CAP: A SCALABLE BENCHMARK FOR EVALUATING CROSS-SITE BROWSER AGENTS WITH COMPLEX ACTIONS AND PERCEPTION

Zejun Xu[1,2], Taiyi Chen[3]*, Jin Li[2]*, Yongtong Gu[4]*, Qi Cheng[2], Aixuan Lv[2], Shuai Zhu, Pengfei Zhu[1], Kaichen Yang[2], Boyu Sun[4], Yixian Yang[1], Mulong Xie[2], Xiaoteng Ma[3], Hongru Wang[5]

[1]Macau University of Science and Technology [2]Fellou AI [3]Tsinghua University
[4]Southeast University [5]University of Edinburgh

zejunx@student.must.edu.mo (Z. Xu) ty-chen20@tsinghua.org.cn (T. Chen)
jinli12334@gmail.com (J. Li) 220255567@seu.edu.cn (Y. Gu)

ABSTRACT

Large language models are increasingly deployed as autonomous agents that interact with the web through browsers. While recent progress has been driven by benchmarks that evaluate end-to-end task success, these evaluations largely overlook two fundamental sources of difficulty in real web browsing: *complex actions over rich user interfaces* and *visual perception of dynamically rendered content*, especially in workflows that span multiple websites. We introduce CAP, a scalable benchmark for evaluating browser agents on cross-site, human-like web tasks that require non-trivial UI interactions and visual understanding. Specifically, we adopt decomposition-recomposition pipeline to first abstract each website into a structured site card, capturing user-facing functions, complex execution operations, and perceptual requirement, and then recomposes these components into realistic cross-site workflows. Therefore, each task is grounded into multiple specific operations in the website, enabling fine-grained diagnosis. Build on this framework, we successfully construct 420 tasks across 108 real-world websites and 24 domains, with carefully quality control. Experiments on state-of-the-art browser agents show low success rates using our newly proposed verifiable agent-as-a-judge evaluation framework and reveal that perception-heavy interactions remain a major bottleneck, highlighting substantial gaps between current agents and real-world web browsing demands.

1 INTRODUCTION

The era of agentic AI has arrived. Large language models are no longer confined to generating text in isolation—they now act as autonomous agents that perceive, reason, and execute actions in real-world environments Wang et al. (2024); Xi et al. (2023). Among the diverse interfaces these agents can operate on, the web browser stands out as arguably the most consequential since it serve as the universal gateway to online information, services, and applications (Yoran et al., 2024; Gou et al., 2025), from booking flights and managing finances to conducting research and shopping. Consequently, mastering browser-based interaction is central to the future of agentic AI.

Yet real-world browser tasks are far more demanding than current benchmarks suggest. Most existing benchmarks focus on single-site interactions and primarily involve simple operations such as clicking links or entering text Deng et al. (2023); Zhou et al. (2023), or address cross-site tasks, but typically evaluate agents based on final answers or high-level task completion, without systematically measuring the difficulty of the underlying interactions Yoran et al. (2024); Mialon et al. (2023). As a result, they offer limited insight into *how* and *why* agents fail in practice. In contrast, modern

*Equal contribution.

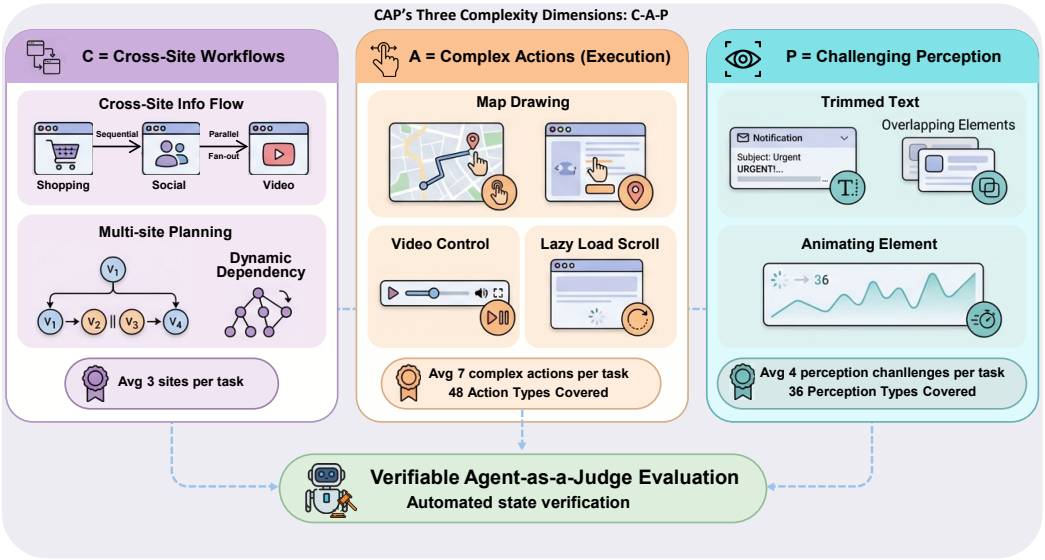

Figure 1: Overview of the CAP framework. The benchmark evaluates agents on **C**ross-site workflows, complex **A**ctions, and challenging **P**erception via a verifiable agent-as-a-judge framework.

web applications present two intertwined challenges that existing evaluations largely overlook: 1) *execution complexity*: completing a task often requires non-trivial interactions (e.g., dragging map regions, manipulating date-range sliders, or navigating nested dropdown menus), operations that go well beyond simple clicking and typing; and 2) *perception complexity*: users must interpret rich visual content including charts, tables, images, and dynamically rendered elements to extract actionable information. Compounding these challenges, realistic workflows demand complex interactions across multiple websites[1]. Together, these gaps highlights the limitations of current benchmarks, motivating the need for more grounded and diagnostic evaluation benchmarks and frameworks.

We introduce **CAP**, a scalable benchmark designed to evaluate browser agents under realistic web browsing conditions, which targets three sources of difficulty that are central to human web use as illustrated in Figure 1: **C**ross-site workflows, complex **A**ction, and challenging visual **P**erception. To enable fine-grained grounding and diagnosis, we adopt *decomposition and recomposition* strategy by first decomposing each website into a structured *site card* that catalogs what the site can do, what complex actions it requires (e.g., map dragging, video control), and what perceptual challenges it presents (e.g., interpreting charts, handling overlapping elements). We then recompose these building blocks into coherent cross-site task proposals through affinity-guided sampling, and finally instantiate them into concrete tasks with automatically generated evaluation criteria. Therefore, it naturally support both expansion and diagnosis. New websites can be added by annotating their site cards, after which they can be integrated into tasks alongside existing sites without manual task engineering. At the same time, because each task is composed of known execution and perception components, evaluation can be grounded in explicit checkpoints rather than end results alone with the proposed verifiable agent-as-a-judge framework. To sum up, our contributions are as follows:

- We present CAP, a benchmark of 420 cross-site tasks spanning 108 real-world websites across 24 functional domains. Each task requires an average of 7 complex execution operations and 4 perception challenges, substantially exceeding the difficulty of prior benchmarks.
- We develop a fully automated pipeline for task synthesis that decomposes websites into structured site cards and recomposes them into diverse, challenging tasks, enabling continuous benchmark expansion and comprehensive coverage.

[1]One example in our data: choosing a laptop might require filtering by specs and budget on BestBuy, comparing stock availability across Amazon and Target, interpreting ratings and parsing negative reviews for recurring complaints, and watching video reviews on YouTube, all within a single task.

Table 1: Comparison with existing web browsing benchmarks. Horizon indicates task length, Dynamic indicates whether answers are time-varying, Cross-site indicates cross-website navigation, and Exec./Perc. Complex indicate requirements for complex execution and perception operations.

Benchmark	# Tasks	Horizon	Dynamic	Cross-site	Exec. C.	Perc. C.	Evaluation
Online-Mind2Web Xue et al. (2025)	300	Short	✓	✗	✗	✗	LLM-as-a-Judge
WebVoyager He et al. (2024)	643	Short	✓	✗	✗	✗	LLM-as-a-Judge
GAIA Mialon et al. (2023)	466	Med.	✗	✗	✗	✗	Answer Match
AssistantBench Yoran et al. (2024)	214	Med.	✗	✗	✗	✗	Answer Match
Mind2Web2 Gou et al. (2025)	130	Long	✓	✓	✗	✗	Agent-as-a-Judge
Ours	420	Long	✓	✓	✓	✓	Verifiable Agent-as-a-Judge

- We propose a verifiable agent-as-a-judge evaluation framework that automatically generates hierarchical rubrics from task structure, providing fine-grained diagnosis of execution and perception capabilities.
- We evaluate five state-of-the-art browser agents and find that even the best-performing system achieves only 7.0% success rate, revealing substantial room for improvement and validating CAP as a meaningful testbed for future research.

