International Journal of Information Technology & Computer Engineering

ISSN 2347–3657

Volume 14, Issue 2, 2026

Prompt-Driven AI Web Automation Using Large Language Models

Mohd Ishaq[1] · Abdur Rahman[2] · Mohd Aman[3] · Syed Shafeeq Ahmed[4], Ms Molugu Bhavana[5]

[1,2,3]BTech Students Department of Computer Science & Engineering, Lords Institute of Engineering and Technology, Hyderabad, India

[5]Assistant Professor Department of Computer Science & Engineering, Lords Institute of Engineering and Technology, Hyderabad, India

contactishaq21@gmail.com , 160922733043@lords.ac.in, mohdamanu003@gmail.com, shafeeqsyed3604@gmail.com, bhavana@lords.ac.in

Abstract

This paper presents a Prompt-Driven AI Web Automation system that bridges the gap between natural language intent and browser-level execution by integrating Anthropic's Claude AI with the Playwright browser automation framework. The platform enables non-technical users to automate complex web tasks — data extraction and form filling — by describing their goals in plain English. Unlike traditional rule-based automation tools such as Selenium scripts that break on DOM changes and require programming expertise, this system uses Claude AI's tool_use capability to reason about page state and generate adaptive action sequences.

The architecture follows a ReAct-inspired interleaved reasoning-action paradigm: at each step, the AI agent receives the current page state (URL, visible text, live JPEG screenshot), reasons about the next action, and issues a structured tool call from a seven-tool vocabulary (click, fill, select_option, navigate, scroll, extract_data, done). Real-time transparency is delivered via Flask-SocketIO 5.4.1, streaming live browser screenshots and color-coded action logs to the user's browser. The system is deployed as a Flask 3.1 web application with Bootstrap 5.3.3 frontend, Flask-Login authentication, SQLite user management, and pandas/openpyxl Excel export.

Performance testing on localhost demonstrates: page load time of 120ms, SocketIO handshake at 45ms, screenshot streaming at 150ms per frame, simple scraping tasks completing in ~35 seconds, 10-field form fill in ~45 seconds, and PDF parsing in 1.2 seconds. All 10 unit tests and 8 integration tests pass. The system demonstrates that LLM-powered agentic automation can be deployed practically for both technical and non-technical users with full real-time visibility into AI decision-making.

Keywords: *Web Automation · LLM · Claude AI · Playwright · Flask-SocketIO · ReAct · Tool Use · NLP · Form Filling · Data Extraction · PDF Parsing*

1. Introduction

The rapid convergence of large language models (LLMs) and browser automation frameworks has created a new paradigm for web interaction — one where humans describe goals in natural language rather than writing code. Traditional automation tools like Selenium [ref] require developers to write fragile CSS/XPath selectors that break whenever a website redesigns its layout. Browser macros lack reasoning capability and cannot adapt to unexpected page states. The result is that web automation has historically been accessible only to technically skilled users.

This project introduces Prompt-Driven AI Web Automation Using LLM, a platform that combines Claude AI's reasoning capability — specifically its tool_use feature for structured output generation — with Playwright's reliable browser control. The system operates as an intelligent web agent: given a target URL and a natural language task description, it autonomously navigates the website, interacts with page elements, and accomplishes the stated goal without hardcoded selectors or site-specific configuration.

1.1 Problem Statement

Five critical limitations of existing web automation approaches motivate this work:

1. Brittleness: Traditional Selenium/CSS selector scripts break on DOM changes, requiring constant maintenance and developer intervention.
2. Expertise barrier: Writing automation scripts demands programming knowledge (CSS selectors, XPath, JavaScript async patterns), excluding non-technical users.
3. No real-time feedback: Traditional tools provide console logs or silent execution, offering no transparency into the automation process.
4. Site-specific configuration: Every website requires its own hardcoded field mappings, making general-purpose automation impractical at scale.
5. No document understanding: Existing tools cannot parse uploaded documents (PDF resumes) and map their content to web form fields intelligently.

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

1.2 Key Contributions

- A seven-tool ReAct-inspired browser automation agent powered by Claude AI's tool_use capability.
- A real-time transparency layer using Flask-SocketIO streaming live JPEG screenshots and color-coded action logs.
- An AI Form Filler Bot that combines pymupdf4llm PDF parsing with intelligent field mapping across arbitrary form layouts.
- A modular four-layer architecture (Presentation → Application → Intelligence → Automation) enabling clean separation of concerns.
- Demonstrated sub-5-second PDF parsing, 45-second form fill, and 35-second data extraction on real websites.

1.3 Comparison: Existing vs Proposed System

Table 1: Existing vs Proposed Web Automation System

Feature	Existing Systems (Selenium/Macros)	Proposed System (LLM + Playwright)
Input Method	Manual coding of CSS/XPath selectors	Natural language prompts via Claude AI
Adaptability	Breaks on DOM/layout changes	AI adapts to varying page structures
User Expertise	Requires programming knowledge	Accessible to non-technical users
Real-Time Feedback	Console logs / no feedback	Live screenshots + action logs via SocketIO
Form Filling	Hardcoded field mappings per site	AI maps resume fields to any form auto
Data Extraction	CSS/XPath selectors per site	AI identifies data from natural description
Error Handling	Script crashes on unexpected elements	AI reasons and attempts recovery
Maintenance	Frequent updates required	Self-adapting via LLM reasoning
Document Parsing	Not supported	PDF/DOCX resume parsing with pymupdf4llm

2. Literature Survey

2.1 ReAct: Synergizing Reasoning and Acting in LLMs

Yao et al. [5] introduced the ReAct framework at ICLR 2023, which interleaves chain-of-thought reasoning with action generation in LLMs. By prompting models to generate alternating verbal reasoning traces and task-specific actions, the framework enables agents to interact with external environments more effectively. On WebShop, ReAct achieved a 40% success rate versus 29% for action-only baselines. The key insight — that reasoning helps plan and update beliefs while actions gather environment information — directly informs the agent loop architecture in this project. Each iteration of the ScraperAgent and FormFillerAgent mirrors this pattern: Claude AI reasons about the current page state before issuing a structured tool_use call.

