jr. No silver bullet—essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. doi: 10.1109/MC.1987.1663532. URL https://doi.org/10.1109/MC.1987.1663532.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. CodeR: Issue resolving with multi-agent and task graphs, 2024. URL https://arxiv.org/abs/2406.01304.

Yi-Hung Chou, Boyuan Jiang, Yi Wen Chen, Mingyue Weng, Victoria Jackson, Thomas Zimmermann, and James A. Jones. Building software by rolling the dice: A qualitative study of vibe coding. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*, 2026. URL https://arxiv.org/abs/2512.22418.

DeepSeek. DeepSeek API change log: DeepSeek-V4, 2026. URL https://api-docs.deepseek.com/updates/. Official V4-Pro and V4-Flash release and update records; accessed September 21, 2026.

Google. Gemini 3.1 Pro Preview, 2026. URL https://ai.google.dev/gemini-api/docs/models/gemini-3.1-pro-preview. Official model documentation; accessed September 21, 2026.

Google DeepMind. Gemini 3.8 Flash: Model card, September 2026. URL https://deepmind.google/models/model-cards/gemini-3-8-flash/. Published September 2, 2026; accessed September 21, 2026.

Grant Hamblin, Kevin Song, Zhanda Zhu, Anand Jayarajan, Sihang Liu, Nandita Vijaykumar, and Gennady Pekhimenko. Specbench: Evaluating specification-level reasoning for software engineering llm agents, 2026. URL https://arxiv.org/abs/2605.30314.

Marc Hassenzahl and Noam Tractinsky. User experience—a research agenda. *Behaviour & Information Technology*, 25(2):91–97, 2006. doi: 10.1080/01449290500330331. URL https://doi.org/10.1080/01449290500330331.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

Kimi Team. Kimi K3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Brendan King and Jeffrey Flanigan. Dialogue swe-bench: A benchmark for dialogue-driven coding agents, 2026. URL https://arxiv.org/abs/2606.13995.

Evgenii Kniazev, Arseny Kravchenko, Igor Rekun, James Broadhead, Nikita Shamgunov, Pranav Sah, Pratik Nichite, and Ivan Yamshchikov. app.build: A production framework for scaling agentic prompt-to-app generation with environment scaffolding, 2025. URL https://arxiv.org/abs/2509.03310.

Chunyang Li, Yilun Zheng, Xinting Huang, Tianqing Fang, Jiahao Xu, Lihui Chen, Yangqiu Song, and Winston Hu. WebDevJudge: Evaluating (M)LLMs as critiques for web development quality. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/bd94ec71bd120e20a12a68e3729f3837-Abstract-Conference.html.

Chenxu Liu, Yingjie Fu, Wei Yang, Ying Zhang, and Tao Xie. Webcoderbench: Benchmarking web application generation with comprehensive and interpretable evaluation metrics, 2026a. URL https://arxiv.org/abs/2601.02430.

Junwei Liu, Chen Xu, Chong Wang, Tong Bai, Weitong Chen, Kaseng Wong, Yiling Lou, and Xin Peng. Towards iterative end-to-end software development: A feature-driven multi-agent framework, 2026b. URL https://arxiv.org/abs/2511.02399.

Lovable. A first look at the build economy, June 2026. URL https://lovable.dev/blog/a-first-look-at-the-build-economy. Published June 9, 2026; accessed September 18, 2026.

Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Junting Pan, Mingjie Zhan, and Hongsheng Li. WebGen-Agent: Enhancing interactive website generation with multi-level feedback and step-level reinforcement learning, 2025. URL https://arxiv.org/abs/2509.22644.

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.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A repository planning graph for unified and scalable codebase generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=VAQq3Y8tIF.

Peter Morville. User experience design. Semantic Studios, June 2004. URL https://semanticstudios.com/user_experience_design/. Published June 21, 2004; accessed September 18, 2026.

OpenAI. Codex CLI, 2026a. URL https://learn.chatgpt.com/docs/codex/cli. Official documentation; accessed September 21, 2026.

OpenAI. GPT-5.6 Sol model, 2026b. URL https://developers.openai.com/api/docs/models/gpt-5.6-sol. Official model documentation; accessed September 21, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. Swe-interact: Reimagining swe benchmarks as user-driven long-horizon coding sessions, 2026. URL https://arxiv.org/abs/2606.30573.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents, 2025. URL https://arxiv.org/abs/2405.14573.

Tencent. Hy3: Official model repository, 2026a. URL https://github.com/Tencent-Hunyuan/Hy3. Release model, distinct from Hy3 Preview; accessed September 21, 2026.

Tencent. Hy4-preview: Official model repository, 2026b. URL https://github.com/Tencent-Hunyuan/Hy4-preview. Accessed September 25, 2026.

Tencent. Weixin mini programs, 2026c. URL https://www.tencent.com/products/weixin-mini-programs/. Official product page; accessed September 18, 2026.

Tencent. Weixin & wechat, 2026d. URL https://www.tencent.com/products/weixin-wechat/. Official product page; monthly active users of Weixin and WeChat combined as of Q1 2026; accessed September 18, 2026.

Yuxuan Wan, Tingshuo Liang, Jiakai Xu, Jingyu Xiao, Yintong Huo, and Michael R. Lyu. Automatically generating web applications from requirements via multi-agent test-driven development, 2025. URL https://arxiv.org/abs/2509.25297.

Xin Wang, Liangtai Sun, Yaoming Zhu, Shuang Zhou, Jiaxing Liu, Fengjiao Chen, Lin Qiu, Xuezhi Cao, Xunliang Cai, Licheng Zhang, and Zhendong Mao. Asuka-Bench: Benchmarking code agents on underspecified user intent and multi-round refinement, 2026. URL https://arxiv.org/abs/2606.05920.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel 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. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

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.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Z.ai. GLM-5.3: Official model documentation, 2026. URL https://github.com/zai-org/GLM-5. GLM-5.3 section of the official model repository; accessed September 21, 2026.

J. D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Bjoern Hartmann. Beyond code generation: LLM-supported exploration of the program design space. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025. doi: 10.1145/3706598.3714154. URL https://doi.org/10.1145/3706598.3714154.

Zexun Zhan, Shuzheng Gao, Ruida Hu, and Cuiyun Gao. Sr-eval: Evaluating llms on code generation under stepwise requirement refinement, 2026. URL https://arxiv.org/abs/2509.18808.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024. URL https://2024.issta.org/details/issta-2024-papers/127/AutoCodeRover-Autonomous-Program-Improvement.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena, 2023. URL https://arxiv.org/abs/2306.05685.

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 Trajectory Collection, Task Construction, and Requirement Grounding

This section expands the trajectory collection and task-construction procedures in Sections 3.1 and 3.2. We first select developer trajectories with modification feedback and sustained application use, then pair each initial public request x_i with a private latent specification z_i reconstructed from the trajectory. Requirement construction establishes what the application should accomplish; the separate VP mining procedure in Section 3.4 determines how to assess generated implementations against these requirements.

A.1 Deployed Service and Trajectory Selection

Service and collected records. We deploy a mini-program generation service powered by DeepSeek-V4-Flash on the WeChat Mini Program platform. Developers describe their desired applications in natural language, interact with the generated implementations, and submit free-form modification requests. Over three months, we collect more than 76,000 generation trajectories, each

recording the initial request, intermediate application versions, subsequent modification requests, and application usage records. We screen and anonymize the collected developer requests and user identity information.

Eligibility and final selection. A trajectory is eligible if it contains subsequent developer-requested modifications and its resulting application meets the sustained-use criterion: at least 10 application invocations after subtracting recorded edits, with at least seven days between the first and last invocation. Modification feedback reveals requirements absent from or insufficiently specified in the initial request; sustained use provides a behavioral signal of practical value. Approximately 7.1% of collected trajectories contain modifications, and 3.94% of all trajectories meet the sustained-use criterion. Their intersection contains 984 eligible trajectories. From this pool, we select 200 cases spanning diverse application domains, core functionalities, and usage workflows. These selected cases undergo additional manual review and anonymization to remove identifying information while preserving the requirements needed for construction and evaluation.

Revision statistics. Among trajectories containing modifications, the revision count has a mean of 3.22 and a median of two, with 84.1% involving only one or two revisions. Among applications with one or two edits, 11.27% meet the sustained-use criterion. These rates describe the observed application groups and do not establish a causal effect of revision. The median of two revisions motivates the two feedback-guided revision rounds used in the benchmark.

A.2 From Trajectory Evidence to Candidate Requirements

We derive the public request from the developer's initial request after removing personally identifiable information. Subsequent modification requests remain private construction evidence; they are not appended to the initial input given to the generation system.

Using GLM-5.3, we extract candidate requirements from the trajectory, including explicitly requested capabilities and undesired behaviors identified through corrective feedback. Each candidate is associated with its supporting source through an `original_from` field. These links distinguish evidence from the initial request from evidence appearing in later feedback and allow reviewers to inspect the context of an extracted requirement.

The extracted candidates are not automatically accepted as evaluation targets. Historical feedback may describe a durable product need, a temporary defect in one application version, a particular interface choice, or a personal preference. We therefore review the underlying meaning of each candidate before including it in z_i. In particular, we distinguish the behavior the user needs from the particular mechanism through which an earlier implementation provided it.

A.3 Requirement Selection and Normalization

We organize candidates by feature and normalize the retained requirements into independently verifiable statements. During this process, a candidate can be retained, rewritten, merged with overlapping candidates, or excluded. Selection follows four principles.

1. **Traceable support.** A requirement must have substantive support in the trajectory. A plausible feature suggested by an annotator or language model is insufficient by itself.
2. **Relevance to the intended workflow.** The requirement must contribute to the product goal represented by the public request. Later feedback is examined for whether it clarifies this goal or introduces an unrelated extension.
3. **Observable and durable meaning.** Requirements describe application behavior or user-visible outcomes under identifiable conditions. Version-specific symptoms are rewritten into the underlying invariant where possible. For example, a complaint about stale values after reopening an application can support a state-continuity requirement without prescribing a particular persistence mechanism.
4. **Appropriate specificity.** We remove unsupported personal preferences and avoid prescribing page layouts, controls, storage technologies, or internal data structures when alternative implementations satisfy the same need. Independently verifiable behaviors remain separate, while duplicate descriptions of the same failure are consolidated.

