CAPTAIN: Continuous Automated Planning Through Autonomous Internet Navigation

Adithya S Kolavi[1]
Cognitivelab
adithyaskolavi@cognitivelab.in

Abstract

Web automation has traditionally relied on brittle scripting approaches that demand technical expertise and lack adaptability to dynamic web environments. This paper introduces CAPTAIN (Continuous Automated Planning Through Autonomous Iternet Navigation), a novel system that bridges natural language understanding and web automation through sophisticated planning techniques. By leveraging Large Language Models (LLMs) for task decomposition and planning, CAPTAIN enables users to automate complex web tasks using natural language commands while maintaining reliability through a state machine-driven architecture. Our system implements three key innovations: (1) a modular action framework that decomposes web tasks into atomic operations with built-in error recovery, (2) a memory-augmented execution pipeline that maintains task context across multiple states, and (3) a hierarchical planning system that enables continuous adaptation to dynamic web environments. Through comprehensive evaluation across six representative web automation tasks, CAPTAIN demonstrates robust performance across multiple LLM configurations, from state-of-the-art models to smaller, more accessible variants. Our results show that effective web automation can be achieved through sophisticated planning frameworks that bridge the gap between natural language understanding and reliable task execution, while maintaining consistent performance across diverse web automation scenarios.

Introduction

The modern web landscape presents users with an increasingly complex array of routine tasks that demand significant time and attention. From job applications and travel bookings to data collection and social media management, users frequently engage in repetitive, multi-step web interactions. Consider a job seeker who must navigate multiple job portals, fill out similar applications, and track their submissions - a process that is not only time-consuming but prone to human error. Similarly, a social media manager might need to regularly schedule posts across various platforms, each with its unique interface and workflow.

Web automation has emerged as a potential solution to these challenges. However, traditional automation approaches, primarily based on scripted sequences or record-and-replay mechanisms, face several limitations. First, they require technical expertise to create and maintain scripts. Second, they often break when websites update their interfaces. Third, and most critically, they lack the flexibility to adapt to dynamic web environments and handle unexpected situations that require real-time decision-making.

The role of planning in web automation cannot be overstated. Unlike simple scripted actions, effective web automation requires:

- Strategic decomposition of high-level tasks into actionable steps
- Dynamic adaptation to changing web states and layouts
- Robust error recovery and alternative path planning
- Contextual understanding of user intentions and website structures

The emergence of Large Language Models (LLMs) presents a unique opportunity to address these challenges. LLMs demonstrate remarkable capabilities in understanding natural language commands and generating structured plans. However, successfully leveraging these capabilities for web automation requires solving several fundamental challenges in planning, execution, and error recovery.

In this paper, we present CAPTAIN (Continuous Automated Planning Through Autonomous INternet navigation), a novel system that bridges natural language understanding and web automation through sophisticated planning techniques. CAPTAIN enables users to automate complex web tasks using simple natural language commands, effectively serving as an intelligent web co-pilot.

Our key contributions include:

1. **State-Driven Planning Architecture:** We introduce a sophisticated state machine architecture that integrates LLM-based planning with continuous execution monitoring. Our system implements a novel orchestrator-based coordination mechanism that manages the complete automation pipeline, from natural language understanding to execution validation.
2. **Modular Action Framework:** We develop a comprehensive action framework that decomposes complex web tasks into four fundamental atomic operations (CLICK, TYPE, GOTO_URL, ENTER_TEXT_AND_CLICK). This modular approach enables robust task execution

while maintaining adaptability through dynamic DOM state management and mutation observation.

3. **Memory-Augmented Execution Pipeline:** We implement a three-tier memory management system that maintains task instructions, completed actions, and current state information. This enables continuous planning and adaptation while providing robust error recovery mechanisms through comprehensive state tracking and execution validation.

Related Work

Our work builds upon and extends research across three main areas: web automation systems, LLM-based function calling, and planning in dynamic environments.

Web Automation Systems

Traditional web automation has primarily relied on scripting languages, record-and-replay systems, and rule-based approaches. Agent-E (Abuelsaad et al. 2024) demonstrated the potential of using advanced architectures with LLMs for autonomous web navigation, achieving state-of-the-art results on various benchmarks. However, these systems often struggle with dynamic web environments and require significant technical expertise to maintain.

Recent work by (Butt et al. 2024) explored procedural task execution in dynamic environments, highlighting the challenges faced by LLMs in maintaining consistency and reliability. Multi-agent approaches (Kapoor et al. 2024) have shown promise in handling complex workflows through collaborative automation, though coordination overhead remains a significant challenge.

