AppBench: Benchmarking AI-Generated Web Applications

Ethan Hellman
Department of Computer Science
Stanford University
hellman1@stanford.edu

Brendan McLaughlin
Department of Computer Science
Stanford University
bmc0407@stanford.edu

Abhinav Lalwani
Department of Computer Science
Stanford University
lalwani@stanford.edu

Belinda Mo
Department of Computer Science
Stanford University
bmo98@stanford.edu

Abstract

Recent advances in large language models (LLMs) have led to significant progress in code generation, with frontier models achieving increasingly human-like performance on a range of programming benchmarks [3, 16]. As evaluation has evolved from function-level tasks to complex, real-world software engineering challenges [12, 5], web application development has emerged as a particularly promising domain—offering the potential to democratize software creation through natural language prompts. However, despite the proliferation of benchmarks evaluating LLMs across various code domains [19, 2], no existing framework holistically assesses an AI system's ability to generate functional, usable, and human-aligned web applications. We introduce **AppBench**, a multi-dimensional benchmark that evaluates AI-generated web applications via automated interaction simulation and structured reasoning, inspired by principles from user experience (UX) design. AppBench is, to our knowledge, the first comprehensive framework to standardize the evaluation of AI-generated web interfaces in a scalable, automatic, and user-centric way.

1 Introduction

Large language models (LLMs) have dramatically expanded the range of tasks AI systems can perform—particularly in software development, where frontier models are now capable of producing multi-file programs and resolving real-world issues [3, 16, 12]. Among these, web application generation has emerged as a particularly compelling domain, given its broad accessibility and the increasing availability of LLM-based tools that can scaffold entire interfaces from a single natural language prompt.

But while these capabilities are advancing rapidly, evaluation methods have not kept pace. Most benchmarks remain focused on function-level correctness or project-scale bug resolution [5**?**], and are ill-suited for assessing the usability of interactive, user-facing applications. A web app is more than just working code—it's an experience that must support intuitive interaction, logical navigation, and visual coherence. Measuring these qualities requires more than unit tests; it requires reasoning about how real users engage with the system.

In this paper, we introduce **AppBench**, a benchmark designed to fill this gap. Rather than evaluating static code correctness, AppBench simulates structured user interaction—spanning navigation, state

Stanford CS 329A: Self-Improving AI Agents, Winter 2025.

transitions, interface responsiveness, and content validation—to evaluate the usability and task completion potential of AI-generated web applications. Drawing from UX design principles and evaluation heuristics, AppBench operationalizes user expectations into automated yet behaviorally grounded scoring scripts.

Our contributions are threefold: (1) we identify and formalize a gap in the LLM evaluation landscape centered on interactive, user-aligned applications; (2) we introduce a novel simulation-based evaluation framework for web applications, built on a suite of dynamic task plans and heuristics; and (3) we demonstrate the feasibility of automated UX evaluation by showing strong alignment with human judgments across several models and agentic systems.

Ultimately, we hope AppBench can serve as a tool for guiding the development of more capable, usable, and human-aligned generative systems—pushing the frontier beyond code correctness toward the design of meaningful user experiences.

2 Related Work

As LLMs increasingly move from code assistants to autonomous agents capable of end-to-end software generation, a critical question arises: how should we evaluate their output when that output is not just code, but experience? This paper sits at the intersection of several important and rapidly evolving research frontiers—code generation, web interaction agents, human-computer interaction (HCI), and AI benchmarking itself—and our work reflects this broader confluence.

Code generation and full-stack evaluation. Early work in LLM-based code generation established foundational benchmarks for correctness at the function level, such as HumanEval [3] and APPS [8]. Subsequent efforts like CodeXGLUE [19], DS-1000 [14], and MultiPL-E [2] expanded the benchmark space to include multi-language, domain-specific, and task-diverse code understanding. Recent work like SWE-bench [12], ClassEval [5], and InterCode [25] has pushed toward project-scale and bug-resolution tasks, recognizing the importance of evaluating models in messier, more human-authored software contexts.

Despite these advances, most code generation benchmarks still center on correctness as judged by test cases or unit assertions. As recent critiques have noted [17, 27], high test pass rates may not translate to reliable or meaningful software—especially in user-facing domains.

Web-based agents and interactive benchmarks. In parallel, the emergence of LLM-powered agents has led to benchmarks focused on web interaction. WebArena [28], Mind2Web [4], AgentBench [18], and WebShop [26] evaluate an LLM's ability to act in simulated or real browser environments, often through reinforcement learning or instruction-following paradigms. These tasks help assess navigation, search, and goal completion across open-ended web environments.