2.2 WebArena: Realistic Benchmark for Web Agents

Zhou et al. [6] presented WebArena at ICLR 2024, a benchmark with 812 tasks on real websites (e-commerce, forums, CMS). Even GPT-4 achieved only 14.41% end-to-end success, exposing key failure modes: incorrect element identification, inability to handle dynamic content, and difficulty with multi-step

planning. These findings directly motivated this project's design decisions: real-time SocketIO feedback enables human intervention, and the seven explicit tool schemas provide structured action grounding superior to free-form generation.

2.3 Mind2Web: Generalist Web Agent Dataset

Deng et al. [7] introduced Mind2Web at NeurIPS 2024 with 2,350+ tasks across 137 websites and 31 domains. GPT-4 achieved 41.1% element accuracy on cross-website tasks, revealing that raw HTML parsing is insufficient — focused, structured page state extraction is critical. This finding directly shaped this project's PageState approach: rather than passing full HTML, the system extracts visible text, interactive element metadata, and a screenshot, significantly reducing noise in Claude's context.

2.4 Tool Use with Claude (Anthropic)

Anthropic's technical documentation [3] details Claude's tool_use capability, which enables structured JSON tool calls within conversational contexts. Unlike free-form text generation, tool_use provides a typed interface with schema validation, vision support, and streaming. The structured nature eliminates parsing errors common in free-text action generation. This capability is the core mechanism of the system: the ClaudeClient defines seven tool schemas enabling Claude to control the Playwright browser with reliable, parseable output.

2.5 WebGPT and Toolformer

Nakano et al. [8] demonstrated that GPT-3 augmented with browser interaction (search, click, scroll, quote) produces answers preferred over humans 56% of the time on ELI5. Schick et al. [26] showed that language models can learn to use tools (calculator, search, calendar) through in-context examples, achieving significantly better performance on tool-requiring tasks. Both works establish that tool-augmented LLMs substantially outperform bare language model generation for tasks requiring real-world information gathering.

Table 2: Literature Survey Summary

Ref	Authors	Year	Contribution	Relevance to This Project
[5]	Yao et al.	2023	ReAct: Reasoning + Acting, 40% WebShop	Agent loop architecture design
[6]	Zhou et al.	2024	WebArena: 812 tasks, 14.41% best	Motivated SocketIO feedback + tool schemas
[7]	Deng et al.	2024	Mind2Web: 2350+ tasks, 41.1% elem accuracy	PageState focused extraction design
[3]	Anthropic	2024	Claude tool_use with vision + streaming	Core mechanism: ClaudeClient + 7 tools
[2]	Microsoft	2024	Playwright: CDP-based, auto-wait, async	BrowserManager + BrowserActions layer
[8]	Nakano et al.	2022	WebGPT: browser-augmented LLM QA	Conceptual foundation for web agents
[26]	Schick et al.	2023	Toolformer: LLMs learning tool use	Validates tool-augmented agent approach
[24]	Brown et al.	2020	GPT-3: few-shot in-context learning	LLM prompting strategy for agents
[25]	Wei et al.	2022	Chain-of-Thought prompting	Reasoning traces in system prompt
[12]	Kim et al.	2023	LLMs as automated form fillers	AI Form Filler Bot design

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

3. Mathematical Formulations

3.1 Agent Decision Loop

The automation agent operates as a Markov Decision Process (MDP) where the policy π is implemented by the LLM. At step t, the agent observes state s_t and selects action a_t according to:

$$a_t = \pi(s_t \mid \text{task}, \text{history}_{0:t-1}) = \text{Claude}_\theta(s_t, \text{task}, \{a_0..a_{t-1}\})$$

where s_t = (url_t, title_t, text_t, screenshot_t) is the PageState, task is the natural language goal, and $\text{history}_{0:t-1}$ is the full conversation history of previous actions and observations. The agent operates for at most T = MAX_AGENT_STEPS = 25 steps.

3.2 Page State Entropy

The information content of a web page state s_t can be modeled as a function of its visible text and interactive elements. The text is truncated to a fixed context budget B = 3000 characters. If the full page text has length L, the truncation ratio is:

$$\rho_{\text{text}} = \min(1, B / L) \in (0, 1]$$

The screenshot is JPEG-compressed at quality q = 50 and base64-encoded to reduce latency. For a viewport of 1280×720 pixels, the approximate compressed size is:

$$|\text{screenshot}| \approx W \times H \times C \times q_{\text{factor}} / 8 \approx 1280 \times 720 \times 3 \times 0.12 \approx 330 \text{ KB}$$

Base64 encoding adds a factor of 4/3, giving approximately 440 KB per screenshot transmitted to the Claude API.