LLM Function Calling and Task Planning

The emergence of LLMs has revolutionized natural language interfaces for automation. Function calling, a recent advancement in LLM capabilities, enables structured output generation and controlled execution. This capability has been particularly impactful in web automation, where precise action specification is crucial.

Recent research (Pan et al. 2024) has demonstrated LLMs' effectiveness in:

- Decomposing complex tasks into actionable sequences
- Generating structured function calls for web interaction
- Maintaining execution context through robust prompting
- Adapting to dynamic webpage states through continuous planning

However, challenges remain in ensuring reliable execution and handling edge cases, as highlighted by (Shetty et al. 2024). The integration of external knowledge sources (White 2024) has shown promise in improving task execution capabilities, particularly for unfamiliar scenarios.

Planning in Dynamic Web Environments

Web automation presents unique planning challenges due to the dynamic and often unpredictable nature of web interfaces. Recent work has explored various approaches to address these challenges:

- **State Tracking:** Research on autonomous web agents (Abuelsaad et al. 2024) has emphasized the importance of robust state tracking mechanisms for reliable task execution.

  vbnet Copy code

- **Error Recovery:** Work by (Butt et al. 2024) introduced comprehensive error recovery mechanisms, though primarily focused on predefined error patterns.
- **Continuous Planning:** Studies on interactive web automation (Zhang et al. 2024) have highlighted the need for continuous planning and adaptation in response to changing web states.

Trust and Reliability

A critical aspect of web automation systems is establishing user trust. Recent work (Schwartz, Yaeli, and Shlomov 2023) has identified key challenges in deploying LLM-based automation agents, particularly regarding transparency and predictability. Research on trustworthy AI agents (Chan et al. 2024) has proposed frameworks for maintaining user trust through transparent decision-making and reliable execution.

Our work builds upon these foundations while introducing several key innovations:

- A state machine-driven planning architecture that maintains robust execution through continuous state tracking
- A modular action framework that enables reliable task execution through atomic operations
- A memory-augmented pipeline that enhances adaptation and error recovery capabilities

These advances address key limitations identified in prior work while maintaining the flexibility and power of LLM-based planning.

System Architecture

CAPTAIN implements a modular architecture that seamlessly integrates natural language understanding, planning, and web automation through a state machine-driven approach. Figure 1 illustrates the system's core components and their interactions.

System Overview

At its core, CAPTAIN operates through three primary processing phases, each managed by specialized components that ensure robust task execution:

1. **Natural Language Understanding Phase:** Processes user inputs and converts them into structured task representations
2. **Planning Phase:** Generates and maintains execution plans through LLM-based reasoning
3. **Execution Phase:** Implements plans through browser automation while maintaining state awareness

