

PAFFA: Premeditated Actions For Fast Agents

Shambhavi Krishna *

University of Massachusetts Amherst
shambhavikri@umass.edu

Zheng Chen

Amazon Alexa
tzchen86@gmail.com

Yuan Ling

Amazon Benchmarking

Xiaojiang Huang

Amazon Alexa

Yingjie Li

Amazon Alexa

Fan Yang

Amazon Alexa

Xiang Li

Amazon Alexa

Abstract

Modern AI assistants have made significant progress in natural language understanding and tool-use, with emerging efforts to interact with Web interfaces. However, current approaches that heavily rely on repeated LLM-driven HTML parsing are computationally expensive and error-prone, particularly when handling dynamic web interfaces and multi-step tasks. We introduce PAFFA (Premeditated Actions For Fast Agents), a method that makes LLMs faster and more accurate in completing tasks on the internet using a novel inference-time technique that requires no task-specific training. PAFFA constructs an “Action Library”, leveraging the parametric knowledge of the base LLM to pre-compute browser interaction patterns that generalize across tasks. By strategically re-using LLM inference *across* tasks — either via “Dist-Map” for task-agnostic identification of key interactive web elements, or “Unravel” for first-encounter, stateful exploration of novel tasks/sites) — PAFFA drastically reduces inference time tokens by 87% while maintaining robust performance (achieving 0.57 vs. 0.50 step accuracy compared to baseline). Further, Unravel’s ability to update its action library based on explorations allows generalization and adaptation to unseen websites. In sum, this work exhibits that LLM reasoning sequences can generalize across prompts, offering a way to scale inference-time techniques for internet-scale data with sublinear token count.

1 Introduction

AI Assistants are increasingly expected to navigate the Internet autonomously to complete user tasks (He et al., 2024; Deng et al., 2023). While AI has advanced in natural language understanding and using structured APIs, interacting reliably with diverse, dynamic web interfaces¹ remains a major challenge, limiting the practical use of these agents (Liu et al., 2018). The web’s semi-structured, visual, and ever-changing nature presents distinct hurdles compared to controlled environments. Effective web interaction requires agents to parse HTML/DOM structure, connect language instructions to visual elements, plan action sequences, and adapt robustly to interface changes - capabilities that challenge current models.

Many current web agents follow an approach requiring repeated computation: they invoke Large Language Models (LLMs) at each step to parse the full HTML and decide the next action (Mazumder & Riva, 2021; Lu et al., 2024; Deng et al., 2023). While this uses the LLM’s strong language understanding, this method has significant drawbacks for web interaction:

- **Efficiency:** Repetitively parsing complex DOMs is computationally expensive.

*Work done as part of an internship at Amazon.

¹i.e., interfaces where content, structure, or element identifiers frequently change

- **Reliability:** Web interfaces change constantly. Agents relying on direct, step-by-step HTML parsing are brittle; small DOM changes can cause action failures and errors (Pan et al., 2024a).
- **Scalability:** Solutions often require website-specific implementations or struggle with the diversity of the web. Furthermore, handling multi-page tasks can quickly exceed LLM context limits when processing cumulative HTML, limiting agent versatility and the complexity of tasks they can handle reliably.

To address these issues, we introduce PAFFA (Premeditated Actions For Fast Agents). PAFFA offers an alternative to repetitive runtime parsing through a structured, efficient method grounded in pre-computation of inference and strategic LLM application. PAFFA’s core is an Action Library: a persistent library of reusable, parameterized functions encoding verified interaction patterns for websites. This library effectively caches the results of complex reasoning and planning required for specific web interactions.

This library is primarily constructed offline, leveraging the inherent parametric knowledge and zero-shot capabilities of base LLMs without requiring task-specific fine-tuning or expert demonstrations. PAFFA offers two distinct strategies for this initial library generation:

- **Dist-Map:** This strategy first performs task-agnostic element distillation using LLM semantic understanding to extract key interactive elements into a simplified, robust representation. Subsequently, task-specific scripts (forming the basis of APIs) are generated using these distilled elements. This approach aims for efficiency and robustness to UI variation by abstracting common structures.
- **Unravel (for Construction):** Alternatively, Unravel can directly generate task-specific interaction scripts (which are then parameterized into actions) by processing tasks incrementally, using the full HTML context one page/state at a time. This method handles complex interactions directly without prior distillation.

Regardless of the initial construction strategy, at runtime, PAFFA employs a novel inference-time technique: the LLM performs lightweight high-level intent recognition (matching user requests to APIs) and parameter extraction, followed by direct execution of the corresponding pre-computed API from the library. This division of labor greatly reduces costly real-time HTML parsing.

Furthermore, the Unravel methodology possesses a unique dual capability. Beyond its potential use in initial library construction, Unravel serves as PAFFA’s primary mechanism for dynamic adaptation and handling novelty. When encountering a task, website, or significant structural change not covered by the existing Action library, PAFFA invokes Unravel to handle the interaction live, performing stateful exploration on the current interface. This needs to be done only once for the first encounter with that specific new scenario.

Crucially, the interaction sequences successfully executed by Unravel during these first encounters can be captured, analyzed, parameterized, and integrated back into the Action Library. This update mechanism allows PAFFA to continuously learn, adapt, and generalize from new experiences, effectively caching solutions for novel or modified interactions and maintaining effectiveness as websites evolve – all without retraining the base LLM.