However, these frameworks evaluate interaction *with* the web, not the capacity to *generate* usable web applications. Moreover, they often assume static, pre-designed interfaces—sidestepping the core challenge we address: whether a model can not only understand but *instantiate* usable human-computer interfaces from scratch.

UX evaluation and HCI principles. From the HCI community, decades of work on usability testing [20], UI evaluation automation [10], and quantitative design assessment [15, 9, 7, 21] provide foundational insight into what makes an interface "usable." Frameworks like WebQEM [23] and methods in layout quality assessment [21] illustrate structured ways to judge effectiveness, efficiency, and aesthetics of design. Yet these traditions rarely intersect with automated code evaluation, and current LLM benchmarks generally overlook these concerns entirely.

Recent efforts like UIcrit [6] have begun bridging these spaces, building datasets for UI design critique. Still, they rely on static screenshots or human annotations, limiting scalability.

Simulation-based evaluation and the road ahead. The idea of using simulation to evaluate AI systems has gained traction in NLP [13], agent benchmarks [18], and even in instructional testing [22, 24]. As AI-generated content becomes more complex, evaluation frameworks are evolving beyond static metrics toward dynamic, behaviorally grounded testing. SimulBench [11] exemplifies this shift by using simulation tasks to assess creativity and procedural understanding.

Our work joins this trend by applying simulation-based evaluation to the web application domain. By using interaction agents to emulate real users and evaluate LLM outputs not only as programs but as experiences, AppBench brings together ideas from AI testing, HCI, and design evaluation into a unified and scalable framework.

3 Methodology

AppBench is a preliminary benchmark composed of 9 diverse and challenging app-generation tasks. Each task in the dataset contains 1) a natural language user query and 2) an executable evaluation script, tailored to the user query, that outputs a scalar value quality score with a corresponding natural language justification that is calibrated to align closely with human evaluation of the web application.

3.1 Task Selection

In selecting user queries to include in the dataset, we aim to test model capability along each of the following six axes of web development: **(1) UI Complexity** – visual and structural sophistication; **(2) Feature Coverage & Functionality** – breadth of supported capabilities; **(3) State Management** – handling of user interactions and data persistence; **(4) API Integration** – ability to connect with external services; **(5) Cross-Page Functionality** – navigation and multi-page interactions; and **(6) Data Processing** – handling and transformation of structured data.

We note the importance of taking a more empirical approach to weighting the importance of each of these axes such that our dataset properly reflects their real-world importance. Additionally, we select tasks that range in difficulty from **L0** (basic static apps) to **L4** (apps that demand mastery of advanced web development skills) (full task descriptions in Appendix A.1).

Below is an example prompt from the AppBench dataset:

- L2: "Generate a basic newsletter sign-up website where the user can input their email and name, click submit, and receive a welcome email. Use the MailSlurp SMTP client with the following credentials: [omitted for brevity]"

User queries are also diverse in their length, sophistication, and specificity. As seen in L2 above, for user queries that imply usage of APIs that are not publicly accessible, we provide credentials in the user query.

3.2 Mapping User Queries to Evaluation Plans

AppBench translates natural language app requirements into structured evaluation plans through a systematic process that preserves the intent, constraints, and expectations embedded in the original query. To illustrate this approach, we analyze the following example query (additional examples in Appendix A.2):

> Develop a multi-page app where the home page displays a list of products (each showing a name, price, and thumbnail image) fetched from an API. Clicking a product navigates to a detail page with its full description, larger image, and 'Add to Cart' button. The app should have a persistent cart icon in the header showing the number of items in cart. Cart state should persist across navigation and page refreshes.