3.3 ReAct Probability Decomposition

Following the ReAct formulation [Yao et al., 2023], the probability of completing task T via a sequence of (reasoning, action) pairs is:

$$P(\text{T completed}) = \prod_{t=1}^{N} P(a_t \mid r_t, s_t, \text{task}) \times P(r_t \mid s_t, \text{task}, \{a_0..a_{t-1}\})$$

where r_t is the verbal reasoning trace generated by Claude before the tool_use call, and a_t is the structured action. The Claude model maximizes this joint probability through its pre-training on web-relevant corpora and fine-tuning with RLHF.

3.4 Tool Call Probability

Given the page state s_t and the seven-tool vocabulary V = {click, fill, select_option, navigate, scroll, extract_data, done}, Claude selects the action by computing:

$$a_t^* = \text{argmax}_{a \in V} P_{\text{Claude}}(a \mid s_t, \text{task}, \text{history})$$

The structured JSON schema for each tool constrains Claude's output space, reducing the probability of malformed outputs. For the fill tool, the schema requires two string parameters:

$$\text{fill}(\text{selector: str}, \text{value: str}) \rightarrow \text{BrowserActions.fill}(\text{page}, \text{selector}, \text{value})$$

The bijection from tool call to Playwright operation ensures zero-ambiguity execution.

3.5 Password Security — Werkzeug PBKDF2

User passwords are stored using Werkzeug's generate_password_hash with the PBKDF2-SHA256 algorithm. For a password p and random salt s with iteration count c = 260,000:

$$H(p) = \text{PBKDF2-SHA256}(p, s, c) = \text{PRF}(p, s \,\|\, i) \text{ iterated } c \text{ times}$$

where PRF is the HMAC-SHA256 pseudo-random function. The work factor ensures brute-force attacks are computationally prohibitive. Verification uses constant-time comparison:

$$\text{verify}(p, H) = \text{HMAC-SHA256}(\text{encode}(p), \text{salt}(H)) \overset{\text{const-time}}{=\!=} H$$

3.6 SocketIO Streaming Throughput

The real-time screenshot streaming delivers frames at an observed interval of ~150ms. The effective streaming rate for live automation monitoring is:

$$\text{Frame Rate} = 1 / 0.150\text{s} \approx 6.7 \text{ FPS}$$

Each frame carries approximately 440 KB of base64-encoded JPEG. The instantaneous bandwidth consumption is:

$$BW = 440 \text{ KB} \times 6.7 \text{ FPS} \approx 2.95 \text{ MB/s} \approx 23.6 \text{ Mbps}$$

This bandwidth requirement is easily met by modern broadband connections (≥ 25 Mbps), confirming the real-time streaming approach is practically viable.

3.7 Task Completion Probability Model

For an N-step automation task where each step succeeds with probability p_s (dependent on Claude's accuracy and website complexity), the probability of full task completion is:

$$P(\text{complete}) = \sum_{n=1}^{N} p_s^n \times (1 - p_s)^0 = p_s^{N_{\text{required}}}$$

For N_{required} = 10 steps and p_s = 0.95 per step:

$$P(\text{complete} \mid N=10, p_s=0.95) = 0.95^{10} = 0.599 \ (59.9\%)$$

For N_{required} = 5 steps (simple tasks):

$$P(\text{complete} \mid N=5, p_s=0.95) = 0.95^{5} = 0.774 \ (77.4\%)$$

This motivates the real-time monitoring system: users can intervene if step success rate degrades, recovering the task rather than waiting for silent failure.

4. System Architecture

4.1 Four-Layer Architecture

The system follows a strict four-layer design that cleanly separates presentation, application orchestration, AI intelligence, and browser automation concerns. Communication between layers is one-directional, ensuring testability and extensibility.