Through these methods, PAFFA demonstrates significant benefits. Evaluations on Mind2Web show **superior performance** over baselines (e.g. element accuracy: 0.74 vs 0.56; step accuracy: 0.57 vs 0.50 on Air.+All Shop. set - see Section 4) while drastically **reducing computational cost**, with an 87% reduction in inference tokens during execution compared to typical LLM-parsing methods.

This work offers:

- A novel Action Library paradigm that pre-computes and caches web interaction patterns, minimizing runtime computation and leveraging LLM parametric knowledge offline without training.

- Two distinct offline strategies (Dist-Map via distillation, Unravel via incremental exploration) for initial library construction using zero-shot LLMs.
- Unravel’s dual role in enabling both initial generation and, uniquely, dynamic runtime adaptation to novel scenarios (new tasks/sites/changes) coupled with a mechanism for library evolution and generalization.
- Empirical results showing significant efficiency gains and strong accuracy improvements on the Mind2Web web agent benchmark.

By replacing expensive runtime parsing with the lightweight execution of pre-computed action plans stored in an adaptive library, PAFFA provides a more scalable, efficient, and robust path for capable web agents. This work contributes to optimizing language model use in complex interactive domains by demonstrating effective strategies for caching, generalizing, and adapting complex reasoning processes.

2 Related Works

Foundation Models and Web Interaction Frameworks: Early web automation frameworks like Mini-WoB++ (He et al., 2024) provided controlled environments for basic web tasks, but failed to capture real-world complexity. Mind2Web (Deng et al., 2023) advanced the field by incorporating real-world websites, though challenges remain in handling dynamic content. Pre-trained models have shown particular promise, with bidirectional models like HTML-T5 achieving state-of-the-art results in document parsing (Li et al., 2022).

Multimodal and Vision-Based Approaches: Recent work has explored multimodal interactions for web navigation. WebVoyager (He et al., 2024) leverages multimodal models for understanding both visual and textual elements, while Pix2Act (Shaw et al., 2023) demonstrates success in screenshot parsing and behavioral cloning using Monte Carlo Tree Search.

Navigation and Planning Systems: Frameworks like FLIN (Mazumder & Riva, 2021) and WebLINX (Lu et al., 2024) have advanced natural language-based navigation, though their step-by-step planning mechanisms create efficiency bottlenecks. Recent work (Gur et al., 2024) has shown promise in task decomposition and multi-step interactions, while MindSearch (Ma et al., 2023) introduces graph-based planning strategies.

Action Abstraction and Evaluation: Current approaches face challenges in balancing accuracy with computational efficiency, requiring repeated HTML parsing and LLM inference. Mind2Web-Live (Pan et al., 2024b) introduces progress-aware evaluation allowing multiple valid paths to task completion. While frameworks have attempted to create reusable components, most focus on low-level actions. Recent work in self-experience supervision (Gur et al., 2024) and frameworks like TPTU-v2 (Kong et al., 2024) explore promising directions in tool use and planning, though gaps remain in developing flexible, high-level action APIs.

3 Methodology

3.1 Overview: Shifting away solely per-interaction reasoning/action sequences

This section details the PAFFA framework, outlining its core components and processes designed to enable efficient, robust, and adaptable web interaction without task-specific training. PAFFA centers around an **Action Library** built using Large Language Models (LLMs) via zero-shot prompting.

We first describe the **two alternative strategies** for initial library construction: **Dist-Map**, which relies on element distillation, and **Unravel**, which performs direct incremental generation (Section 3.2). We then detail **Unravel’s unique role in runtime adaptation** to novel scenarios and the mechanism for **evolving the Action API Library** based on these runtime experiences (Section 3.3). Finally, we cover the common processes of **API synthesis** from generated scripts/traces and the **runtime execution model** using the library (Section 3.3.2).

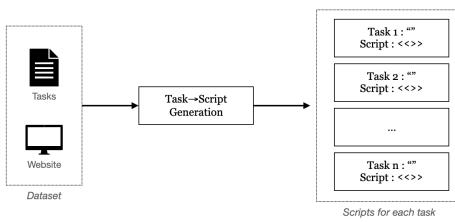


Figure 1: Creating task-specific scripts.

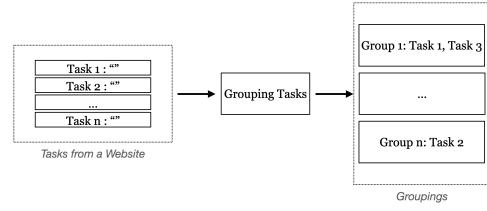


Figure 2: Grouping tasks solvable by one API.

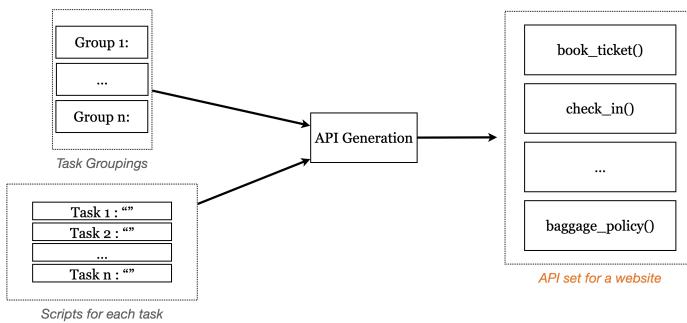


Figure 3: Creating APIs per group.

The Action Library is constructed by leveraging the inherent document understanding, reasoning, and code generation capabilities of LLMs².