Figure 1: Example user query with highlighted evaluation axes: Cross-Page Functionality, UI Components, API Integration, Navigation, State Management

3.2.1 Evaluation Checklist Extraction

We first construct a sequential evaluation checklist that captures both explicit and implicit requirements. This checklist serves as an intermediate representation between the user query and the final evaluation graph. For our example query, the human evaluators organically identified the following checklist items:

Core Requirements (Explicit)

- Has home page with list of products
- Each product shows name, price, thumbnail at least
- There is a persistent header with an add to cart button
- When you click a product, you navigate to a new page
- That new page has a description, larger image, and an add to cart button
- The persistent cart icon should still be in the header on detail page
- If you click add to cart, the cart icon shows a 1 item added to cart
- If you refresh or go back page, the cart state persists

Quality-Enhancing Features (Implicit)

- There is a reasonable title for the website in the header or on home page
- There is a back page button to navigate back to home page
- User can add multiple items at once to cart using a quantity incrementer
- User can view the items in cart (by clicking on cart icon)
- User can remove items from cart

Figure 2: Sequential evaluation checklist derived from user query with highlighted categories: UI Components, Navigation, State Management

3.2.2 Dependency Modeling and Graph Construction

The sequential checklist reveals natural dependencies between requirements. For instance, evaluating the "add to cart" functionality first requires successful navigation to a product detail page, which itself depends on the existence of clickable product listings on the home page. These dependencies form the basis of our evaluation graph.

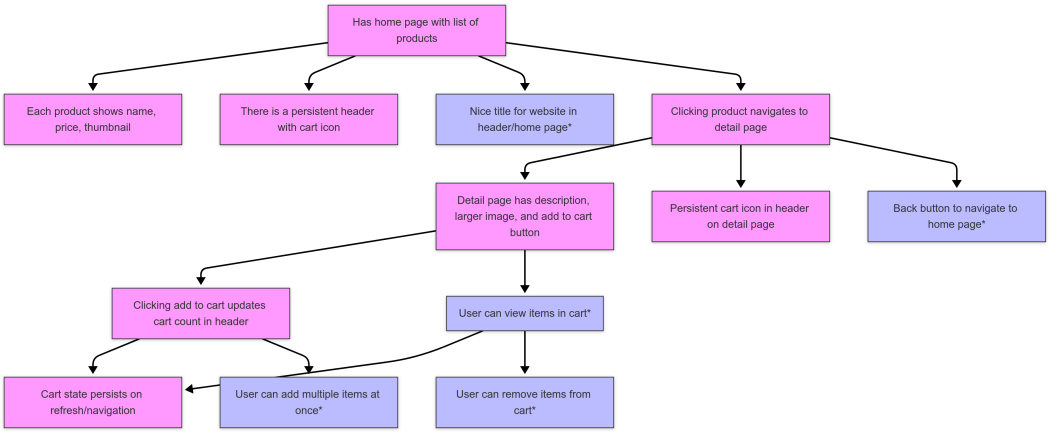

Figure 3: Evaluation dependency graph for the *l3-add-to-cart* example. Pink nodes represent explicit core requirements, while purple nodes represent implicit quality-enhancing features. Edges indicate dependencies between requirements.

The evaluation graph models user journeys through the application, with each node representing a testable requirement and each edge representing a dependency. Core requirements (pink nodes) are prioritized in the evaluation while quality-enhancing features (purple nodes) contribute additional points to the overall assessment.

3.2.3 Evaluation Harness

Our evaluation harness enables us to convert organically constructed evaluation graphs into executable scripts by providing a shared toolbox of flexible evaluation primitives that can be chained together to evaluate each graph node.

Some common evaluation primitives include:

1. **Multimodal LLM Reasoning** – Screenshots paired with natural language prompts provide state observations without DOM dependencies
2. **Adaptive LLM-Augmented Browser Interaction** – Stagehand [1] enables natural language commands for app manipulation (e.g., "click the add to cart button")
3. **Network Manipulation** – API interception allows testing error states and edge cases (implementation details in Appendix A.3)

At the top of each script, each task's assessment criteria are defined in a structured configuration that maps directly to the evaluation graph, with weighted categories and subcategories (details in Appendix A.4.

By mapping natural language queries to executable evaluation graphs in this manner, AppBench creates a flexible and scalable framework for evaluating AI-generated web applications across various dimensions of functionality, usability, and alignment with user intent.

4 Experiments

4.1 Model Performance

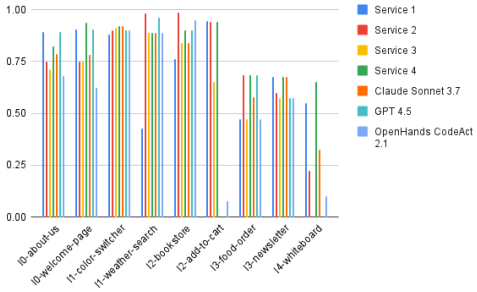

Figure 4: Individual agent performance across all examples.