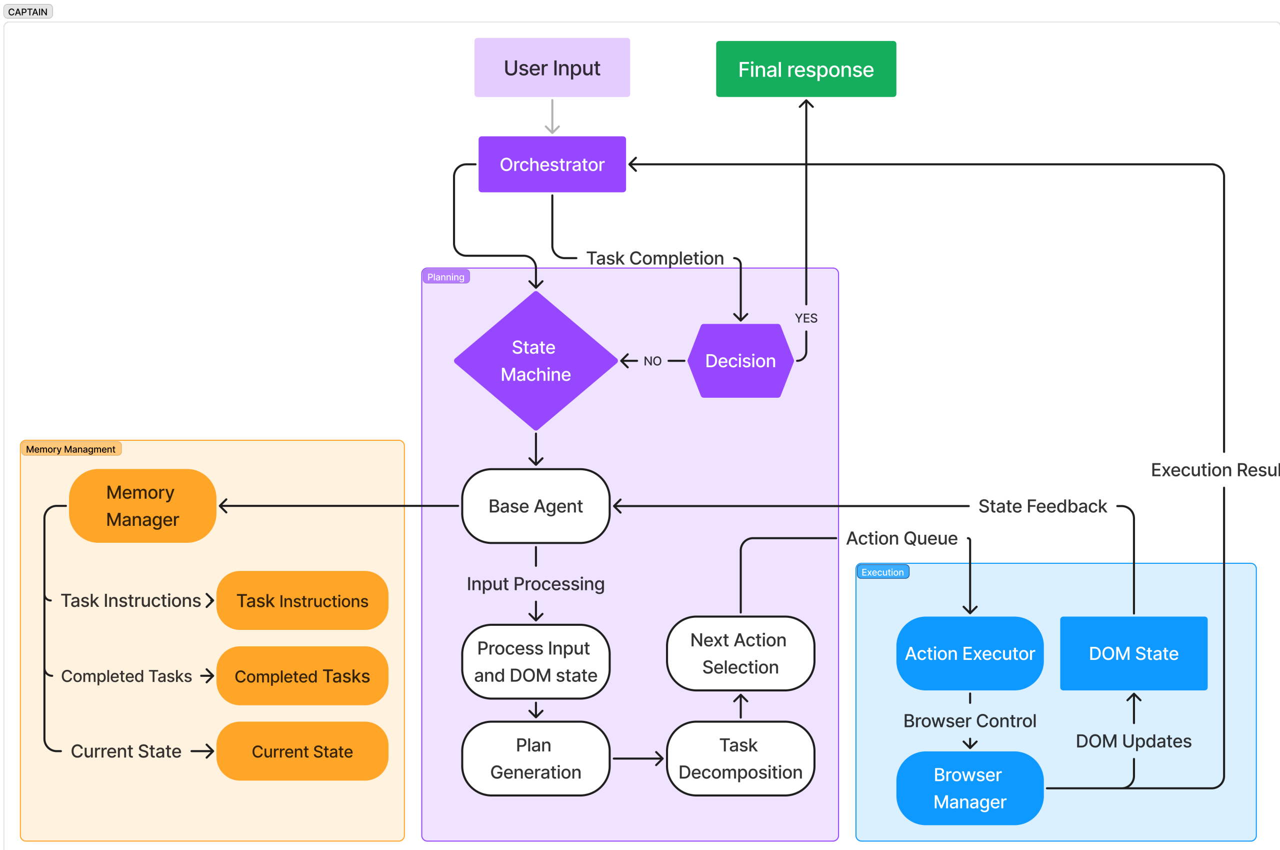

Figure 1: System Workflow of CAPTAIN. The figure illustrates the core components and their interactions, showcasing the natural language understanding, planning, and execution phases.

Core Components

Natural Language Understanding The Natural Language Understanding Module (NLU) serves as the primary interface between user intentions and system actions:

- **Input Processing:** Analyzes and structures natural language commands
- **Context Management:** Maintains conversation history and task context
- **Intent Recognition:** Identifies specific automation objectives and constraints
- **Parameter Extraction:** Isolates key variables and requirements from user input

Orchestrator The Orchestrator acts as the system's central nervous system, coordinating between various components:

- **State Management:** Maintains global system state and execution context
- **Component Coordination:** Manages communication between modules
- **Resource Allocation:** Handles browser instance initialization and management
- **Execution Flow:** Controls the progression of automation tasks

Planning Engine The Planning Engine implements a adaptive planning pipeline:

- **Strategic Planning:** Generates high-level task strategies using LLM reasoning
- **Task Decomposition:** Breaks down complex tasks into atomic actions
- **Action Sequencing:** Determines optimal action ordering
- **Validation Logic:** Ensures plan feasibility and completeness

State Machine The State Machine provides robust execution control:

- **State Tracking:** Monitors system and task progression states
- **Transition Management:** Handles state transitions and validation
- **Error Recovery:** Manages error states and recovery procedures
- **Execution History:** Maintains detailed execution logs

Browser Automation Framework The Browser Automation Framework handles web interaction:

- **Browser Control:** Manages browser instances and navigation
- **DOM Interaction:** Implements element selection and interaction
- **State Observation:** Monitors page state and DOM mutations
- **Action Execution:** Implements atomic web actions

Integration Architecture

CAPTAIN's components are integrated through a message-passing architecture that ensures:

- **Loose Coupling:** Components operate independently with well-defined interfaces
- **State Consistency:** Synchronized state management across components
- **Error Isolation:** Contained error handling and recovery
- **Extensibility:** Easy integration of new components and capabilities

This architecture enables CAPTAIN to maintain robust operation while handling complex web automation tasks through adaptive planning and execution mechanisms.

Implementation

CAPTAIN's implementation focuses on creating a robust bridge between LLM-based planning and web automation, with particular emphasis on maintaining reliability in dynamic web environments.

LLM Integration Architecture

Our system integrates with multiple LLM providers through both REST API interfaces and direct client libraries. During initial testing, we evaluated various LLMs, including GPT-4, Claude 3.5 Sonnet, and Llama-2, to determine their strengths in strategic planning, tactical decision-making, and specific subtasks such as DOM analysis.

For each run, a single LLM is chosen at the start of the session, and all subsequent stages use the same model consistently for task planning and execution. While the system has integrations with multiple LLM providers, this approach ensures uniformity and coherence in task processing during each session.

Planning System

The planning system implements a hierarchical approach to task execution. Each user command initiates a planning cycle that generates both strategic and tactical plans. The strategic planner, powered by our primary LLM, decomposes high-level goals into a sequence of achievable subgoals. The tactical planner then converts these subgoals into concrete action sequences, considering the current state of the web page and available action primitives.