3.2 Initial Library Construction Strategies

3.2.1 Strategy 1: Dist-Map (Distillation then Mapping)

The goal of this method is to search for reusable actions that will generalize across interface variations. We hypothesize that leveraging the LLM's inherent understanding of webpages to "distill" the semantic meaning of page elements will make the agent invariant to superficial UI changes. Then, we leverage the LLM's coding ability to write scripts that operate on the distilled semantic representations of webpage elements. More specifically, we break the algorithm into 3 parts:

- Phase 1: Task-Agnostic Element Distillation: Use the LLM for targeted semantic analysis of HTML to identify and extract key interactive elements (buttons, fields, links) and map them to their attributes, creating a structured, distilled representation (e.g., JSON). This aims to capture the functional essence independent of precise layout, building upon work from (Gur et al., 2024; Zheng et al., 2023).
- Phase 2: Verification: Ask the LLM to review the distilled elements to ensure their correctness.
- Phase 3: Contextual Script Generation: Map a specific task description to a sequence of interactions using only the relevant verified, distilled elements. The LLM generates an executable Selenium script (`sel`) based on this constrained context.

²All LLM operations described in this section utilize Anthropic's Sonnet 3.5 (Anthropic, 2025) model.

Because the action library constructed by Dist-Map is static, this algorithm is well-suited for stable, frequently used interaction patterns where element abstraction provides robustness.

3.2.2 Strategy 2: Unravel (Direct Incremental Exploration)

This methodology directly generates scripts by simulating the interaction incrementally, leveraging the full context of the current page state at each step.

For a given task description, Unravel decomposes it into sequential subtasks corresponding to interactions within single page views. In a loop, the LLM processes the *full HTML* of the current page, along with the **overall task goal and interaction history**. Unlike methods that greedily select only the single next best action, this richer context allows the LLM to perform more sophisticated localized planning – potentially identifying multiple necessary actions on the current page or generating code that implicitly anticipates subsequent steps within the current view – before generating the code for the immediate interaction(s).

By using the full page context per step, we observe that this method handles more complex interactions than Dist-Map with more robust error handling (see Section 3.3.1) in the initial generated action scripts. It avoids the potential information loss of distillation but processes more raw HTML during script generation compared to Dist-Map’s final phase.

3.3 Runtime Adaptation and Library Evolution via Unravel

A unique aspect of PAFFA is its ability to adapt to novelty and evolve its library over time, primarily driven by the Unravel methodology operating at runtime.

3.3.1 Handling Novelty with Unravel

When a user request corresponds to a task, website, or significantly changed interface not covered by the existing Action Library, PAFFA invokes Unravel dynamically:

- Trigger: The initial API retrieval fails, or a previously reliable API encounters repeated execution errors indicative of major site changes.
- Execution: Unravel engages in incremental, stateful exploration, using the process described in Section 3.2.2 (chunked execution based on full current-page HTML, maintaining task coherence via history) to attempt the task live on the interface. This exploration needs to be performed only once for the first successful completion of that specific novel scenario.
- Robustness during Exploration: As Unravel processes the full page context at each step, it can generate interaction code with sophisticated error handling (e.g., multi-step try-except blocks targeting different selectors), increasing the chance of success even on unfamiliar or dynamic interfaces.

The successful execution traces generated by Unravel during runtime adaptation are valuable new interaction patterns. PAFFA incorporates a mechanism to integrate this knowledge back into the library:

- Capture: Record the scripts used by Unravel for a novel task completion.
- Synthesize: Process this trace using API Synthesis (Section 3.3.2) to create a new, parameterized API.
- Integrate: Add the new API to the Action Library.

This feedback loop allows PAFFA to continuously learn and generalize. Solutions discovered for novel scenarios are effectively cached as new APIs, making future execution of the same or similar tasks highly efficient. This adaptation occurs without retraining the base LLM, relying instead on capturing and structuring the results of its runtime problem-solving.

3.3.2 Library Evolution

Once initial scripts are generated (by either Dist-Map or Unravel) or new traces are captured (from runtime Unravel³), the following steps finalize the API library and enable runtime use:

1. Task Clustering: All scripts/traces for a website are used to prompt the model to reason and identify groups of tasks sharing common interaction sequences or sub-goals based on semantic similarity (e.g., various login flows, different search types). This LLM-driven step ensures related functionalities are grouped before parameterization.
2. API Synthesis and Parameterization: For each task cluster, the LLM performs program/API synthesis. It analyzes the associated scripts/traces, identifies variations (e.g., different search terms, credentials, dates), and generates a single parameterized Python function (an API) capable of executing all tasks within the cluster. For increased robustness and guarantee of task completion, we employ a 2-step reasoning-based self-corrective prompt that is directed to list any possible shortcomings of the script, and then to address them fully.

3.3.3 Runtime Usage

During deployment (Figure 5 in Appendix B), when a user request arrives:

- API Retrieval/Intent Recognition: The LLM selects the most appropriate API from the library based on the user’s natural language request.
- Parameter Extraction/Slot Filling: The LLM extracts necessary arguments from the request to call the selected API.
- Execution: The chosen, parameterized API is executed directly using browser automation (e.g., Selenium).

This runtime workflow replaces expensive, iterative full-page HTML parsing with a lightweight API call, leveraging the pre-computed knowledge encoded in the action library.

4 Results

We evaluate PAFFA’s effectiveness through multiple lenses: performance on standard web agent benchmarks, computational efficiency during task execution, and quality of the constructed action library.

4.1 Mind2Web Benchmark Performance

To compare PAFFA against established benchmarks, we evaluate on the Mind2Web dataset (Deng et al., 2023), which is the predominant web agent benchmark (to the authors’ knowledge). We report performance using two standard metrics:

1. Element Accuracy: The percentage of correctly identified interactive elements.
2. Step Accuracy: The percentage of correct element-action pairs executed.