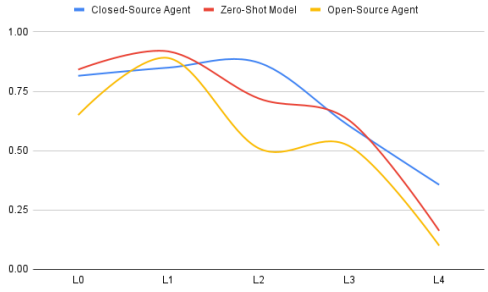

Figure 5: Agent performance across each level by agent type.

To test the model performance, we operated in the "one-shot" regime: each agent was prompted with the same prompt and given one turn to build a functioning web application. For Claude 3.7 and GPT 4.5, they were fed context that first detailed the default structure of a Next.js application created as a scaffold to work with (details in Appendix A.5). This was done to help these models understand the task better and avoid generations that were overly simplistic and would necessarily under-perform compared to their agentic counterparts. All generations were then evaluated using the same prompt-specific evaluator. Figure 4 shows all scores per agent, per prompt. The same scores have been averaged across model type (Figure 5) for further analysis.

4.2 Evaluator Performance

Metric	Value
Mean MSE	0.0302
Mean MAE	0.1313
Pearson Corr.	0.8173
Spearman Corr.	0.6827

Table 1: Overall Metrics

Agent	MSE
Service 1	0.0466
Service 2	0.0307
Service 3	0.0303
Service 4	0.0160
Claude 3.7	0.0199
GPT 4.5	0.0179
CodeAct	0.0501

Table 2: MSE per Agent

Prompt	MSE
l0-about-us	0.0194
l0-welcome-page	0.0179
l1-color-switcher	0.0194
l1-weather-search	0.0163
l2-bookstore	0.0212
l2-add-to-cart	0.0042
l3-food-order	0.0751
l3-newsletter	0.0653
l4-whiteboard	0.0334

Table 3: MSE per Prompt

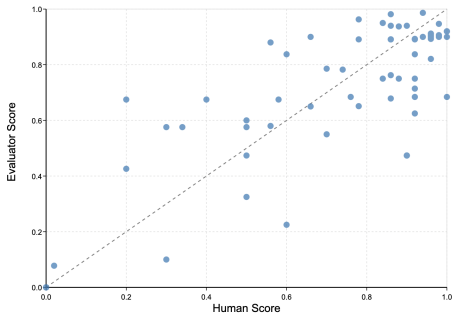

Figure 6: Human vs. Evaluator Scoring

In order to test our approach in creating intelligent evaluators, for each web application generation human evaluations were conducted. The human evaluation consisted of 3 independent evaluators checking over the web applications and reporting their scores. The human evaluations were conducted without knowing the evaluation results from the previous experiment. Subsequent analysis can be seen in Tables 1, 2, and 3. The correlation between human and machine scores is visualized in Figure 6.

5 Discussion

Our evaluation reveals several promising—and in some cases, surprising—insights into the current state of AI-generated web applications. As shown in Figure 4, model performance declines steadily as task complexity increases, confirming that AppBench captures a meaningful gradient of difficulty across core web development skills. This aligns with broader trends in LLM research, where strong performance on simpler tasks often masks brittleness on more structured, high-context, or interactive problems.

One particularly notable finding is the relatively narrow margin between zero-shot outputs and fully agentic systems (Figure 5). While we expected agent frameworks to significantly outperform single-prompt completions—particularly on higher-level tasks—the performance gap was modest, and in some cases negligible. This raises important questions: Are today's agentic pipelines offering meaningful returns on their added complexity and inference cost? Or are we nearing diminishing returns without deeper architectural or supervisory advances?

This finding becomes especially salient at higher difficulty levels such as L4, where most systems—regardless of orchestration—struggle to generate usable experiences. At such low performance bands, the difference between a partially functional and a completely broken web app becomes moot from a user experience standpoint. Although agentic systems occasionally produce cleaner codebases or better-structured designs, they do not consistently result in applications that are more usable or functionally successful. The data suggests that, at least for now, the gap between well-engineered agents and high-performing zero-shot models like Claude or GPT remains surprisingly narrow.

Equally encouraging is the performance of our evaluation harness. As shown in Table 1 and Figure 6, our scoring pipeline correlates strongly with human judgments—demonstrating that simulation-based UX evaluation can meaningfully approximate human assessment. This is not only a technical validation, but a conceptual one: automation and user-centered design can coexist, and even complement each other in benchmark construction.

