

TeleGen: Improving LLM-Based Web Application Generation via Runtime Telemetry

Yujia Luo¹ Haonan Zhang² Jiasi Shen³ Zishuo Ding^{1,†} Weiyi Shang²

¹The Hong Kong University of Science and Technology (Guangzhou)

²University of Waterloo

³The Hong Kong University of Science and Technology

y1uo246@connect.hkust-gz.edu.cn haonan.zhang@uwaterloo.ca sjs@cse.ust.hk
zishuoding@hkust-gz.edu.cn wshang@uwaterloo.ca

Abstract

Large language models can generate runnable web applications from natural-language requirements, but many generated applications still fail interactive tasks. Existing generate-execute-repair pipelines execute the generated application and use task outcomes or error messages to guide code revision. However, this feedback often misses the runtime behavior between a browser action and the final task outcome, making interaction-level failures difficult to diagnose. Therefore, we propose **TeleGen**, an observability-enhanced framework for LLM-based web application generation. **TeleGen** instruments generated applications, collects runtime telemetry during task execution, and compresses raw telemetry logs into concise briefs for repair. We evaluate **TeleGen** on WebGen-Bench and Web-Bench. On WebGen-Bench, **TeleGen** improves task success from 67.7% with repair without telemetry to 76.2%, an increase of 8.5 percentage points. On Web-Bench, it improves cumulative Pass@2 from 21.7% to 29.8%. Ablation results show that runtime telemetry provides a useful diagnostic signal, while telemetry briefs make this signal more effective and less costly to use. Further analysis shows that telemetry is especially helpful for failures involving hidden execution paths, such as navigation, form workflows, and frontend-backend coordination.

1 Introduction

Large language models (LLMs) have made substantial progress in code generation (Chen et al., 2021; Austin et al., 2021; Jiang et al., 2026; Hui et al., 2024), moving from standalone programming problems (Hendrycks et al., 2021; Li et al., 2022b; Jain et al., 2025; Liu et al., 2023) to repository-level tasks (Liu et al., 2024b; Zhang et al., 2023; Jimenez et al., 2023; Wang et al., 2025). Within web code

generation, the task has similarly shifted from producing static webpages to building interactive web applications from requirements expressed in natural language, screenshots, or domain-specific languages (Lu et al., 2025a; Xiao et al., 2025; Li et al., 2026; Yang et al., 2026; Wan et al., 2025a; Dang et al., 2025). Static web page generation asks a model to translate a visual design or screenshot into HTML/CSS that reproduces the intended layout and appearance (Si et al., 2025; Wan et al., 2025b). Interactive web application generation goes further: the model must produce runnable website code from a high-level natural-language requirement, implementing user-facing workflows such as search, navigation, and form submission (Xiao et al., 2025). Correctness therefore depends not only on the rendered appearance or individual code fragments, but also on whether user interactions trigger the right frontend logic, state updates, and backend behavior during execution (Mesbah et al., 2012a).

To improve the quality of generated code, recent work adopts a generate-execute-repair workflow (Lu et al., 2025b). After generating an initial application, an execution step exercises its behavior by performing user-oriented tasks, either through browser agents that interact with the application in realistic web environments (Deng et al., 2023; Zhou et al., 2024; He et al., 2024; Koh et al., 2024) or through end-to-end testing frameworks that programmatically verify behavior against expected outcomes (Zhang et al., 2025; Teoh et al., 2026; Chevrot et al., 2025). This step exposes failures that the source code alone does not reveal, such as a button that does not trigger its handler (Artzi et al., 2011), or a form submission that does not update the interface (Mesbah et al., 2012b). When execution reveals a task failure, the repair agent (Pabba et al., 2025; Maddila et al., 2025; Xia et al., 2023) receives the feedback and revises the generated code.

[†]Corresponding author.

While effective, this feedback is often incomplete for interactive web applications: interaction-based execution can reveal that a generated application fails a task, but it can miss critical information that is not captured through the visible UI layer alone (Alimadadi et al., 2016). As a result, the feedback that reaches the repair agent (Fan et al., 2023; Bouzenia et al., 2025) is typically limited to task-level pass/fail outcomes or surface-level error messages. For example, after a browser agent clicks a button, the available feedback may not show if the click reached the intended event handler, sent a request, received the expected backend response, or the application state changed but was not rendered (Jr. et al., 2017). From the outside, these failures may look similar as the expected result was not reached. For repair, however, they point to different defects and require different code changes (Jr. et al., 2016).

This gap motivates our central question: can making generated applications more observable during execution provide better runtime evidence for repair? To answer it, we add an observability step before repair. Drawing on the use of observability to diagnose complex software systems (Li et al., 2022a, 2021), we instrument the generated application with lightweight logging mechanisms and record its runtime behavior while the browser agent executes each task. The resulting runtime telemetry (Barik et al., 2016) captures interaction events, runtime errors, state changes, and other task-relevant execution signals. Rather than passing the raw telemetry logs directly to the repair agent, we employ a separate agent to compress them into a telemetry brief. Raw logs often contain duplicated messages (Li et al., 2019, 2022c), framework noise, and other low-level details that can obscure the evidence relevant to a failed task (Locke et al., 2022). They are also lengthy, increasing the context size and cost of each repair attempt (Liu et al., 2024a). The telemetry brief is designed to preserve concise diagnostic evidence while avoiding the noise and cost of long execution logs.

We evaluate our approach on two web code-generation benchmarks, WebGen-Bench (Lu et al., 2025b) and Web-Bench (Xu et al., 2025), covering both full web application generation and sequential web-development tasks. Across both benchmarks, adding runtime observability to the generation pipeline improves the quality of the final generated applications compared with standard generation and repair without telemetry. On WebGen-

Bench, task success improves from 61.7% for the initially generated applications to 67.7% after repair without telemetry, and further to 76.2% when repair is guided by telemetry-derived evidence. At the task level, telemetry-guided generation fixes 106 tasks that the no-telemetry repair baseline fails, while losing 51 tasks solved by the baseline, giving a net gain of 55 tasks. On Web-Bench, our framework improves Pass@2 from 21.7% to 29.8%. Our further analysis shows that runtime evidence is especially useful for failures involving navigation, form and input workflows, authentication, search and filtering, and report or export behavior.

Our contributions can be summarized as follows:

- We identify an observability gap in interactive web applications pipelines: existing feedback often misses the runtime behavior behind task failures.
- We propose and evaluate an observability-enhanced framework that instruments generated applications, collects runtime logs, and compresses them into telemetry briefs for repair, improving final application quality on WebGen-Bench and Web-Bench.
- We show that telemetry is especially useful for failures involving hidden execution paths such as interaction handling, state updates, navigation, network behavior, and frontend-backend coordination.

Availability. The code, prompts, and experimental results are publicly available on [GitHub](#).

2 Related Work

LLM-based web application generation. Recent work has studied generating webpages and website codebases from screenshots, design inputs, interaction traces, and natural language requirements (Si et al., 2025; Yun et al., 2024; Xiao et al., 2025; Lu et al., 2025b,a; Li et al., 2026). These tasks move beyond standalone code generation because the output must render a plausible interface and support user behavior through routing, events, state updates, and backend logic. Si et al. (2025) and Yun et al. (2024) study front end code generation from visual inputs, while Xiao et al. (2025) shows that interactive behavior remains difficult to generate reliably. Lu et al. (2025b) is closest to our study because it evaluates whether generated web applications can complete functional UI tasks.

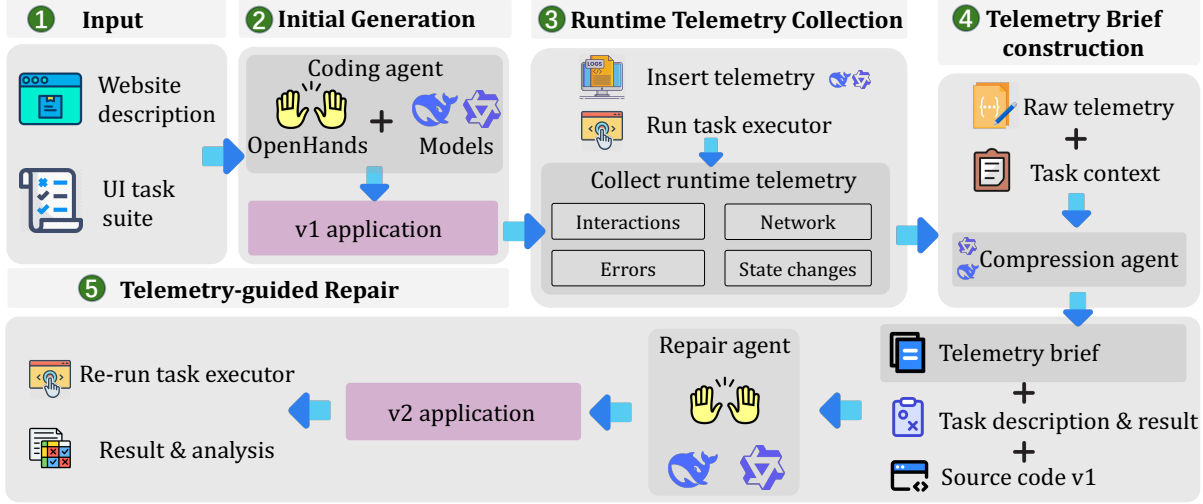


Figure 1: Overview of **TeleGen**. Given a website description and UI task suite, a coding agent first generates an initial web application, denoted as the *v1 application*. **TeleGen** then instruments a separate copy of the *v1* application and re-runs the task executor to collect runtime telemetry, including actions, network events, errors, and state changes. A compression agent combines the raw traces with the task context and produces a compact telemetry brief. Finally, the repair agent edits the clean *v1* source code using the telemetry brief and task feedback, producing a repaired *v2 application*, which is re-run by the task executor for final result analysis.