We follow the Mind2Web evaluation protocol, considering various cross-task and cross-website splits across the Airlines and Shopping domains (details in Appendix C). Crucially, web task annotations can be ambiguous, and multiple valid interaction paths often exist. Therefore, in addition to exact match accuracy ('Exact'), we perform human re-evaluation to account for functionally correct but non-annotated paths ('Inexact'), providing a more realistic measure of task success.

³This would be after a full task has been successfully completed by Unravel

Dataset Split	MindAct					Dist-Map		Unravel	
	DeB	T5 B	T5 L	T5 XL	S3.5	Ex	Inex	Ex	Inex
Airlines	0.306	0.477	0.562	0.626	0.492	0.618	0.67	0.75	0.76
Air.+Shop.-Task	0.285	0.428	0.508	0.566	0.418	0.699	0.78	0.67	0.701
Shop.-Cross Task	0.202	0.273	0.333	0.357	0.202	0.779	0.89	0.59	0.642
Shop.-Task+CW	0.283	0.372	0.45	0.472	0.316	0.74	0.86	0.61	0.67
Shop.-Cross-Web	0.354	0.510	0.552	0.552	0.395	0.7	0.83	0.62	0.72
Air.+All Shop.	0.298	0.449	0.526	0.562	0.422	0.699	0.79	0.65	0.74

Table 1: Element Accuracies: MindAct vs. PAFFA

Dataset Split	MindAct				Dist-Map		Unravel	
	T5 B	T5 L	T5 XL	S3.5	Ex	Inex	Ex	Inex
Airlines	0.462	0.564	0.616	0.467	0.35	0.42	0.38	0.58
Air.+Shop.-Task	0.421	0.503	0.548	0.38	0.35	0.525	0.29	0.55
Shop.-Cross Task	0.276	0.331	0.338	0.173	0.35	0.63	0.2	0.52
Shop.-Task+CW	0.286	0.366	0.384	0.231	0.344	0.60	0.29	0.56
Shop.-Cross-Web	0.357	0.385	0.362	0.249	0.33	0.56	0.36	0.58
Air.+All Shop.	0.409	0.482	0.500	0.357	0.34	0.57	0.32	0.57

Table 2: Step Accuracy: MindAct vs. PAFFA

Tables 1 and 2 present the Element and Step Accuracy results, comparing both PAFFA algorithms (Dist-Map and Unravel) on a Sonnet 3.5 base against MindAct baselines (fine-tuned DeBERTa+FLAN-T5, and Sonnet 3.5 with 3-shot prompting⁴). They show that PAFFA’s methodologies generally achieve superior or competitive performance compared to MindAct baselines across various splits.

First, comparing PAFFA (using zero-shot Sonnet 3.5) against the fine-tuned MindAct baselines (DeBERTa (He et al., 2021) + FLAN-T5 (Chung et al., 2024) models up to 3B parameters), PAFFA demonstrates strong performance despite using no task-specific training whatsoever. While Sonnet 3.5 is a larger, more capable base model, PAFFA’s effectiveness coupled with its massive reduction in runtime computation (Section 4.2) highlights the power of its pre-computation and strategic LLM usage.

Second, to isolate the contribution of the PAFFA algorithm itself from the underlying model’s power, we compare PAFFA (zero-shot Sonnet 3.5) against MindAct using the same Sonnet 3.5 model with 3-shot prompting (S3.5 column), following the few-shot strategy used in the original MindAct paper. Even under this controlled comparison using the identical base model, PAFFA often outperforms the MindAct few-shot approach, particularly evident in the more forgiving ‘Inexact’ metrics. This strongly suggests that PAFFA’s architectural design – the Action library built via Dist-Map and Unravel – provides significant advantages beyond just leveraging a powerful LLM.

Third, the Cross-Website generalization challenge starkly highlights PAFFA’s advantage. As seen in the Shop.-Cross-Web split, the performance of MindAct (both fine-tuned and few-shot Sonnet) degrades significantly when faced with websites unseen during training or prompting setup. PAFFA’s Unravel method, however, maintains performance comparable to its other splits. This is a direct consequence of PAFFA’s design: because PAFFA requires no training data and Unravel adapts dynamically using the base LLM’s generalization capabilities, it does not suffer from the same generalization gap when encountering new website structures.

Therefore, PAFFA not only demonstrates superior performance in several scenarios but achieves this without needing fine-tuning data or expert demonstrations. Its training-free nature makes it inherently more robust to domain shifts, like encountering new websites, a critical advantage for real-world deployment.

⁴They follow the same three-shot strategy for GPT 3.5 and GPT4 in their paper

MindAct			PAFFA		
# Tokens	Calls	Total Tokens	# Tokens	Calls	Total Tokens
1,565	126	197,190	25,000	1	25,000

Table 3: After Setup Usage Per Task/Request

4.2 Cost Comparison

A core motivation for PAFFA is reducing the computational cost associated with repeated LLM calls for HTML parsing. We compare the typical inference cost per task at deployment time. MindAct requires stepwise LLM calls involving ranked candidate elements⁵ and action selection. PAFFA, leveraging its Action Library, requires only a single, simpler LLM call to map the user request to the appropriate pre-computed API and extract parameters.

PAFFA achieves an 87% reduction in estimated inference tokens per task compared to the MindAct workflow. This substantial efficiency gain stems directly from replacing expensive, iterative runtime parsing and planning with lightweight calls to pre-computed, verified action primitives stored in the Action Library. This calculation is conservative, as it doesn't include the token cost associated with MindAct's candidate element ranking performed by DeBERTa.