The public request is held fixed when assessing these candidates. A weakly supported requirement is revised or excluded rather than justified by adding its answer to the public request.

A.4 Grounding Requirements in the Public Request

We distinguish *historical provenance* from *public-request grounding*. Provenance explains where a requirement was observed in the trajectory. Grounding explains how it relates to the goal and context available in x_i. These are complementary: a later developer request can establish that a need is authentic without making every detail explicitly recoverable from the initial request.

To make this distinction inspectable, the authoring materials associate each requirement with a grounding route:

Direct. The public request explicitly states the relevant operation, outcome, or constraint. The annotation identifies the supporting text span.

Indirect. A concrete clue in the public request supports the requirement through reasoning about the workflow, its objects, or its state transitions. The annotation explains the intermediate inference and the practical failure that the requirement prevents.

Consensus. The public request invokes a concept whose established functional or domain semantics support the requirement, such as an accounting identity, a temporal relation, or a distinction between measurement units. The relevant convention must be stated explicitly; generic appeals to correctness or good design are insufficient.

These labels describe the basis of a requirement, not its importance, difficulty, or the extent of agreement among annotators. In particular, an indirect requirement is a supported completion of the product intent rather than a claim of unique logical entailment.

For indirect and consensus requirements, the review materials represent the justification as a derivation graph. Evidence nodes contain exact spans from the public request; intermediate nodes state individual inferences; and the terminal node identifies the requirement. Dependencies make explicit when an inference relies on multiple premises. The accompanying rationale records the failure prevented and the remaining design freedom. A direct requirement instead uses a text-span-to-requirement mapping.

The graph is an auditable justification, not independent evidence that its conclusion is correct. Reviewers must still examine whether its premises support each inference and whether the conclusion overstates what the available evidence warrants. Private persona preferences and the behavior of a reference implementation do not supply missing public premises.

A.5 Illustrative Requirement Derivations

Consider the cargo-handling calculator task. An English translation of its public request is:

> During ship loading and unloading, I need to keep track of the remaining quantity and estimated completion time for each hold. Please make a calculation tool that is convenient to use on site.

The following examples summarize three grounding routes from its authoring materials.

Direct: per-hold estimates. The request explicitly asks for the remaining quantity and estimated completion time for each hold. This supports a requirement to calculate and display both estimates separately for each hold.

Consensus: quantity and time have distinct semantics. The same phrase names two quantities with different dimensions. An estimated completion time must therefore be represented as a duration or a completion timestamp, rather than as a cargo quantity. This requirement preserves flexibility in time format and presentation while ruling out a semantically incorrect result.

Indirect: continuity after leaving and returning. The phrases "keep track" and "use on site" support reasoning about a continuing operation that does not stop when the user leaves the interface:

> On-site work continues while the tool is closed → the user may return after time has elapsed → the displayed state should reflect the current operation rather than the earlier screen state.

The corresponding requirement concerns restoring the recorded state and updating estimates for ongoing work upon re-entry. It does not require a continuously running background process, cloud storage, or cross-device synchronization; different mechanisms may produce the same observable result.

The final example also illustrates why derivations require review. Its intermediate premises must be justified by the task context and trajectory evidence. Merely observing that persistence would be useful does not establish a requirement for every application.

A.6 Human Review and Final Task Assembly

Ten experienced mini-program developers review the extracted specifications, as described in Section 3.2. The annotation interface organizes the review around the requester description, the product goal, and individual requirements. For each requirement, reviewers can inspect the relevant evidence, its grounding route, the derivation graph, and the stated implementation boundaries.

The review asks whether the evidence supports the requirement, whether the derivation omits a necessary premise, whether the wording changes the developer's intent, and whether it unnecessarily restricts valid implementations. Reviewers also identify duplicate or conflicting requirements and missing behaviors essential to the stated workflow. The interface distinguishes no identified issue, an identified issue, and insufficient information. This allows missing evidence to remain explicit rather than being treated as approval.

Reference application clips, where available, help reviewers understand the scenario. They do not define the correct implementation or establish that a requirement is valid. Review comments can be attached to individual derivation nodes or links, making unsupported inference steps and overly restrictive conclusions actionable.

The resulting task retains the public request, the reviewed latent specification, and supporting authoring records. During evaluation, the generation system initially receives x_i; the latent specification z_i guides simulated-developer exploration and grounded feedback; and the separately mined, frozen verification points define the evaluation rubric. The provenance records and derivation graphs support task auditing without exposing the full specification in the initial generation prompt.

A.7 Task and Verification-Point Formats

Listings 1 and 2 illustrate the structured task and verification-point formats used in MiniProgramBench. In the task example, `original_from` summarizes supporting evidence from the developer's modification history; this evidence is not part of the public request.

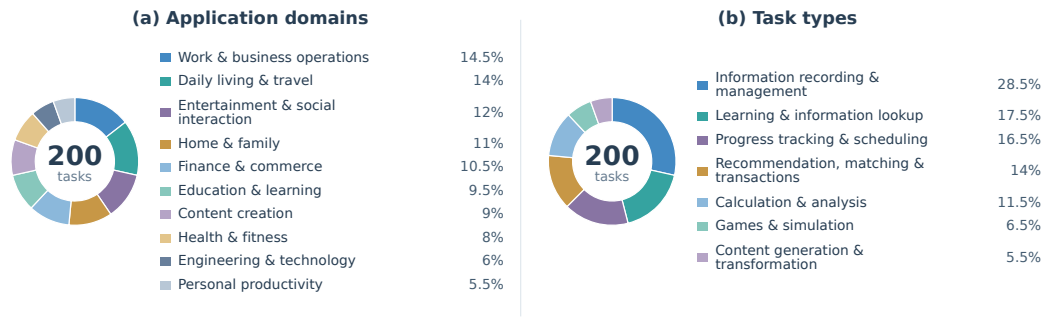


Figure 2: Distribution of the 200 benchmark tasks by (a) application domain and (b) task type. Each task has one primary label on each axis. Percentages are reported within each panel.

```
{
  "taskId": "case-2",
  "title": "Mahjong Score Calculator",
  "userRequest":
    "Record chip scores and calculate results.",
  "latentSpec": {
    "scope": [{
      "feature": "Score entry",
      "kind": "required",
      "requirements": [{
        "text":
          "Enter chip denomination and quantity.",
        "original_from":
          "modification history: denomination x
  quantity"
      }]
    },
    ...]
  }
}
```

Listing 1: Abbreviated example of a trajectory-grounded task.

```
{
  "evaluationDimension": {
    "category": "useful",
    "verificationPoints": [{
      "focus": "Chip calculation process",
      "setup":
        "Enter chip denomination and quantity.",
      "checks": [{
        "given":
          "Denomination and quantity are entered.",
        "when":
          "The user clicks Calculate.",
        "then":
          "The chip score is calculated correctly."
      },
      ...]
    }]
  }
}
```

Listing 2: Abbreviated example of a verification point.

B Task Coverage and Evaluation Environment

MiniProgramBench provides an integrated environment for running generated applications and evaluating their behavior across interactions, state changes, and user roles. Execution-based benchmarks such as WebArena and AndroidWorld assess agents through functional outcomes in working applications (Zhou et al., 2024; Rawles et al., 2025); FullStack-Bench extends application-generation evaluation to frontend, backend, and database functionality (Lu et al., 2026). Our environment brings GUI interaction, generated cloud-function execution, shared data, and independent user identities into the evaluation of applications built from under-specified requests.

Application domains and task types. We characterize the 200 benchmark tasks along two independent axes. The *application domain* describes the context of use, while the *task type* describes the primary activity supported by the application. Each task receives one primary label on each axis. Figure 2 summarizes both distributions. The dataset spans 10 application domains and seven task types. Each domain accounts for 5.5%–14.5% of the tasks, covering diverse everyday application needs. Information recording and management is the most common task type (28.5%), followed by learning and information lookup (17.5%) and progress tracking and scheduling (16.5%).

Execution requirements of the task collection. Figure 3 complements the domain and task-type distributions in Figure 2 with two execution-oriented classifications. Personal stateful and collaborative/service applications together account for 81% of the 200 tasks. Separately, 45 tasks (22.5%) specify multiple accounts in their execution plans. This composition motivates evaluating state continuity, data updates, and interactions across participants.

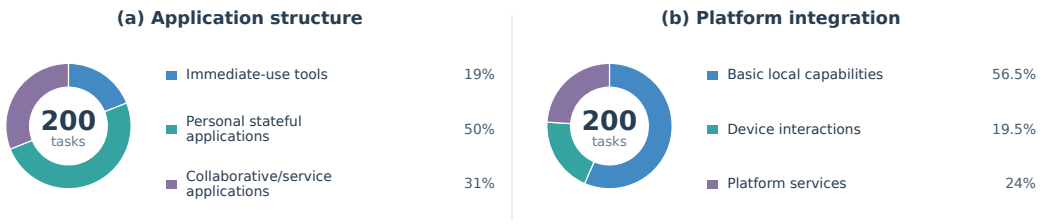


Figure 3: Distribution of the 200 tasks by (a) application structure and (b) platform integration, based on authoring annotations. Each task has one primary label on each axis; percentages are reported within each panel.