We build on this setting by studying the next step, repairing the generated application after browser evaluation exposes failed workflows.

Web agents and browser-based evaluation.

Web agents have been widely used to test whether language models can complete user-oriented tasks through browser interaction (Furuta et al., 2024; Deng et al., 2023; Zhou et al., 2024; He et al., 2024; Koh et al., 2024; Drouin et al., 2024; de Chezelles et al., 2025; Gou et al., 2025; Hu et al., 2025). These benchmarks evaluate behavior through execution rather than static inspection. In web application generation, browser agents can also act as testers by interacting with a generated application and checking whether the expected result is reached (Lu et al., 2025b). This task completion signal is useful, but it is often too coarse for repair. It can show that a workflow failed, but not whether the cause was an inactive event handler, a failed request, a missing route transition, or a state update that was not reflected in the interface. We study whether runtime evidence collected during browser execution can provide this missing diagnostic signal.

LLM-based program repair. Large language models have been studied for automated program repair, including repairing incorrect programs produced by LLMs, generating patches for standard repair benchmarks, and resolving repository-level

software issues (Fan et al., 2023; Xia et al., 2023; Jimenez et al., 2023). Recent repair systems further treat the model as an agent that can gather context, reason about defects, use tools, and validate candidate fixes (Bouzenia et al., 2025; Pabba et al., 2025). Generated web applications pose a related but different repair problem. The target is an interactive system verified through browser workflows, and the failure cause may only become visible during execution. Our work complements prior repair research by making the generated application observable during evaluation and by studying whether telemetry evidence improves repair beyond source code, task descriptions, and final outcomes.

3 Approach

To study how runtime telemetry can improve LLM-generated web applications, we propose **TeleGen**, an observability-enhanced framework for web application generation. Figure 1 illustrates the pipeline. Given a website description, a coding agent first generates an initial application, which we call the *v1 application*. A task executor then runs the corresponding task suite on this application and produces task feedback, including pass/fail outcomes. Then, **TeleGen** adds an observability step, where it creates an instrumented copy of the *v1 application* and runs the same tasks on this copy. During execution, the instrumented application records runtime telemetry. These raw telemetry logs are

then compressed into a telemetry brief. The repair agent receives the clean *v1* source code, task feedback, and, in the **TeleGen** setting, the telemetry brief. It then edits the clean source code to produce a revised *v2 application*. Finally, the task executor re-runs the same task suite on the *v2 application* to measure whether the final generated application better satisfies the intended workflows.

3.1 Initial Generation and Task Execution

TeleGen augments existing web code-generation pipelines rather than replacing their original generation or execution protocols. For each benchmark, we therefore follow the setup used in the corresponding original work (cf. Section 4). For example, in WebGen-Bench (Lu et al., 2025b), we use the OpenHands-based (Wang et al., 2025) generation setting to produce applications from natural-language website specifications, then execute the generated applications with the benchmark’s WebVoyager (He et al., 2024) UI-task protocol. WebBench (Xu et al., 2025) is handled analogously by following its original Web Agent and Playwright-based (Playwright, 2026) testing workflow.

In these benchmarks, the original execution step already provides simple feedback. This feedback may come from task outcomes, build failures (Vassallo et al., 2020), assertion mismatches (Xie and Memon, 2007), or test errors from automation frameworks (Zhang et al., 2025). **TeleGen** keeps this feedback information unchanged at this stage. This design lets us study whether observability improves existing generate-execute-repair pipelines without changing the generator, executor, or benchmark protocol.

3.2 Runtime Telemetry Collection

To collect runtime telemetry, **TeleGen** instruments a separate copy of the *v1* application while preserving the clean *v1* source for later repair. This separation keeps telemetry collection independent from the repair step and avoids confounding the repair results with possible side effects introduced during instrumentation.

Specifically, the instrumentation is performed by an LLM-based coding agent. Given the application source, website specification, and task suite, the agent inserts lightweight logging statements at code locations that it judges useful for diagnosing task failures. This design makes the logging repair-oriented: since the telemetry is ultimately consumed by a coding agent during repair, we ask

a coding agent to decide what execution signals are likely to be useful. The prompt explicitly instructs the agent to add telemetry only, without changing application behavior.

After instrumentation, the task executor runs the same tasks on the instrumented application. During this run, the inserted logging statements record runtime telemetry and capture how browser actions propagate through the application. This makes intermediate application behavior visible, instead of leaving the repair agent with only the original task feedback.

3.3 Telemetry Brief Construction

The raw telemetry logs collected during task execution are not passed directly to the repair agent. Although they contain complete runtime details, they also include duplicated messages or other types of noise. Long logs further increase context cost and can make the useful evidence harder to identify (Liu et al., 2024a).

TeleGen therefore uses a separate compression agent to convert raw telemetry logs into a telemetry brief. The agent receives the task context together with the collected logs, then preserves execution behavior that is likely to explain the failure while removing repeated or irrelevant entries. The brief is not intended to be a complete execution transcript. Instead, it gives the repair agent a compact, repair-oriented view of what happened during task execution. For example, it may show that a request succeeded but no visible update followed, or that data was loaded but no selection event occurred. Such evidence helps distinguish failures that appear similar from the final task outcome alone.

3.4 Telemetry-Guided Repair

After the telemetry brief is constructed, the repair agent receives the clean *v1* source code together with the task description, expected result, pass/fail outcome and telemetry brief. It edits the clean *v1* application rather than the instrumented copy, producing the *v2 application*. This separation ensures that instrumentation is used only for observation, while functional changes are made during repair. Guided by the telemetry brief, the repair agent can localize the likely cause of a failure before modifying the source code.

The *v2 application* is then executed with the same task suite and evaluation protocol as the original pipeline. Comparing this setting with standard repair measures whether runtime telemetry

improves the final generated application.

4 Experimental Settings

We evaluate whether runtime telemetry improves LLM-based web application generation. This section describes the benchmarks, agents, models, and metrics used in our experiments. The main comparisons follow the original baselines of each benchmark (cf. Section 3.1), while the component-level ablation is described in Section 5.2.

4.1 Benchmarks and Task Setting

We evaluate **TeleGen** on two web code-generation benchmarks with different task formats. WebGen-Bench evaluates full web application generation from natural-language requirements. The input is a website specification describing the intended application and its user-facing functionality, while the expected output is a runnable multi-file web application. The generated application is tested with UI tasks, where each task contains a user goal and an expected final result. WebGen-Bench contains 101 projects and 647 UI tasks, which are grouped into Content Presentation (174 tasks), User Interaction (313 tasks), and Data Management (160 tasks).

Web-Bench evaluates sequential web development tasks. Each project contains an initial codebase and a sequence of dependent tasks. At each step, the input is the current project state together with the next task description, and the expected output is a set of code edits that implement the requested feature. The result is checked with end-to-end tests. Web-Bench contains 50 projects, each consisting of 20 tasks. We use it to examine whether runtime telemetry remains useful when the benchmark already includes a second attempt with test-error feedback. The benchmark also provides three difficulty groups: Easy (277 tasks), Moderate (236 tasks), and Challenging (487 tasks).

4.2 Agents and Models

For each benchmark, we follow its original code-agent framework and execution setup. On WebGen-Bench, we use the OpenHands-based generation setting and execute UI tasks with WebVoyager. WebVoyager interacts with the generated application through the browser and judges whether each expected result is reached. On Web-Bench, we use its own Web-Agent framework, which generates file edits for each development task and relies on Playwright-based end-to-end tests to check results.

Our main experiments use DeepSeek-V4-Flash as the text/code model for generation, telemetry insertion, telemetry-brief construction and repair. Additional results with DeepSeek-V4-Pro, Qwen3.5-Plus and Gemini-3-Flash-Preview are reported in Appendix A. For WebGen-Bench, all WebVoyager model-driven steps use Qwen3.5-Plus.

4.3 Metrics and Analysis

For WebGen-Bench, our main metric is task success rate, defined as the fraction of UI tasks judged successful by the browser-agent evaluator. We report overall task success and category-level success. We also report token usage to measure the context cost introduced by runtime telemetry and telemetry-brief construction.