4.3 Qualitative Evaluation of Generated Actions

Furthermore, to evaluate the generalization and coverage of the generated Action Library beyond the initial set of tasks, we employed LLM-based synthetic task generation and evaluated the system's ability to correctly ground these diverse requests to the appropriate API calls (see Appendix E for examples).

Beyond task success rates and computational cost, we analyze the intrinsic quality of the initial scripts generated by PAFFA's core methodologies, Dist-Map and Unravel, before potential API synthesis. We employ LLM-based evaluation using Sonnet 3.5⁶ on the Airlines dataset subset. Scripts were assessed along three dimensions: Script Task Alignment (functional goal completion vs. task requirements), Action Representation Fidelity (faithfulness of code to necessary actions), and Script Efficiency (directness of the interaction path). Detailed definitions and LLM prompts for these metrics are provided in Appendix D⁷.

Findings: Table 4 summarizes the average scores, comparing scripts generated via the Dist-Map strategy versus the Unravel strategy. Unravel consistently outperforms Dist-Map across all metrics.

Table 4: LLM-based Qualitative Evaluation Scores (Avg. on Airlines Dataset). Higher is better.

Metric	Dist-Map	Unravel	Rel. Improv.
Script Task Alignment (1-5)	3.2	3.8	+18.8%
Action Repr. Fidelity (1-5)	2.1	2.77	+31.9%
Script Efficiency (1-5)	2.1	3.0	+42.9%

The qualitative evaluation consistently favors scripts generated by Unravel over Dist-Map across all measured dimensions. The significant relative improvements, particularly in efficiency (+42.9%) and fidelity (+31.9%), strongly suggest that Unravel's direct, incremental

⁵This is with k=50 as is preferred in the Mind2Web paper.

⁶We employ Sonnet 3.5 for qualitative assessment, providing consistent evaluations at scale, though we acknowledge this is an emerging evaluation technique.

⁷Since PAFFA does not use any expert trajectories/action sequences, expert actions mentioned in Appendix D prompts serve only as an evaluation reference standard.

exploration using full page context produces higher-quality initial scripts. Based on these evaluated metrics, Unravel appears superior for generating effective, well-aligned, and relatively efficient scripts in this domain, reinforcing its importance not only for runtime adaptation (Section 3.3) but also as a strong candidate for the initial library construction phase.

5 Discussion and Conclusion

This work introduced PAFFA, an algorithm designed to make LLM-driven web agents more efficient, robust and adaptable. By leveraging an Action Library to pre-compute and cache common interaction patterns offline using zero-shot LLMs, PAFFA significantly reduces the computational burden at runtime. Compared to methods requiring step-by-step HTML parsing and planning, PAFFA employs a lightweight inference-time technique involving API retrieval and execution. Our empirical results demonstrate PAFFA’s effectiveness: it achieves superior or competitive accuracy on the Mind2Web benchmark while realizing an 87% reduction in inference tokens during task execution.

A key aspect of PAFFA is its training-free nature. Both initial library construction strategies (Dist-Map and Unravel) rely solely on the parametric knowledge of base LLMs, eliminating the need for costly fine-tuning or expert demonstrations. This inherent characteristic, coupled with the Unravel methodology for runtime adaptation, grants PAFFA strong generalization capabilities, particularly evident in its robust performance on cross-website tasks where traditional trained models falter. Unravel’s ability to handle novel scenarios dynamically and feed successful execution traces back into the library allows PAFFA to continuously adapt and evolve without retraining. Our qualitative evaluations further suggest that Unravel, when used for initial script generation, produces higher-fidelity and more efficient code than the distillation-based Dist-Map approach, reinforcing its central role in the framework.

While PAFFA demonstrates significant advantages, we acknowledge limitations. Our current evaluation relies partly on LLM-based qualitative metrics, an emerging technique, and human evaluation for ‘Inexact’ accuracy, which can be resource-intensive. The scope of tested websites and tasks, while based on a standard benchmark, is not exhaustive. Furthermore, while Unravel’s design promotes robustness, dedicated empirical evaluation of resilience to various types of website changes is needed. Future work should focus on automating API grouping and grounding, enhancing verification modules for library entries, developing strategies for automated library maintenance (e.g., detecting and updating stale APIs), and integrating PAFFA’s capabilities within broader conversational AI assistants.

In conclusion, the Action API Library concept represents a fundamental shift from purely reactive, iterative LLM application towards leveraging pre-computation and strategic caching of complex reasoning processes. PAFFA demonstrates that this approach can yield substantial gains in efficiency and adaptability for autonomous web agents, offering a promising direction for scaling LLM capabilities in complex, dynamic interactive domains.

References

- Webdriver. URL <https://www.selenium.dev/documentation/webdriver/>. Accessed: 2024-12-02.
- Anthropic. Claude 3.5 sonnet. <https://www.anthropic.com>, 2025. Large language model developed by Anthropic.
- Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellar, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *J. Mach. Learn. Res.*, 25:70:1–70:53, 2024.

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

Izzeddin Gur, Hiroki Furuta, Austin V. Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. In *ICLR*. OpenReview.net, 2024.

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. In *ACL (1)*, pp. 6864–6890. Association for Computational Linguistics, 2024.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: decoding-enhanced bert with disentangled attention. In *ICLR*. OpenReview.net, 2021.