2 Related Work

Benchmarks. Browser-based agents are studied as a general interface for real web tasks, motivating benchmarks that address diverse functional demands Deng et al. (2023); Zhou et al. (2023); Hallominate and Skyvern (2025). Earlier benchmarks predominantly focused on programmatic interaction and structural information extraction within DOM-based environments Liu et al. (2018); Chen et al. (2021). Subsequently, richer multimodal signals like vision were introduced. This enriched agent interaction and brought new challenges for evaluation Xu et al. (2025); Song et al. (2025). Driven by the demand for practical utility, evaluation has evolved to encompass broader dimensions. Recent benchmarks have transitioned toward real-world, open environments to assess agent robustness against the stochastic nature of the live web Pan et al. (2024); Kara et al. (2025). Simultaneously, there is a growing emphasis on long-horizon and complex tasks, which require agents to sustain multi-step reasoning and maintain long-term memory over extended trajectories Gou et al. (2025). However, real-world browser tasks often involve extended interactions across multiple websites with substantial execution and perceptual demands. As shown in Table 1, existing benchmarks lack systematic evaluation of agents under such multifaceted, real-world conditions.

Evaluation Methodologies. Evaluation methods can be categorized by their grading mechanisms. Rule-based graders Mialon et al. (2023); Yoran et al. (2024) use deterministic comparators over canonicalized outputs, offering high precision but requiring task-specific scripts that hinder scalability across diverse tasks. To address this limitation, LLM-as-a-judge approaches He et al. (2024); Xue et al. (2025) apply rubric-based scoring to assess constraint satisfaction or answer quality. However, their reliance on static observation of final outputs makes them insufficient for validating complex claims or multi-step reasoning processes. More recently, agent-as-a-judge methods Gou et al. (2025) extend evaluation capabilities by implementing judges as active agents that can execute verification operations (e.g., fetching and examining cited sources). This shift from passive assessment to active verification enables comprehensive evaluation of execution trajectories, though at the cost of increased computational overhead.

3 CAP Construction Pipeline

We introduce CAP to evaluate agentic browsers on realistic cross-site tasks with non-trivial interactions. To efficiently construct this benchmark, we develop an automatic pipeline that generates plausible cross-site tasks with challenging interactions. The pipeline consists of four phases: 1) site card annotation (§ 3.1); 2) cluster sampling (§ 3.2); 3) task proposal (§ 3.3); and 4) task instantiation (§ 3.4).

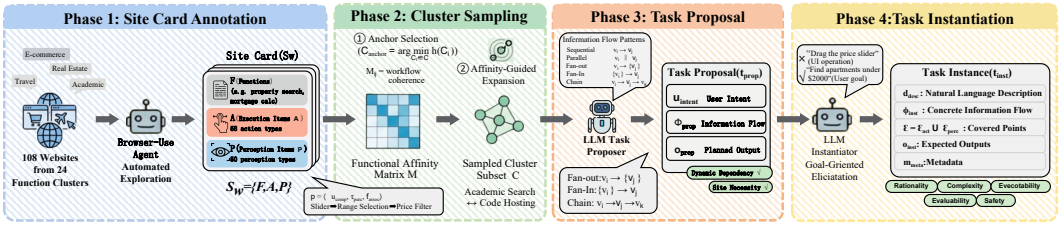

Figure 2: Overview of the CAP construction pipeline. The pipeline operates in four phases: (1) Site Card Annotation decomposes websites into structured functional representations; (2) Cluster Sampling selects coherent website subsets based on functional affinity; (3) Task Proposal generates abstract cross-site workflows via an LLM proposer; and (4) Task Instantiation transforms proposals into concrete, verifiable task instances.

3.1 Site Card Annotation

We use websites as the units for task construction. To capture what each website contributes to a task, we annotate it with a *site card*—a structured summary that records what functions the website offers and what complex interactions those functions involve.

For each website w, the site card is defined as $s_w = \langle \mathcal{F}, \mathcal{A}, \mathcal{P} \rangle$. $\mathcal{F}$ denotes the set of user-facing functions. Each function $f \in \mathcal{F}$ describes what users can accomplish on the website (e.g., property search, mortgage calculation). $\mathcal{A}$ denotes the set of execution items, defined as non-trivial interactions such as map dragging and map zooming. Each execution item $a \in \mathcal{A}$ is defined as $a = \langle u_{comp}, \tau_{act}, f_{assoc} \rangle$, where u_{comp} is the UI component, τ_{act} is the action type, and $f_{assoc} \in \mathcal{F}$ is the associated function. $\mathcal{P}$ denotes the set of perception items—elements requiring visual understanding such as chart trend recognition and image content understanding. Each perception item $p \in \mathcal{P}$ is defined as $p = \langle u_{comp}, \tau_{perc}, f_{assoc} \rangle$, where τ_{perc} is the perception type. By linking execution items a and perception items p to functions f_{assoc}, we can phrase tasks in terms of user goals (e.g., "find apartments under $2000") instead of UI operations (e.g., "drag the price slider"), making the tasks better reflect real user scenarios.

To determine the scope of action types τ_{act} and perception types τ_{perc} that CAP should cover, we ground our definitions in UI components, as modern web applications are fundamentally built upon them and each component naturally constrains the forms of user interaction it affords. We adopt the Ant Design component library[2] as a reference framework, as it is widely used in industry and provides a systematic categorization of common UI components. Specifically, we retain its core categories (*Navigation*, *Data Entry*, *Data Display*, and *Feedback*), exclude *General* and *Layout* components that primarily involve basic styling, and augment the taxonomy with *Data Visualization*, *Media*, and *Editor* components that frequently appear in real-world websites but are not explicitly covered by the library. Based on this grounded component taxonomy, domain experts identify 64 action types and 56 perception types. The complete taxonomy is provided in Appendix B.

To annotate site cards s_w efficiently, we use Browser-Use[3], an open-source browser agent (prompt details in Appendix C.1). Given a website w as input, the agent automatically explores the website, identifies user-facing functions $\mathcal{F}$, and catalogs execution items $\mathcal{A}$ and perception items $\mathcal{P}$, producing s_w as output. To ensure broad coverage of real-world web applications, we define 24 functional clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_{24}\}$ commonly encountered in browser agent tasks: e-commerce, travel booking, academic search, real estate, code hosting, video streaming, news media, healthcare, and others. We select high-traffic websites w within each cluster C_i based on SimilarWeb rankings, yielding 139 candidates. After excluding websites that restrict automated access or require mandatory login, we obtain 108 site cards covering 1,167 functions, 58 execution item types τ_{act} (90.6% coverage), and 50 perception item types τ_{perc} (89.2% coverage).

[2] https://ant.design/

[3] https://github.com/browser-use/browser-use

3.2 CLUSTER SAMPLING

In this stage, we select which s_w to combine into cross-site tasks from the functional clusters $\mathcal{C}$. Not all combinations make sense—browsing Zillow and then searching GitHub feels disjointed, whereas reading arXiv and then locating code on GitHub is a natural workflow. To construct realistic tasks, we assess which combinations are coherent. Since websites $w \in C_i$ share similar functionality, coherence between clusters C_i and C_j generalizes to their member websites. This allows us to assess coherence once per cluster pair rather than per website pair—we term this cluster-level coherence *functional affinity*.

To quantify functional affinity, we prompt a language model to assess pairwise cluster compatibility based on the plausibility of information flow scenarios, producing an affinity matrix $\mathbf{M}$ where $\mathbf{M}_{ij}$ represents the likelihood that clusters C_i and C_j form coherent workflows (prompt details in Appendix C.2). For instance, *academic search* and *code hosting* receive high $\mathbf{M}_{ij}$ since researchers frequently find papers and then locate implementations, whereas *real estate* and *code hosting* receive low $\mathbf{M}_{ij}$ due to the absence of natural information flow. Based on $\mathbf{M}$, we sample a cluster subset $\mathcal{C}^* \subseteq \mathcal{C}$ through two steps:

Step 1: Anchor selection. To ensure balanced coverage, we maintain historical usage statistics $h(C_i)$ that record how often cluster C_i has appeared in generated tasks. We select the cluster with the lowest usage as the anchor, preventing over-representation of common domains:

$$C_{\text{anchor}} = \arg\min_{C_i \in \mathcal{C}} h(C_i) \tag{1}$$

Step 2: Affinity-guided expansion. Starting from $\mathcal{C}^* = \{C_{\text{anchor}}\}$, we iteratively add clusters that exhibit high functional affinity with the anchor (i.e., $\mathbf{M}_{\text{anchor},i}$ exceeds a threshold η) while prioritizing those with low historical co-occurrence $h(C_i, C_j)$.