For Web-Bench, we follow the original benchmark and report Pass@1 together with cumulative Pass@2. Pass@1 measures the fraction of tasks that pass all end-to-end tests on the first attempt. Pass@2 measures the fraction of tasks passed after the retry step, where failed tasks receive error context and the agent is given a second attempt. Because **TeleGen** only changes the second attempt in Web-Bench, Pass@1 remains unchanged between the original and telemetry-augmented settings.

For paired task-level comparisons, we use two-sided exact McNemar tests on the discordant outcomes between settings.

We further analyze tasks solved by **TeleGen** but not by the no-telemetry baseline to identify where runtime telemetry is most useful. These cases are grouped by failure type to study which kinds of web-application failures benefit most from runtime telemetry.

5 Results and Analysis

5.1 Effectiveness of TeleGen

Overall, **TeleGen** improves the final generated applications on both benchmarks, showing that runtime telemetry is useful across both full web application generation and sequential web-development tasks. Table 1 reports the main results under the DeepSeek-V4-Flash configuration. On WebGen-Bench, the initially generated applications pass 399 out of 647 tasks, corresponding to 61.7% task success. Standard one-round repair without telemetry increases this to 438 tasks, or 67.7%. With runtime telemetry, **TeleGen** further improves task success to 493 tasks, or 76.2%, yielding an 8.5 percentage-point gain over the matched No-

Benchmark	Setting	Passed / Total	Overall	Δ	Categories		
					Content	User	Data
WebGen-Bench	Initial generation	399 / 647	61.7	–	71.3	55.0	64.4
	No-Telemetry repair	438 / 647	67.7	+6.0	78.7	60.4	70.0
	TeleGen	493 / 647	76.2	+14.5	88.5	68.1	78.8
					Easy	Moderate	Challenging
WebBench	Pass@1	142 / 1000	14.2	–	44.0	7.2	0.6
	Original Pass@2	217 / 1000	21.7	+7.5	59.6	16.9	2.5
	TeleGen Pass@2	298 / 1000	29.8	+15.6	72.9	32.2	4.1

Table 1: For WebGen-Bench, Overall, Content, User and Data report task-level success rates for the full benchmark and its three project categories. For Web-Bench, Pass1 is the first-attempt success rate, while Original Pass2 and **TeleGen** Pass2 report cumulative success after the original and telemetry-augmented second attempts. Easy, Moderate and Challenging report category-level cumulative success rate. For WebGen-Bench, Δ denotes the overall improvement of **TeleGen** over Initial generation; for Web-Bench, Δ denotes the overall improvement over Pass1. All values are percentages except passed-task counts.

Telemetry repair baseline. The paired improvement over No-Telemetry repair is statistically significant under a two-sided exact McNemar test ($p = 1.35 \times 10^{-5}$).

The improvement is consistent across all three WebGen-Bench project categories. Compared with No-Telemetry repair, **TeleGen** improves success by 9.8 points on Content Presentation (78.7% to 88.5%), 7.7 points on User Interaction (60.4% to 68.1%), and 8.8 points on Data Management (70.0% to 78.8%). These gains suggest that runtime telemetry provides useful repair evidence across different kinds of generated web applications rather than only one project type.

Because WebGen-Bench relies on WebVoyager for outcome judgment, we additionally conduct an outcome-blinded human audit of 60 sampled tasks. Human judgments agree with WebVoyager on 54 of 60 cases (90.0%), with Cohen’s $\kappa = 0.80$. The full audit protocol is reported in Appendix B.2.

Web-Bench provides a complementary setting based on sequential web-development tasks. The first-attempt Pass@1 is 14.2%, with 142 out of 1000 tasks passing before retry. The original Web-Bench retry pipeline improves cumulative Pass@2 to 21.7%. When the second attempt is augmented with runtime telemetry, **TeleGen** further improves cumulative Pass@2 to 29.8%. This corresponds to 81 additional passed tasks over the original retry pipeline and a 15.6% gain over the first attempt. The paired improvement over Original Pass@2 is also statistically significant under a two-sided exact McNemar test ($p = 4.49 \times 10^{-10}$).

The category-level results show that the improvement is not limited to easier tasks. On Easy tasks,

TeleGen improves cumulative Pass@2 from 59.6% to 72.9%, suggesting that telemetry helps the agent recover failures even when the underlying task is relatively accessible. On Moderate tasks, the gain is larger in absolute terms, rising from 16.9% to 32.2%, which indicates that runtime telemetry is particularly useful when the model needs more diagnostic information to decide how to repair the code. Challenging tasks remain difficult overall, but **TeleGen** still improves performance from 2.5% to 4.1%. These results suggest that telemetry strengthens the original Web-Bench retry pipeline by providing additional execution context beyond standard test-error feedback, with the clearest benefit on tasks where failures are recoverable but not obvious from error messages alone.

5.2 Ablation Studies

We conduct a step-by-step ablation of **TeleGen** to analyze how runtime telemetry and telemetry-brief construction contribute to performance. We perform the ablation on WebGen-Bench, where all variants start from the same generated v1 applications and use the same one-round repair protocol. Table 2 reports the results.

The first variant is no-telemetry repair, where the repair agent receives the standard repair inputs, including the clean v1 source code, task description, expected result and pass/fail outcome. The second variant is no-brief repair, where the repair agent additionally receives raw telemetry logs collected from the instrumented v1 application. The final variant is **TeleGen**, where the raw telemetry logs are compressed into telemetry briefs before repair.

Compared with no-telemetry repair, adding raw

Setting	Passed	Success	Δ
No-telemetry	438	67.70%	–
No-brief	484	74.81%	+7.11 pp
TeleGen	493	76.20%	+8.50 pp

Table 2: Repair effectiveness under DeepSeek-V4-Flash. Δ is measured against no-telemetry repair.

telemetry logs increases the number of passed tasks from 438 to 484, improving task success from 67.70% to 74.81%. This 7.11-point gain shows that the runtime signals collected by **TeleGen** contain diagnostic information that is not available from the standard repair inputs alone. In other words, the repair agent benefits from observing what happened during execution, not only from knowing the final task outcome.

Compressing raw logs into telemetry briefs further improves success to 493 passed tasks, or 76.20%. The gain over raw logs is smaller than the gain from adding telemetry itself, but it is still meaningful because both settings use the same underlying runtime information. The difference is how that information is presented to the repair agent. This suggests that telemetry is most useful when it is organized into a concise, task-centered form rather than passed as long raw logs.

Table 3 further shows that telemetry-brief construction improves the effectiveness-cost tradeoff. Compared with no-brief repair, **TeleGen** reduces average total token cost from 1481.7K to 1458.1K tokens per project while improving task success rate. Across the full run, this saves 2.39M tokens, or 1.6%. Overall, the ablation shows that runtime telemetry is the primary source of improvement, while telemetry briefs help the repair agent use this information more effectively and with lower context cost.

Beyond token usage, the additional monetary overhead of runtime telemetry is small under the model pricing used in our experiments. TeleGen costs approximately \$0.41 per project, compared with \$0.28 for No-Telemetry repair, while improving task success from 67.7% to 76.2%. Detailed cost accounting and runtime measurements are reported in Appendix B.4.

5.3 Failures Helped by Runtime Telemetry

To understand where runtime telemetry helps, we analyze the 106 tasks solved by **TeleGen** but not by no-telemetry repair. These tasks provide a di-

Setting	Input	Output	Total
No-telemetry	971.8K	19.0K	990.8K
No-brief	1462.1K	19.6K	1481.7K
TeleGen	1439.6K	18.4K	1458.1K

Table 3: Average token cost per project under DeepSeek-V4-Flash. Values are reported in thousands of tokens.

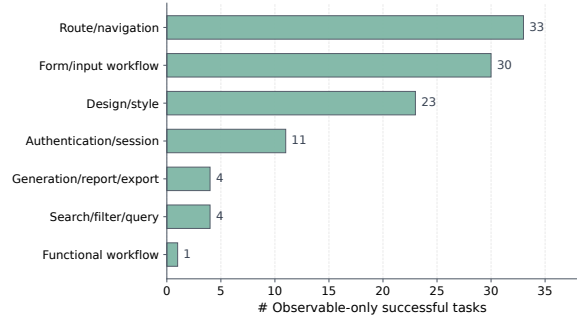


Figure 2: Failure types among tasks solved only by Telemetry-Guided Repair under DeepSeek-V4-Flash. These tasks fail under No-telemetry repair but succeed when telemetry brief is provided.

rect view of the added value of telemetry, since both settings start from the same generated applications and use the same repair protocol, while only **TeleGen** receives telemetry-derived execution information.

Figure 2 shows the distribution of these **TeleGen**-only successes. Route/navigation failures form the largest group, with 33 tasks. These failures are difficult to diagnose from final outcomes because the repair agent may only know that the expected page or view did not appear. Runtime telemetry can reveal whether the click reached the intended handler, whether the route update was attempted, or whether the application remained on the same page after interaction. This helps the repair agent distinguish missing navigation links from broken routing logic or state-dependent redirects.