Yilun Kong, Jingqing Ruan, Yihong Chen, Bin Zhang, Tianpeng Bao, Shiwei Shi, Du Qing, Xiaoru Hu, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, and Xueqian Wang. Tptu-v2: Boosting task planning and tool usage of large language model-based agents in real-world industry systems. In *EMNLP (Industry Track)*, pp. 371–385. Association for Computational Linguistics, 2024.

Junlong Li, Yiheng Xu, Lei Cui, and Furu Wei. Markuplm: Pre-training of text and markup language for visually rich document understanding. In *ACL (1)*, pp. 6078–6087. Association for Computational Linguistics, 2022.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *ICLR (Poster)*. OpenReview.net, 2018.

Xing Han Lu, Zdenek Kasner, and Siva Reddy. Weblinx: Real-world website navigation with multi-turn dialogue. In *ICML*. OpenReview.net, 2024.

Kaixin Ma, Hongming Zhang, Hongwei Wang, Xiaoman Pan, and Dong Yu. LASER: LLM agent with state-space exploration for web navigation. *CoRR*, abs/2309.08172, 2023.

Sahisnu Mazumder and Oriana Riva. FLIN: A flexible natural language interface for web navigation. In *NAACL-HLT*, pp. 2777–2788. Association for Computational Linguistics, 2021.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. *CoRR*, abs/2404.06474, 2024a.

Yichen Pan, Dehan Kong, Sida Zhou, Cheng Cui, Yifei Leng, Bing Jiang, Hangyu Liu, Yanyi Shang, Shuyan Zhou, Tongshuang Wu, and Zhengyang Wu. Webcanvas: Benchmarking web agents in online environments. *CoRR*, abs/2406.12373, 2024b.

Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina N Toutanova. From pixels to ui actions: Learning to follow instructions via graphical user interfaces. *Advances in Neural Information Processing Systems*, 36:34354–34370, 2023.

Longtao Zheng, Rundong Wang, and Bo An. Synapse: Leveraging few-shot exemplars for human-level computer control. *CoRR*, abs/2306.07863, 2023.

A Common Prior Workflow

Existing solutions iteratively call the LLM after every single action, which is inefficient (see Figure 4), and does not take into account planning that Agents can leverage to understand the web like a human would, by constructing viable workflows and following those action sequences. Previously executed tasks *should* inform similar future tasks.

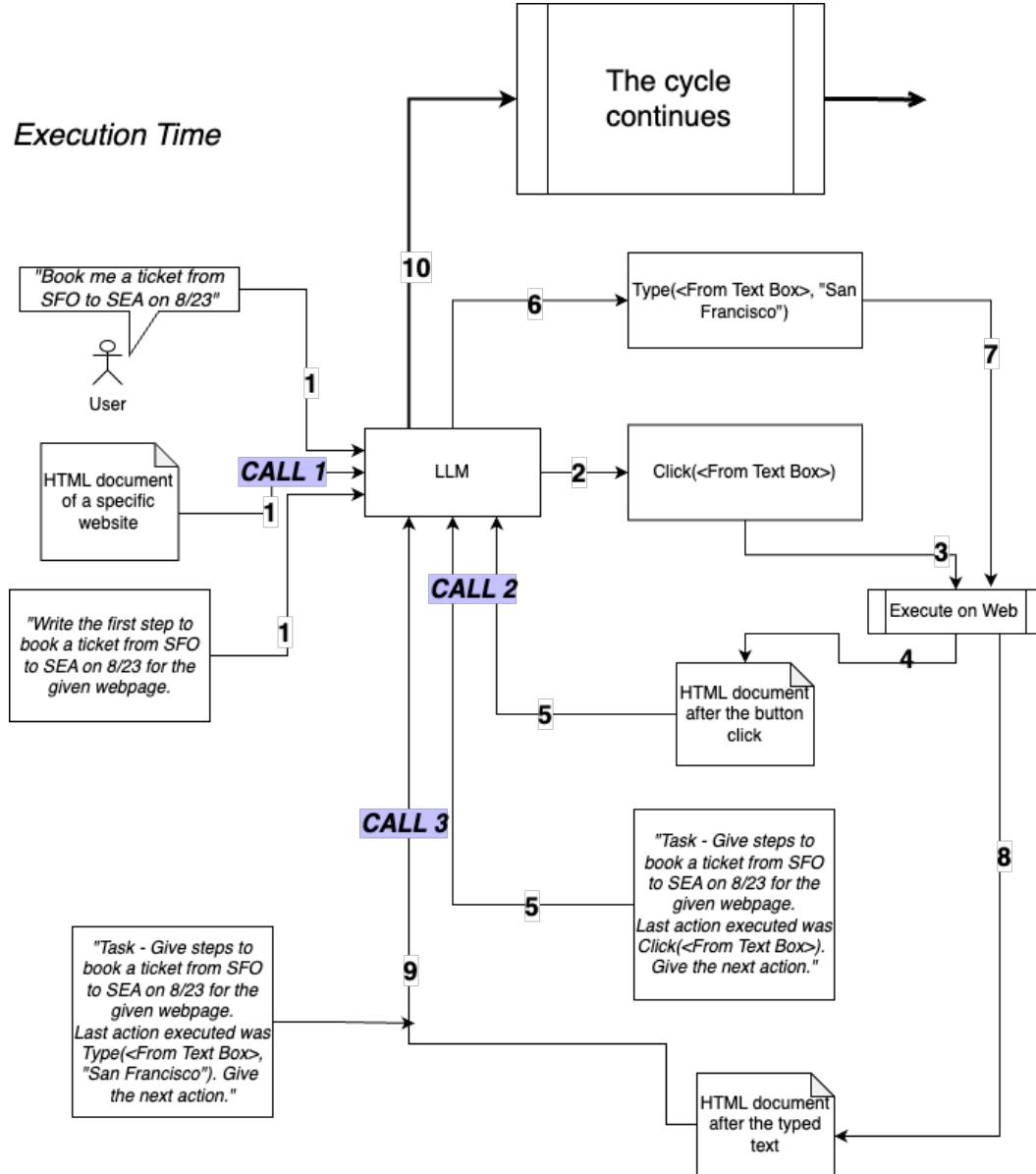


Figure 4: Common workflow of existing solutions like MindAct (Deng et al., 2023).