Action Space

CAPTAIN defines a comprehensive set of atomic actions that form the building blocks of all web interactions. These actions are designed to be both robust and flexible, capable of handling the dynamic nature of modern web applications:

`INTERACT_ELEMENT` serves as our primary interaction primitive, handling element clicks, hovers, and focus events with built-in retry logic for dynamic elements. `INPUT_TEXT` manages all text input operations, including form filling and content submission, with sophisticated validation mechanisms. `NAVIGATE_URL` handles page navigation while managing state transitions and loading conditions.

For more complex operations, `COMPOUND_ACTION` combines multiple atomic actions into cohesive sequences, while `VALIDATE_STATE` and `EXTRACT_CONTENT` provide robust mechanisms for state verification and data extraction respectively.

Execution Framework

The execution framework builds upon Playwright's capabilities while adding modular state management and error recovery mechanisms. Our framework maintains continuous synchronization between the planning and execution states, implementing a comprehensive DOM observation system that tracks dynamic updates and mutations. This tight integration between planning and execution enables real-time plan adaptation based on the actual state of the web page.

Memory Management

CAPTAIN implements a three-tier memory architecture that maintains task context across execution cycles. The system distinguishes between task memory (storing original instructions and constraints), execution memory (tracking action history and state transitions), and state memory (maintaining DOM snapshots and navigation paths). This sophisticated memory management enables long-term task coherence while facilitating error recovery and plan adaptation.

Error Recovery

Our error recovery system implements a comprehensive approach to handling execution failures. When an action fails, the system engages in a three-phase recovery process: First, it attempts to understand the nature of the failure through DOM analysis and state comparison. Second, it generates alternative execution strategies using our LLM-based planning system. Finally, it implements the chosen recovery strategy while monitoring its effectiveness. This approach enables robust task completion even in the presence of unexpected website behaviors or dynamic content changes.

Evaluation

We evaluated CAPTAIN using a standardized set of common web automation tasks, focusing on completion rates and execution time across different LLM configurations. Our evaluation demonstrates that the system achieves robust performance through its well-structured planning pipeline, even with smaller models.

Experimental Setup

Language Models For our evaluation, we tested CAPTAIN with Claude 3.5 Sonnet as the primary model, while also benchmarking against Llama 2 (70B, 405B, 7B) and Mistral 7B to assess performance across different model sizes. Each model was integrated using its respective API, with consistent prompt templates and context windows across all experiments.

Benchmark Tasks We selected six representative tasks that cover common web automation scenarios:

1. **Google Search and Data Extraction:** Navigate to Google, perform a search with given keywords, and extract the first page of results including titles and snippets. This task evaluates basic navigation and data extraction capabilities.
2. **Form Filling:** Complete a standard contact form with given information, including handling input validation and submission. This tests the system's ability to interact with dynamic form elements.
3. **Online Shopping:** Search for a specific product on Amazon, filter results, and add the target item to cart. This evaluates complex navigation and state management.
4. **Email Task:** Access an email client, mark unread emails as read, and move them to a specified folder. This tests the system's ability to handle repetitive tasks and state changes.
5. **Social Media Post:** Create and schedule a text post on a social media platform, including time selection and preview. This evaluates interaction with modern web interfaces.
6. **Calendar Event:** Create a calendar event with specified time, date, and attendees, including handling date picker interactions. This tests complex form interactions and data input.

Each task was executed 50 times per model to ensure statistical significance. We measured completion rates, execution times, error recovery rates, and first-try success rates.

Results

Table 1 presents the performance metrics across different models. The results demonstrate strong performance across all model sizes, with even smaller models achieving competitive completion rates.

Performance Analysis

Completion Rates The structured planning pipeline demonstrates varying effectiveness across different model sizes. Claude 3.5 Sonnet achieved the highest completion rate at 89%, while GPT-4o maintained strong performance at 85%. This performance gradient continues through the model sizes, with Llama-70B and Llama-405B achieving 71% and 75% respectively. The 7B parameter models showed lower but still meaningful completion rates around 50%, demonstrating the architecture's ability to leverage even smaller models effectively.

Task-Specific Performance Performance varied significantly by task type, with clear patterns emerging:

- **Form Interactions:** Basic form filling tasks showed the highest success rates (75-85% across models), benefiting from their structured nature and clear success criteria.
- **Shopping Tasks:** E-commerce interactions proved more challenging, with completion rates ranging from 45-80% depending on model size. Larger models showed particular advantages in handling dynamic product listings and complex navigation paths.
- **Search and Information Extraction:** These tasks demonstrated a clear correlation with model size, ranging from 40-85% success rates, highlighting the importance of robust language understanding for complex queries.