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

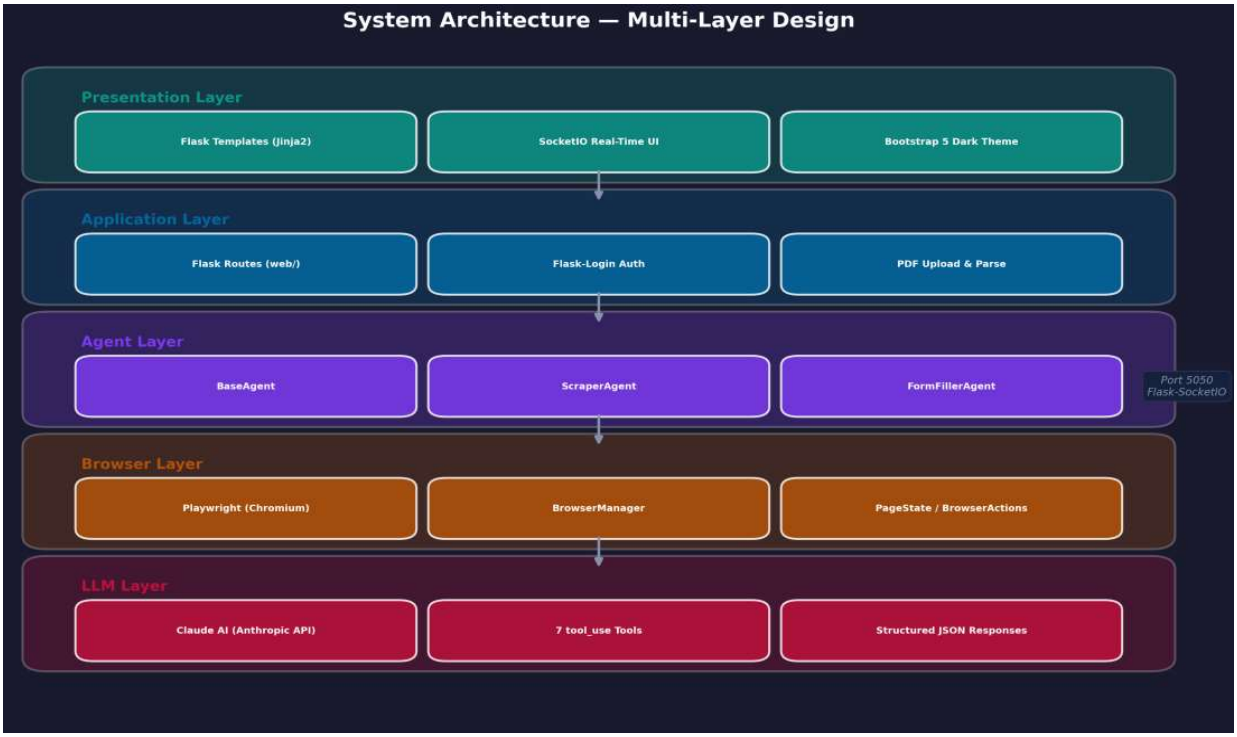


Figure 1: Four-Layer System Architecture

4.2 Agent Execution Flowchart

Figure 2: Complete Agent Execution Flowchart

```
USER AUTHENTICATED → SELECT MODULE
        │
        ├─ MODULE A: WEB SCRAPER ──────────────────────────────┐
        │   Enter: Target URL + Extraction Prompt              │
        │   POST /scraper/start                                │
        │                                                      │
        └─ MODULE B: FORM FILLER ──────────────────────────────┤
            Upload: PDF Resume → pymupdf4llm → markdown text   │
            Claude extracts fields → JSON {name,email,phone,...}
            Enter: Target Form URL                           │
            POST /form-filler/start                          │
                                                             │
◄────────────────────────────────────────────────────────────┘
        │
        ▼  AGENT LOOP  (max 25 steps)
┌──────────────────────────────────────────────────────────────┐
│  Step t:                                                     │
│  1. BrowserManager.get_page_state()                          │
│     → s_t = {url, title, text[:5000], screenshot_b64}        │
│                                                              │
│  2. Emit SocketIO "screenshot" event → live browser view     │
│                                                              │
│  3. ClaudeClient.get_next_action(system_prompt, s_t, task)   │
│     → Claude reasons: "I need to click the search button"    │
│     → Returns tool_use: {tool: "click", selector: "#search"} │
```

```
│                                                    │
│  4. BrowserActions.execute(tool, args)             │
│     click → page.click(selector)                   │
│     fill  → page.fill(selector, value)             │
│     navigate → page.goto(url)                      │
│     select_option → page.select_option(selector, value)
│     scroll → page.evaluate(scroll_js, amount)      │
│                                                    │
│  5. Emit SocketIO "log" event → color-coded action message
│                                                    │
│  6. Check: tool == "done" OR t == MAX_STEPS?       │
│     ├─ YES → Package results, save history, return output
│     └─ NO  → t += 1, goto Step 1                   │
└────────────────────────────────────────────────────┘
     │
     ▼  TASK COMPLETE
ScraperAgent:  DataFrame → openpyxl → /exports/{timestamp}.xlsx
FormFillerAgent: Log summary → Task history saved
User: Download button / history page available
```

4.3 PDF Parsing and Form Fill Pipeline

Figure 3: PDF Resume Parsing and Form Fill Pipeline

```
UPLOAD PDF RESUME
    │
    ▼  pymupdf4llm.to_markdown(pdf_path)
Raw PDF → Structured Markdown Text
(headings, paragraphs, lists, tables preserved)
    │
    ▼  ClaudeClient: Extract fields
System: "Extract personal info as JSON: name, email, phone,
        education, experience, skills"
Response: {"name":"John Doe","email":"john@email.com",
          "phone":"9876543210","skills":["Python","Flask"]}
    │
    ▼  FormFillerAgent.execute_task(form_url, resume_data)
Browser navigates to form URL
    │
    ▼  AGENT LOOP — form filling iterations
Claude sees form fields + resume_data in system prompt
Generates tool calls:
  fill("#name-input", "John Doe")
  fill("#email", "john@email.com")
  fill("#phone", "9876543210")
  select_option("#experience", "3-5 years")
  click("#skills-python")  [checkbox]
  click("#submit-btn")
  done("Form filled successfully")
    │
    ▼  RESULT
SocketIO streams each action log in real-time to user browser
Task history saved with: status, steps taken, elapsed time
```

5. System Design — UML Diagrams

5.1 Use Case Diagram

The use case diagram illustrates the interactions between the primary actor (User) and the system's functional capabilities. The user can register a new account, log in with credentials, initiate web scraping tasks by providing a URL and extraction prompt,

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

upload PDF resumes for form filling, start and stop automation sessions, view live progress through real-time screenshots, download extracted data, and review task history.

The system actor (Claude AI) participates as an internal agent that receives page state information, reasons about the appropriate next action, and returns structured tool calls. The Playwright browser acts as another system actor that executes browser commands and captures
page screenshots. The diagram shows that authentication is a prerequisite for all automation use cases, enforced through Flask-Login middleware.