There remains substantial room to refine and expand our evaluator suite. Calibration with expert UX reviewers could improve threshold sensitivity, enable more nuanced detection of failure modes, and enhance inter-model differentiation—particularly in mid-performance bands where outputs are neither clearly successful nor clearly broken. With broader scale, we may also better characterize the nature of common failure modes across systems. While AppBench spans a diverse range of task types, its limited scope makes it premature to draw strong generalizations. Still, early patterns are emerging: closed-source agentic systems tend to produce more visually polished interfaces, yet they do not substantially outperform zero-shot models in supporting task completion. This underscores a key insight—functional usability and visual appeal are distinct axes of quality, and agentic infrastructure alone is not yet sufficient to close the gap.

6 Conclusion

As generative models expand their role from code completion to full-stack application development, the need for benchmarks that reflect real user expectations becomes increasingly urgent. In this work, we introduced **AppBench**, a novel evaluation framework for AI-generated web applications that foregrounds usability, interactivity, and goal-completion—qualities essential to delivering functioning software experiences.

Our approach reframes benchmark design around the end user, leveraging interaction simulation and structured UX heuristics to evaluate AI outputs not just as code, but as experiences. Through both quantitative and qualitative analysis, we demonstrated that AppBench can reliably capture differences in model performance across varying levels of task complexity, while remaining strongly correlated with human evaluation.

While our current benchmark is limited in scale, we believe AppBench offers a compelling foundation for evaluating human-aligned software generation. As future work explores larger datasets, automated evaluator construction, and tighter coupling with real-world UX paradigms, we hope this line of research inspires a shift toward benchmarks that reflect how AI systems will actually be used—in practice, by people.

References

[1] BrowserBase. Stagehand: AI-powered browser automation. GitHub repository, 2024. [Online]. Available: `https://github.com/browserbase/stagehand`.

[2] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Xiang Deng, Yu Gu, Boyuan Zheng, et al. Mind2web: Towards a generalist agent for the web, 2023.

[5] Xueying Du, Mingwei Liu, Kaixin Wang, et al. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.

[6] Peitong Duan, Chin-yi Chen, Gang Li, et al. Uicrit: Enhancing automated design evaluation with a uicritique dataset, 2024.

[7] Basma K. Eldrandaly, Ahmed A. Al, Ripon K. Chakrabortty, and Mohamed Abdel-Basset. An efficient framework for evaluating the usability of academic websites: Calibration, validation, analysis, and methods. *Neutrosophic Sets and Systems*, 53(1), 2023.

[8] Dan Hendrycks, Steven Basart, Saurav Kadavath, et al. Measuring coding challenge competence with apps, 2021.

[9] Huicong Hu, Ying Liu, Wen Feng Lu, and Xin Guo. A quantitative aesthetic measurement method for product appearance design. *Advanced Engineering Informatics*, 53:101644, 2022.

[10] Melody Y. Ivory and Marti A. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4):470–516, 2001.

[11] Qi Jia, Xiang Yue, Tianyu Zheng, et al. Simulbench: Evaluating language models with creative simulation tasks, 2024.

[12] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[13] Douwe Kiela, Max Bartolo, Yixin Nie, et al. Dynabench: Rethinking benchmarking in nlp, 2021.

[14] Yuhang Lai, Chengxi Li, Yiming Wang, et al. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

[15] Xiaosong Li, Ye Liu, Zizhou Fan, and Will Li. A quantitative approach in heuristic evaluation of e-commerce websites. *International Journal of Artificial Intelligence amp; Applications*, 9(1):01–13, January 2018.

[16] Yujia Li, David Choi, Junyoung Chung, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.

[18] Xiao Liu, Hao Yu, Hanchen Zhang, et al. Agentbench: Evaluating llms as agents, 2023.

[19] Shuai Lu, Daya Guo, Shuo Ren, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[20] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. *SIGCHI Bulletin*, 21(1):249–256, 1990.

[21] Yumei Pu, Danfei Liu, Siyuan Chen, and Yunfei Zhong. Research progress on the aesthetic quality assessment of complex layout images based on deep learning. *Applied Sciences*, 13(17), 2023.

[22] Jon Saad-Falcon, Rajan Vivek, William Berrios, Nandita Shankar Naik, Matija Franklin, Bertie Vidgen, Amanpreet Singh, Douwe Kiela, and Shikib Mehri. Lmunit: Fine-grained evaluation with natural language unit tests, 2024.