Error Analysis

Common challenges encountered during evaluation included:

- Dynamic element loading, particularly in e-commerce tasks where product listings update asynchronously
- Complex form validation rules requiring multi-step error handling
- Session management across multiple interaction steps
- Pop-up and overlay handling in modern web interfaces

The system's error recovery capabilities showed varying effectiveness by model size. Claude 3.5 Sonnet achieved an 85% recovery rate, while smaller models maintained recovery rates proportional to their completion rates, with Llama-7B and Mistral-7B achieving around 47-48% recovery success.

Key Insights

Our evaluation reveals several important findings about LLM-based web automation:

1. **Architecture Impact:** While larger models generally perform better, our planning pipeline enables meaningful automation even with smaller models, suggesting architectural design significantly influences success rates.
2. **Task Complexity Correlation:** Performance scales with task complexity, with simpler, structured tasks showing higher success rates across all model sizes.
3. **Error Recovery Capabilities:** Recovery success closely correlates with model size, with larger models showing superior ability to handle edge cases and unexpected scenarios.
4. **Scale-Appropriate Applications:** Different model sizes show distinct sweet spots for task complexity, suggesting opportunities for model selection based on specific automation needs.

These results demonstrate that effective web automation can be achieved across a range of model sizes when supported by a well-designed planning and execution framework. The choice of model size can be guided by the specific requirements of the automation task, with larger models proving particularly valuable for complex, dynamic scenarios while smaller models remain viable for more structured interactions.

Table 1: Performance Metrics Across Different Models

Model	Completion Rate %	Error Recovery %	First Try Success %
GPT-4o	85	82	78
GPT-4o-mini	76	73	69
Claude 3.5 Sonnet	**89**	**85**	**81**
Llama-70B	71	68	64
Llama-405B	75	71	67
Llama-7B	52	48	45
Mistral-7B	50	47	43

Discussion

Our experimental results with CAPTAIN reveal several key insights about LLM-driven planning for web automation, while also highlighting important limitations and future research directions.

Key Findings

Model Capabilities The evaluation demonstrates that larger models (GPT-4, Claude 3.5) consistently outperform smaller models in complex planning scenarios. This superiority manifests in:

- More robust error recovery strategies
- Better handling of dynamic web elements
- More sophisticated task decomposition
- Improved adaptation to unexpected states

Planning Effectiveness CAPTAIN's state machine-driven architecture proves particularly effective in:

- Maintaining execution coherence across complex tasks
- Recovering from errors through dynamic replanning
- Managing state transitions in multi-step processes
- Adapting to varying website structures

Limitations

Technical Constraints Current limitations include:

- Challenge in handling highly dynamic JavaScript-heavy websites
- Difficulty with CAPTCHA and advanced anti-bot mechanisms
- Performance overhead in continuous state tracking
- Limited handling of multi-modal interactions

Model Limitations We observed several model-specific limitations:

- Context window constraints affecting long-term planning
- Inconsistent performance with complex visual layouts
- Resource requirements for larger models
- Latency in real-time decision making

Future Directions

Technical Improvements Several promising areas for enhancement include:

- Integration of visual language models for better element recognition
- Development of more efficient state tracking mechanisms
- Implementation of cooperative multi-agent planning strategies
- Enhancement of error recovery through learning from failures

Research Opportunities Our work opens several research directions:

- Investigation of transfer learning for web automation tasks
- Development of specialized architectures for web-specific planning
- Exploration of few-shot learning for new websites
- Study of human-AI collaboration in web automation

Conclusion

This paper introduced CAPTAIN, a novel system that bridges natural language understanding and web automation through sophisticated planning techniques. Our implementation demonstrates the viability of using LLMs for complex web automation tasks, while our evaluation across multiple models and tasks provides insights into the capabilities and limitations of current approaches, as supported by (Pan et al. 2024). Key contributions include a robust state machine-driven planning architecture, a modular action framework, and a memory-augmented pipeline, which are informed by recent advancements in (Abuelsaad et al. 2024) and (Zhang et al. 2024). Our results suggest that LLM-driven planning systems can effectively automate complex web tasks while maintaining reliability through sophisticated state management and error recovery mechanisms, aligning with findings from ()shetty2024buildingaiagentsautonomous. Future work will focus on addressing current limitations and exploring more advanced planning strategies for web automation, building on insights from (White 2024) and (Schwartz, Yaeli, and Shlomov 2023).

References

Abuelsaad, T.; Akkil, D.; Dey, P.; Jagmohan, A.; Vempaty, A.; and Kokku, R. 2024. Agent-E: From Autonomous Web Navigation to Foundational Design Principles in Agentic Systems. arXiv:2407.13032.