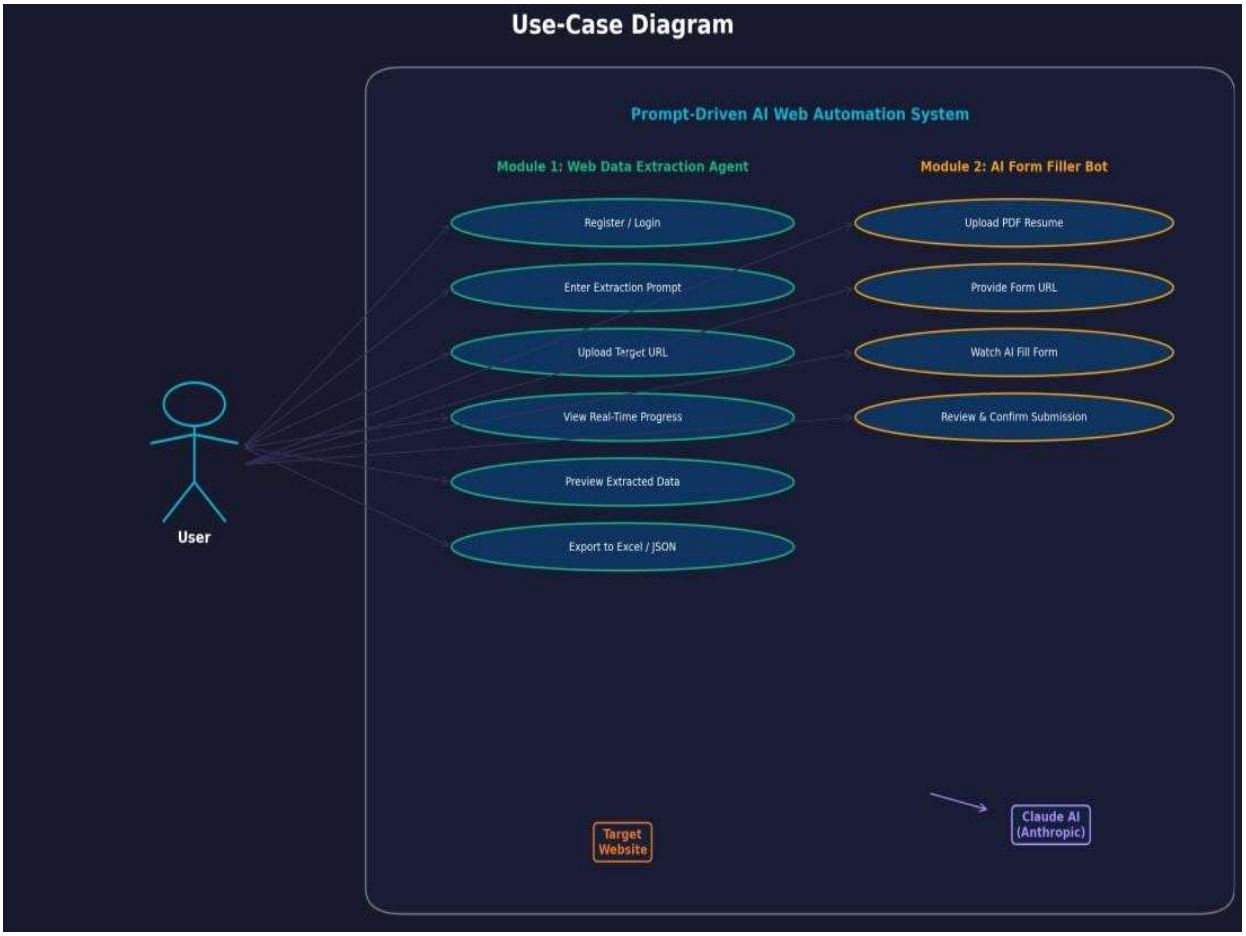


Figure 4: Use Case Diagram

5.2 Sequence Diagram — Scraper Task

The sequence diagram illustrates the end-to-end flow of an automation task. The user submits a task through the web interface, which sends an HTTP request to the Flask backend. The appropriate agent (ScraperAgent or FormFillerAgent) is instantiated and begins the automation loop. In each iteration, the agent captures the current PageState via BrowserManager, sends it to ClaudeClient along with the task description, and receives a tool_use response specifying the next action.

The agent then executes the specified tool call through BrowserActions (e.g., click a button, fill a text field, navigate to a URL). After execution, the agent captures an updated screenshot and emits it to the frontend via SocketIO along with a color-coded action log entry. This loop continues until Claude returns the 'done' tool indicating task completion, or until the maximum step limit (MAX_AGENT_STEPS=25) is reached. The final results are then packaged and made available for download