The resulting subset $\mathcal{C}^*$ forms a coherent task domain, which we pass to the next stage for task proposal.

3.3 TASK PROPOSAL

After identifying a coherent cluster subset $\mathcal{C}^*$, the goal of this stage is to define what cross-site tasks should be performed within this domain. Manually authoring such tasks is labor-intensive and difficult to scale. We observe that the functions $\mathcal{F}$ recorded in site cards provide natural building blocks for task composition: by explicitly exposing what each website can do, we can prompt a language model to automatically combine functions across sites into coherent workflows (see Appendix C.3 for prompt details and Appendix E for information flow patterns).

Given a sampled cluster subset $\mathcal{C}^*$ and site cards s_w of websites within $\mathcal{C}^*$, each generated proposal $t_{prop} = \langle u_{intent}, o_{prop}, \phi_{prop} \rangle$ contains three components: the user intent u_{intent} describing who needs the task and why, the planned output o_{prop} defining what the user expects to receive, and the information flow ϕ_{prop} specifying how data transfers between websites as a directed acyclic graph (DAG) over workflow steps.

Using this approach, we generate 600 task proposals (5 proposals per sampled cluster subset over 120 sampling rounds). Eight annotators manually reviewed all proposals, assessing whether user intent is meaningful and cross-site information is dynamically obtained during execution rather than preset. Results show 512 proposals (85.3%) meet these criteria.

3.4 TASK INSTANTIATION

Given task proposals t_{prop} from the previous stage, we instantiate each into a concrete benchmark task $t_{inst} = \langle d_{desc}, \phi_{inst}, \mathcal{E}, o_{inst}, m_{meta} \rangle$, comprising: a natural language description d_{desc} as a first-person user request, a concrete information flow ϕ_{inst} with specific entities and values, covered points $\mathcal{E} = \mathcal{E}_{act} \cup \mathcal{E}_{perc}$ denoting execution and perception items, expected outputs o_{inst}, and metadata m_{meta} recording involved websites and functions.

To generate t_{inst}, we select execution items $a \in \mathcal{A}$ and perception items $p \in \mathcal{P}$ from relevant site cards to include in $\mathcal{E}$, then prompt a language model to instantiate ϕ_{prop} into ϕ_{inst} by binding abstract

Category	Types	Avg	Min	Max
Clusters	24	2	1	5
Websites	108	3	2	14
Functions	405	5	2	15
Execution	48	7	2	20
Perception	36	4	2	15

Table 2: Statistics of our benchmark. Types refers to the total distinct types across all queries. Avg, Min, and Max refer to the number of operations/elements per task.

slots to concrete entities (prompt details in Appendix C.4). A key principle is *goal-oriented elicitation*: descriptions express user goals (e.g., "find what people are saying about this product") rather than UI operations, ensuring complex interactions are triggered implicitly through task requirements. The outputs o_{inst} serve dual purposes: providing value to users and supporting verification, where each covered point $e \in \mathcal{E}$ links to at least one output reflecting its result.

After manual review for rationality, complexity, executability, evaluability, and topic safety (details in Appendix D), we finalize 420 task instances (192 public, 228 private). The benchmark spans 24 task clusters and 108 websites, covering 405 functions with 82.8% execution item coverage and 72% perception item coverage. Each task involves 2 clusters and 3 websites on average. Detailed statistics are in Table 2 and Figure A.2.

4 Automated Evaluation Protocol

To evaluate browser agents on composite, cross-site tasks, there are two key challenges: (1) final-answer accuracy alone cannot reveal whether agents struggle with complex UI operations (e.g., slider manipulation) or perceptual reasoning (e.g., trend identification); and (2) manually authoring evaluation scripts for hundreds of heterogeneous tasks is prohibitively expensive. We address both challenges through a unified framework that leverages the covered points annotated during task construction (§3) to automatically generate fine-grained, verifiable evaluation criteria.

4.1 Problem Formulation

Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ denote the set of task instances in CAP. Recall from §3.4 that each task $t \in \mathcal{T}$ is associated with a set of covered points $E = E_{\text{act}} \cup E_{\text{perc}}$, where $E_{\text{act}} = \{e_1^a, \ldots, e_m^a\}$ denotes execution points—non-trivial interactive operations the task requires (e.g., date-range filtering, map region selection)—and $E_{\text{perc}} = \{e_1^p, \ldots, e_k^p\}$ denotes perception points—elements requiring visual understanding (e.g., chart trend recognition, table data extraction). Each covered point $e \in E$ is formally defined as a tuple $e = (id, \tau, d)$, where *id* is a unique identifier, $\tau \in \{\texttt{action}, \texttt{perception}\}$ indicates the type, and d is a natural language description. Given an agent's response $\mathcal{A}$ to task t, our goal is to compute a score $S(t, \mathcal{A}) \in [0, 1]$ that reflects both overall task completion and fine-grained capability along the execution and perception dimensions.

4.2 Evaluation Script Generation

Given the scalability bottlenecks inherent in manual verification, we employ an LLM-based code generation pipeline, as illustrated in Figure 3. Guided by the detailed instruction prompt provided in Appendix F, the generator synthesizes a Python evaluation script $\mathcal{S}$ by conditioning on the task description D and the set of covered points E. To ensure operational safety and standardization, the generation process is strictly constrained by an API whitelist Ω and a reference template $\mathcal{R}$. The resulting script $\mathcal{S}$ instantiates the rubric tree structure G (defined in §4.3), encapsulating both leaf-node verification and hierarchical score aggregation. Furthermore, to guarantee robustness, we implement an iterative refinement mechanism: execution failures trigger a self-debugging loop, complemented by a self-reflection phase that validates logical correctness against edge cases.

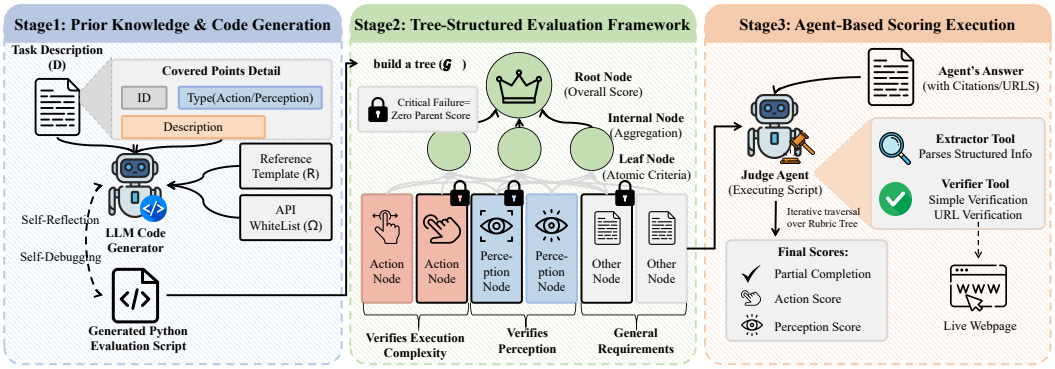

Figure 3: Overview of the proposed evaluation framework. The pipeline operates in three phases: (1) Prior Knowledge & Code Generation, where an LLM generates evaluation scripts from task descriptions; (2) Tree-Structured Evaluation Framework, which constructs a hierarchical rubric tree; and (3) Agent-Based Scoring Execution, where a judge agent performs extraction and verification to compute final scores.

4.3 Rubric Tree Construction

Each evaluation script $\mathcal{S}$ implements a rubric tree $G = (V, \mathcal{E})$, a rooted tree where each node corresponds to a verification criterion. Leaf nodes $V_{\text{leaf}} \subset V$ represent atomic criteria, each producing a binary score $s(v) \in \{0, 1\}$; internal nodes aggregate scores from their children; and the root node v_{root} yields the final task score. To enable capability diagnosis, we classify leaf nodes into three categories: action nodes V_{act} verify execution of complex UI operations, perception nodes V_{perc} verify extraction of visually-presented information, and other nodes V_{other} verify general requirements such as output format. Nodes are classified as critical—prerequisites whose failure invalidates downstream results, zeroing the parent score—or non-critical, which permit partial credit. This classification is automatically determined by the LLM during script generation based on task-specific dependency analysis.

The rubric tree is constructed automatically from the task description D and covered points E. Each covered point $e \in E$ maps to one or more leaf nodes according to its type: action-type points generate nodes in V_{act}, while perception-type points generate nodes in V_{perc}. This mapping enables indirect verification of whether agents successfully complete the complex execution and perception operations required by each task.