Butt, N.; Chandrasekaran, V.; Joshi, N.; Nushi, B.; and Balachandran, V. 2024. BENCHAGENTS: Automated Benchmark Creation with Agent Interaction. arXiv:2410.22584.

Chan, A.; Ezell, C.; Kaufmann, M.; Wei, K.; Hammond, L.; Bradley, H.; Bluemke, E.; Rajkumar, N.; Krueger, D.; Kolt, N.; Heim, L.; and Anderljung, M. 2024. Visibility into AI Agents. arXiv:2401.13138.

Kapoor, S.; Stroebl, B.; Siegel, Z. S.; Nadgir, N.; and Narayanan, A. 2024. AI Agents That Matter. arXiv:2407.01502.

Pan, J.; Zhang, Y.; Tomlin, N.; Zhou, Y.; Levine, S.; and Suhr, A. 2024. Autonomous Evaluation and Refinement of Digital Agents. arXiv:2404.06474.

Schwartz, S.; Yaeli, A.; and Shlomov, S. 2023. Enhancing Trust in LLM-Based AI Automation Agents: New Considerations and Future Challenges. arXiv:2308.05391.

Shetty, M.; Chen, Y.; Somashekar, G.; Ma, M.; Simmhan, Y.; Zhang, X.; Mace, J.; Vandevoorde, D.; Las-Casas, P.; Gupta, S. M.; Nath, S.; Bansal, C.; and Rajmohan, S. 2024. Building AI Agents for Autonomous Clouds: Challenges and Design Principles. arXiv:2407.12165.

White, R. W. 2024. Advancing the Search Frontier with AI Agents. arXiv:2311.01235.

Zhang, Z. J.; Schoop, E.; Nichols, J.; Mahajan, A.; and Swearngin, A. 2024. From Interaction to Impact: Towards Safer AI Agents Through Understanding and Evaluating UI Operation Impacts. arXiv:2410.09006.

Case Studies

Case Study 1: Stock Price Lookup

We demonstrate CAPTAIN's ability to handle a simple but common web task: looking up a stock price on Google. This case study illustrates the system's planning and execution processes in detail.

User Command: "What is NVIDIA's current stock price?"

Initial State

```
Browser started and ready
```

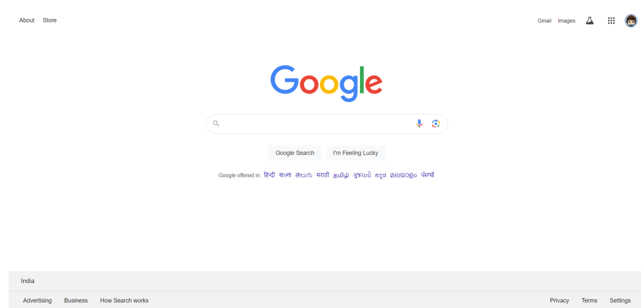

Figure 2: Initial state: Google homepage

Stage 1: Query Input The system generates and executes its first action:

```
Current Thought: I see it's the Google homepage in the
provided DOM representation. To find NVIDIA's current
stock price, I can simply type 'NVIDIA stock price' in
the search bar and click on the search button.

Plan:
 1. Type 'NVIDIA stock price' in the search bar
 2. Click on the search button
 3. Extract the current stock price from the results

Action: TYPE
Result: Success. Text "NVIDIA stock price" set successfully
```

Stage 2: Search Execution The system proceeds with the search action:

```
Current Thought: Since the search query has been entered,
the next step is to click on the 'Google Search' button
to get the search results.

Action: CLICK
Result: Click action completed, page may have navigated
```

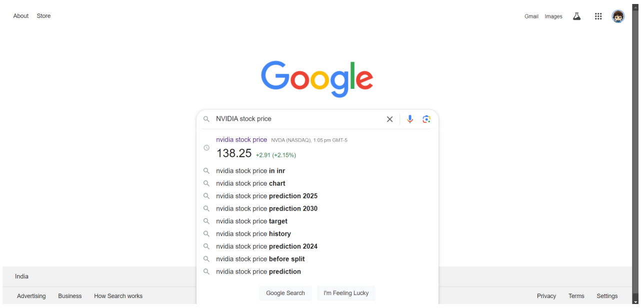

Figure 3: Stage 1: Entering search query

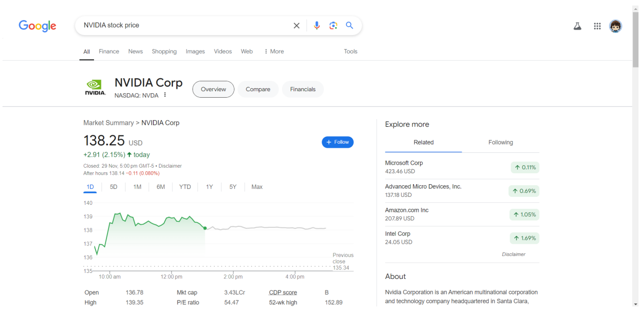

Figure 4: Stage 2: Search results showing stock price

Final Result The system successfully completes the task:

```
Final Response:
The current stock price of NVIDIA is
    $138.25 USD.
```