Form/input workflow failures are the second largest group, with 30 tasks. These failures often involve several intermediate steps between user input and visible confirmation. A form may render correctly, but the submitted values may not be captured, validation may reject the input, or the submit handler may fail to update the interface. Runtime telemetry exposes where the workflow stops, making it easier to decide whether the fix should target input binding, validation rules, handler logic or rendering after submission.

Design/style failures account for 23 tasks. Al-

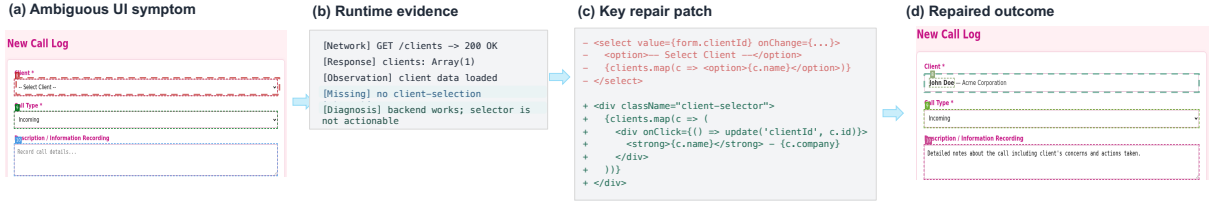


Figure 3: Representative form/input workflow repair enabled by runtime evidence. Although the form appears valid, runtime telemetry shows that client data is loaded but no client-selection interaction occurs. Telemetry-Guided Repair replaces the hard-to-operate dropdown with explicit clickable client cards, allowing the browser agent to complete the workflow.

though these failures appear visual, many are tied to dynamic UI behavior rather than static appearance alone. For example, a required element may be missing only after filtering, navigation or state changes. Telemetry helps determine whether the application produced the expected state internally but failed to render it, or whether the relevant state was never reached.

Authentication/session failures account for 11 tasks. These failures often depend on coordination between frontend state and backend responses. Runtime telemetry can reveal whether login actions sent the expected request, whether session state changed after the response, or whether protected content failed to render after authentication. This is especially useful because the final UI may simply show that the user remains unauthenticated, without explaining where the authentication flow broke.

The remaining **TeleGen**-only successes involve generation/report/export, search/filter/query and general functional workflows. These categories are smaller, but they follow the same pattern: telemetry helps when the failure occurs between the browser action and the visible result. For export tasks, it can show whether the generation action ran but no artifact became available. For search and filtering, it can reveal whether the query state changed, the filtering logic executed, or the result list failed to update.

Overall, the failure-type analysis shows that runtime telemetry is most useful when task failure hides multiple possible causes. By exposing intermediate execution behavior, telemetry helps the repair agent distinguish failures that look similar from the outside but require different code changes.

Representative case study. Figure 3 shows a representative form/input workflow failure from the CRM call logging application. The task requires the browser agent to create a new call log by select-

ing a client and filling in the required fields. The visible UI alone is ambiguous: the form is rendered, the client selector appears available, and the static code contains a standard `<select>` control with an `onChange` handler populated from the client API. From task-level feedback alone, the repair agent only knows that the task is not completed; it cannot determine whether the failure comes from data loading, navigation, form submission, state update, or the client-selection control.

Runtime evidence exposes the missing interaction step. The trace shows that the agent reaches the call-log creation page and that the client-loading request succeeds, returning valid client data. However, the execution timeline then contains no follow-up client-selection interaction. This “silence” after successful data loading localizes the failure to the frontend affordance: the backend is functional, but the native dropdown is not sufficiently actionable for the browser agent. Guided by this evidence, **TeleGen** replaces the `<select>` dropdown with explicit clickable client cards and direct `onClick` handlers. After this change, the browser agent can select a client and complete the call-log creation workflow.

Overall, the results show that observability is most useful when the cause of failure lies between the user action and the final UI state. Runtime evidence helps bridge this gap by exposing hidden interaction paths, state transitions, handler execution, and frontend–backend behavior that task-level feedback alone does not reveal.

6 Conclusion

We presented **TeleGen** for LLM-generated web applications. Instead of relying only on task-level pass/fail feedback, our approach instruments generated applications, collects runtime evidence during browser-based execution, compresses the resulting traces into a repair-oriented telemetry brief,

and uses this evidence to guide code repair. Experiments on WebGen-Bench show that **TeleGen** improves task success over both initial generation and no-telemetry repair under the main DeepSeek-V4-Flash configuration. Category-level results further show gains across content presentation, user interaction, and data management projects. Our ablation demonstrates that runtime logs provide useful repair signal, while telemetry-brief compression further improves success and reduces token usage compared with using logs directly. Finally, failure-type analysis and a representative case study show that runtime evidence is especially useful for failures involving hidden interaction paths, form workflows, state transitions, and frontend–backend coordination. Overall, our results suggest that making generated web applications observable is a practical way to turn opaque browser-level failures into actionable evidence for LLM-based code repair.

7 Limitations

While our results suggest that observability provides useful runtime evidence for repairing generated web applications, our current implementation has several limitations.

First, the observability layer relies on LLM-based telemetry instrumentation. Although we constrain the instrumentation model to output patches and apply these patches locally, the process can still introduce variance. The model may miss important execution points, insert noisy or redundant logs, or modify code locations that are already correct. In rare cases, instrumentation may affect application behavior rather than only exposing it. Our telemetry sanitization and compression step reduces the impact of noisy logs, but it does not fully eliminate the risk of incomplete or imperfect instrumentation.

Second, our current instrumentation strategy is intentionally conservative. The model inserts telemetry based on the source code and task context in order to expose as much potentially relevant runtime behavior as possible. This broad insertion strategy can increase runtime overhead, produce long execution logs, and consume additional storage and context budget. While the telemetry brief reduces the amount of evidence eventually provided to the repair agent, it does not remove the cost of collecting and storing the raw traces. More selective or adaptive instrumentation strategies could reduce this overhead in future work.

Despite these limitations, the results show that

making generated web applications observable provides useful repair evidence across a diverse set of browser-based tasks. Future work could explore more deterministic instrumentation, adaptive logging policies, and lightweight runtime tracing mechanisms to further improve the reliability and efficiency of observable repair.

Acknowledgments

This work is supported by the Guangdong Science and Technology Department (Internal Project No. G_2025_082).

References

- Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2016. [Understanding javascript event-based interactions with clematis](#). *ACM Trans. Softw. Eng. Methodol.*, 25(2):12:1–12:38.
- Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Möller, and Frank Tip. 2011. [A framework for automated testing of javascript web applications](#). In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, pages 571–580. ACM.
- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *CoRR*, abs/2108.07732.
- Titus Barik, Robert DeLine, Steven Mark Drucker, and Danyel Fisher. 2016. [The bones of the system: a case study of logging and telemetry at microsoft](#). In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 92–101. ACM.
- Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2025. [Repairagent: An autonomous, llm-based agent for program repair](#). In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 2188–2200. IEEE.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#). *CoRR*, abs/2107.03374.
- Antoine Chevrot, Alexandre Vernotte, Jean-Rémy Falleri, Xavier Blanc, Bruno Legeard, and Aymeric Cretin. 2025. [Are autonomous web agents good testers?](#) *Proc. ACM Softw. Eng.*, 2(ISSTA).

- Truong Hai Dang, Jingyu Xiao, and Yintong Huo. 2025. [Envisioning future interactive web development: Editing webpage with natural language](#). In *2nd IEEE/ACM International Conference on AI-powered Software, AIware 2025, Seoul, Republic of Korea, November 19-20, 2025*, pages 61–66. IEEE.
- Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Lacoste, Massimo Caccia, Alexandre Drouin, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omid Shayegan, Lawrence Kunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Graham Neubig, Quentin Cappart, Russ Salakhutdinov, and Nicolas Chapados. 2025. [The browsergym ecosystem for web agent research](#). *Trans. Mach. Learn. Res.*, 2025.
- Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. [Mind2web: Towards a generalist agent for the web](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vázquez, Nicolas Chapados, and Alexandre Lacoste. 2024. [Workarena: How capable are web agents at solving common knowledge work tasks?](#) In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 11642–11662. PMLR / OpenReview.net.
- Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. [Automated repair of programs from large language models](#). In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1469–1481. IEEE.
- Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. 2024. [Multimodal web navigation with instruction-finetuned foundation models](#). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanov, Botao Yu, Bernal Jiménez Gutiérrez, Yiheng Shu, Chan Hee Song, Jiaman Wu, Shijie Chen, Hanane Nour Moussa, Tianshu Zhang, Jian Xie, Yifei Li, Tianci Xue, Zeyi Liao, and 7 others. 2025. [Mind2web 2: Evaluating agentic search with agent-as-a-judge](#). *CoRR*, abs/2506.21506.
- Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. [Webvoyager: Building an end-to-end web agent with large multimodal models](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL 2024, Bangkok, Thailand, August 11-16, 2024, pages 6864–6890. Association for Computational Linguistics.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. [Measuring coding challenge competence with APPS](#). In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.
- Minda Hu, Tianqing Fang, Jianshu Zhang, Jun-Yu Ma, Zhisong Zhang, Jingyan Zhou, Hongming Zhang, Haitao Mi, Dong Yu, and Irwin King. 2025. [Webcot: Enhancing web agent reasoning by reconstructing chain-of-thought in reflection, branching, and rollback](#). In *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 5155–5173. Association for Computational Linguistics.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. [Qwen2.5-coder technical report](#). *CoRR*, abs/2409.12186.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Livecodebench: Holistic and contamination free evaluation of large language models for code](#). In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sung Hun Kim. 2026. [A survey on large language models for code generation](#). *ACM Trans. Softw. Eng. Methodol.*, 35(2):58:1–58:72.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. [Swe-bench: Can language models resolve real-world github issues?](#) *CoRR*, abs/2310.06770.
- Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2017. [A study of causes and consequences of client-side javascript bugs](#). *IEEE Trans. Software Eng.*, 43(2):128–144.
- Frolin S. Ocariza Jr., Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. 2016. [Automatic fault localization for client-side javascript](#). *Softw. Test. Verification Reliab.*, 26(1):69–88.
- Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. [Visualwebarena: Evaluating multimodal agents on realistic visual web tasks](#). In

- Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL 2024, Bangkok, Thailand, August 11-16, 2024, pages 881–905. Association for Computational Linguistics.
- Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022a. [Enjoy your observability: an industrial survey of microservice tracing and analysis](#). *Empir. Softw. Eng.*, 27(1):25.
- Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. 2021. [A qualitative study of the benefits and costs of logging from developers’ perspectives](#). *IEEE Trans. Software Eng.*, 47(12):2858–2873.
- Yan Li, Zezi Zeng, Yifan Yang, Yuqing Yang, Ning Liao, Weiwei Guo, Lili Qiu, Mingxi Cheng, Qi Dai, Zhendong Wang, Zhengyuan Yang, Xue Yang, Ji Li, Lijuan Wang, and Chong Luo. 2026. [Mm-webagent: A hierarchical multimodal web agent for webpage generation](#). *CoRR*, abs/2604.15309.
- Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022b. [Competition-level code generation with alphacode](#). *CoRR*, abs/2203.07814.
- Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. 2022c. [Studying duplicate logging statements and their relationships with code clones](#). *IEEE Trans. Software Eng.*, 48(7):2476–2494.
- Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2019. [Dlfinder: characterizing and detecting duplicate logging code smells](#). In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 152–163. IEEE / ACM.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. [Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. [Lost in the middle: How language models use long contexts](#). *Trans. Assoc. Comput. Linguistics*, 12:157–173.
- Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024b. [Repobench: Benchmarking repository-level code auto-completion systems](#). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Steven Locke, Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Wei Liu. 2022. [Logassist: Assisting log analysis through log summarization](#). *IEEE Trans. Software Eng.*, 48(9):3227–3241.
- Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2025a. [Webgen-agent: Enhancing interactive website generation with multi-level feedback and step-level reinforcement learning](#). *CoRR*, abs/2509.22644.
- Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025b. [Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch](#). *CoRR*, abs/2505.03733.
- Chandra Shekhar Maddila, Adam Tait, Claire Chang, Daniel Cheng, Nauman Ahmad, Vijayaraghavan Murali, Marshall Roch, Arnaud Avondet, Aaron Meltzer, Victor Montalvao, Michael Hopko, Chris Waterson, Parth Thakkar, Renuka Fernandez, Kristian Kristensen, Sivan Barzily, Sherry Chen, Rui Abreu, Nachiappan Nagappan, and 5 others. 2025. [Agentic program repair from test failures at scale: A neuro-symbolic approach with static analysis and test execution feedback](#). *CoRR*, abs/2507.18755.
- Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012a. [Crawling ajax-based web applications through dynamic analysis of user interface state changes](#). *ACM Trans. Web*, 6(1):3:1–3:30.
- Ali Mesbah, Arie van Deursen, and Danny Roest. 2012b. [Invariant-based automatic testing of modern web applications](#). *IEEE Trans. Software Eng.*, 38(1):35–53.
- Anvith Pabba, Alex Mathai, Anindya Chakraborty, and Baishakhi Ray. 2025. [Semagent: A semantics aware program repair agent](#). *CoRR*, abs/2506.16650.
- Playwright. 2026. [Playwright enables reliable web automation for testing, scripting, and AI agents](#). Accessed: 2026-05-25.
- Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. [Design2code: Benchmarking multimodal code generation for automated front-end engineering](#). In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 3956–3974. Association for Computational Linguistics.
- Xiwen Teoh, Yun Lin, Duc-Minh Nguyen, Ruofei Ren, Wenjie Zhang, and Jin Song Dong. 2026. [Webtestpi-lot: Agentic end-to-end web testing against natural language specification by inferring oracles with symbolized GUI elements](#). *CoRR*, abs/2602.11724.
- Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2020. [Every build you break:](#)

- developer-oriented assistance for build failure resolution. *Empir. Softw. Eng.*, 25(3):2218–2257.
- Yuxuan Wan, Tingshuo Liang, Jiakai Xu, Jingyu Xiao, Yintong Huo, and Michael R. Lyu. 2025a. [Automatically generating web applications from requirements via multi-agent test-driven development](#). *CoRR*, abs/2509.25297.
- Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael R. Lyu. 2025b. [Divide-and-conquer: Generating UI code from screenshots](#). *Proc. ACM Softw. Eng.*, 2(FSE):2099–2122.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 2 others. 2025. [Openhands: An open platform for AI software developers as generalist agents](#). In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.
- Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. [Automated program repair in the era of large pre-trained language models](#). In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1482–1494. IEEE.
- Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R. Lyu. 2025. [Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping](#). In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pages 241–253. IEEE.
- Qing Xie and Atif M. Memon. 2007. [Designing and comparing automated test oracles for gui-based software applications](#). *ACM Trans. Softw. Eng. Methodol.*, 16(1):4.
- Kai Xu, YiWei Mao, XinYi Guan, and ZiLong Feng. 2025. [Web-bench: A LLM code benchmark based on web standards and frameworks](#). *CoRR*, abs/2505.07473.
- Zhen Yang, Wenyi Hong, Mingde Xu, Xinyue Fan, Weihang Wang, Jiale Cheng, Xiaotao Gu, and Jie Tang. 2026. [Ui2codeⁿ: Ui-to-code generation as interactive visual optimization](#). *Preprint*, arXiv:2511.08195.
- Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Qazim Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, Timothy Baldwin, Zhengzhong Liu, Eric P. Xing, Xiaodan Liang, and Zhiqiang Shen. 2024. [Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms](#). In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [Repocoder: Repository-level code completion through iterative retrieval and generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.
- Haonan Zhang, Kundi Yao, Zishuo Ding, Lizhi Liao, and Weiye Shang. 2025. [Who’s to blame? rethinking the brittleness of automated web GUI testing from a pragmatic perspective](#). In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pages 3058–3070. IEEE.
- 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. 2024. [Webarena: A realistic web environment for building autonomous agents](#). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

A Additional Model Results

The main paper focuses on DeepSeek-V4-Flash as the primary text/code model configuration. In this appendix, we report additional model-level results to examine whether the benefit of runtime telemetry is specific to the main configuration. Table 4 includes additional WebGen-Bench results with DeepSeek-V4-Pro and Qwen3.5-Plus as text/code models, as well as an additional Web-Bench validation result under Qwen3.5-Plus.

On WebGen-Bench, **TeleGen** achieves the highest success rate for both additional text/code models. For DeepSeek-V4-Pro, **TeleGen** improves overall success from 66.8% for initial generation and 68.5% for no-telemetry repair to 73.6%, increasing the number of passed tasks from 432 and 443 to 476 out of 647 tasks. For Qwen3.5-Plus, **TeleGen** improves overall success from 50.1% for initial generation and 54.3% for no-telemetry repair to 60.0%, increasing the number of passed tasks from 324 and 351 to 388 out of 647 tasks. For Gemini-3-Flash-Preview, **TeleGen** improves overall success from 66.3% under No-Telemetry repair to 69.2%, corresponding to 19 additional successful tasks and a 2.9 percentage-point gain. The same

trend also holds across the WebGen-Bench project categories: **TeleGen** achieves the best result on Content Presentation, User Interaction, and Data Management for both additional text/code models.

The additional Web-Bench result further supports this trend under Qwen3.5-Plus. Compared with Original Pass@2, **TeleGen** Pass@2 improves cumulative success from 21.9% to 24.2%, increasing the number of passed tasks from 219 to 242 out of 1000 tasks. The improvement is also consistent across difficulty groups, with Easy increasing from 58.5% to 61.0%, Moderate from 19.1% to 22.0%, and Challenging from 2.5% to 4.3%.

Overall, these additional results suggest that the benefit of runtime telemetry is not limited to the main DeepSeek-V4-Flash configuration. Across additional text/code models and an auxiliary Web-Bench validation setting, **TeleGen** preserves the same overall trend of improving repair outcomes over the corresponding non-telemetry baselines.