B Action Library Workflow

The Figure 5 depicts PAFFA’s core inference-time process. Instead of parsing HTML, the system retrieves a relevant pre-computed API from the library based on the user’s request, extracts parameters, and executes the API directly to interact with the website.

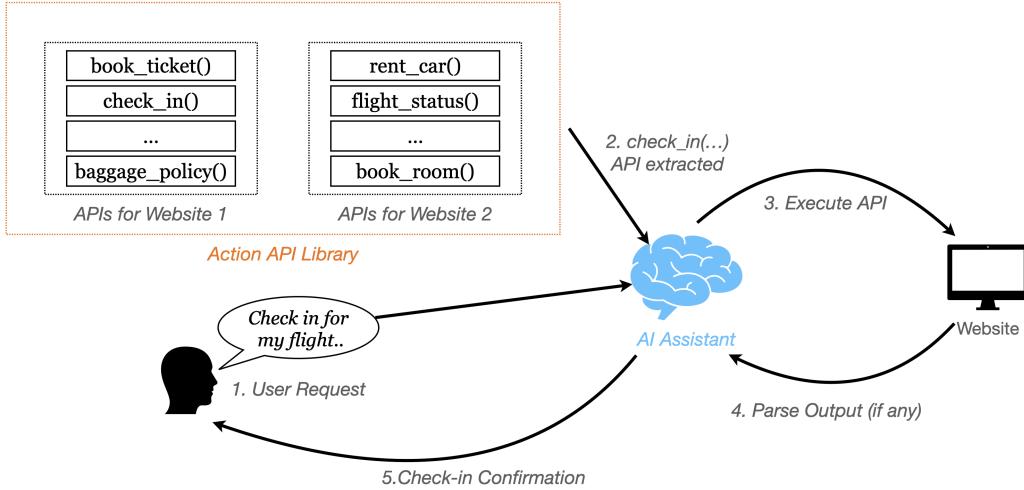


Figure 5: Using Action Library.

C Dataset

We evaluate our algorithm on the Mind2Web benchmark (Deng et al., 2023), which provides annotated actions for diverse tasks across real-world websites. This dataset’s key advantage is its use of complex, real-world websites rather than simplified environments like MiniWoB++ (Liu et al., 2018), enabling more realistic evaluation of generalization capabilities.

For our evaluation, we extract a focused subset of Travel-Airlines domain and Shopping-General Cross-Task splits. We additionally include the Cross-Website split of Shopping-General to assess generalization in our no-train setting (see Appendix C.1 for more details). From each website’s training data, we extract ‘unique web pages’ - pages with distinct HTML structures whose content may change dynamically (e.g., a homepage’s structure remains constant while its content varies with date and location).

We note two key limitations of the benchmark: lack of interactivity compared to toy environments, and its evaluation methodology requiring exact matching of user actions. As the dataset expects models to map HTML documents, previous actions, and tasks to single-step actions, it presents a potential disadvantage for our approach, which aims to avoid task-specific implementations and live HTML parsing at test time.

C.1 Dataset Details

The three datasets are:

1. Cross-Task Split for Travel-Airlines : contains 31 test tasks, across 7 websites : American Airlines, Delta, Jetblue, Kayak, Qatarairways, Ryanair, United.
2. Cross-Task Split for Shopping-General : contains 8 test tasks, across 3 websites : Amazon, Target, Instacart.

3. Cross-Website Split for Shopping-General : contains 17 test tasks, for one website : Google Shopping.

Note : There are no Cross-Website test cases for Travel-Airlines.

D Qualitative Evaluation Metrics

We employed LLM-based evaluation (using Sonnet 3.5) to assess the quality of initially generated scripts along the following dimensions:

- Script Task Alignment (Scale 1-5): Measures how completely the generated script fulfills all explicit and implicit requirements of the specified task, comparing against the user instruction and available expert action sequences used solely as an evaluation reference. Crucially, similar to the ‘Inexact’ evaluation in Section 4.1, this metric prioritizes functional goal completion. A high score indicates the script successfully achieves the task’s objectives, even if the sequence of actions differs from a specific reference path. A lower score indicates failure to meet requirements or incomplete execution.
- Action Representation Fidelity (Scale 1-5): Assesses how accurately the essential actions and parameters within the generated script capture the necessary steps required by the task, reflecting the faithfulness of the generated code structure to the underlying interaction logic.
- Script Efficiency (Scale 1-5): Assesses whether the script uses a reasonably direct path, avoiding unnecessary steps or redundant operations. A higher score indicates greater efficiency.

E Example Action Generation and Usage

This appendix provides a concrete example of the Action API Library components for the Delta website, illustrating the output of the API Synthesis process and how APIs are used.