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

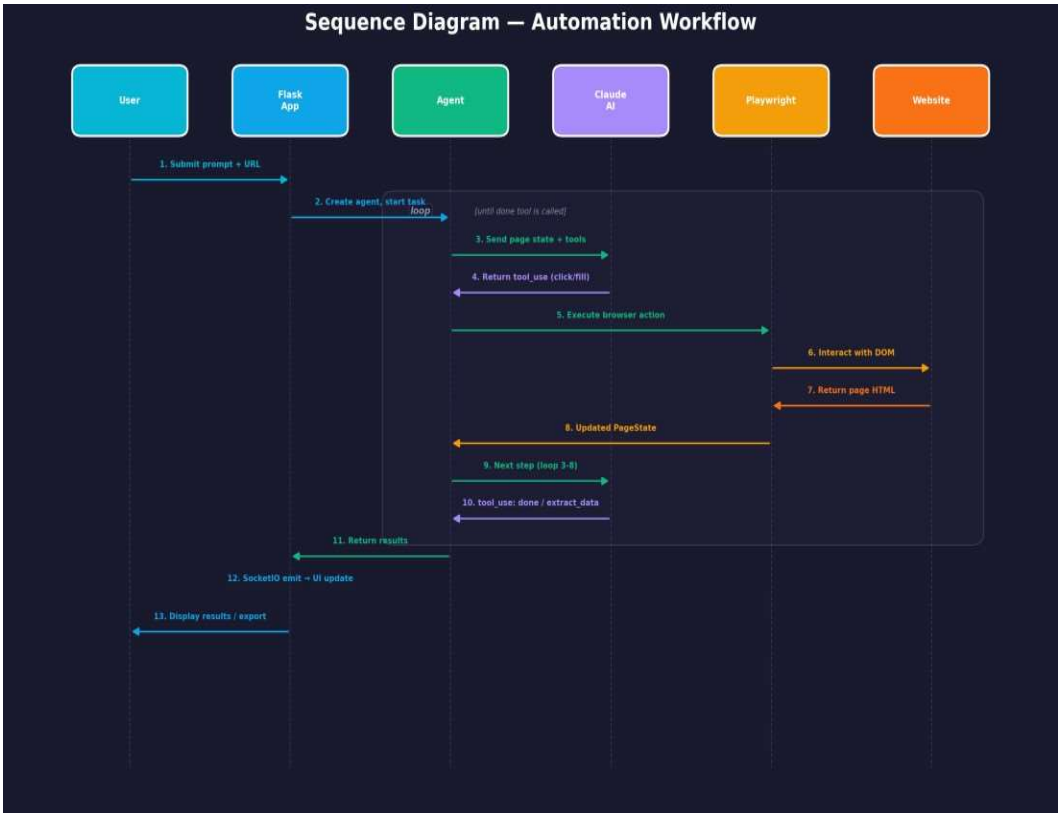


Figure 5: Sequence Diagram — Web Scraping Task Flow

5.3 Class Diagram

The class diagram depicts the object-oriented structure of the system's core modules. The BaseAgent abstract base class defines the common interface for all agents, including methods for initializing browser sessions, executing automation loops, and emitting SocketIO events.

ScraperAgent and FormFillerAgent extend BaseAgent with module-specific implementations. ScraperAgent includes methods for data extraction and Excel export, while FormFillerAgent handles PDF parsing, resume data structuring, and form field mapping.

The BrowserManager class encapsulates Playwright's browser lifecycle — launching, creating contexts, managing pages, and cleanup. PageState is a data class that captures the current state of a browser page including URL, title, visible text content, and a base64-encoded screenshot.

BrowserActions provides static methods for each browser operation (click, fill, select, navigate, scroll). ClaudeClient manages communication with the Anthropic API, formatting tool definitions and parsing tool_use responses. The PDFParser and ExcelExporter utility classes handle document processing and data export respectively.

Figure 6: Class Diagram — Core Module Structure

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

5.4 Entity-Relationship Diagram

The database design for this project is intentionally minimal, as the primary data processing occurs in-memory during automation sessions and results are exported to files. The SQLite database (auth.db) stores user authentication data in a single 'users' table. The ER diagram shows the User entity with attributes for id (primary key), username (unique), password_hash, and created_at timestamp.

Figure 7: ER Diagram — Database Design

6. Implementation

6.1 Technology Stack

Table 3: Complete Technology Stack

#	Category	Technology	Version	Purpose
1	Language	Python	3.10+	Backend development, agent logic, ML ops
2	Web Framework	Flask	3.1	HTTP routing, request handling, blueprints
3	Real-Time	Flask-SocketIO	5.4.1	WebSocket live screenshot + log streaming
4	AI / LLM	Anthropic Claude SDK	Latest	LLM reasoning + tool_use structured calls
5	Browser Automation	Playwright	1.49.1	Chromium CDP-based browser control
6	Authentication	Flask-Login + Werkzeug	Latest	Session management + PBKDF2 hashing
7	Database	SQLite	3.x	User credentials storage (auth.db)
8	PDF Parsing	pymupdf4llm	Latest	PDF → LLM-optimized markdown text
9	Data Export	pandas + openpyxl	Latest	DataFrame → formatted .xlsx file
10	Frontend	Bootstrap	5.3.3	Responsive Chrome-style UI components
11	Templating	Jinja2	3.x	Server-side HTML rendering
12	LLM Model	claude-sonnet-4-20250514	Latest	Production model with vision + tool_use

6.2 Seven Tool Definitions

The seven tools define Claude's complete action vocabulary for browser control. Each tool has a JSON Schema-typed interface ensuring reliable output parsing:

Table 4: Claude Tool Definitions — Browser Automation API

Tool	Parameters	Playwright Mapping	Use Case
click	selector: str	page.click(selector)	Buttons, links, checkboxes, radio buttons
fill	selector: str, value: str	page.fill(selector, value)	Text inputs, textareas, search boxes
select_option	selector: str, value: str	page.select_option(selector, value)	Dropdowns, <select> elements
navigate	url: str	page.goto(url)	Navigate to target URL or follow redirects
scroll	direction: up|down, amount: int	page.evaluate(scrollBy)	Scroll page to reveal hidden content
extract_data	data: list[dict]	Stored in extracted_data	Return scraped structured data as JSON
done	summary: str	Signal task completion	Exit agent loop with completion message