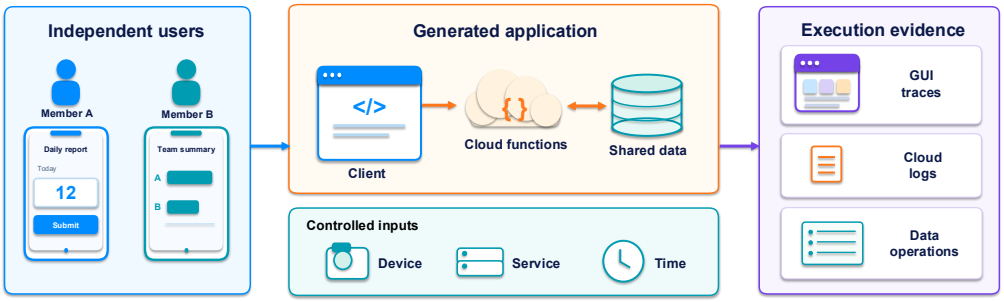


Figure 4: Integrated execution environment. Generated client and cloud-function code operate with independent actor sessions, shared application data, and controlled device and service inputs. GUI observations and backend traces record the resulting application behavior.

Integrated application execution. Figure 4 summarizes the environment. Each actor interacts with the generated application through a separate GUI session with its own identity and client storage. Generated cloud functions execute application logic against a shared application data store, enabling workflows in which one participant creates or updates information that another subsequently accesses. Task-specific fixtures supply device inputs and simulated external-service responses through the corresponding interfaces, while controlled time advancement supports time-dependent scenarios. The generated code implements the business rules that consume these inputs, update records, and present the resulting state. This arrangement makes data persistence, role-dependent behavior, and service-dependent workflows executable within a common evaluation environment.

Observed execution coverage. Table 6 summarizes observed execution activity across eleven models and three generation rounds. Generated cloud functions execute in 132 tasks, database write operations occur in 92, and multi-account interactions and controlled time advancement are observed in 32 and 22, respectively. These records show the environment exercising application behavior across backend execution, shared data, user identities, and time-dependent scenarios. Figure 5 presents this task-level coverage alongside the task requirements.

An example of cross-user state propagation. A team-reporting workflow illustrates how these facilities work together. Member A submits a daily value, Member B opens the team summary under an independent identity, and Member A subsequently revises the value. The corresponding behavioral checks ask whether the submission persists, whether Member B can access the shared report, and whether reopening the summary reflects the revised value. Independent sessions distinguish participants, shared data carries the report across accounts, and repeated GUI interaction exposes the resulting state changes. GUI observations ground simulated-user feedback, while source code and execution evidence support the frozen VP evaluation described in Section 3.4.

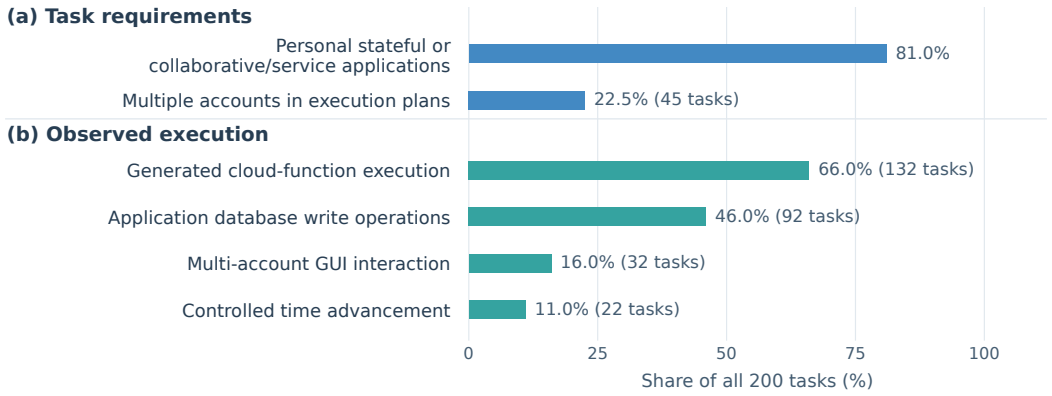


Figure 5: Task requirements and observed execution coverage. Requirements follow task annotations and execution plans. Execution coverage pools archived records across 11 models and three generation rounds. Each task is counted once per category, categories overlap, and all percentages use 200 tasks. Observed activity does not establish task success or coverage by every model.

Execution evidence	Tasks	Evaluations
GUI interaction	190	5,813
Multi-account interaction	32	657
Generated cloud-function responses	132	1,152
Application database reads	123	951
Application database write operations	92	547
Controlled time advancement	22	220

Table 6: Execution coverage across eleven models and three rounds. An evaluation is one model's application for one task in one round. Each evaluation is counted once per evidence type. Cloud-function and database operations must originate from user interaction; multi-account interaction requires GUI actions under at least two distinct identities. Counts indicate execution activity, not task success.

C Simulated-Developer Protocol

Simulated users enable interactive evaluation through scenario-grounded exchanges with an agent (Yao et al., 2024). In MiniProgramBench, the simulated developer inspects generated applications and communicates unmet requirements to guide revision. It combines a GLM-5.3-Flash controller with a Qwen3.6-35B-A3B GUI agent, as introduced in Section 3.3. This section describes their interaction protocol.

C.1 Controller-Guided Interaction and Feedback

Requirement-guided exploration. The controller uses the private latent specification to organize exploration into scenarios with designated actors, behavioral goals, and dependencies. Each scenario specifies the outcome to examine, while the GUI agent discovers an interaction path through the generated application. Dependencies preserve the workflow: a scenario that examines a shared report, for example, follows the scenario that creates it. The generation system receives the public request initially and the resulting user-facing feedback in subsequent rounds.

Role assistance. When a scenario requires information or an action from another participant, the GUI agent can request assistance from that actor. The controller preserves the active session, invokes the assisting actor in its own session, and returns the information obtained through GUI interaction. For example, Member B may need an invitation code that Member A can retrieve from the team-management page. Member B then resumes the original workflow using the returned code. Assistance is scoped to the dependency, preserving each participant's identity and role.

Controller-guided recovery. After exploration, the controller reviews the actions and observations to determine whether they support an assessment of the targeted requirements. When additional interaction can resolve an evidence gap, it requests at most two further attempts directed at the missing observation. For instance, after a report is submitted, the controller may ask the GUI agent to reopen the page and inspect the updated summary. This mechanism connects requirement assessment to the application state reached by the relevant actor.

Feedback generation and replay. If an application fails compilation, the generation system receives the compiler errors as feedback for the next revision. Otherwise, the controller assesses the latent requirements using the accumulated GUI observations, execution evidence, and source code. If all requirements are satisfied, it returns `accept` and issues no further feedback. The accepted application version is retained for evaluation in all remaining rounds. Otherwise, it selects at most one unmet requirement for feedback. The message uses the requesting user's perspective, contains at most two sentences, and describes the required behavior without exposing private requirement identifiers or evaluator terminology. After revision, the simulated user explores the new application version and reassesses the relevant behavior.

For example, if a team member submits a daily value but another member's summary does not include it, the feedback may read: "After another team member submits their daily value, the team summary still shows only my report. Please make the summary include the latest reports from all team members." The next round checks the revised application's submission and summary behavior through renewed interaction. The simulated user supplies revision feedback; the frozen VP rubric supplies the benchmark scores.

D Verification-Point Construction and Judge Validation

D.1 Evaluation Dimensions and VP Design

Our verification rubric comprises three task-grounded quality dimensions—Useful, Usable, and Desirable—drawing on established accounts of user experience Morville (2004); Hassenzahl & Tractinsky (2006). Each dimension contains four categories that guide candidate VP generation.

Useful: correct goal completion. *Core capability* checks whether the intended workflow can be completed end to end. *Domain correctness* checks calculations and business rules. *Data integrity* checks whether saved and retrieved data preserve their meaning. *Transaction state* checks whether actions produce only permitted state changes, including under cancellation, failure, or repeated triggering.

Usable: reliable interaction. *Reachability* checks whether users can find and invoke an operation. *Input guard* checks prevention and explanation of invalid inputs or accidental actions. *Status feedback* checks whether current states and action outcomes are understandable. *Recovery continuity* checks whether users can resume after failure, interruption, or return navigation.

Desirable: experience and presentation quality. *Information hierarchy* checks grouping and emphasis for efficient scanning. *Context semantics* checks whether labels and state expressions convey task-specific meaning. *Visual coherence* checks consistent, readable typography, spacing, and visual conventions. *Microinteraction flow* checks whether feedback and transitions convey changes without disrupting the task. These categories concern observable qualities supported by the task, rather than prescribed decoration or subjective liking.

Category boundaries and VP design. We classify each VP by its primary user consequence. In an expense tracker, losing a committed record concerns Useful, losing entered values during retry concerns Usable, and poor organization of an otherwise readable history concerns Desirable. Similarly, missing success feedback concerns Usable, whereas clear but unnecessarily disruptive feedback concerns Desirable. Each VP must be grounded in task requirements, expressed through observable checks, and permit alternative valid implementations. All three dimensions are evaluated using the frozen rubric, independently of whether another dimension passes.

Table 7: CoreVP-search configuration.

Configuration	Value
Tasks	200
Quality layers	Useful, Usable, Desirable
Reference implementations per task	10
Train / Validation / Test	6 / 2 / 2
Search rounds (R)	3
Candidates per round (N)	16
Train point budget per round (B)	4
Listwise ranking repetitions	3
Ranking aggregation	Normalized Borda score
Checks per point	2–6; core and boundary required
Minimum Train failures (τ)	2
Validation eligibility	Complete verdicts; positive effect
Maximum points per layer (K)	5
Candidate generator	GLM-5.3
Semantic gate	DeepSeek-V4-Flash
Relation assessor	DeepSeek-V4-Flash
Listwise ranker	Qwen3.8-27B
Search-stage code judge	GLM-5.3-Flash

D.2 Reference Implementations and Search Configuration

Table 7 summarizes the configuration used for the 200-task search experiment. The reference pool contains one first-round implementation per task from each of ten models: GPT-5.6 Sol, GPT-OSS-120B, Claude Sonnet 5, Gemini-3.1-Pro-Preview, Qwen3.8-27B, Qwen3.6-35B-A3B, Qwen3.5-4B, DeepSeek-V4-Pro, Hy3, and GLM-5.2. This yields 2,000 reference implementations across 200 tasks. The reference-pool roster is distinct from the eleven-model benchmark comparison. For each task, the ten implementations are randomly partitioned into six Train, two Validation, and two Test implementations. Assignments vary across tasks but remain fixed throughout search and are shared across the three quality dimensions. Candidate pools and search histories are maintained separately for each dimension. Validation is accessed only after all search rounds, and Test only after the final selections are frozen.