4.4 Score Computation

Given the rubric tree G and an agent's response $\mathcal{A}$, scoring proceeds in two phases. Each leaf node $v \in V_{\text{leaf}}$ is first evaluated by a judge agent—an LLM-based tool operating in two modes: extraction + verification, which parses structured information from $\mathcal{A}$ and applies deterministic checks, and URL verification, which fetches cited web pages to verify supporting content. Internal nodes then aggregate child scores via a gate-then-average strategy. Let $C(v)$ denote the children of node v, partitioned into critical nodes $K(v)$ and non-critical nodes $N(v)$. The aggregated score is:

$$s(v) = \begin{cases} 0 & \text{if } \exists u \in K(v) : s(u) < 1 \\ \bar{s}_N & \text{if } \forall u \in K(v) : s(u) = 1 \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

where $\bar{s}_N = \frac{1}{|N(v)|} \sum_{u \in N(v)} s(u)$ when $N(v) \neq \emptyset$. We report four metrics to capture different aspects of performance.

Method	Core Performance (%)		Complexity (%)		Output	
	Partial Completion	Success Rate	Complex A	Complex P	Time (min)	Len. ($\times 10^3$)
Commercial Browser Agents						
Genspark	16.0 $\pm$2.0	5.0 $\pm$1.0	30.0 $\pm$2.1	28.0 $\pm$1.8	1.7	3.37
Dia	19.0 $\pm$2.0	4.0 $\pm$1.0	28.0 $\pm$1.5	23.0 $\pm$2.2	1.1	3.00
Fellou	**24.0** $\pm$2.0	**7.0** $\pm$2.0	28.0 $\pm$1.9	25.0 $\pm$2.1	22.1	3.47
Open Source Agents						
GPT-5	15.0 $\pm$1.0	2.0 $\pm$1.0	29.0 $\pm$2.0	23.0 $\pm$1.5	15.3	2.95
Claude-4.5-Sonnet	21.0 $\pm$3.0	5.0 $\pm$2.0	**32.0** $\pm$2.4	**32.0** $\pm$2.5	29.9	5.52

Table 3: Main results on CAP benchmark. All performance metrics are reported in percentages (%). Error bars (indicated in gray) represent standard deviation. Note that **GPT-5** and **Claude** in the Open Source section refer to the *Browser-use* agent framework driven by the respective models.

5 EXPERIMENTS

5.1 SETUP

Evaluated Systems. We evaluate both commercial and open-source browser agent systems on the CAP benchmark consisting of 198 queries (see §3 for dataset construction details). Given the diversity of task requirements, we select systems capable of performing complex web interactions across multiple domains, including both commercial products and open-source implementations. We evaluate four commercial browser agent products: Genspark[4], Fellou[5], and Dia[6]. All commercial systems are accessed through official interfaces with default configurations. Additionally, we assess Browser-Use[3], an open-source browser automation framework deployed with GPT-5 as the backbone LLM via the OpenAI API.

Evaluation Metrics. We report metrics computed following the evaluation methodology in §4. Our primary metrics are **Partial Completion**, indicating whether agents achieve at least partial task completion, and **Success Rate**, measuring full task completion. To understand how performance varies across task characteristics, we categorize results by complexity: **Complex-A** evaluates task nodes with complex execution scenarios, and **Complex-P** assesses task nodes with complex perception scenarios. To further contextualize system performance, we report the average execution **Time** (in minutes) and output **Length** (in thousands of words) as supplementary output statistics.

5.2 MAIN RESULTS

Finding 1: State-of-the-art browser agents demonstrate limited capabilities on complex execution and perception tasks. The main results are presented in Table 3. As CAP specifically focuses on challenging scenarios involving complex execution and perception, the results reflect the difficulty of these tasks. The best-performing system Fellou achieves 24.0% partial success rate and 7.0% full success rate, while GPT-5 shows 15.0% partial success and 2.0% success rate. The limited completion rates across all evaluated systems underscore the significant challenges present in handling complex browser automation tasks.

Finding 2: Perception tasks are more challenging than execution tasks for current browser agents. We analyze performance across different complexity dimensions to identify specific bottlenecks. For tasks requiring complex execution operations (Complex A), systems achieve scores ranging from 28.0% to 32.0%, with Claude achieving the highest score (32.0%). For complex perception tasks (Complex P), scores range from 23.0% to 32.0%, with Claude performing best (32.0%). Complex P tasks show an average performance of 26.7% compared to 30.4% for Complex A tasks—a 3.7 percentage point gap.

[4] https://www.genspark.im/

[5] https://fellou.ai/

[6] https://www.diabrowser.com/

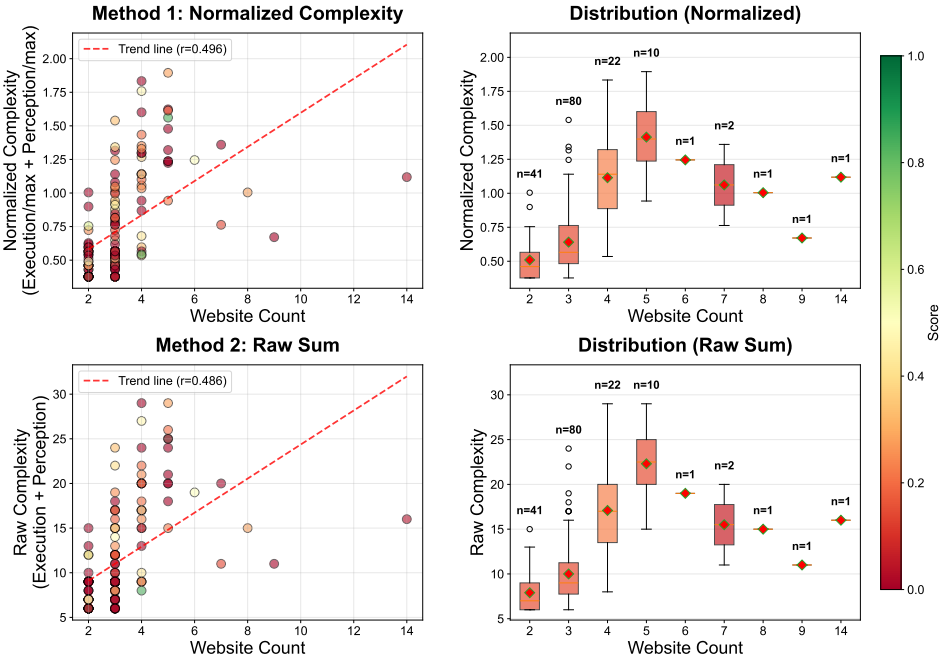

Figure 4: Relationship between website count and task complexity under two definitions. Left: scatter plots with linear fits. Right: boxplots grouped by website count. Colors encode task scores.

Finding 3: Extended reasoning is critical for navigating complexity. Top agents like Fellou and Claude sacrifice efficiency (running > 20 mins or generating $> 5k$ tokens) to achieve superior success (7.0%) or complexity handling (32.0%), suggesting that rigorous, time-intensive processing is essential for robust web agents.

5.3 Complexity Analysis

Finding 4: Fine-grained visual reasoning and precise state inference pose the greatest challenges.

Figure A.1(a-b) reveals the most challenging task components in our benchmark. For execution operations, node manipulation (0.087) and date range selection (0.110) exhibit the lowest performance. In perception tasks, value reading achieves a near-zero score (0.000), while expansion state awareness (0.092) also shows significant difficulty, suggesting that extracting numerical information and inferring UI state changes remain major bottlenecks for current web agents.

To quantify task complexity, we define two metrics. **Raw-sum complexity** directly sums the number of execution operations and perception requirements for each task. **Normalized complexity** separately divides the execution count by the maximum execution count and the perception count by the maximum perception count observed in the benchmark, then sums these two normalized values, yielding a metric in $[0, 2]$.

Figure 4 examines the relationship between task complexity and performance. Point colors encode task scores (0–1). Notably, **task difficulty (score) shows no significant correlation with complexity quantity**. High-performing tasks (green points) and low-performing tasks (red points) distribute across the entire complexity range under both definitions.

5.4 Cross-site Analysis

Finding 5: Domain-specific websites with specialized data structures present disproportionate challenges.

Figure A.1(c) demonstrates significant performance variance across different websites. Cdc.gov (0.033) and google.com/flights (0.048) present the greatest challenges, while accuweather.com (0.139) and wikipedia.org (0.131) achieve relatively higher scores.

Figure 4 reveals a moderate positive correlation between website count and task complexity (normalized: r=0.496; raw-sum: r=0.486), indicating that tasks involving more websites tend to require more interaction steps and perceptual operations.