This case study demonstrates several key capabilities of CAPTAIN:

- Natural language understanding to interpret the user's query
- Strategic planning to break down the task into steps
- Dynamic interaction with web elements
- Information extraction from search results

Case Study 2: Flight Price Search

This case study demonstrates CAPTAIN's ability to handle complex travel booking interfaces, showcasing its staged approach to task completion and adaptive planning capabilities.

User Command: "Tell me the cheapest flights from Bangalore to SF"

Stage 1: Task Initialization The system begins by analyzing the task and initializing its components:

Stage 2: Travel Site Navigation The system executes its first action by navigating to an appropriate travel search engine:

```
Action - GOTO
Action results: ['Page loaded: https://
    www.skyscanner.co.in/
?previousCultureSource=COOKIE&
    redirectedFrom=www.skyscanner.com,
```

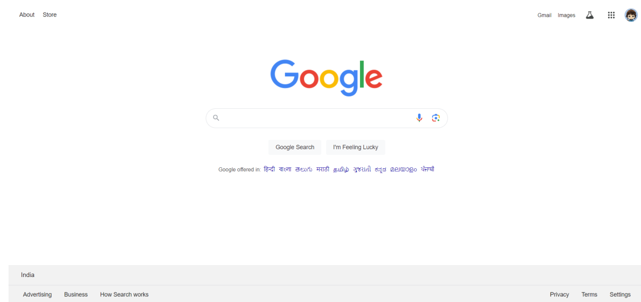

Figure 5: Stage 1: System initialization state

```
Title: Flight Ticket Booking: Cheap
    Flights & Air Tickets
| Skyscanner']
```

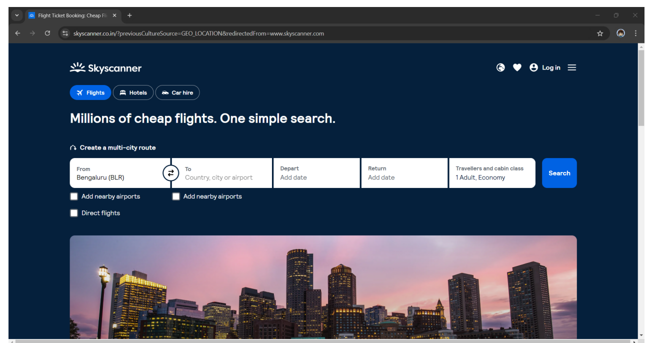

Figure 6: Stage 2: Successfully navigated to Skyscanner homepage

Stage 3: Strategic Planning The system formulates a comprehensive plan for completing the task:

```
Current Thought: I see it looks like the
     Google homepage
in the provided DOM representation. In
    order to find the
cheapest flights from Bangalore to SF, I
     should go to a
website like skyscanner and carry my
    searches over there...

Plan:
 1. Go to www.skyscanner.com
 2. List interaction options available
 3. Set from airport to 'Bangalore'
 4. Set destination airport to SF
 5. Select travel dates
 6. Confirm field values
 7. Click search button
 8. Confirm search results page
 9. Apply price filters
 10. Extract cheapest flight price

Current Task: Go to www.skyscanner.com
Completed Tasks:
  [\   ] 1. Go to www.skyscanner.com
```

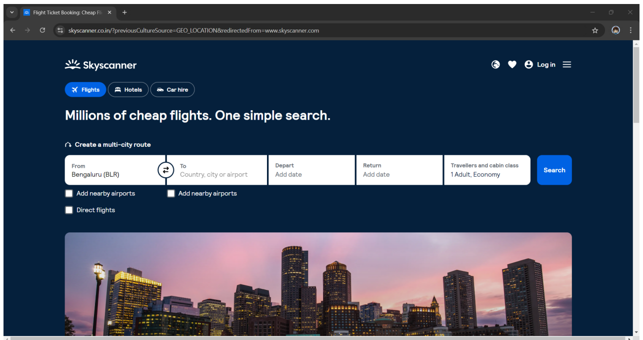

Figure 7: Stage 3: Plan formulation and initial task completion

Stage 4: Form Interaction The system begins interacting with the search form:

```
Action - TYPE
Action results: ['Successfully entered "
    Bangalore" in origin field']
Current Task: Set destination airport to
    SF
Completed Tasks:
  [\   ] Set from airport to 'Bangalore'
```