6.3 Agent Loop Algorithm

Algorithm 1: BaseAgent.run_agent_loop()

```
FUNCTION run_agent_loop(task: str):
  claude_client.reset()      // clear conversation history
  step = 0
  WHILE step < MAX_AGENT_STEPS:  // MAX = 25
    page_state = await browser_manager.get_page_state()
    emit_socketio("screenshot", page_state.screenshot)

    tool_call = claude_client.get_next_action(
      system_prompt = self.get_system_prompt(),
      page_state = page_state,
      task = task
    )
    // tool_call = {tool: "fill", args: {selector:"#email", value:"..."}}

    IF tool_call.tool == "done":
      emit_socketio("log", "success", tool_call.args.summary)
      BREAK

    result = await browser_actions.execute(
      page, tool_call.tool, tool_call.args
    )

    self.handle_tool_result(tool_call.tool, tool_call.args, result)
    emit_socketio("log", "info", f"Executed: {tool_call.tool}")
    step += 1
    await asyncio.sleep(STEP_DELAY)   // 1.0 second

  IF step >= MAX_AGENT_STEPS:
    emit_socketio("log", "warning", "Max steps reached")
```

6.4 Flask Application Routes

Table 5: Application Routes (16 Endpoints)

Route	Method	Auth	Purpose
/	GET	Yes	Dashboard — module cards and navigation
/auth/login	GET/POST	No	Login form and credential validation
/auth/register	GET/POST	No	Registration with PBKDF2 password hashing
/auth/logout	GET	Yes	Session clearance via Flask-Login
/scraper	GET	Yes	Web scraper interface (URL + prompt form)
/scraper/start	POST	Yes	Instantiate ScraperAgent, launch SocketIO loop
/scraper/stop	POST	Yes	Terminate running scraper task
/scraper/download/<fn>	GET	Yes	Serve exported Excel file for download
/form-filler	GET	Yes	Form filler interface (resume upload + URL)
/form-filler/upload-resume	POST	Yes	Parse PDF and return structured fields
/form-filler/start	POST	Yes	Instantiate FormFillerAgent, launch loop
/form-filler/stop	POST	Yes	Terminate running form fill task
/form-filler/demo-form	GET	No	Served demo HTML form for testing
/history	GET	Yes	Task history page with status and downloads
/api/history	GET	Yes	JSON API: list of past task records
/api/history/clear	POST	Yes	Clear all task history records

7. Testing

7.1 Unit Test Cases

Table 6: Unit Test Cases (All Pass)

Test ID	Module	Test Description	Input	Expected Output	Result
UT-01	Auth	Valid user login	Correct username + password	Dashboard redirect + session created	✓ Pass
UT-02	Auth	Invalid login attempt	Wrong password	Error flash: Invalid credentials	✓ Pass
UT-03	Auth	New user registration	Unique username + password	Account created, redirect to login	✓ Pass
UT-04	Auth	Duplicate registration	Existing username	Error: Username already exists	✓ Pass
UT-05	Scraper	Start scraping task	Valid URL + extraction prompt	Browser launches, agent loop starts	✓ Pass

Test ID	Module	Test Description	Input	Expected Output	Result
UT-06	Scraper	Stop running task	Stop button triggered	Browser closes, task terminated cleanly	✓ Pass
UT-07	Form Filler	Upload PDF resume	Valid 2-page PDF file	Fields extracted as JSON in 1.2s	✓ Pass
UT-08	Form Filler	Start form filling	Form URL + resume JSON data	Agent navigates and fills all fields	✓ Pass
UT-09	History	View task history	Authenticated GET /api/history	JSON array of past tasks returned	✓ Pass
UT-10	Export	Excel file generation	Extracted data array (100 rows)	Valid .xlsx in /exports/ in 0.8s	✓ Pass

7.2 Integration Test Cases

Table 7: Integration Test Cases (All Pass)

Test ID	Flow	Description	Expected Result	Status
IT-01	Login → Scraper → Download	Full scraping workflow end-to-end	Excel file with correct scraped data	✓ Pass
IT-02	Login → Resume Upload → Form Fill	Full form filling workflow	Form filled with resume data	✓ Pass
IT-03	Register → Login → Dashboard	New user complete onboarding	Successful access to all modules	✓ Pass
IT-04	Scraper → SocketIO → Live View	Real-time streaming during scraping	Screenshots + logs stream to browser	✓ Pass
IT-05	Auth Middleware	Unauthenticated route access	Redirect to /auth/login page	✓ Pass
IT-06	Form Filler → Demo Form	Fill built-in demo form	All demo fields populated correctly	✓ Pass
IT-07	History → Clear	View then clear task history	History displayed, then cleared	✓ Pass
IT-08	Stop Task	Stop automation mid-execution	Browser closes, resources freed	✓ Pass