The distribution shows that most tasks involve 2–5 websites (n=80, 41, 22, 10 respectively), reflecting realistic multi-platform workflows. The increasing variance in complexity for higher website counts demonstrates the diversity of cross-site task designs in our benchmark.

6 Conclusion

We introduce CAP, a benchmark evaluating browser agents on cross-site workflows, complex actions, and perception. We employ a Verifiable Agent-as-a-Judge framework for fine-grained diagnosis. Experiments reveal that current agents struggle with complex interactions, highlighting visual perception as a critical bottleneck.

References

Xingyu Chen, Zihan Zhao, Lu Chen, JiaBao Ji, Danyang Zhang, Ao Luo, Yuxuan Xiong, and Kai Yu. Websrc: A dataset for web-based structural reading comprehension. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 4173–4185, 2021.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *NeurIPS*, 2023.

Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanev, Botao Yu, Bernal Jiménez Gutiérrez, Yiheng Shu, et al. Mind2web 2: Evaluating agentic search with agent-as-a-judge. *arXiv preprint arXiv:2506.21506*, 2025.

Halluminate and Skyvern. Webbench: Ai web browsing agent benchmark, 2025. `https://webbench.ai/`.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Su Kara, Fazle Faisal, and Suman Nath. Warex: Web agent reliability evaluation on existing benchmarks. *arXiv preprint arXiv:2510.03285*, 2025.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations (ICLR)*, 2018. URL `https://arxiv.org/abs/1802.08802`.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Yichen Pan, Dehan Kong, Sida Zhou, Cheng Cui, Yifei Leng, Bing Jiang, Hangyu Liu, Yanyi Shang, Shuyan Zhou, Tongshuang Wu, et al. Webcanvas: Benchmarking web agents in online environments. *arXiv preprint arXiv:2406.12373*, 2024.

Yixiao Song, Katherine Thai, Chau Minh Pham, Yapei Chang, Mazin Nadaf, and Mohit Iyyer. Bearcubs: A benchmark for computer-using web agents. *arXiv preprint arXiv:2503.07919*, 2025.

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

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Kevin Xu, Yeganeh Kordi, Tanay Nayak, Adi Asija, Yizhong Wang, Kate Sanders, Adam Byerly, Jingyu Zhang, Benjamin Van Durme, and Daniel Khashabi. Turkingbench: A challenge benchmark for web agents. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3694–3710, 2025.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. *arXiv preprint arXiv:2504.01382*, 2025.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. Assistantbench: Can web agents solve realistic and time-consuming tasks? *arXiv preprint arXiv:2407.15711*, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A EXPERIMENTAL RESULTS

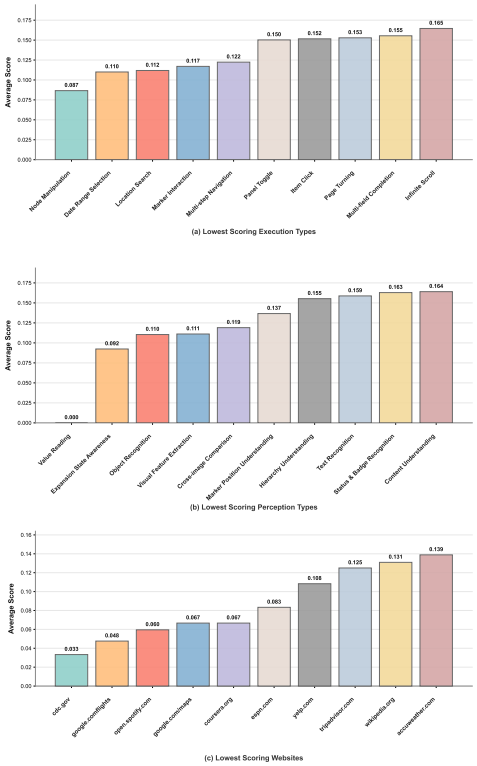

Figure A.1: Distributions over the bottom-10 (by mean score) complex-execution types, complex-perception types, and websites.

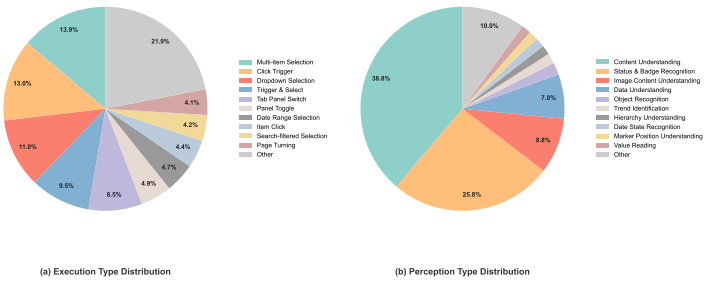

Figure A.2: Distribution of (a) complex execution types and (b) complex perception types in our benchmark.

B TAXONOMY OF COMPLEX INTERACTION TYPES

We define a taxonomy comprising 64 action types and 56 perception types. The taxonomy is grounded in the Ant Design component library, excluding basic categories (*General*, *Layout*) and extending with *Data Visualization*, *Media*, and *Editor* to cover complex interactions in real-world applications. The complete lists are provided in Table B.1 and Table B.2.

B.1 ACTION TYPES

Action types (τ_{act}) define the interactive operations that an agent must perform on UI components. We categorize these into seven primary categories: *Navigation*, *Data Entry*, *Data Display*, *Feedback*, *Data Visualization*, *Media*, and *Editor*.

ID	Component	Action Type	Description
Navigation			
AT01	Dropdown	Trigger & Select	Click or hover to trigger dropdown and select target item
AT02	Dropdown	Multi-level Navigation	Navigate through nested dropdown menus to locate target
AT03	Menu	Submenu Expansion	Expand multi-level submenus via hover or click (e.g., mega menu)
AT04	Pagination	Page Turning	Click page numbers or prev/next buttons to switch pages
AT05	Pagination	Page Jump	Input specific page number to jump directly
AT06	Steps	Multi-step Navigation	Execute prev/next transitions in wizard workflows
AT07	Tabs	Tab Panel Switch	Click tab labels to switch content panels
AT08	Tabs	Tab Bar Scrolling	Scroll tab bar to reveal hidden tab options
AT09	Breadcrumb	Path Navigation	Click breadcrumb items to navigate to target hierarchy
Data Entry			
AT10	Checkbox	Multi-item Selection	Select multiple checkboxes for combined filtering
AT11	Checkbox	Select All/Inverse	Execute select all, inverse selection, or deselect all
AT12	Cascader	Hierarchical Selection	Expand levels sequentially to complete selection (e.g., region)
AT13	Cascader	Search-based Selection	Search keywords to directly locate and select deep options
AT14	DatePicker	Single Date Selection	Open calendar panel and select date with quick options
AT15	DatePicker	Date Range Selection	Select start/end dates with shortcuts (e.g., last 7 days)
AT16	DatePicker	DateTime Selection	Select both date and specific time simultaneously
AT17	Form	Multi-field Completion	Identify and complete forms with multiple input types
AT18	Select	Dropdown Selection	Open dropdown list for single or multi-select scenarios
AT19	Select	Search-filtered Selection	Filter options via keywords then select target item
AT20	Slider	Single Value Drag	Drag slider to select a single value
AT21	Slider	Range Selection	Drag dual sliders to set value range (e.g., price range)
AT22	TimePicker	Time Selection	Select specific hour/minute via scroll or input
AT23	TreeSelect	Tree Node Selection	Expand tree structure and select target nodes
AT24	Upload	File Upload	Upload files via click or drag-and-drop
Data Display			
AT25	Calendar	Date Selection	Click to select target date in calendar view
AT26	Calendar	View Switch	Switch between different month or year views
AT27	Card	Click Trigger	Click card for details or action buttons (favorite, add to cart)
AT28	Card	Hover Trigger	Hover to reveal quick actions or additional information
AT29	Carousel	Slide Switch	Switch via arrows/indicators or drag gestures
AT30	Collapse	Panel Toggle	Click header to expand/collapse, including accordion mode
AT31	Image	Preview Operations	Open thumbnail for zoom, rotate, or image switching
AT32	Popover	Popover Interaction	Trigger popover via click/hover and perform actions within
AT33	Table	Column Sorting	Click header to sort by column in ascending/descending order
AT34	Table	Column Filtering	Use column filters to filter table data by conditions
AT35	Table	Row Operations	Select rows, expand details, or click inline action buttons
AT36	Tree	Node Manipulation	Expand/collapse nodes, select, or drag to adjust structure
AT37	List	Infinite Scroll	Scroll to bottom to trigger loading more items
AT38	List	Item Click	Click list item to enter corresponding detail page
AT39	List	Drag Reorder	Drag list items to adjust display order
Feedback			
AT40	Drawer	Drawer Operations	Open/close side drawer for filtering or form operations
AT41	Modal	Modal Operations	Open/close modal dialog for form or content selection
AT42	Popconfirm	Confirmation Action	Trigger confirmation popover and execute confirm/cancel
Data Visualization			
AT43	Chart	Data Point Hover	Hover chart elements to display data tooltip
AT44	Chart	Legend Filtering	Click legend to toggle data series visibility
AT45	Chart	Zoom & Pan	Box-select zoom or drag to pan for data details
AT46	Chart	Data Drill-down	Click chart elements to drill into finer-grained data
AT47	Map	Zoom & Pan	Adjust map view via scroll zoom and drag pan
AT48	Map	Marker Interaction	Click map markers to view POI details
AT49	Map	Location Search	Search place names and navigate to location
AT50	Map	Route Planning	Set origin/destination and plan navigation route
AT51	Map	3D View Adjustment	Adjust pitch and bearing angles via Ctrl+drag
AT52	Map	Street View	Enter street view, click to move, drag to look around
AT53	Panorama	Panoramic Browsing	Drag to rotate view, scroll to zoom details
AT54	Panorama	Hotspot Navigation	Click hotspot markers to switch panorama scenes
AT55	Model3D	Model Interaction	Drag to rotate, scroll to zoom, switch configurations