Model assignments. GLM-5.3 proposes and refines candidate VPs, while GLM-5.3-Flash supplies code-based verdicts. We assign semantic validity checks and relation assessment to DeepSeek-V4-Flash, and listwise ranking to Qwen3.8-27B, to control computational cost and reduce reliance on a single model throughout rubric construction. This separates candidate proposal from semantic screening and prioritization. The resulting selections undergo the additional quality review and human adjudication described in Appendix D.4.

The Judge returns pass, fail, or unknown for each check. A verification point fails if any check fails, passes if all checks pass, and is otherwise unknown. An unresolved point–implementation pair receives at most one additional evaluation attempt within an evaluation version; successful pairs are reused. Unknown verdicts receive no failure credit and do not satisfy the completeness requirement for final selection.

D.3 Verification-Point Search

Figure 6 summarizes the search workflow, and Algorithm 1 specifies the procedure for one task. The append-only ledger stores all candidates, including rejected candidates and earlier versions of rewritten points. MEMORY deterministically exposes candidate history, gate decisions, current semantic rankings, relations, category counts, and anonymized Train verdict counts. Train counts are included only when the corresponding verdict matrix is complete; Validation and Test outcomes are excluded.

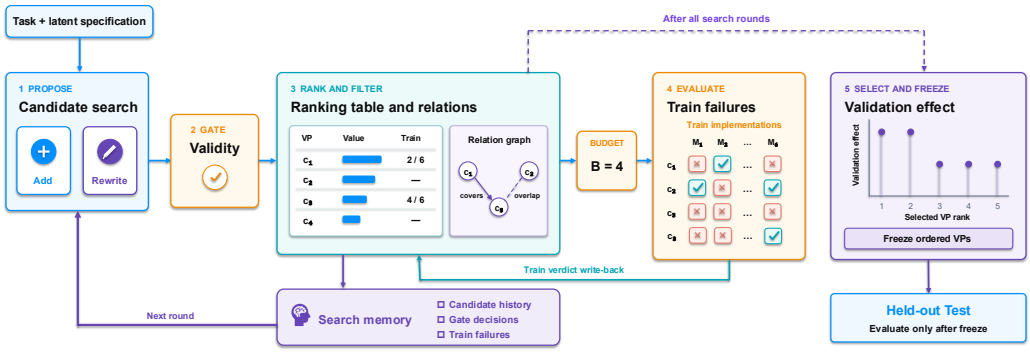


Figure 6: CoreVP-search workflow for one task and quality dimension. Candidates pass a semantic validity gate and enter a ranking table with pairwise semantic relations. Budgeted Train evaluations update the search memory for subsequent rounds. After search, eligible candidates are ranked by Validation effect and selected subject to nonredundancy constraints. The right panel schematically shows individual VP effects in the selected order, with ties allowed; it does not represent cumulative coverage. The ordered selections are frozen before held-out Test evaluation. Table entries, graph relations, and plotted effects are illustrative.

For each layer, the relation graph contains every gate-admissible candidate. Each pair is labeled as independent, equivalent, directional coverage, or partial overlap. New rounds assess all missing pairs while preserving existing labels, without transitive inference. Relation assessment uses candidate semantics and task context, not implementation outcomes. We define $\mathrm{Conflict}_G(c, s)$ to hold when c and s are equivalent or either covers the other; partial overlap does not constitute a conflict.

For split A, $\mathrm{Complete}(c, A)$ requires a known pass/fail verdict for every implementation in A. Using the binary failure indicator y_{ca} defined in Section 3.4, the selection effect for complete candidates is

$$e_A(c) = \frac{1}{|A|} \sum_{a \in A} y_{ca}.$$

Semantic value determines which candidates receive Train evaluation, whereas Validation effect determines the final selection order. The two rankings serve different purposes. Each point in the right panel of Figure 6 represents the Validation effect of one selected VP. Selected effects are nonincreasing, with ties resolved by semantic value and then candidate ID; at most five VPs are retained per quality dimension.

D.4 Quality Review and Rubric Finalization

After search fixes the ordered per-layer selections, GPT-5.6 Sol reviews candidate VPs for requirement grounding, atomicity, internal consistency, observable Given–When–Then checks, implementation independence, core and boundary coverage, user value, layer assignment, and redundancy. Each reviewed VP receives a `keep`, `revise`, `drop`, or `uncertain` decision. The review flags 246 of the 2,818 search-selected VPs. Three expert reviewers independently examine the flagged cases and any directly related changes using the review rationales, public task materials, and sibling VPs. Disagreements are resolved through discussion. Generated implementations, model identities, search scores, and benchmark outcomes are excluded from this review.

Table 8 summarizes the selected rubric before and after review. Fourteen selected VPs are removed without backfilling, and the original selection order is retained. The final rubric contains 2,804 VPs and 5,627 atomic checks across 200 tasks, with at most five VPs per quality layer for each task. The reviewed VPs are then frozen as the actual evaluation points used across all generation rounds of the benchmark comparison.

Algorithm 1 CoreVP-search for one task

Require: Task context x, latent specification z; disjoint implementation sets T, D, H; rounds R, proposals N, Train budget B, support threshold τ, final budget K
Ensure: Frozen point sequences $\{S_\ell\}$ and Test verdicts

for each quality layer ℓ **do**
  Initialize ledger L_ℓ and relation graph G_ℓ
  for $r = 1, \ldots, R$ **do**
    $M \leftarrow \textsc{Memory}(L_\ell)$
    $C \leftarrow \textsc{Propose}(x, z, \ell, M, N)$
    Append C and their gate decisions to L_ℓ
    $V_\ell \leftarrow$ all gate-admissible candidates in L_ℓ
    $q_\ell \leftarrow \textsc{ListwiseBorda}(V_\ell, x, z, \ell)$
    Complete missing pair labels in G_ℓ over V_ℓ
    $O \leftarrow \textsc{Sort}(V_\ell;\, q_\ell \downarrow, \text{ID} \uparrow)$
    $P \leftarrow \textsc{Greedy}(O, G_\ell, \infty)$
    $Q \leftarrow$ first B candidates in P with incomplete Train verdicts and remaining attempts
    Evaluate Q on T and append verdicts to L_ℓ
  end for
end for
for each quality layer ℓ **do**
  $E \leftarrow \{c \in V_\ell : \text{Complete}(c, T) \wedge \text{FailCount}(c, T) \geq \tau\}$
  Evaluate E on D
  $E \leftarrow \{c \in E : \text{Complete}(c, D) \wedge e_D(c) > 0\}$
  $O \leftarrow \textsc{Sort}(E;\, e_D \downarrow, q_\ell \downarrow, \text{ID} \uparrow)$
  $S_\ell \leftarrow \textsc{Greedy}(O, G_\ell, K)$
end for
Freeze all selected sequences $\{S_\ell\}$
Evaluate $\bigcup_\ell S_\ell$ on held-out set H
Report each k using prefixes $S_\ell[:k]$ and the same recorded verdicts
return $\{S_\ell\}$ and Test verdicts
function $\textsc{Greedy}(O, G, K)$
  $S \leftarrow [\,]$
  for c in ordered sequence O **do**
    if $|S| = K$ **then**
      break
    end if
    if $\nexists s \in S : \text{Conflict}_G(c, s)$ **then**
      Append c to S
    end if
  end for
  return S
end function

Table 8: Finalization of the selected verification rubric across 200 tasks. Retained counts include both unchanged and revised VPs. Removal does not trigger replacement or re-ranking.

Quality layer	Search-selected	Removed	Final retained
Useful	876	4	872
Usable	962	3	959
Desirable	980	7	973
Total	2,818	14	2,804

D.5 Held-Out Failure Detection and Budget Analysis

We analyze whether the failure-detection behavior of the selected verification points persists on held-out implementations and how the retained-set size affects implementation-level coverage. For each

Table 9: Failure detection for the search-selected VP sequences at $k = 5$, before the subsequent quality audit. The 2,818 selected VPs are evaluated on the fixed Train, Validation, and Test splits. Failure rate is the fraction of VP–implementation pairs judged as fail, computed within each task and then macro-averaged across tasks. Unknown verdicts remain in the denominator and receive no failure credit. Confidence intervals are 95% percentile intervals from 10,000 task-level bootstrap resamples. Only Test is held out from both mining and selection. All values are percentages.

Split	Failure rate	95% CI	Unknown rate (%)
Train	83.17	[81.63, 84.72]	0.00
Validation	93.09	[91.81, 94.33]	0.00
Test	83.58	[81.24, 85.85]	0.48

of the 200 tasks, the ten reference implementations are partitioned into six Train implementations, two Validation implementations, and two Test implementations. Train outcomes guide candidate generation and refinement, Validation outcomes determine final eligibility and ranking, and Test outcomes remain inaccessible until the ordered VP sequences have been frozen. For the prefix analysis, each facet uses the first $\min(k, n)$ points from the same Validation-ranked sequence of length n, for $k \in \{1, \ldots, 5\}$; mining and selection are not rerun for different values of k, and Test outcomes never affect the selected points or their order. We report two task-macro metrics: the fraction of VP–implementation pairs judged as failures and the fraction of implementations with at least one failing VP across the three facets. This analysis uses the 2,818 VPs selected by search before the subsequent quality audit.

Failure detection persists on held-out implementations. At $k = 5$, the Test VP-level failure rate is 83.58%, compared with 83.17% on Train (Table 9). The similarity indicates that the selected VPs continue to expose failures in held-out implementations of the same tasks. Validation reaches 93.09%, which is higher than Test because Validation outcomes were used to determine eligibility and final ordering. This comparison measures transfer to held-out implementations of the same tasks within the reference pool.