B Additional RQ2 Analysis

The main text reports the primary RQ2 results, including repair effectiveness and token cost. Here we provide a finer-grained comparison between Observable repair and the two baselines under the main DeepSeek-V4-Flash configuration. We summarize the comparison at both project and task levels.

B.1 Paired Significance Analysis

We further assess the task-level differences using two-sided exact McNemar tests. On WebGen-Bench, **TeleGen** solves 106 tasks that No-Telemetry repair fails, while No-Telemetry repair solves 51 tasks that **TeleGen** fails. This paired difference is statistically significant ($p = 1.35 \times 10^{-5}$). In contrast, Raw-Telemetry and **TeleGen** have 56 and 65 exclusive successes, respectively, and their difference is not statistically significant ($p = 0.4672$). On Web-Bench, Original Pass2 and **TeleGen** Pass2 have 45 and 126 exclusive successes, respectively, yielding a statistically significant difference ($p = 4.49 \times 10^{-10}$).

B.2 Human Audit of WebVoyager Evaluation

Because WebGen-Bench relies on WebVoyager for task-outcome judgment, we conduct an independent human audit to assess the reliability of the automatic evaluation. The audit population contains 493 WebVoyager-judged successes and 154

judged failures under **TeleGen**. We use balanced stratified sampling without replacement, selecting 30 judged successes and 30 judged failures with random seed 20260713. We select at most one task per project, resulting in 60 tasks from 60 distinct projects, with no manual replacement.

One independent human annotator reviews the task description, expected outcome, execution trajectory, actions, observations, screenshots, final screenshot, and final application state. The WebVoyager binary outcome and evaluator response are hidden during annotation.

Human judgment agrees with WebVoyager on 28 of 30 automatically successful cases (93.3%; Wilson 95% CI: 78.7–98.2%) and 26 of 30 automatically unsuccessful cases (86.7%; 95% CI: 70.3–94.7%). Overall agreement is 54/60, or 90.0% (95% CI: 79.9–95.3%), with Cohen’s $\kappa = 0.80$. Here, κ measures agreement between the human judgment and WebVoyager rather than agreement between multiple human annotators.

The six disagreements consist of two cases judged successful by WebVoyager but unsuccessful by manual review, and four cases judged unsuccessful by WebVoyager but successful by manual review. Because the audit intentionally contains equal numbers of WebVoyager successes and failures, the 90.0% value is a balanced agreement rate. Weighting the two strata according to the full **TeleGen** outcome distribution gives an estimated agreement of 91.7%.

B.3 Audit of TeleGen-Only Successes

We separately inspect 30 tasks sampled from the 106 cases where **TeleGen** succeeds but No-Telemetry repair fails. This audit examines whether the measured gains correspond to functional repairs or primarily to changes that make an already-correct outcome easier for the browser agent or evaluator to observe.

We categorize 24 cases as genuine functional repairs, two as mixed functional and judgeability improvements, and four as judgeability-only improvements.

Functional repairs change application behavior required to complete the evaluated task. Mixed cases combine a functional change with clearer evaluator-facing feedback, while judgeability-only cases make an already successful application state easier to operate or observe. Thus, 24 of the 30 audited cases are clearly functional repairs, and 26 involve at least some functional repair. The

Benchmark	Model	Setting	Passed / Total	Overall	Δ	Categories		
						Content	User	Data
WebGen-Bench	DeepSeek-V4-Pro	Initial generation	432 / 647	66.8	–	85.6	55.9	67.5
		No-telemetry repair	443 / 647	68.5	+1.7	83.9	61.0	66.2
		TeleGen	476 / 647	73.6	+6.8	88.5	66.5	71.2
	Qwen3.5-Plus	Initial generation	324 / 647	50.1	–	71.8	43.1	40.0
		No-telemetry repair	351 / 647	54.3	+4.2	75.9	47.0	45.0
		TeleGen	388 / 647	60.0	+9.9	81.0	54.0	48.8
	Gemini-3-Flash-Preview	Initial generation	396 / 647	61.2	–	75.9	52.4	62.5
		No-telemetry repair	429 / 647	66.3	+5.1	78.2	58.1	69.4
		TeleGen	448 / 647	69.2	+8.0	82.2	62.3	68.8
						Easy	Moderate	Challenging
Web-Bench	Qwen3.5-Plus	Pass1	150 / 1000	15.0	–	45.8	6.4	1.6
		Original Pass2	219 / 1000	21.9	+6.9	58.5	19.1	2.5
		TeleGen Pass2	242 / 1000	24.2	+9.2	61.0	22.0	4.3

Table 4: Additional model results. For WebGen-Bench, Overall, Content, User, and Data report task-level success rates for the full benchmark and three project categories. For Gemini-3-Flash-Preview, all rates use the canonical 647-task denominator, with missing evaluations counted as unsuccessful. For Web-Bench, Pass1 is the first-attempt success rate, while Original Pass2 and **TeleGen** Pass2 report cumulative success after the original and telemetry-augmented second attempts. Easy, Moderate, and Challenging report category-level cumulative success rate. Δ denotes the overall improvement over Initial generation for WebGen-Bench and over Pass1 for Web-Bench. All values are percentages except passed-task counts.

	Human +	Human –	Total
WV success	28	2	30
WV failure	4	26	30
Total	32	28	60

Table 5: Agreement between WebVoyager (WV) and the independent human annotator on the balanced 60-task audit sample.

Category	Count
Functional repair	24
Mixed functional/judgeability	2
Judgeability-only	4

Table 6: Manual categorization of 30 audited **TeleGen**-only successes.

audit also shows that judgeability contributes to a minority of the measured gains.

B.4 Cost and Runtime Analysis

We additionally analyze the computational overhead of runtime telemetry over all 101 WebGen-Bench projects. Table 7 reports the recorded model calls, API cost, and project-level wall-clock time for the three repair settings. This monetary accounting is reported separately from the token statistics in Table 3, which retain the aggregation used in the original experiments.

Setting	Calls	Cost (\$)	Time (s)
No-Tel.	24.0	0.28	711
Raw-Tel.	29.9	0.42	747
TeleGen	29.5	0.41	710

Table 7: Mean per-project cost and runtime accounting on WebGen-Bench. Cost is API cost under the DeepSeek-V4-Flash pricing used in this analysis; time is recorded repair-stage wall-clock time.

Compared with No-Telemetry repair, **TeleGen** increases the average number of model calls from 24.0 to 29.5 and the API cost from \$0.28 to \$0.41 per project. Thus, the additional model-side cost is approximately \$0.12 per project. Raw-Telemetry has a similar cost of \$0.42 per project. The recorded repair-stage wall-clock times are 711, 747, and 710 seconds per project for No-Telemetry, Raw-Telemetry, and **TeleGen**, respectively.

The observability pipeline also has modest instrumentation and evidence overhead. Instrumentation processes 8.9 source files and modifies 7.6 files per project on average, with no recorded patch-application errors. Telemetry collection is integrated into the instrumented WebVoyager execution: No-Telemetry uses one clean WV1 execution for task feedback, whereas **TeleGen** uses one instrumented WV1 execution that produces both task feedback and runtime telemetry, so telemetry col-

Base	Level	Gain	Loss	Net
No-tel.	Project	48	16	+32
	Task	106	51	+55
No brief	Project	34	14	+20
	Task	65	56	+9

Table 8: Project- and task-level comparison between **TeleGen** and each baseline. Gain denotes cases where telemetry-guided repair performs better, Loss denotes cases where the baseline performs better, and Net is Gain minus Loss.

Pair (A/B)	A-only	B-only	p
No-Tel./ TeleGen	51	106	1.35×10^{-5}
Raw-Tel./ TeleGen	56	65	0.4672
Orig.P@2/ TeleGen P@2	45	126	4.49×10^{-10}

Table 9: Two-sided exact McNemar tests on paired task outcomes. A-only and B-only denote tasks solved exclusively by the first and second settings in each pair.

lection does not add another full browser-execution pass. The resulting sanitized telemetry report averages 52.9 KB per project and is compressed to a 477-byte telemetry brief, corresponding to a 99.1% reduction in evidence size.

The remaining projects or tasks have the same outcome under both settings. Against No-telemetry repair, telemetry-guided repair yields a net gain of 32 projects and 55 tasks. Against No brief, it yields a net gain of 20 projects and 9 tasks. These results show that the benefit of Observable repair is positive at both granularities: compared with both a no-runtime-telemetry baseline and a raw-log baseline, telemetry-guided repair improves more projects and tasks than it hurts.

C Implementation Details

All repair settings use a single repair round. This design keeps the comparison controlled: after the initial browser-based evaluation, each setting receives one opportunity to modify the generated application. We use the same task set and evaluation protocol across settings.