E.1 Example Synthesized APIs for Delta

Following the process described in Section 3.3.2, scripts generated for various Delta tasks (e.g., finding reservations, searching for flights using different criteria) are clustered and synthesized into parameterized Python functions. Below are two examples of such APIs generated for the Delta website:

- API for Retrieving Trip Information: This function encapsulates the steps needed to look up a reservation using confirmation details.

```

1  def retrieve_trip_information(
2      driver: webdriver.Chrome,
3      confirmation_number: str,
4      first_name: str,
5      last_name: str,
6      wait_time: int = 10
7  ) -> None:
8      """
9          Retrieves trip information by navigating to the 'My Trips'
10         page,
11         entering the confirmation number, first name, and last name,
12         then submitting the form.
13
14     Args:
15         driver (webdriver.Chrome): The Selenium WebDriver
16             instance.
17             confirmation_number (str): The trip confirmation number.

```

```

17         first_name (str): The traveler's first name.
18         last_name (str): The traveler's last name.
19         wait_time (int, optional): Max wait time. Defaults to 10.
20     """
21     def safe_action(locator: tuple, action: str, value:
22         Optional[str]
23         = None) -> None:
24         element = WebDriverWait(driver, wait_time).until(
25             EC.presence_of_element_located(locator)
26         )
27         if action == "click":
28             driver.execute_script("arguments[0].click()", element)
29         elif action == "input":
30             # Use arguments array for safer JS execution with
31             values
32
33             driver.execute_script("arguments[0].value=arguments[1];"
34             , element, value)
35
36             # Navigate to My Trips
37             safe_action((By.ID, "headPrimary3"), "click")
38             sleep(2) # Consider replacing sleep with WebDriverWait if
39             possible
40
41             # Enter details
42             safe_action((By.ID, "confirmationNo"), "input",
43             confirmation_number)
44             safe_action((By.ID, "firstName"), "input", first_name)
45             safe_action((By.ID, "lastName"), "input", last_name)
46
47             # Submit the form
48             safe_action((By.ID, "btn-mytrip-submit"), "click")
49             sleep(3) # Wait for potential results page
50

```

- API for Searching Flights: This function handles various flight search scenarios, including booking, checking status, using miles, and specifying class.

```

1     def search_flights(
2         driver: webdriver.Chrome,
3         origin: str,
4         destination: str,
5         depart_date: str, # Expects format like MM/DD/YYYY
6         use_miles: bool = False,
7         wait_time: int = 10
8     ) -> None:
9         """
10            Searches for flights using origin, destination, date, and
11            miles option.
12
13            Args:
14                driver (webdriver.Chrome): Selenium WebDriver instance.
15                origin (str): Origin airport/city.
16                destination (str): Destination airport/city.
17                depart_date (str): Departure date (e.g., "06/05/2024").
18                use_miles (bool, optional): Search using miles. Defaults
19                to False.
20                wait_time (int, optional): Max wait time. Defaults to 10.
21
22            def safe_action(locator: tuple, action: str, value:
23                Optional[str] = None) -> None:
24                element = WebDriverWait(driver, wait_time).until(
25                    EC.element_to_be_clickable(locator)
26                    # Use clickable for inputs/buttons

```

```

24
25     )
26     if action == "click":
27         # Using execute_script often more reliable than
28         element.click()
29         driver.execute_script("arguments[0].click();",
30         element)
31     elif action == "input":
32         element.click() # Focus element first
33         element.clear()
34         element.send_keys(value)
35         # May need Keys.RETURN or specific waits for
36         #autocomplete here
37         sleep(0.5) # Small pause
38
39     # Enter Origin
40     safe_action((By.ID, "fromAirportName"), "input", origin)
41     sleep(1)
42
43     # Enter Destination
44     safe_action((By.ID, "toAirportName"), "input", destination)
45     sleep(1)
46
47     # Enter Departure Date
48     safe_action((By.ID, "input_departureDate_1"), "input",
49     depart_date)
50     sleep(1)
51
52     # Handle Shop with Miles
53     if use_miles:
54         safe_action((By.ID, "shopWithMiles"), "click")
55         sleep(1)
56
57     # Click Search Button
58     safe_action((By.ID, "btn-book-submit"), "click")
59
60     sleep(3) # Wait for search results page

```

E.2 Synthetic Task Generation and API Grounding

To test the coverage and robustness of the generated API library beyond the initial tasks used for its creation, we also employed LLM-based synthetic task generation (zero-shot prompting with Sonnet 3.5). For a given website (e.g., Delta), we prompted the LLM to generate diverse, realistic user requests. **Example Synthetically Generated Tasks for Delta:**

- Find my reservation with confirmation code DLTX7Y including passenger name Sarah Johnson
- Check available non-stop flights from Atlanta to Los Angeles on September 12th 2025 for two adults in Comfort+
- Show me first class round-trip options from Boston to Miami departing April 15, 2026 and returning April 22, 2026
- Find flights from Chicago to San Francisco on July 8th 2025 that can be booked using SkyMiles and have Wi-Fi onboard

These synthetic tasks are then processed by PAFFA’s runtime system (Section 3.3). The LLM performs API Retrieval/Intent Recognition and Parameter Extraction/Slot Filling to map the natural language task to the most appropriate API call.

Example Grounded API Call: For a request like “Find flights from Seattle to New York on June 5th, 2025 using miles”, the system might generate the following call:

```
1      search_flights(
2          driver=driver_instance,
3          origin="Seattle",
4          destination="NewYork",
5          date="2025-06-05",
6          use_miles=True
7      # travel_class might be None if not specified
8      )
```

This process of generating synthetic tasks and grounding them to API calls allows for broader testing of the library's capabilities and the LLM's ability to correctly map diverse requests to the pre-computed functions.