[23] K. K. Singh. A quantitative method for evaluation of websites quality using webqem tool. 2014.

[24] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing, 2024.

[25] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023.

[26] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023.

[27] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks, 2024.

[28] Shuyan Zhou, Frank F. Xu, Hao Zhu, et al. Webarena: A realistic web environment for building autonomous agents, 2024.

A Appendix

A.1 Task Definitions and Complexity Levels

Table 4 provides a breakdown of the 9 AppBench tasks by complexity level and primary evaluation axes.

The complexity levels correspond to increasing difficulty:

- **L0**: Static applications with minimal interactivity
- **L1**: Basic interactivity with simple state management
- **L2**: Multi-page applications with navigation and moderate state management
- **L3**: Complex state management across multiple views with API integration
- **L4**: Advanced applications requiring sophisticated state management, real-time updates, and complex user interactions

Task ID	Level	Primary Axes	Description
l0-about-us	L0	1, 2	Static about page with responsive design
l0-welcome-page	L0	1, 2	Minimal landing page with hero section
l1-color-switcher	L1	1, 2, 3	Theme switcher with state management
l1-weather-search	L1	1, 3, 4	Weather app with API integration
l2-bookstore	L2	1, 2, 5, 6	Multi-page book catalog
l2-add-to-cart	L2	1, 3, 5	Simple e-commerce with cart functionality
l3-food-order	L3	1, 2, 3, 5, 6	Stateful online restaurant menu
l3-newsletter	L3	1, 3, 4	Newsletter signup with email integration
l4-whiteboard	L4	1, 2, 3, 4, 6	Collaborative drawing app with persistent state

Table 4: AppBench tasks by complexity level and evaluation axes

A.2 Complete Task Queries

Table 5 provides the complete text of all task queries in the AppBench dataset. For space reasons, we show evaluation graphs only for the add-to-cart example (Figure 3) in the main text.

A.3 Evaluation Primitives Implementation

AppBench implements a set of core evaluation primitives that can be flexibly combined to assess any node in the evaluation graph. These primitives are implemented using a combination of browser automation and LLM-augmented observation:

A.3.1 Schema-Based Screenshot Evaluation

Our *LLMEvaluator* class provides structured assessment of application state through screenshots among other methods:

```
const uiEval = await this.llmEvaluator.evaluateScreenshot(
  screenshot,
  weatherUISchema,
  "Evaluate the weather app's UI elements and layout...",
  "UI"
);
```

The evaluator uses strongly typed schemas to ensure quantifiable assessment:

```
const weatherUISchema = z.object({
  has_search: z.boolean(),
  has_button: z.boolean(),
  layout_score: z.number().min(0).max(getMaxScore(weatherAppConfig, "initialUI", "layout")),
  has_title: z.boolean(),
  reasoning: z.string(),
});
```

A.3.2 Error State Validation

A specialized primitive for assessing application responses to error conditions:

```
const errorEval = await this.llmEvaluator.evaluateErrorState(
  screenshot,
  testEmptySchema,
```

```
  "empty search submission",
  "Errors/Empty"
);
```

A.3.3 Conditional Test Execution

AppBench implements dependency-aware testing that respects the evaluation graph structure:

```
// Only proceed with weather tests if basic UI exists
if (uiScore >= 4) {
  await this.testWeatherDisplay(stagehand);
  await this.testErrorHandling(stagehand);
} else {
  this.logger.warn(
    "Skipping weather tests due to insufficient UI score",
    "WeatherApp"
  );
}
```

This approach prevents cascading failures when fundamental requirements are not met, providing a more nuanced evaluation that mirrors how human testers would approach application assessment.

A.4 Evaluation Scoring Configuration

Each AppBench evaluator defines a structured scoring configuration that maps directly to the evaluation graph and UX priorities. Below is the example configuration for the weather search application:

```
const weatherAppConfig: ScoringConfig = {
  categories: {
    initialUI: {
      name: "Initial UI Elements",
      maxScore: 8,
      subcategories: {
        searchInput: { name: "Search Input", maxScore: 2.5 },
        searchButton: { name: "Search Button", maxScore: 2.5 },
        layout: { name: "Layout Quality", maxScore: 2 },
        title: { name: "Weather Title", maxScore: 1 },
      },
    },
    weatherDisplay: {
      name: "Weather Display",
      maxScore: 15,
      subcategories: {
        temperature: { name: "Temperature Display", maxScore: 4 },
        condition: { name: "Weather Condition", maxScore: 3 },
        additionalData: { name: "Additional Data", maxScore: 3 },
        quality: { name: "Display Quality", maxScore: 5 },
      },
    },
    errorHandling: {
      name: "Error Handling",
      maxScore: 4,
      subcategories: {
        emptySearch: { name: "Empty Search", maxScore: 2 },
        invalidCity: { name: "Invalid City", maxScore: 2 },
      },
    },
  },
};
```