Small prefixes provide broad implementation coverage. With at most one VP per facet, the selected prefixes flag at least one Judge-identified failure in 97.75% of Test implementations; this implementation hit rate reaches 100% with at most three VPs per facet (Figure 7). Across $k = 1$ to 5, the Test VP-level failure rate changes only modestly, from 85.38% to 83.58%. Implementation hit rate records whether any selected VP fails, while the additional VPs retain distinct requirement checks. The two metrics therefore characterize Judge-identified failure yield at the point and implementation levels.

Uncertainty in the mining ablation. For the comparison between the mined and specification-only rubrics in Table 5, 95% confidence intervals for the task-macro differences are computed from 10,000 paired task-level bootstrap samples. The intervals are $[16.14, 22.10]$ for Useful, $[14.24, 19.76]$ for Usable, and $[9.63, 14.27]$ for Desirable, all in percentage points. This comparison uses the final mined rubric, whereas the prefix analysis above uses the search-selected VPs before review.

D.6 Judge Validation

Reference set and input conditions. We use GLM-5.3-Flash with source code and GUI traces for benchmark scoring. Following the empirical validation of model-based judges Zheng et al. (2023), we compare ten judging configurations on a reference set of 967 VPs from 87 tasks. Binary metrics are computed on the 967 VPs with explicit pass/fail reference annotations (163 pass and 804 fail). The analysis retrospectively scores archived predictions against reference annotations revised after disagreement review, while the predictions were generated from the original VP requirements.

The source-only condition (S) supplies the public task context, application source code, and VP checks. The GUI-trace-augmented condition (E) additionally supplies archived textual GUI execution traces from the corresponding application. GLM denotes GLM-5.3-Flash, Hy3 denotes Hy3, and DeepSeek denotes DeepSeek-V4-Flash. For consensus, a check receives a binary verdict only when GLM and Hy3 agree on that verdict. Arbitration sends disputed or unresolved checks to DeepSeek

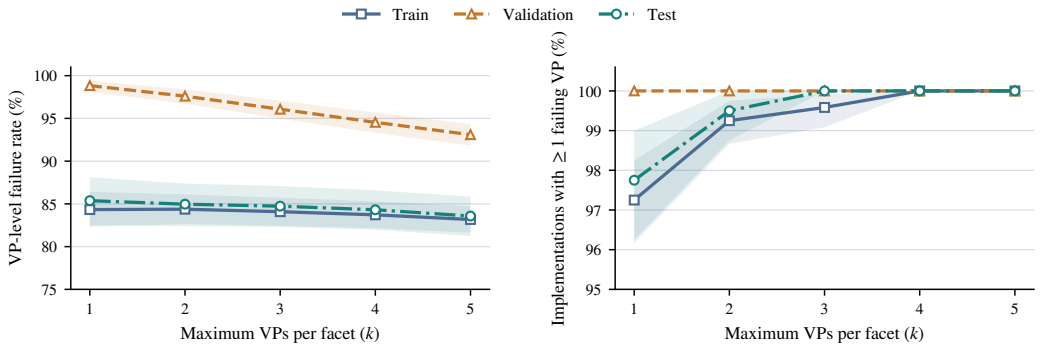


Figure 7: Sensitivity to the number of retained VPs. For each facet, we take the first $\min(k, n)$ points from its Validation-ranked post-search sequence of length n, giving at most $3k$ VPs per task, without rerunning mining or using Test outcomes for reselection. Left: task-macro VP-level failure rate. Right: task-macro fraction of implementations with at least one failing VP across the three facets. Shaded regions show pointwise 95% confidence intervals from 10,000 task-level bootstrap resamples. Unknown verdicts receive no failure credit, and both vertical axes are zoomed.

Table 10: Judge agreement with 967 explicit pass/fail reference labels. S uses source code, task context, and VP checks; E additionally includes textual GUI execution traces. Sel. BA and Sel. Acc. are computed on answered VPs, while Coverage uses all 967 reference-labeled VPs. Consensus combines GLM and Hy3; arbitration uses DeepSeek for disputed or unresolved checks. Results are retrospective rescoring of archived predictions. All values are percentages.

Configuration	Sel. BA	Sel. Acc.	Coverage
Source-only (S)			
GLM	89.39	87.93	94.21
Hy3	84.24	89.23	92.14
GLM–Hy3 consensus	92.41	92.74	81.18
+ selective arbitration	89.31	89.68	98.24
+ forced-binary arbitration	88.88	88.83	100.00
GUI-trace-augmented (E)			
GLM (benchmark judge)	90.44	88.80	92.35
Hy3	85.43	89.69	95.24
GLM–Hy3 consensus	**94.69**	**94.88**	78.80
+ selective arbitration	89.95	91.46	98.04
+ forced-binary arbitration	88.77	89.87	**100.00**

together with the original inputs and anonymized judgments. The selective variant permits unknown, whereas the forced-binary variant requires pass or fail. Check verdicts are aggregated into a VP verdict: any failing check makes the VP fail; all checks must pass for the VP to pass; otherwise, the VP remains unresolved.

Metrics. Coverage is the fraction of the 967 reference-labeled VPs for which a configuration returns pass or fail. Selective accuracy (Sel. Acc.) is the fraction of correct verdicts among those answered VPs. Selective balanced accuracy (Sel. BA) is the mean of pass-class and fail-class recall within that answered subset. Unknown or missing predictions reduce coverage. Accuracy and balanced accuracy are therefore interpreted jointly with coverage.

Agreement and coverage. The benchmark judge, GLM-5.3-Flash with GUI traces, achieves 90.44% selective balanced accuracy and 88.80% selective accuracy at 92.35% coverage. Evidence-augmented consensus achieves higher selective accuracy (94.88%) with lower coverage (78.80%). Selective arbitration raises coverage to 98.04%, with 91.46% selective accuracy. These comparisons

characterize the trade-off between agreement and coverage; the benchmark scores use the single-model GLM configuration.

Judge instructions and evidence requirements. The Judge evaluates each supplied check against its stated setup, action, and expected outcome. A pass requires supporting evidence for the expected outcome; a fail requires a concrete contradiction or an absent required behavior; insufficient evidence yields unknown. Code reasoning follows the relevant event handlers, conditions, data updates, and rendered states. Execution evidence is matched to the scenario, participant, and action under evaluation. Each check is returned exactly once with its identifier, status, applicability, reason, and source or execution references. The following translated summary states the central decision rules of the prompt:

> Evaluate only the supplied verification checks. Do not add requirements or prescribe an implementation. Trace the stated scenario through the available code and, when supplied, execution evidence. Return pass when the expected outcome is supported, fail when a required behavior is contradicted or absent, and unknown when the evidence cannot establish a verdict. Cite the evidence supporting each pass or fail. Judge each check separately; the evaluation pipeline aggregates check verdicts into the VP result.

E Additional Results and Intent Alignment

E.1 Generation Configurations

Models and reasoning configurations. The evaluated models comprise GPT-5.6 Sol (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026b) and Claude Sonnet 5 (Anthropic, 2026c), Gemini 3.1 Pro Preview (Google, 2026) and Gemini 3.8 Flash (Google DeepMind, 2026), DeepSeek-V4-Pro and DeepSeek-V4-Flash (DeepSeek, 2026), Kimi K3 (Kimi Team, 2026), GLM-5.3 (Z.ai, 2026), and Hy3 (Tencent, 2026a) and Hy4-preview (Tencent, 2026b). GPT-5.6 Sol and both Gemini models use `high` reasoning. Both Claude models use adaptive reasoning with `high` effort. Both DeepSeek models, Hy3, and Hy4-preview use reasoning enabled with `high` effort. Kimi K3 and GLM-5.3 use always-on reasoning with `max` effort. These settings follow each model's configuration interface and do not imply identical reasoning budgets across models.

Harness execution. Codex and Claude Code operate in isolated Docker workspaces, where the paired model can inspect, create, and modify project files. Both harnesses use `high` reasoning effort and a 30-minute generation timeout per round. As harness-assisted generation is more resource-intensive, the comparisons with direct generation use a randomly selected 50-task subset. Both conditions use the same tasks and feedback protocol. The task IDs will be released with the benchmark. Appendix E.3 reports the per-round results.

E.2 Per-Round Fixed-Rubric Results

Figure 9 compares Product scores across three generation rounds, while Table 11 reports the corresponding Useful, Usable, and Desirable scores for all 11 models. All rounds use the same frozen rubric, which retains at most five VPs per quality dimension; Top-5 therefore denotes the complete released rubric. Models in the figure are ordered by Round-3 Product score, whereas the table preserves the model-family grouping used in the main results.

Product scores increase across both revision steps for every model. Claude Opus 5 achieves the highest final score (56.97) and the largest R1-to-R3 gain (+25.92 percentage points), followed in final score by GLM-5.3 (55.53) and Kimi K3 (54.05). Initial performance and final performance do not yield identical rankings. GLM-5.3 leads in R1, while Claude Opus 5 leads in R3. Nevertheless, every model remains below 57 Product after two feedback-based revisions, indicating substantial remaining room for improvement under the fixed rubric. Figure 8 illustrates the four models with the largest proportional decline in gains from the first revision to the second.

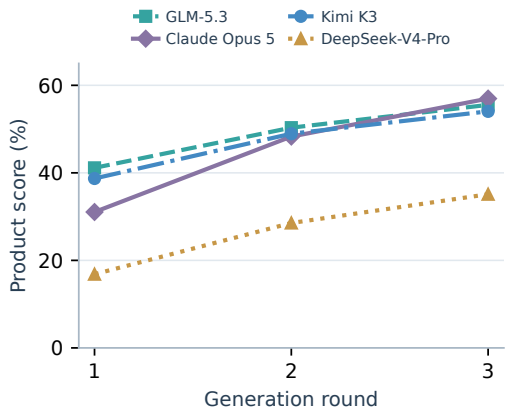


Figure 8: Product across three rounds for the four models with the largest proportional decline in revision gains. Table 11 reports all 11 models.