We separate the text/code model from the browser-execution model. The text/code model is responsible for initial generation, telemetry instrumentation, telemetry-brief construction, and repair. In the main experiments, this model is DeepSeek-V4-Flash. All WebVoyager model-driven steps, including browser exploration, task execution, and final task evaluation, are routed to Qwen3.5-Plus. Additional text/code model results are reported in

Appendix A.

Telemetry instrumentation is performed as an LLM-based patching step. The instrumentation model is instructed to add runtime telemetry only and not to repair functional behavior during this stage. To reduce uncontrolled edits, the model outputs code patches rather than directly modifying the workspace. These patches are then applied locally before executing the instrumented application.

The no-telemetry baseline receives the task description, source code, and browser-level task feedback, but no runtime logs. The no-brief setting receives raw runtime logs collected from the instrumented application without telemetry sanitization and compression. Observable repair receives the compressed telemetry brief, which removes duplicated or irrelevant logs and preserves task-relevant observations such as navigation, user interactions, network requests, runtime errors, and state changes.

For observable repair, we first execute the instrumented application with WebVoyager and collect runtime observations during browser-based task execution. The collected logs are then sanitized and compressed into a repair-oriented telemetry brief. The repair agent receives this brief together with the task feedback and source code, and produces a repair patch. The repaired application is evaluated using the same WebVoyager protocol as the other settings.

For RQ2, token usage is aggregated at the project level for each repair setting. We separately record input and output tokens and report average per-project usage in the main text. The no-brief and observable settings include the additional runtime evidence provided to the repair agent, while the no-telemetry setting does not.

Computational budget. All experiments use proprietary API-based models. DeepSeek-V4-Flash is used as the main text/code model, while Qwen3.5-Plus is used for WebVoyager execution and screenshot-based task judgment. The exact parameter counts of these proprietary models are not publicly disclosed. We do not train or fine-tune any model. The main computational cost comes from API calls during generation, telemetry instrumentation, telemetry-brief construction, repair, and browser-based evaluation. We report token usage for the main repair settings in Table 3. Web application execution and browser testing are run in a local sandboxed environment using the benchmark

execution protocols.

D Prompt Templates

This section summarizes the prompt templates used in our pipeline. The actual prompts include project-specific instructions, file paths, source code, raw telemetry logs, task feedback, and task metadata. For readability, we replace such instance-specific content with placeholders while preserving the key instructions and constraints.

D.1 Initial Generation Prompt

The initial generation prompt asks the text/code model to generate a complete runnable web application from the website specification and UI task suite. This step follows the original generation protocol of the benchmark. It does not include telemetry instrumentation; telemetry is inserted later on a separate copy of the generated v1 application.

```
You are a professional full-stack developer.
Generate a complete, working web application from the
provided website specification and UI task suite.

## Project ID
{{PROJECT_ID}}

## Website Specification
{{WEBSITE_SPECIFICATION}}

## Category Information
- Primary Category: {{PRIMARY_CATEGORY}}
- Subcategories: {{SUBCATEGORIES}}

## UI Task Suite
The generated application will be evaluated with the
following UI tasks and expected results.

{{UI_TASK_SUITE}}

## Technical Requirements
Generate a full-stack application with:
1. Backend: Node.js/Express RESTful API on port 5001
2. Frontend: React + Vite on port 3000
3. CORS enabled between frontend and backend
4. All required dependencies declared in package.json files
5. Setup scripts and clear run instructions

## Implementation Requirements
1. Create the project directory structure.
2. Implement the backend routes, controllers, and data logic
   needed by the website specification.
3. Implement the frontend pages, components, navigation,
   state updates, forms, and user interactions needed by
   the UI tasks.
4. Ensure frontend API calls match backend routes, methods,
   and payload formats.
5. Prepare the environment and perform basic syntax or smoke
   checks when possible.
6. Generate complete working code with no placeholders or
   TODOs.

## Telemetry Constraint
Do not add telemetry instrumentation, debug logging, or
semantic tracing markers during initial generation.
Runtime telemetry will be inserted later by a separate
instrumentation step.

Generate COMPLETE, WORKING code.
```

D.2 Telemetry Instrumentation Prompt

Telemetry instrumentation is performed as an LLM-based patching step on a separate copy of the v1 application. The instrumentation model receives the task context and one source file at a time, and returns structured patch operations rather than directly modifying the workspace. This design allows us to inspect and apply instrumentation locally, while preserving the clean v1 source code for repair.

```
You are a code instrumentation assistant for {{LANGUAGE}}.

Your job is to add lightweight runtime telemetry logs that
help diagnose the tested web application flows.
You must not rewrite the full file.
You must return PATCH OPERATIONS ONLY as JSON.

## Goal
Add telemetry that makes task-relevant runtime behavior
visible during browser-based execution.
Use the website specification, UI task suite, and expected
behaviors to infer important user journeys and likely
code paths.

Add logs only at meaningful points, such as:
- navigation entry points
- click and input event handlers
- form submissions
- state updates
- fetch or axios requests
- network response handling
- route handlers
- controller entry points
- critical branch decisions
- error handling paths

Prefer consistent telemetry prefixes:
- [Telemetry] Interaction
- [Telemetry] Network Request
- [Telemetry] Network Response
- [Telemetry] StateChange
- [Telemetry] Branch
- [Telemetry] Error

## Strict Instrumentation Rules
1. Instrumentation is for observation only.
2. Do not change functional behavior.
3. Do not repair bugs.
4. Do not refactor unrelated logic.
5. Do not remove existing code.
6. Do not invent APIs, routes, variables, or data structures
   not supported by the current file.
7. Keep patches minimal.
8. Prefer adding logging lines or the smallest structural
   change required to place logging safely.
9. Do not modify imports unless required for a safe logging
   pattern.
10. If this file is not relevant to the tested flows, return
    {"operations": []}.
11. At most 8 operations.

## Output Format
Return exactly one JSON object.
Do not include markdown or prose.

JSON schema:
{
  "operations": [
    {
      "action": "replace",
      "search": "exact existing snippet from the file",
      "replace": "replacement snippet with added telemetry
logs",
      "reason": "short reason"
    }
  ]
}
```

Every search snippet must appear exactly as-is in the current file and match exactly once. Include enough surrounding code in the search snippet to make repeated lines unique.

```
## Task Context
{{TASK_CONTEXT}}

## Current File
Path: {{FILE_PATH}}

## Current File Content
{{FILE_CONTENT}}
```

The task context supplied to the instrumentation model contains the website specification, project category metadata, and the UI task suite with expected behaviors. This context is used only to identify task-relevant execution paths for telemetry, not to repair the application.

```
Project ID: {{PROJECT_ID}}

Website specification:
{{WEBSITE_SPECIFICATION}}

Primary category: {{PRIMARY_CATEGORY}}
Subcategories: {{SUBCATEGORIES}}

UI tasks and expected behaviors:
1. Task: {{TASK_1}}
   Expected result: {{EXPECTED_RESULT_1}}

2. Task: {{TASK_2}}
   Expected result: {{EXPECTED_RESULT_2}}

...

Use these tasks to infer the likely user journey, navigation
path, frontend interactions, backend endpoints,
network behavior, and state transitions that should be
logged.
```

D.3 Telemetry Brief Construction Prompt

The telemetry brief construction prompt compresses raw telemetry logs into a concise, repair-oriented summary. It removes duplicated or irrelevant entries while preserving task-relevant runtime evidence. The brief is not intended to be a complete execution transcript; it summarizes what happened during execution in a form that is useful for repair.

```
You are a telemetry analysis assistant.

You will receive:
1. Task context
2. Task feedback
3. Raw telemetry logs collected during browser-based
   execution

Your job is to clean noisy logs, extract meaningful runtime
evidence, and reconstruct a concise execution summary
for the repair agent.

Do NOT provide code patches.
Do NOT provide implementation instructions.
Do NOT speculate beyond the evidence.
Only describe what happened factually and objectively.

## Discard or Compress
- Framework noise
- Browser extension messages
```

- Development server messages unrelated to the task
- Exact duplicate log entries
- Repeated low-level messages that do not change the execution state
- Third-party library noise
- Logs unrelated to the tested task flow

```
## Retain and Organize
Preserve task-relevant evidence, including:
- user actions and interaction events
- navigation attempts
- network requests and responses
- JavaScript exceptions
- backend errors
- state changes
- missing or stalled interactions
- data loaded but not rendered
- successful requests followed by no visible update
- source file and line references when available
```

```
## Output Format
```

```
### Task Context
Briefly summarize the task being executed and the expected
result.
```

```
### Task Feedback
Summarize the observed pass/fail outcome and any task
executor feedback.
```

```
### User Operation Timeline
Provide a chronological timeline of task-relevant events.
```

```
Use event types such as:
```

- USER_ACTION
- NAVIGATION
- API_REQUEST
- API_RESPONSE
- STATE_CHANGE
- ERROR
- MISSING_INTERACTION
- OBSERVED_GAP

```
### Runtime Evidence Summary
Summarize the most important runtime evidence.
Focus on evidence that helps distinguish why the task failed
.
```

```
### Affected Code Locations
List source files, functions, routes, components, or line
numbers referenced in the logs, if available.
```

```
### Compression Report
Report the approximate number of raw entries, retained
entries, and discarded or compressed entries.
```

```
## Constraints
- Use only information present in the task context, task
  feedback, and raw telemetry logs.
- Do not invent missing code locations.
- Do not recommend specific patches.
- Do not include sections titled Root Cause or
  Recommendations.
- Preserve precise file references when present.
```

```
## Task Context
{{TASK_CONTEXT}}
```

```
## Task Feedback
{{TASK_FEEDBACK}}
```

```
## Raw Telemetry Logs
{{RAW_TELEMETRY_LOGS}}
```

D.4 Repair Prompt Variants

The three repair prompts differ only in the runtime evidence available to the repair agent. No-telemetry repair receives task feedback without runtime telemetry. No-brief repair receives raw telemetry logs. **TeleGen** receives the compressed telemetry brief. All repair variants edit the clean v1

source code rather than the instrumented copy.