This declarative approach allows each evaluator to precisely weight different aspects of the application according to their importance. Moreover, modifications to score weights and test sections to improve alignment with human evaluators is made trivial by this implementation. Scores are normalized to a 0-100 scale for consistency across tasks of different complexity.

A.5 Prompting

```
You are an AI coding agent that builds web applications.
You are given default started code for a next.js built with:
Next.js (version 15.2.3) - A React framework for building web applications
TypeScript - For type-safe JavaScript development
Tailwind CSS - For styling
ESLint - For code linting
The project structure follows Next.js conventions:
my-app/
|-- src/                    # Source code directory
|   |-- pages/             # Next.js pages directory (routing)
|   `-- styles/            # CSS and styling files
|-- public/                # Static assets
|-- node_modules/          # Dependencies
|-- package.json           # Project configuration and dependencies
|-- tsconfig.json          # TypeScript configuration
|-- next.config.ts         # Next.js configuration
|-- postcss.config.mjs     # PostCSS configuration (for Tailwind)
|-- eslint.config.mjs      # ESLint configuration
`-- .gitignore             # Git ignore rules
Key features of the project:
It's a TypeScript-based Next.js application
Uses modern React (version 19)
Implements Tailwind CSS for styling
Has ESLint configured for code quality
Follows the standard Next.js project structure with a src directory.
Build the following application as described starting from the default code:
```

Task ID	Complete User Query
l0-about-us	"Design a static 'About Us' page with a header, descriptive text in a content section, and a footer with company details."
l0-welcome-page	"Generate a basic landing page with a header, a centered welcome message, and a footer. The page should be a warm, welcoming landing page."
l1-color-switcher	"Create a simple theme switcher web page where users can toggle between three color schemes: light, dark, and blue. The page should have a clear title at the top, followed by three clickable theme buttons arranged horizontally. Each button should show which theme is currently active. Below the buttons and include a sample content section with a heading, short paragraph of text to demonstrate how the theme affects different elements. When users click a theme button, the entire page (background, text colors, and button styles) should immediately update to match the selected theme. Keep everything on a single page with no backend functionality or data persistence needed - just simple state management to track the current theme."
l1-food-order	"Create a single-page web application that contains food items from a menu. There should be the name of the food and price listed. Without having to login or create an account, the user should be able to quickly construct an order by clicking the quantity of a given food item to add to their order. The bottom of the page should dynamically display the order items and the total cost of the order."
l2-newsletter	"Generate a basic newsletter sign-up website where the user can input their email and name, click submit, and receive a welcome email. Use the MailSlurp SMTP client with the following credentials: [omitted for brevity]"
l2-weather-search	"Create a page that includes a search box; when a user enters a city name and clicks a button, fetch and display basic weather data from an the OpenWeatherMap API using this api key: 442b472f13319ac99f6ecb231e3c2fe0"
l3-add-to-cart	"Develop a multi-page app where the home page displays a list of products (each showing a name, price, and thumbnail image) fetched from an API. Clicking a product navigates to a detail page with its full description, larger image, and 'Add to Cart' button. The app should have a persistent cart icon in the header showing the number of items in cart. Cart state should persist across navigation and page refreshes."
l3-bookstore	"Create an app with a landing page containing a search bar for books, and upon submission, navigate to a results page displaying a list of books with their titles, authors, and cover images fetched from the Google Books API. Use this Google API token: [omitted for brevity]."
l4-whiteboard	"Create a real-time collaborative whiteboard where users can draw together. The whiteboard should have a canvas where users can draw with the mouse, a tool bar with at least 3 different colors to choose from, and a section that shows the connected users and an invite button that opens a modal/overlay where the user can copy the unique white board URL that can be used to join the board from another browser window. Users on the same board should be able to see live drawing updates to the whiteboard from other users."

Table 5: Complete user queries for all AppBench tasks