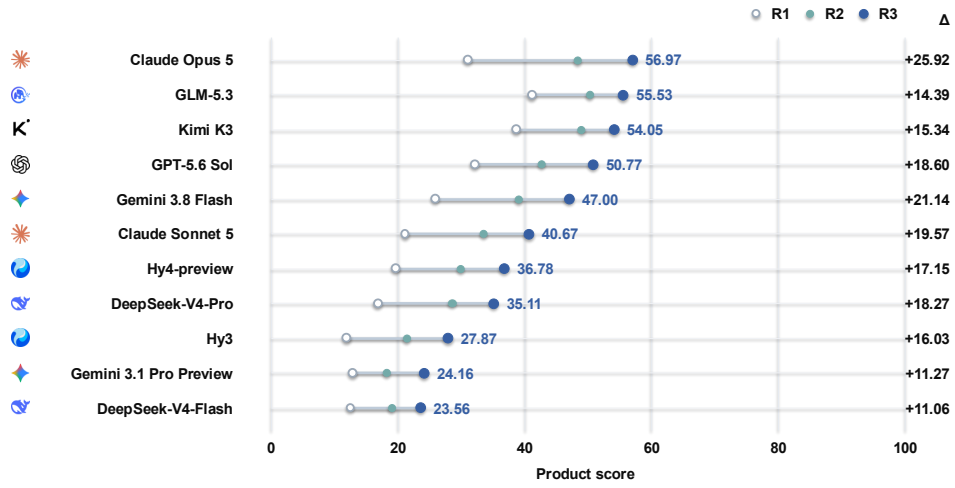


Figure 9: Product scores across three generation rounds. R1 denotes initial generation; R2 and R3 incorporate feedback-based revisions. Models are ordered by R3 score. Labels beside the final markers report R3 scores, and the rightmost column reports $\Delta = \text{R3} - \text{R1}$ in percentage points. Scores use the full frozen rubric and task macro-averaging, with the same scoring and exclusion rules as Table 11.

E.3 Full Results for Agentic Harnesses

Table 12 expands the Product comparison in Table 3 with Useful, Usable, Desirable, and compilation pass rates for the same matched comparisons across all three rounds.

E.4 Compilation Reliability and Static Code Characteristics

We additionally analyze compilation reliability and the static structure of the generated code. Table 13 reports the percentage of tasks with a compilable application in the initial generation and in the final retained workspace after at most two feedback-based revisions. An application that was not delivered or did not pass the compilation gate is counted as a failure. For trajectories that terminated early following acceptance, the last accepted workspace is retained as the final workspace.

Compilation reliability does not by itself imply product completeness. For example, several models reach a final compilation rate above 99%, while their final frozen-rubric scores remain substantially

Table 11: Per-model results for 11 models across three generation rounds using the full Top-5 frozen rubric, point-level scoring, and task macro-averaging. R1 denotes initial generation; R2 and R3 incorporate feedback-based revisions. Compilation failures score zero for Product and applicable layers. Scoring and aggregation follow Section 4.2. Bold denotes the highest score in each column. All scores are percentages.

Model	Product			Useful			Usable			Desirable		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
GPT-5.6 Sol	32.17	42.68	50.77	34.46	49.09	57.61	35.04	46.82	56.14	25.70	31.11	38.89
Claude Opus 5	31.05	48.29	**56.97**	34.50	52.88	**64.73**	35.20	52.78	**60.97**	23.37	38.66	**45.58**
Claude Sonnet 5	21.10	33.53	40.67	23.03	36.94	47.30	22.83	35.65	44.89	16.50	26.42	30.05
Gemini 3.1 Pro Preview	12.89	18.22	24.16	14.62	20.33	28.92	14.00	20.41	25.99	9.53	12.88	17.36
Gemini 3.8 Flash	25.86	39.02	47.00	27.98	43.59	54.61	27.03	39.69	48.50	21.80	32.60	37.68
DeepSeek-V4-Pro	16.84	28.54	35.11	19.72	33.49	42.03	18.89	32.62	38.08	11.53	19.32	25.14
DeepSeek-V4-Flash	12.50	19.02	23.56	13.71	21.38	27.94	14.51	20.45	24.91	9.22	14.61	17.43
Kimi K3	38.71	48.94	54.05	40.52	53.27	58.95	43.18	**54.76**	59.34	**32.32**	38.67	43.38
GLM-5.3	**41.14**	**50.32**	55.53	**44.33**	**54.12**	62.52	**46.56**	54.54	59.18	32.23	**41.12**	45.08
Hy3	11.84	21.37	27.87	14.35	27.99	33.66	13.49	22.70	30.65	7.12	13.47	19.30
Hy4-preview	19.63	29.83	36.78	24.29	37.50	45.89	20.14	30.05	37.98	14.88	22.26	27.30

Table 12: Application generation with and without agentic harnesses on the 50-task proxy set. Scores are macro-averaged over matched evaluable tasks within each model pair and round, with compilation failures scored as zero. All values are percentages.

Round	Direct generation (%)					Agentic generation (%)				
	Product	Useful	Usable	Desirable	Compile	**Product**	Useful	Usable	Desirable	Compile
GPT-5.6 Sol	Harness: Codex									
1	30.93	33.90	36.10	19.96	**100.00**	**33.66**	**36.38**	**40.74**	**23.19**	85.11
2	41.58	44.83	48.54	28.33	100.00	**46.40**	**53.51**	**52.12**	**32.15**	100.00
3	45.11	50.43	51.38	31.88	100.00	**47.75**	**50.87**	**51.52**	**38.84**	100.00
Claude Sonnet 5	Harness: Claude Code									
1	22.01	22.40	23.72	**17.29**	**95.35**	**27.02**	**37.44**	**27.91**	16.28	88.37
2	33.28	34.28	33.18	27.99	100.00	**40.40**	**48.45**	**43.56**	**28.64**	100.00
3	42.09	39.23	41.92	35.38	100.00	**51.79**	**60.00**	**52.31**	**38.85**	100.00

Bold values indicate the higher score for each metric within the same model and round. Ties are not highlighted. Compile denotes the compilation pass rate on the same matched subset.

different (Table 11). Conversely, the large compilation improvements of Claude Opus 5, Gemini 3.8 Flash, and DeepSeek-V4-Flash show that feedback-based revision frequently repairs invalid initial implementations, although it does not necessarily close the remaining product-quality gap.

Table 14 characterizes the final compilable workspaces using deterministic static source-code statistics. Project-level line and file counts are summarized by the median across compilable tasks. These measurements describe implementation structure and should not be interpreted as direct measures of functional correctness or maintainability.

We relate these static characteristics to the Round-3 Product scores in Table 1 using Pearson correlations across the 11 model-level aggregates reported in the tables. Product is computed over all evaluated tasks, with compilation failures scored as zero, while each static characteristic is summarized over the final compilable workspaces for that model. The resulting correlations are descriptive and do not establish that a static property causes higher Product performance.

Table 13: Task-level compilation pass rates. Initial denotes Round 1; final denotes the retained workspace after at most two feedback-based revisions, keeping the accepted version for trajectories that terminate early. Undelivered or non-compiling applications count as failures.

Model	Initial (%)	Final (%)	Δ (pp)
GPT-5.6 Sol	97.0	99.0	+2.0
Claude Opus 5	79.0	99.5	**+20.5**
Claude Sonnet 5	96.5	**100.0**	+3.5
Gemini 3.1 Pro Preview	87.5	99.5	+12.0
Gemini 3.8 Flash	84	97.5	+13.5
DeepSeek-V4-Pro	93.5	99.0	+5.5
DeepSeek-V4-Flash	80	93.5	+13.5
Kimi K3	94.5	98.5	+4.0
GLM-5.3	**98.5**	99.5	+1.0
Hy3	**98.5**	99.5	+1.0
Hy4-preview	**98.5**	**100.0**	+1.5

Table 14: Static characteristics of the final compilable applications. LoC denotes non-empty lines of code. Project-level counts and comment ratios are reported as medians. Modular and componentized columns report percentages of compilable projects satisfying the corresponding criterion.

	JavaScript structure			Project organization			Interface source	
Model	JS LoC	JS files	Largest JS file	Modular projects	Componentized projects	Comment ratio	WXML LoC	WXSS LoC
GPT-5.6 Sol	1,034	4	504	62.6%	1.5%	0.0%	379	876
Claude Opus 5	1,642	9	456	98.0%	1.5%	0.5%	392	661
Claude Sonnet 5	596	7	200	83.5%	0.5%	0.3%	195	332
Gemini 3.1 Pro Preview	207	3	118	22.6%	0.0%	1.0%	74	178
Gemini 3.8 Flash	1,381	7	420	93.8%	2.1%	3.4%	746	1,484
DeepSeek-V4-Pro	398	5	179	60.6%	0.0%	0.0%	112	234
DeepSeek-V4-Flash	371	5	161	43.3%	0.0%	2.0%	157	323
Kimi K3	1,136	9	322	95.4%	2.5%	1.6%	306	592
GLM-5.3	1,594	9	444	97.0%	4.5%	1.5%	378	764
Hy3	213	4	107	29.6%	0.0%	0.0%	64	134
Hy4-preview	348.5	5	160	63.0%	0.0%	0.0%	81	142

Notes. A project is modular when it uses at least one explicit module construct: `import`, `require`, `export`, or `module.exports`. A project is componentized when it contains at least one custom-component directory. The comment ratio is the fraction of non-empty JavaScript lines identified as comment lines. Static statistics are computed only over the final workspaces that pass the compilation gate.

Table 15: Pearson correlations between Round-3 Product and static code characteristics across all 11 evaluated models, computed from the reported model-level aggregates.

Static characteristic	Pearson r
JavaScript LoC	0.92
JavaScript files	0.79
Largest JavaScript file	0.89
Modular projects	0.90
Componentized projects	0.80
Comment ratio	0.13
WXML LoC	0.68
WXSS LoC	0.65

E.5 User-Intent Grounding and Alignment through Revision

We examine whether revisions bring generated applications closer to the requirements underlying user requests. The analysis connects delivered feedback to its target requirement and then to the frozen verification points (VPs) associated with that requirement. This enables us to distinguish improvement on the requested behavior, completion of its associated checks, and preservation of previously satisfied behavior.