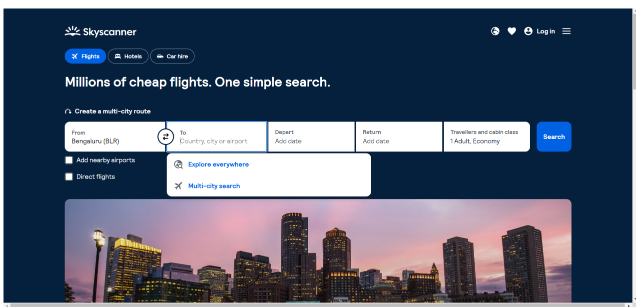

Figure 8: Stage 4: Origin city entry completed

Stage 5: Search Form Completion The system completes the form with destination details:

```
Action - TYPE
Action results: ['Successfully entered "
    San Francisco"
in destination field']
Current Task: Validate form entries
Completed Tasks:
  [\   ] Set origin and destination
      cities
  [\   ] Configure search parameters
```

Stage 6: Search Execution The system initiates the search and handles the loading state:

```
Action - CLICK
Action results: ['Click executed
    successfully, search
initiated']
Current Task: Monitor search results
    loading
```

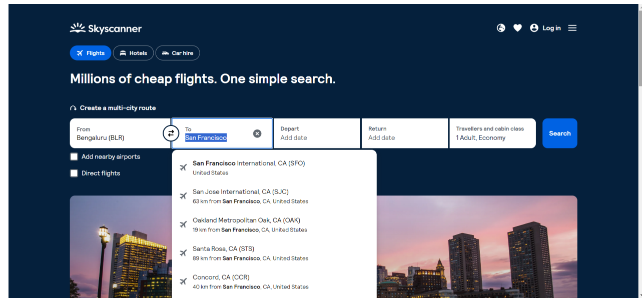

Figure 9: Stage 5: Search form fully populated

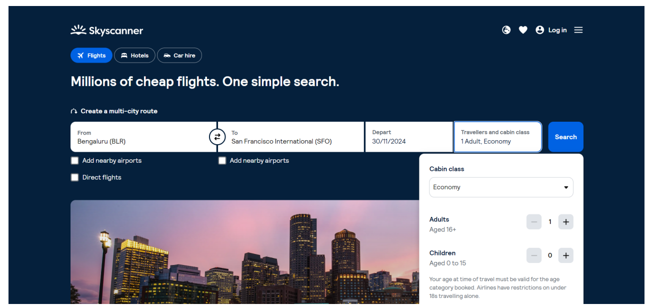

Figure 10: Stage 6: Search execution and loading state

Stage 7: Results Processing The system processes the search results and extracts pricing information:

```
Current Thought: Search results are
    loaded. Analyzing
prices to find the cheapest option.

Final Response:
The cheapest flight from Bengaluru to
    San Francisco
is  102 ,514.
```

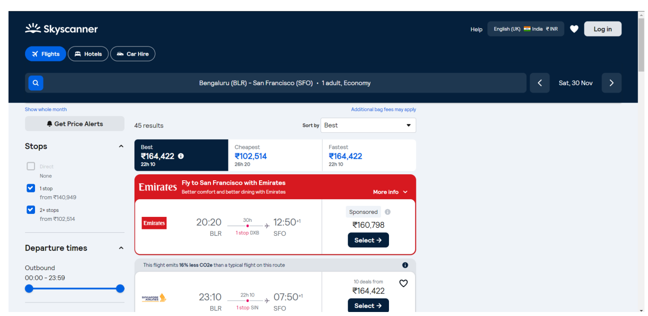

Figure 11: Stage 6: Search execution and loading state

This case study demonstrates several advanced capabilities of CAPTAIN:

1. **Strategic Planning:** Breaking down complex tasks into manageable steps and adapting the plan based on website state.
2. **Form Interaction:** Handling sophisticated web forms including autocomplete fields and dynamic validation.

3. **State Management:** Maintaining task context across multiple steps and handling asynchronous content loading.
4. **Error Resilience:** Managing potential failure points such as network delays and dynamic content updates.
5. **Data Extraction:** Processing structured data from complex search results and identifying relevant price information.

This demonstrating CAPTAIN's efficiency in handling complex web interfaces. The system successfully managed multiple state transitions, form interactions, and dynamic content loading while maintaining consistent progress toward the task objective.