ID	Component	Action Type	Description
Media			
AT56	Video	Playback Control	Play/pause video, drag progress bar to seek
AT57	Video	Playback Settings	Adjust volume, speed, quality, subtitles, or fullscreen
Editor			
AT58	TextEditor	Text Editing	Edit rich text including formulas and special symbols
AT59	TextEditor	Format & Content	Use toolbar for formatting or insert images/links/tables
AT60	Spreadsheet	Cell Editing	Select and edit spreadsheet cell contents
AT61	Spreadsheet	Formula Input	Enter formulas in cells for calculations
AT62	Spreadsheet	Drag Fill	Drag cell edges to batch fill data
AT63	Canvas	Element Manipulation	Select, move, and resize elements on canvas
AT64	Canvas	Node Connection	Drag from anchors to create connection lines

Table B.1: Action types for UI components.

B.2 Perception Types

Perception types (τ_{perc}) characterize the visual understanding capabilities required to interpret UI states and content. These span four primary categories: *Data Display*, *Data Visualization*, *Media*, and *Editor*.

ID	Component	Perception Type	Description
Data Display			
PT01	Tooltip	Content Understanding	Hover to obtain and understand tooltip text content
PT02	Tooltip	Type Recognition	Distinguish help, warning, error, and other tooltip types
PT03	Calendar	Date Marker Recognition	Identify dates with event or price markers on calendar
PT04	Calendar	Date State Recognition	Identify available, disabled, and selected date states
PT05	Card	Image Content Understanding	Recognize objects, scenes, and product types in card images
PT06	Card	Visual Feature Extraction	Extract color, material, and style features from card images
PT07	Card	Status & Badge Recognition	Identify status markers (sold out, promo) and badge values
PT08	Carousel	Image Content Understanding	Recognize carousel image themes and promotional content
PT09	Carousel	Embedded Text Recognition	OCR text embedded within carousel images
PT10	Carousel	Position Awareness	Perceive current carousel position (n of total)
PT11	Image	Object Recognition	Identify specific objects and their categories in images
PT12	Image	Scene Understanding	Understand scene semantics (indoor/outdoor, purpose)
PT13	Image	Feature Extraction	Extract color, material, and style visual features
PT14	Image	Text Recognition	OCR text embedded within images
PT15	Image	Cross-image Comparison	Compare similarity across images, identify key differences
PT16	Collapse	Expansion State Awareness	Identify whether collapse panel is expanded or collapsed
PT17	Popover	Content Understanding	Understand comprehensive information in popover cards
PT18	List	Content Understanding	Understand list item content and extract key information
PT19	List	Sort State Awareness	Identify current list sorting method and order
PT20	Table	Data Understanding	Understand table data semantics and row-column relations
PT21	Table	Cell State Awareness	Identify row/cell states (highlighted, disabled, abnormal)
PT22	Table	Sort State Awareness	Identify current sort column and direction
PT23	Table	Cross-cell Understanding	Understand data relations across rows or columns
PT24	Tree	Hierarchy Understanding	Understand tree structure levels and parent-child relations
PT25	Tree	Node State Awareness	Identify node selection, expansion, and disabled states
PT26	Tree	Path Understanding	Understand complete path from root to target node
Data Visualization			
PT27	Chart	Value Reading	Accurately read specific values from visual positions
PT28	Chart	Trend Identification	Identify data trends (rising, falling, fluctuating, turning)
PT29	Chart	Multi-series Comparison	Compare differences and relations across data series
PT30	Chart	Proportion Understanding	Understand proportional relations in pie/donut charts
PT31	Chart	Extrema Identification	Find maximum/minimum points and corresponding positions
PT32	Chart	Anomaly Detection	Detect outliers, anomalies, and sudden changes in data
PT33	Chart	Legend Mapping	Understand correspondence between legend and data series
PT34	Map	Marker Position Understanding	Understand geographic positions and spatial relations
PT35	Map	Spatial Distribution Awareness	Identify distribution patterns, clusters, and coverage

ID	Component	Perception Type	Description
PT36	Map	Route Understanding	Understand route path, distance, and estimated time
PT37	Map	Region Boundary Understanding	Understand regional divisions and administrative boundaries
PT38	Map	View State Awareness	Identify current view mode (2D, 3D, street) and tilt degree
PT39	Map	Street View Understanding	Understand buildings, roads, and environment in street view
PT40	Map	Street View Orientation	Understand directional orientation in street view
PT41	Map	3D Occlusion Understanding	Understand occlusion relations between buildings in 3D
PT42	Panorama	Spatial Layout Understanding	Understand room layout and spatial structure in panorama
PT43	Panorama	Hotspot Recognition	Identify interactive hotspot marker positions
PT44	Panorama	Orientation Awareness	Understand current panorama viewing direction
PT45	Panorama	Environment Understanding	Understand decoration style, furniture in panorama
PT46	Model3D	Structure Understanding	Understand 3D model components and overall structure
PT47	Model3D	Material Perception	Identify surface materials and colors of model
PT48	Model3D	Proportion Perception	Understand relative proportions and size relations
PT49	Model3D	Viewpoint Recognition	Identify current viewing angle (front, side, back, etc.)
Media			
PT50	Video	Frame Content Understanding	Recognize content, objects, and scenes in video frames
PT51	Video	Playback State Awareness	Identify playback state (playing, paused, buffering)
PT52	Video	Progress Position Awareness	Understand current position within total duration
Editor			
PT53	Spreadsheet	Data Understanding	Understand spreadsheet data semantics and calculation logic
PT54	Spreadsheet	Format Semantics	Recognize conditional formatting and color-coded meanings
PT55	Canvas	Element Recognition	Identify shapes, lines, and text elements on canvas
PT56	Canvas	Relation & Layout Understanding	Understand element connections, logic flow, and layout

Table B.2: Perception types for UI components.

C Prompts for Task Construction

This appendix presents the prompts used in each phase of the CAP construction pipeline.

C.1 Site Card Annotation Prompt

This prompt guides the annotation of site cards that document website capabilities. Given a target website URL, the annotation agent browses the site to identify functional modules, then enumerates complex execution items and perception items by analyzing the rendered HTML structure. The key insight is that complex UI components can be identified through their HTML signatures—for instance, a date range picker can be recognized by its characteristic DOM structure and attributes (e.g., `<input type="date">`, calendar container classes) without actually completing date selection.