E.5.1 Grounding User Intent in Annotated Requirements

We operationalize user intent through the reviewed latent requirements described in Appendix A. Their provenance records identify supporting evidence, while their grounding annotations describe their relationship to the initial public request. Among 1,408 requirements, 217 are direct, 859 indirect, and 332 consensus-grounded. These categories describe the basis for a requirement; they are not labels of personal preference. A requirement initially left implicit may subsequently be made explicit through feedback.

The task materials associate all 2,804 frozen VPs with latent requirements, recording a primary requirement where one can be identified, additional related requirements, and a mapping rationale. There are 2,772 VPs with a primary requirement and 32 with multiple related requirements but no unique primary. Requirement-level analyses are restricted to requirements with at least one mapped VP. We verify the mapping source hashes and match all VP contents to the historical scoring rubric. The mappings support retrospective analysis without changing the evaluation criteria.

Analysis cohorts. The three-round profiles use 26,225 of the 30,844 possible model–VP trajectories (eleven models and 2,804 VPs), retaining only trajectories with pass/fail verdicts in all three rounds. Unknown verdicts, compilation-gate zeros, Judge execution failures, and missing results are excluded from these profiles. Each model's eligible subset is fixed across rounds; category rates pool eligible model–VP observations. Feedback-linked revision analyses instead use adjacent-round binary pairs with confirmed delivery of requirement-bound feedback, as specified below. The same judge configuration is used across all three generation rounds.

E.5.2 Feedback Content Across Generation Models

We use GLM-5.3 to analyze the specific requirements expressed in feedback. Table 16 summarizes the reported proportions and qualitative patterns for each generation model. Percentages describe the feedback received by each model, rather than failure rates over all tasks. The indicators include complaint content, requests for additional capabilities, and descriptions of concrete interaction paths, and do not form an exclusive partition. Where a numerical proportion is not available, we report the qualitative finding only.

E.5.3 Which Models Address Feedback Successfully?

We identify 2,593 requirement-bound feedback events whose text is confirmed in the next generation input; 2,205 have at least one mapped VP. A VP is target-linked if its primary or related requirements contain the feedback target. We measure targeted repair as the fraction of initially failed target-linked VPs judged passed after the next revision. Preservation is the fraction of all previously passed VPs that remain passed after that revision. Both rates use binary paired verdicts. The figure reports the two revision steps separately; pooled rates summarize each model across both steps.

Figure 10 separates these two aspects of revision. Pooling both steps, Claude Opus 5 has the highest observed targeted repair rate (57.1%), followed by Kimi K3 (50.1%) and GLM-5.3 (46.8%). Hy4-preview reaches 35.2%, compared with 29.5% for Hy3; their preservation rates are 89.4% and 81.3%, respectively. DeepSeek-V4-Flash has a targeted repair rate of 21.3% and preservation rate of 77.3%. These results characterize the observed feedback trajectories; models receive different feedback and begin with different residual failures, so the rates are not controlled comparisons on identical repair requests.

Complete satisfaction is stricter than repair of individual checks. For each feedback-targeted requirement with at least one initially failed associated VP, we also ask whether all associated VPs pass after revision. Among events with complete binary post-revision coverage, this occurs in 41/120 events

Table 16: Model-level feedback profiles from GLM-5.3 analysis. Each percentage uses the corresponding model's feedback messages as its denominator.

Model	Reported proportions	Characteristic feedback
GPT-5.6 Sol	Missing capabilities: 61.2%; functional failures: 11.2%.	Missing editing, filtering, or state-recovery support despite usable core functionality.
Claude Opus 5	Qualitative finding only.	Relatively frequent behavioral corrections and data-sharing complaints, particularly incorrect semantics or cross-user behavior.
Claude Sonnet 5	Functional failures: 28.9%.	More saving, database, and sharing problems than GPT-5.6 Sol and Claude Opus 5.
Gemini 3.1 Pro Preview	Functional failures: 41.9%; concrete interaction paths: 69.4%.	Highest reported functional-failure share, including blank screens, failed saves, and lists remaining empty.
Gemini 3.8 Flash	Direct functional failures: 10.5%.	More missing capabilities or behavior inconsistent with requirements despite an application that runs.
DeepSeek-V4-Pro	Explicit requests to add or provide capabilities: 48.8%.	The strongest tendency toward requests for additional capabilities.
DeepSeek-V4-Flash	Functional failures: 24.4%.	Frequent editing and repair expressions alongside functional failures.
Kimi K3	Concrete interaction paths: 59.4%.	Descriptions of completed actions followed by difficulty finding the next entry point.
GLM-5.3	Qualitative finding only.	Frequent requests for additional capabilities and relatively few direct runtime failures.
Hy3	Functional failures: 30.0%.	Empty lists, loading failures, missing data displays, or actions without an observable effect.
Hy4-preview	Missing capabilities: 54.1%; functional failures: 36.9%; semantic mismatches: 15.9%.	Requests for missing controls and views, failed loading or saving, and corrections to business rules and misleading application states.

for Claude Opus 5 (34.2%), 36/138 for GLM-5.3 (26.1%), 36/146 for Kimi K3 (24.7%), and 34/227 for Hy4-preview (15.0%). Across all eleven models, the rate is 340/1,940 (17.5%). Thus, successful local repairs frequently leave additional conditions of the targeted requirement unresolved.

E.5.4 Which Verification Points Remain Difficult?

We use the twelve authored VP categories to examine which behaviors improve through revision and which remain difficult. Figure 11 groups the categories into Useful, Usable, and Desirable, with three bars per category showing R1, R2, and R3 pass rates. All rates use the same model–VP observations with binary verdicts in all three rounds, irrespective of feedback targeting.

Progress concentrates on some behaviors more than others. Data integrity has the highest final pass rate, increasing from 35.8% through 50.1% to 57.0%. Domain correctness improves by 22.0 percentage points, from 24.5% to 46.5%, and core capability improves by 20.6 points, from 29.8% to 50.4%. Information hierarchy improves by 9.6 points to 29.8%, while context semantics reaches 30.1% and input guard reaches 38.4%. Among initially failed checks, 38.2% of data-integrity VPs pass by the final round, compared with 15.0% for information hierarchy and 17.6% for context semantics. Thus, improvements in functional and data behavior coexist with persistent gaps in information organization and task-specific expression. Figure 12 shows how these profiles vary across models.

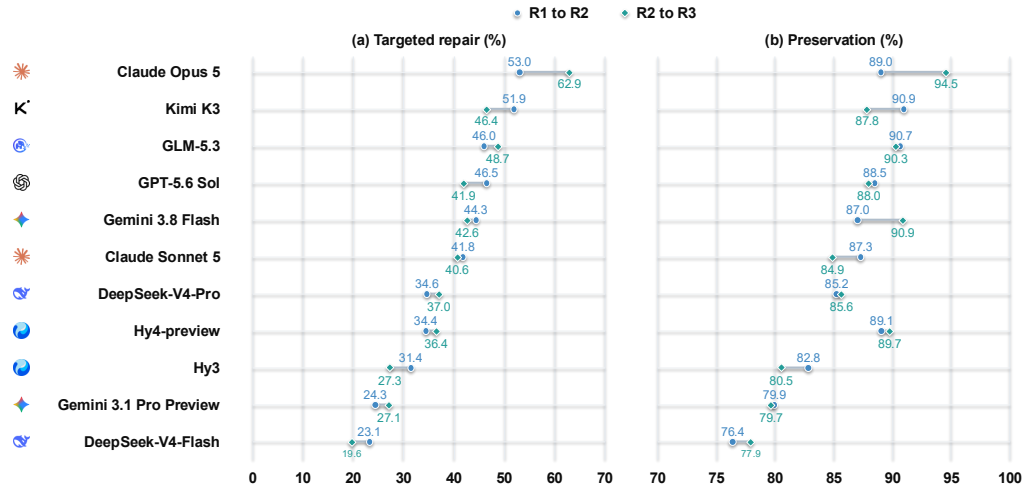


Figure 10: Model-level revision outcomes. Left: next-round fail-to-pass rate for feedback-linked VPs, shown separately for each revision step. Right: preservation of previously passed VPs, including target-linked and other checks. Unknown and unavailable verdicts are excluded. Both steps include only revisions with confirmed delivery of requirement-bound feedback.

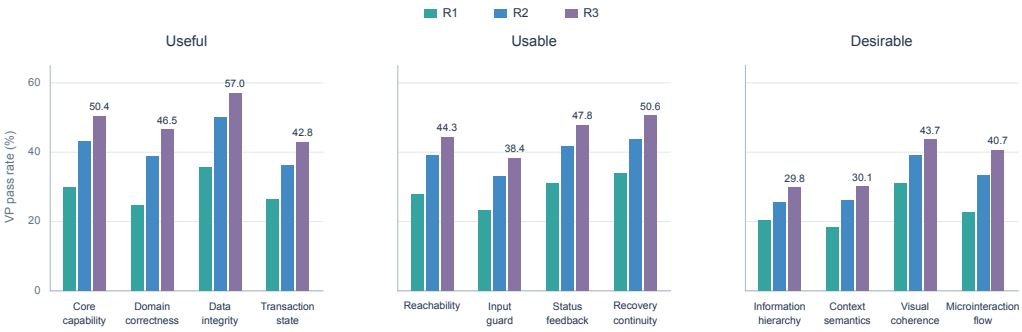


Figure 11: VP-category pass rates across eleven models, grouped by quality dimension. Within each category, bars show R1, R2, and R3; numeric labels report R3 rates. All panels share the same vertical scale. Rates pool the same model–VP observations with binary judgments in all three rounds and are point-weighted rather than task-macro averaged. Categories are authored metadata, not inferred failure causes. The analysis includes all eligible trajectories, irrespective of feedback targeting.