D.4.1 No-Telemetry Repair Prompt

```
# Core Objective
You are an expert full-stack implementation agent.
The mounted workspace contains the clean v1 source code of a
generated decoupled frontend/backend web application.

Your goal is to maximize the number of task executor tasks
that pass using only:
1. the website specification,
2. the task descriptions and expected results,
3. the v1 task feedback,
4. static source-code inspection.

# Baseline Constraint
This is the no-telemetry repair branch.
Do not rely on telemetry_brief.md or telemetry_report.md.
Do not add telemetry instrumentation, debug logging, or
tracing as part of the repair.

# Analysis Strategy
1. Read the task evidence section. It lists the exact tasks,
expected results, and v1 pass/fail observations.
2. Inspect the frontend and backend code to identify missing
features, broken navigation, bad API contracts, state
-update issues, or runtime-prone logic.
3. Compare the expected behavior with the current
implementation.
4. Implement the minimum targeted changes needed for task
completion.
5. Preserve the existing application structure and
technology choices.

# Execution Constraints
1. Improve task completion, not just superficial errors.
2. Do not add logging or instrumentation.
3. Do not refactor technology stacks or file structures.
4. Ensure frontend API calls match backend routes, methods,
and payloads.
5. Do not run sandbox tests.
6. Finish once all targeted modifications are complete.
```

D.4.2 No-Brief Repair Prompt

```
# Core Objective
You are an expert full-stack implementation agent.
The mounted workspace contains the clean v1 source code of a
generated decoupled frontend/backend web application.

Your goal is to maximize the number of task executor tasks
that pass.

In addition to the standard task feedback, you are given raw
telemetry logs collected from a separately
instrumented copy of the v1 application.
These logs provide runtime evidence about what happened
during browser-based execution.
They may be noisy, duplicated, or incomplete, so they should
be interpreted together with the task feedback and
source code.

# Evidence Available
You may use:
1. website specification,
2. task descriptions and expected results,
3. v1 task feedback,
4. clean v1 source code,
5. telemetry_report.md containing raw telemetry logs.

# Clean Source Constraint
The current workspace is the clean v1 source code used for
repair.
Do not assume that telemetry instrumentation is part of the
code being repaired.
Do not add new telemetry instrumentation as part of the
repair unless it is strictly necessary for application
functionality, which should normally not be the case.

# Analysis Strategy
Step 1: Understand what should work.
```

Read the task evidence section, including task descriptions, expected results, and pass/fail observations.

Step 2: Understand what actually happened.
Read telemetry_report.md.
Use raw telemetry logs to identify:

- HTTP errors
- JavaScript exceptions
- failed or missing interactions
- navigation attempts
- network requests and responses
- state changes
- frontend-backend data mismatches
- events that occurred internally but did not produce the expected UI result

Step 3: Gap analysis.
Compare expected behavior with observed runtime behavior and source-code implementation.

Step 4: Implement fixes.
Prioritize:

1. missing task features,
2. runtime errors,
3. broken navigation paths, routes, links, and buttons,
4. frontend-backend request/response mismatches,
5. state updates that fail to render expected UI changes.

Execution Constraints

- Make targeted changes only.
- Do not add telemetry instrumentation as a repair strategy.
- Do not refactor technology stacks or file structures.
- Do not run sandbox tests.
- Finish once all targeted modifications are complete.

D.4.3 TeleGen Repair Prompt

```
# Core Objective
You are an expert full-stack implementation agent.
The mounted workspace contains the clean v1 source code of a
generated decoupled frontend/backend web application.

Your goal is to maximize the number of task executor tasks
that pass.

In addition to the standard task feedback, you are given a
telemetry brief produced from raw telemetry logs
collected on a separately instrumented copy of the v1
application.
The telemetry brief summarizes task-relevant runtime
evidence and should help you localize failures before
editing code.

# Evidence Available
You may use:
1. website specification,
2. task descriptions and expected results,
3. v1 task feedback,
4. clean v1 source code,
5. telemetry_brief.md.

# Clean Source Constraint
The current workspace is the clean v1 source code used for
repair.
Do not edit the instrumented copy.
Do not add telemetry instrumentation, debug logging, or
tracing as part of the repair.

# Analysis Strategy
Step 1: Understand what should work.
Read the task evidence section, including the exact tasks,
expected results, and v1 pass/fail observations.
```

Step 2: Understand what actually happened.
Read telemetry_brief.md.
Use the telemetry brief to identify:

- failed or missing interactions
- navigation attempts that did not reach the expected page
- successful requests followed by missing UI updates
- JavaScript exceptions
- backend errors
- frontend-backend data mismatches
- state changes that did not produce the expected visible result
- task-relevant code locations mentioned in the brief

Step 3: Gap analysis.
Compare the expected behavior, v1 task feedback, telemetry evidence, and source code.
Identify missing features, broken UI flows, failed actions, data contract mismatches, and runtime errors.

Step 4: Implement fixes.
Prioritize:
1. missing task features,
2. runtime errors,
3. broken routes, links, buttons, and entry points,
4. frontend-backend request/response mismatches,
5. state updates or rendering logic needed to complete the task.

Execution Constraints
- Make minimal targeted changes.
- Do not add telemetry instrumentation.
- Do not refactor technology stacks or file structures.
- Do not run sandbox tests.
- Finish once all targeted modifications are complete.

D.5 Task Evidence Block

All repair prompts receive a dynamically constructed task evidence block describing the application purpose, expected task behavior, and v1 task executor outcomes.

```
---
# Task Evidence from v1 Baseline

The following information describes what the application is
supposed to do and what actually happened during
automated task execution of the v1 application.

## Application Purpose
{{WEBSITE_SPECIFICATION}}

## Expected Behaviors
### Expected Task 1
Task: {{TASK_1}}
Expected Result: {{EXPECTED_RESULT_1}}

### Expected Task 2
Task: {{TASK_2}}
Expected Result: {{EXPECTED_RESULT_2}}

...

## v1 Baseline Task Results
Summary:
{{N_PASSED}} passed, {{N_FAILED}} failed, {{N_UNKNOWN}}
unknown or timeout

### {{TASK_NAME}} -- {{VERDICT}}
Task executor observation: {{OBSERVATION}}

...
```

D.6 WebVoyager Execution and Evaluation Prompts

For WebGen-Bench, WebVoyager is used both as a browser-interaction task executor and as a screenshot-based evaluator. The following templates summarize the browsing and evaluation prompts.

D.6.1 Browsing Agent Prompt

Imagine you are a robot browsing the web, just like humans.
In each iteration, you receive an observation with a screenshot of a webpage and text context.
The screenshot contains numerical labels placed at web elements.

Analyze the screenshot and choose exactly one action:

- Click [Numerical_Label]
- Type [Numerical_Label]; [Content]
- Scroll [Numerical_Label or WINDOW]; [up or down]
- Wait
- GoBack
- ANSWER; [content]

Key rules:

1. Execute only one action per iteration.
2. Do not repeat the same ineffective action if the webpage remains unchanged.
3. Use ANSWER only after completing all required steps.
4. Pay attention to search boxes, buttons, filters, sort controls, dates, forms, and labeled web elements.
5. Focus on the numerical labels attached to web elements.
6. When a task requires checking a final state, navigate and interact until the expected result is visible or clearly unavailable.

Your reply must follow:

Thought: {{BRIEF_REASONING}}
Action: {{ONE_ACTION}}

D.6.2 Evaluation Prompt

As an evaluator, you will be presented with:

1. Web task instruction
2. Result screenshots
3. Result response

Your responsibility is to assess whether the result satisfies the task instruction.

You do not need to interact with web pages.

Do not assume information not present in the screenshot or response.

If the screenshot and response conflict, the screenshot prevails.

Evaluate whether the task has been successfully accomplished

.

Explain the evidence for your evaluation, then provide a final verdict:

SUCCESS or NOT SUCCESS.

Input:

TASK: {{TASK_INSTRUCTION}}

Result Response: {{AGENT_ANSWER}}

Screenshots: {{SCREENSHOTS}}