Site Card Annotation Prompt

```
You are a frontend evaluation expert agent with browser automation capabilities.
Your task is to visit the target website and systematically document its functions,
UI components, and complex interaction points.

INPUT:
- A target website URL to analyze
- A component taxonomy reference defining action types and perception types

IDENTIFICATION METHOD:

Identify complex UI components through HTML analysis:
- Inspect DOM structure, element tags, class names, and attributes
- Match against known component signatures from the taxonomy
- Record components based on their HTML presence, not execution success

Examples of HTML-based identification:
- Date picker: <input type="date">, calendar containers, date-range classes
- Map component: <div class="map-container">, Leaflet/Google Maps scripts
- Video player: <video>tags, player control elements, playback-rate options
```

```
- Slider: <input type="range">, slider track/handle elements
- Expandable section: aria-expanded attributes, collapse/accordion classes

IMPORTANT: Your task is to IDENTIFY components through HTML inspection, not to
COMPLETE complex operations. A component should be recorded if its DOM signature
is present, regardless of whether you can fully operate it.

VERIFICATION PRINCIPLES:

1. Record What You Can Identify in HTML
  - DOM structure matches known component pattern -> record
  - Relevant classes/attributes present -> record
  - No HTML evidence found -> do not record

2. When In Doubt, Leave It Out
  - Missing a real feature: causes incompleteness (acceptable)
  - Recording a non-existent feature: causes downstream failure (unacceptable)

STEP 1: IDENTIFY FUNCTIONS (F)

Browse the website and identify user-facing functional modules.

Naming: Use verb + domain-specific noun
- WRONG: "Search", "View Details"
- RIGHT: "Video Search", "Property Listing", "Course Reviews"

Tags (combinable):
- [Core]/[Standard]: whether this is the primary draw of the site
- [Read]/[Write]: whether operations have persistent effects

STEP 2: IDENTIFY EXECUTION ITEMS (A)

For each function, identify complex UI components through HTML inspection.

Two roles in tasks:

1. Parameter Constraint - specifies how to perform an operation
  - "Filter by price $50-100, rating above 4.5, pet-friendly"
  - "Play at 1.5x speed with subtitles enabled"

2. Navigation Bridge - required step to access deeper capabilities
  - "Enter detail page" -> enables further actions/perceptions
  - "Expand fee breakdown" -> reveals hidden pricing

Complexity Threshold:
- INCLUDE: Multi-criteria filtering, parameter configuration, map interactions,
media control, complex forms
- EXCLUDE: Simple clicks, single inputs, basic navigation

STEP 3: IDENTIFY PERCEPTION ITEMS (P)

For each function, identify elements requiring visual understanding.

Two roles in tasks:

1. Information Output - agent must extract and return this
  - "What is the monthly rent and deposit?"
  - "Describe the price trend over 3 months"

2. Filter Criteria - perception determines selection
  - "Find pet-friendly apartments" (requires policy recognition)
  - "Find items with recent price drops" (requires trend detection)

Complexity Threshold:
- INCLUDE: Status recognition, structured data extraction, chart interpretation,
spatial understanding
- EXCLUDE: Direct text reading, fixed-position extraction

OUTPUT FORMAT:

functions.csv:
  id, name, tags, description
```

```
  // description: core capabilities and typical use cases

execution_items.csv:
  id, component, action_type, associated_functions, description
  // description: location path and concrete usage scenario

perception_items.csv:
  id, component, perception_type, associated_functions, description
  // description: location path and what to extract or verify
```

C.2 Functional Affinity Assessment Prompt

This prompt assesses pairwise cluster compatibility to produce the affinity matrix $\mathbf{M}$. Given two functional clusters as input, the model evaluates whether they can form coherent cross-site workflows and outputs an affinity score with justification.

Functional Affinity Assessment Prompt

```
You are assessing whether two functional clusters can form coherent cross-site
workflows.

INPUT: Two functional clusters C_i and C_j, each with member websites and their
representative functions.

TASK: Evaluate the plausibility of information flow scenarios:
- Can information from C_i naturally serve as input for tasks in C_j?
- Can information from C_j naturally serve as input for tasks in C_i?
- Do real user workflows commonly involve both clusters?

RATING SCALE:
- HIGH: Frequent real-world workflows connect these clusters
   Example: Academic Search + Code Hosting (researchers find papers then locate
implementations)
- MEDIUM: Plausible but less common workflows exist
- LOW: No natural information flow
  Example: Real Estate + Code Hosting (no coherent user scenario)

OUTPUT: Affinity score (HIGH/MEDIUM/LOW) with brief justification.
```

C.3 Task Proposal Prompt

This prompt generates task proposals that define the semantic structure of cross-site workflows. Given a sampled cluster subset $\mathcal{C}^*$ and the corresponding site cards, the model produces an array of task proposals, each specifying user intent, information flow pattern, and planned outputs. The proposals serve as semantic skeletons for downstream instantiation.

Task Proposal Prompt

```
You are designing cross-site tasks for an AI browser agent benchmark. Your goal
is to create tasks that real users would genuinely want AI assistance with—NOT
artificial capability tests.

INPUT:
- Website cluster C containing functional clusters to use
- Site cards for each website, including: functions F, execution items A (complex UI
operations like map dragging, date picker interaction), perception items P (visual
understanding requirements like chart reading, table parsing)

CORE PRINCIPLE:
Imagine yourself as a user facing a tedious multi-site problem, thinking "I wish AI
could handle this." That authentic feeling is the starting point for a good task.
Do NOT design tasks just to test capabilities.

INFORMATION FLOW PATTERNS:
```

```
Compose tasks using these primitives:
- Sequential (A -> B): A's output feeds B's input
- Parallel (A || B): Same goal across platforms for comparison
- Fan-out (A -> {B1, B2}): One source, multiple follow-ups
- Fan-in ({A1, A2} -> B): Multiple sources, unified processing
- Chain (A -> B -> C): Multi-hop information refinement

AUTHENTICITY REQUIREMENTS:
- Task must address genuine user pain points (cross-platform switching, repetitive operations, scattered information)
- Manual completion should be tedious enough to warrant AI assistance
- Must require actual website operations (filtering, sorting, pagination, state verification), NOT just answerable by web search

CROSS-SITE NECESSITY:
- Information flow between websites must be natural, not contrived
- Each website must provide unique, essential information
- Removing any website should break the task (Skip Test)

DYNAMIC DEPENDENCY (CRITICAL):
All information passed between sites must be obtained during task execution, NOT preset in the task description.

Reject these anti-patterns:

1. Preset Keywords
  WRONG: "Find 'Roman Architecture' courses on Coursera, then search 'Basilica' and 'Forum' on Getty Museum"
  Problem: 'Basilica' and 'Forum' are preset, not discovered from Coursera

2. Hypothetical Linking
  WRONG: "Assuming CVPR is one of the conferences you found, search for hotels in San Francisco..."
 Problem: "Assuming" indicates the input is preset by task designer, not dynamically obtained

3. Isolated Outputs
  WRONG: "Check flu activity level on CDC, find immunity-boosting ingredients on WebMD, then find recipes on FoodNetwork"
  Problem: CDC output doesn't affect subsequent steps; removing it doesn't break the task

GOOD EXAMPLES:

1. "Find 5 Greek mythology artworks currently on display at Getty, note the artwork names and mythological figures, then find educational videos about EACH figure on Bilibili"
  Why good: Must obtain specific artwork/figure names from Getty before knowing what to search on Bilibili

2. "Check Saturday's precipitation probability on AccuWeather and AQI on EPA; if precipitation > 30% OR AQI > 50, find indoor oven recipes, otherwise find outdoor BBQ recipes"
 Why good: Weather data determines which recipe category to search—true conditional branching

OUTPUT FORMAT:
Return a JSON array of task proposals:
[
  {
    "task_id": "PROP-001",
    "user_scenario": {
      "persona": "...",  // Specific role, e.g., "ML researcher preparing a survey"
      "situation": "...",  // Concrete context triggering this task
      "pain_point": "...",  // Why manual execution is tedious
      "decision_goal": "..."  // Actionable outcome user seeks
    },
    "information_flow": {
```

```
      "pattern": "...",  // One of: sequential, parallel, fan-out, fan-in, chain,
mixed
      "description": "..."  // How data transfers between sites
    },
    "websites_involved": ["site1.com", "site2.com", ...],
    "function_points": ["site1.com:F1", "site2.com:F3", ...],
    "planned_outputs": ["...", "..."]  // What user expects to receive
  },
  ...  // More proposals
]
```

C.4 Task Instantiation Prompt

This prompt transforms each task proposal into a concrete, executable benchmark instance. Given a single proposal and the relevant site cards, the model produces a detailed task specification including natural language description, structured output fields, and covered execution/perception items. The instantiation process binds abstract slots to concrete entities while ensuring complex interactions are triggered implicitly through goal-oriented descriptions.