Category averages conceal substantial variation between checks. In task 051, the VP requiring creation time to survive subsequent edits passes in only one of eleven models initially but all eleven finally. The same transition occurs for the daily-hours aggregation display in task 055, even though information hierarchy has the lowest category-wide final pass rate. Conversely, all eleven models fail the checks for preserving existing tags during a batch append (task 027) and separating fixed from flexible expenses in their presentation (task 006) in both the initial and final rounds. These are archived VP judgments and illustrate specific strengths and remaining gaps without treating a whole category as uniformly easy or difficult.

VP difficulty recurs across models. To test whether difficulty is specific to individual checks, we hold out each of the eleven models and predict its final-round binary VP outcomes using only the other ten models. The VP predictor averages outcomes for the same task and VP; the task predictor averages all eligible VPs for that task; the global predictor averages all eligible training observations.

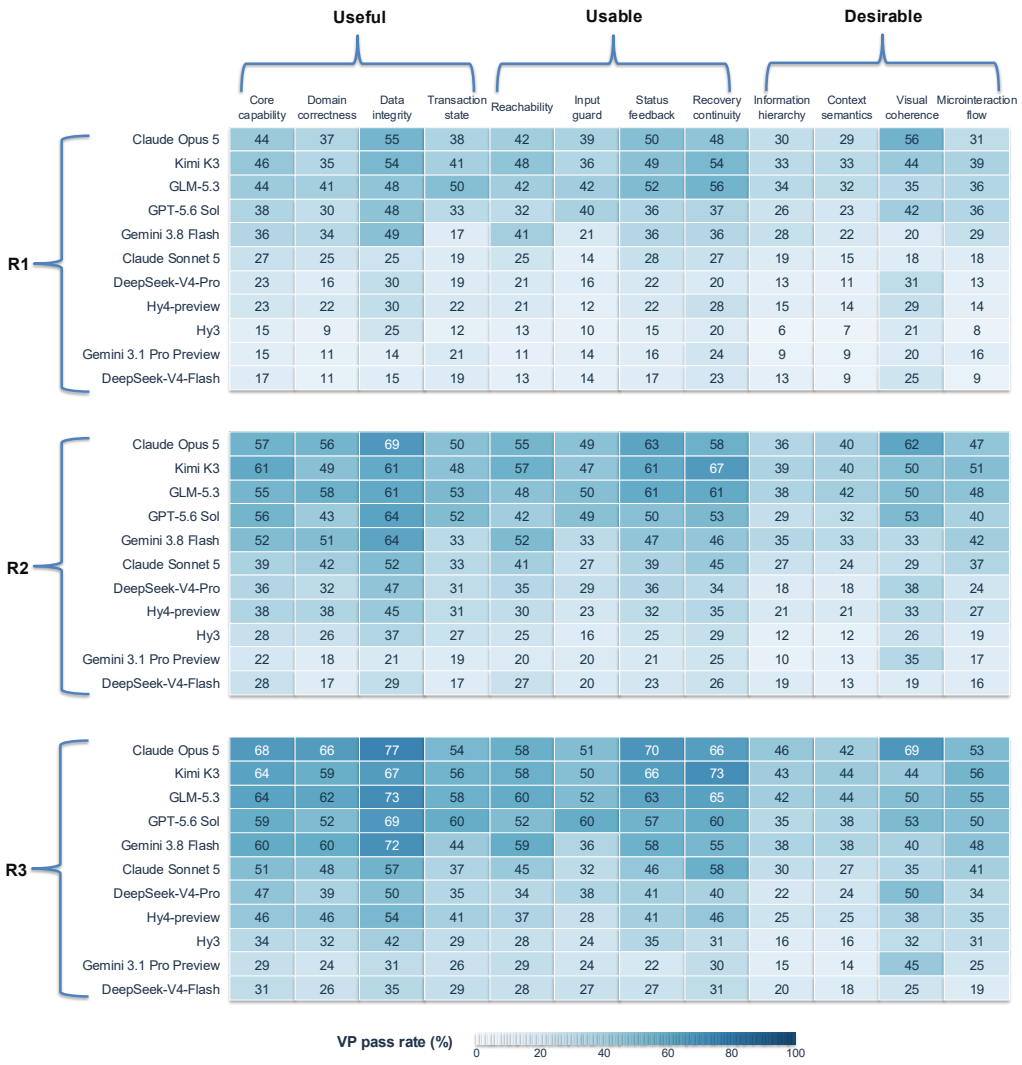


Figure 12: Three-round VP pass rates by model and authored category. All panels use the same complete-case observations and a common color scale. Cell denominators vary with category size and available verdicts. These are category profiles, not the task-macro Product metric.

Table 17: Leave-one-model-out prediction of final-round VP outcomes across eleven models. Predictions use the other ten models only. MAE and RMSE are pooled over the same 26,225 observations and reported on the 0–100 scale; lower is better.

Predictor	MAE	RMSE
Same task and VP	**31.07**	**41.72**
Same task	42.81	46.81
Global mean	48.94	49.63

All predictors are evaluated on the same 26,225 complete-case observations. As Table 17 shows, the VP predictor has lower error than the task and global predictors; its MAE is also lower than the task predictor for every held-out model. This indicates that individual VPs capture recurring behavioral difficulties beyond a task's overall difficulty.

Table 18: VP pass rates (%) across three rounds, grouped by the grounding category inherited from each VP's primary mapped requirement. Each model uses a fixed subset with binary verdicts in all three rounds; eligible subsets may differ across models. Rates are point-weighted rather than task-macro averaged.

Model	Direct			Indirect			Consensus-grounded		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
GPT-5.6 Sol	41.9	48.8	50.9	30.3	42.4	51.5	35.0	47.6	52.6
Claude Opus 5	44.2	53.9	54.7	38.0	50.9	60.0	39.9	52.1	54.4
Claude Sonnet 5	22.6	30.2	33.3	19.9	33.9	41.4	25.4	40.3	43.1
Gemini 3.1 Pro Preview	17.9	20.6	19.6	12.3	17.2	22.6	16.1	20.0	26.5
Gemini 3.8 Flash	32.0	36.8	37.1	27.6	41.3	50.6	35.5	49.1	54.1
DeepSeek-V4-Pro	19.7	25.6	29.1	16.6	28.2	35.6	19.3	33.7	39.2
DeepSeek-V4-Flash	18.8	22.7	25.8	12.9	20.6	26.3	14.5	19.2	23.3
Kimi K3	46.9	53.4	53.4	39.4	49.9	56.0	42.9	54.7	57.4
GLM-5.3	52.5	54.8	56.9	38.9	50.2	56.6	44.5	52.0	54.6
Hy3	15.8	20.9	19.1	10.3	21.2	28.4	12.6	23.2	29.8
Hy4-preview	23.5	26.9	30.0	17.6	29.2	39.2	21.8	32.7	35.5

E.5.5 VP Performance by Requirement Grounding

We further group VPs by the grounding category of their primary mapped requirement. This yields 1,711 indirect, 699 consensus-grounded, and 362 direct VPs; 32 VPs without a primary requirement are excluded from this breakdown. The inherited label characterizes the source requirement, rather than independently classifying the VP itself.

Table 18 reports three-round VP pass rates for each model, grouped by the grounding category inherited from the primary mapped requirement.

Convergence across revision rounds. Pooling eligible model–VP observations, indirect VPs improve from 23.9% to 34.9% and 42.5%, whereas consensus-grounded VPs improve from 27.9% to 38.6% and 42.8%. Their aggregate gap narrows from 4.0 to 0.2 percentage points. Nine of the eleven models show a larger first-to-final-round gain on indirect VPs. Thus, the initial gap in these observed cohorts largely closes over successive revisions. The initial ordering is sensitive to cohort selection, however, and does not establish that one grounding category is intrinsically harder.

Similar repair rates for feedback-linked failures. We additionally consider VPs linked to a requirement identified in feedback confirmed to reach the next generation input. Pooling the two revision steps, 2,042 of 5,101 initially failed indirect VP events pass after revision (40.0%), compared with 486 of 1,207 consensus-grounded events (40.3%). These denominators include only fail-to-pass or fail-to-fail pairs; a VP may contribute to both steps. The similar observed rates suggest that the initial performance gap does not translate into a comparable gap in feedback-linked repair. Nevertheless, approximately three fifths of these failed checks remain unresolved after the next revision. Models encounter different feedback and residual failures, so these rates are descriptive rather than controlled estimates of feedback effectiveness.

Explicit goals still leave behavioral details unresolved. VPs associated with direct requirements improve by only 6.7 percentage points, from 30.6% to 37.3%. Among initially failed direct VPs, 16.1% pass in the final round. This group includes behavioral details derived from an explicit high-level goal: for example, a requirement to support reservations is linked to a check that prevents booking on non-business days. That check fails in both the first and final rounds for all eleven models. The inherited direct label therefore does not imply that every checked detail was explicitly requested.

E.5.6 From Requested Changes to Implemented Behavior

Recovering a missing data representation. In the chip-settlement task, GPT-5.6 Sol initially represents chips using an aggregate quantity and a shared unit price. After feedback requests quantities by denomination, the revision introduces denomination-specific counts and weighted aggregation. The frozen VPs checking settlement totals and the use of edited counts change from fail to pass and remain passed in the final round. The edit implements the clarified data representation and its computational consequences, although other input-validation checks remain unresolved.

Adding a field without recovering its full semantics. In the consumption-tracking task, DeepSeek-V4-Flash adds a user-entered consumption date and later uses that field to calculate the next eligible date. However, it selects the most recently inserted record rather than the latest consumption event. Backfilling an older event can therefore move the calculated date backwards. The corresponding backfill VP remains failed. An offline probe of the archived implementation confirms this behavior. This case illustrates partial alignment: the requested concept appears in the implementation, while a related temporal invariant remains unsatisfied.

A revision can undo existing behavior. In the receipt task, GLM-5.3 adds price editing after targeted feedback. A subsequent revision under a generic improvement prompt retains the quantity-button bindings but removes their handlers; archived GUI attempts no longer change quantities. Source inspection confirms the missing handlers. This case illustrates loss of previously available behavior. It is a qualitative complement to the linked-feedback analysis, since the triggering prompt is generic rather than bound to a specific requirement.