Task Instantiation Prompt

```
You are instantiating a task proposal into a concrete, executable, and verifiable
benchmark task.

INPUT:
- A single task proposal with user scenario and information flow
- Site cards with functions F, execution items A and perception items P for involved
websites

GOAL-ORIENTED ELICITATION:
Express requirements through user goals, NOT UI operations. This ensures complex
interactions are triggered implicitly while keeping task descriptions natural.

WRONG: "Use the duration filter to select videos under 10 minutes"
RIGHT: "Find tutorial videos under 10 minutes"

WRONG: "Scroll down to load more comments, then click expand"
RIGHT: "Check what users are discussing about this product"

WRONG: "Click the expand button to show the full description"
RIGHT: "Verify if the video description mentions the original paper"

WRONG: "Hover over the seller name to view reputation score"
RIGHT: "Find products from sellers with reputation above 4.8"

OUTPUT FIELD DESIGN:
Every output field must serve one of three verification roles:

1. User Value: Core information the user actually needs
  Examples: product name, course title, paper abstract

2. Condition Verification: Fields to validate filter criteria used in the task
  CRITICAL: If task says "rating >= 4.5", output MUST include the rating
  If task says "published within 2 years", output MUST include date
  If task says "distance under 5 miles", output MUST include distance

3. Provenance: URLs for source verification to prevent hallucination
  Every claimed fact should be traceable to a source URL

ROBUSTNESS GUIDELINES:

1. Prefer Range Filtering over Superlatives
  WRONG: "Find the most downloaded model" (sorting may not be supported)
  RIGHT: "Find 5 models with downloads exceeding 10,000"
  Reason: Superlatives require specific sorting features; range filters are more
robust
```

```
2. Prefer Relative Time over Absolute Dates
  WRONG: "Find events on June 20, 2026" (task expires)
  RIGHT: "Find events next weekend"
  Reason: Absolute dates make tasks time-sensitive and eventually invalid

3. Include Fallback Instructions for Dynamic Content
  "If this weekend has no events, check next weekend"
  "If the course is unavailable, find alternatives from the same instructor"
  Reason: Web content changes; fallbacks ensure task remains completable

4. Specify Quantities Explicitly
  WRONG: "Find some highly-rated restaurants"
  RIGHT: "Find 5 restaurants with rating above 4.5"
  Reason: Explicit quantities enable objective evaluation

5. Consolidate Output Requirements
   Place all output requirements at the END of task description, not scattered
throughout
   WRONG: "Find books on Goodreads, note title and author. Then check reviews on
StoryGraph, record the mood tags. Finally..."
   RIGHT: "Find books on Goodreads... check reviews on StoryGraph... Output for
each book: title, author, Goodreads rating, StoryGraph mood tags, and URLs for both
platforms."

TASK DESCRIPTION STYLE:
- First person ("I want...", "Help me...")
- Natural conversational tone, as if talking to an AI assistant
- 100-250 words
- Include context that implicitly triggers complex operations

OUTPUT FORMAT:
{
  "task_id": "TASK-001",
  "source_proposal": "PROP-001",  // Reference to the source proposal
  "task_description": "...",  // First-person natural language, 100-250 words
  "information_flow_summary": "...",  // e.g., "Site A -> Site B -> Site C"
  "core_outputs": [
    {
      "id": "o1",  // Unique identifier for this output field
      "name": "...",  // Field name, e.g., "listing_price"
      "type": "...",  // One of: text, number, boolean, url, list
      "source": "...",  // Website this field comes from
      "role": "..."  // One of: user_value, condition_verify, provenance
    },
    ...
  ],
  "covered_points": [
    {
      "point_id": "...",  // Format: website.com:Fx:Ay or website.com:Fx:Pz
      "ability_type": "...",  // e.g., "date range filter", "scroll loading"
      "trigger_method": "...",  // How task description triggers this ability
      "output_ref": "...",  // Which output field (e.g., "o2") verifies execution
      "verify_method": "..."  // How to verify, e.g., "check value >= threshold"
    },
    ...
  ],
  "metadata": {
    "websites_involved": ["site1.com", "site2.com", ...],
    "action_points": ["site1.com:F1:A2", ...],  // Covered execution items
    "perception_points": ["site2.com:F3:P1", ...]  // Covered perception items
  }
}
```

D Quality Control Criteria

Each instantiated task undergoes rigorous manual quality control by twelve annotators. Tasks are first grouped by function sets and websites, with redundant ones removed within each group. Each task is then reviewed against five criteria: rationality, complexity, executability, evaluability, and topic safety. To further ensure quality, annotators personally execute tasks on live websites, followed by agent-based testing to identify potential issues.

The finalized benchmark contains 420 task instances with an average of $|\mathcal{E}_{act}| = 7$ execution items and $|\mathcal{E}_{perc}| = 4$ perception items per task, involving up to 5 clusters and 14 websites.

E Information Flow Patterns

We represent the information flow ϕ_{prop} as a directed acyclic graph (DAG) over workflow steps. Formally, $\phi_{prop} = (V, E)$ where each step $v_i = (w_i, f_i, o_i) \in V$ invokes function f_i on website w_i to produce output o_i, and each edge $(v_i, v_j) \in E$ indicates that o_i serves as input to v_j. To promote diversity, we provide the language model with five primitive patterns (listed below) and let it compose tasks by freely combining them. For example, a workflow can merge a sequential step with a parallel branch and a fan-in aggregation, represented symbolically as

$$v_1 \to (v_2 \parallel v_3) \to v_4,$$

where v_1 triggers $v_2 \parallel v_3$, both feeding into v_4. By exploring such combinations, we generate workflows ϕ_{prop} that are plausible across websites and structurally varied.

Pattern	Structure	Example
Sequential	$v_i \to v_j$	Find paper on arXiv, then locate code on GitHub
Parallel	$v_i \parallel v_j$	Cross-platform price comparison
Fan-out	$v_i \to \{v_j\}$	Obtain recommendations, verify each on separate sites
Fan-in	$\{v_i\} \to v_j$	Aggregate reviews from multiple sources for decision
Chain	$v_i \to v_j \to v_k$	News article $\to$ company website $\to$ financial tracker

Table E.1: Primitive cross-site information flow patterns.

F Prompts for Eval Construction

This appendix presents the prompt used for Evaluation Script Generation, corresponding to the methodology described in §4.2. This prompt instructs the LLM to translate the structured task metadata into executable Python verification scripts.

Evaluation Script Generation Prompt

```
You are a skilled evaluation annotator. Please write a Python evaluation script based
on the Mind2Web 2 framework for a new task, using the following task information
and reference code.

### New Task Information
{json.dumps(context, indent=2, ensure_ascii=False)}

### Core Requirements
1. Focus on covered_points: Design evaluation points based on the covered_points
field.
 - point_id examples: "ikea.com:F2:A8" (Action) or "ikea.com:F9:P6" (Perception).
 - Action Node desc format: "<complex action: [point_id]> ..."
 - Perception Node desc format: "<complex perception: [point_id]> ..."
```

2. **Strict Critical Node Logic (Error Prevention)**:
 - **Cause of Error**: The framework stipulates that if a parent node is critical=True, all its child nodes must also be critical=True.
 - **Design Recommendation**: To avoid errors and allow for partial scoring, **please set top-level container nodes (e.g., step1, step2) to critical=False**. Only set leaf nodes (specific verification nodes) to critical=True if the failure of that step implies total task failure.
3. **Do not deviate from the task**: Do not add irrelevant evaluation points not mentioned in the task description.

Code Writing Specifications (API Whitelist)
You must strictly adhere to the following APIs; do not invent non-existent methods:
1. **Allowed Methods**:
 - evaluator.initialize(...): Initialize, returns the root node.
 - evaluator.add_sequential(...): **Can** be used (Note: not add_sequence). Used to add sequential execution container nodes.
 - evaluator.add_parallel(...): Used to add parallel execution container nodes.
 - evaluator.add_leaf(...): Add a leaf node.
 - evaluator.add_custom_node(...): Pass in a boolean result node directly.
 - evaluator.extract(...): Extract information.
 - evaluator.verify(...): Verification logic.
 - evaluator.get_summary(): Return the result.
2. **Advanced Technique (verify with node=None)**:
 - If you need to obtain a verification result (True/False) in the Python code to determine subsequent logic, but do not want to draw this node on the evaluation tree, use await evaluator.verify(node=None, ...).
3. **Strictly Prohibit Hallucinations (Negative Constraints)**:
 - **Prohibited Usage**: add_sequence (missing 'ial'), add_serial, update_node_result.
 - **Prohibited Attribute Access**: The Evaluator object does not have .logger or .answer attributes.
 - **Correct Practice**: logger and answer are arguments of the evaluate_answer function; use the argument variables directly (e.g., logger.info(...)).
4. **Data Structures**:
 - Use pydantic.BaseModel to define extraction structures.
 - The evaluate_answer function signature must remain complete (containing all arguments: client, answer, cache, semaphore, logger, model etc.).

Reference Code Template
Please imitate the structure, import style, and verification logic of the following code:
—
{reference_code}
—
Please output the Python code directly. Do not include Markdown code block markers, and ensure the code is directly executable.