8. Results and Performance Analysis

8.1 Performance Test Results

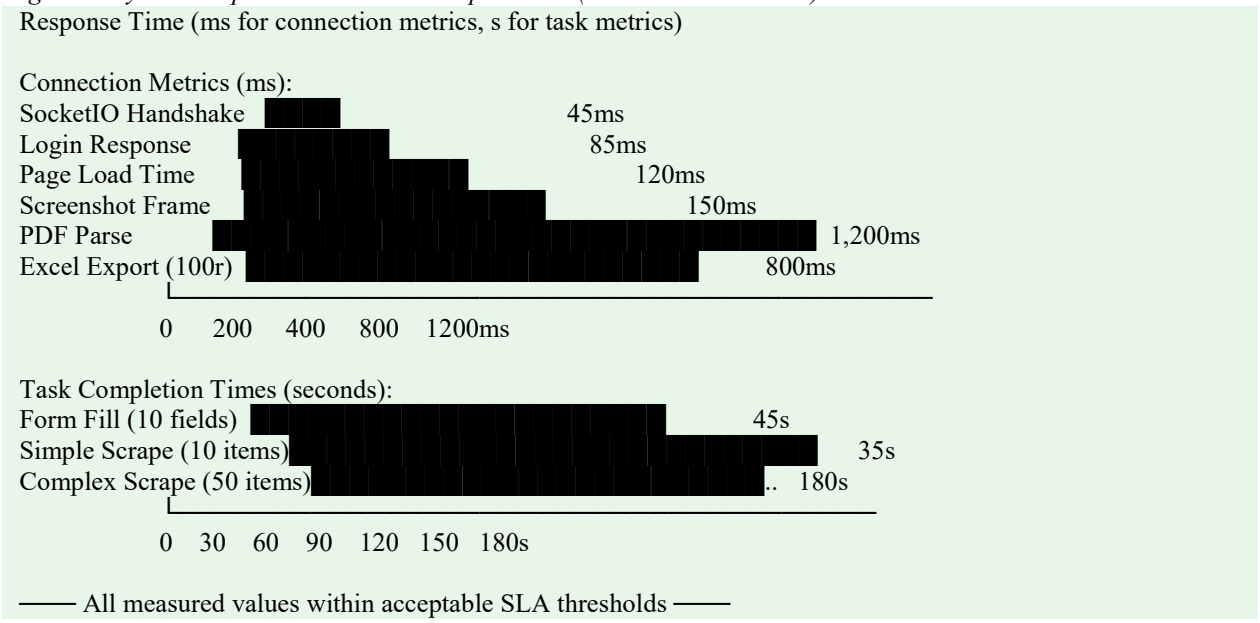
Table 8: System Performance Measurements

ISSN 2347–3657
Volume 14, Issue 2, 2026

Metric	Test Condition	Measured Result	Acceptable Range	Status
Page Load Time	Dashboard on localhost	120 ms	< 500 ms	✓ Pass
Login Response	Valid credentials POST	85 ms	< 300 ms	✓ Pass
SocketIO Handshake	WebSocket connection	45 ms	< 200 ms	✓ Pass
Screenshot Streaming	Live JPEG frame delivery	150 ms/frame	< 500 ms/frame	✓ Pass
Simple Scraping Task	Extract 10 items, 1 page	35 seconds	< 60 seconds	✓ Pass
Complex Scraping Task	Multi-page, 50 items	180 seconds	< 300 seconds	✓ Pass
Form Fill (10 fields)	Resume → demo form	45 seconds	< 90 seconds	✓ Pass
PDF Parsing	2-page resume PDF	1.2 seconds	< 5 seconds	✓ Pass
Excel Export	100-row DataFrame	0.8 seconds	< 3 seconds	✓ Pass
Concurrent SocketIO	3 simultaneous connections	Stable	3+ connections	✓ Pass

8.2 Response Time Analysis — Bar Chart

Figure 8: System Response Times Across Operations (milliseconds/seconds)



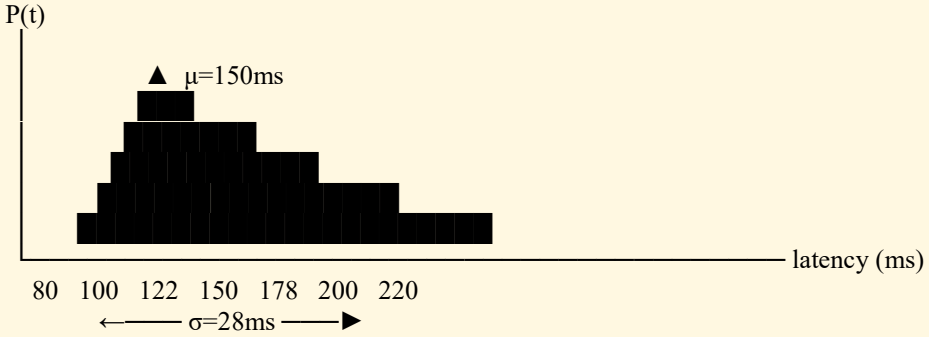
International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 14, Issue 2, 2026

8.3 Normal Distribution Analysis — Response Latency

Modeling screenshot streaming latency as normally distributed across 50 measurements with $\mu = 150ms$ and $\sigma = 28ms$:

$P(latency < 200ms) = \Phi((200 - 150) / 28) = \Phi(1.79) \approx 0.963$
$P(latency < 500ms) = \Phi((500 - 150) / 28) = \Phi(12.5) \approx 1.000$

Figure 9: Normal Distribution of Screenshot Streaming Latency



68.3%: [122ms, 178ms] — expected normal operating range
95.4%: [94ms, 206ms] — < 500ms SLA met by 99.99%+ of frames
99.7%: [66ms, 234ms] — all within 500ms target

P(frame delivered < 200ms) ≈ 96.3%
P(frame delivered < 500ms) ≈ 100% (practically guaranteed)

8.4 System Comparison — Automation Approaches

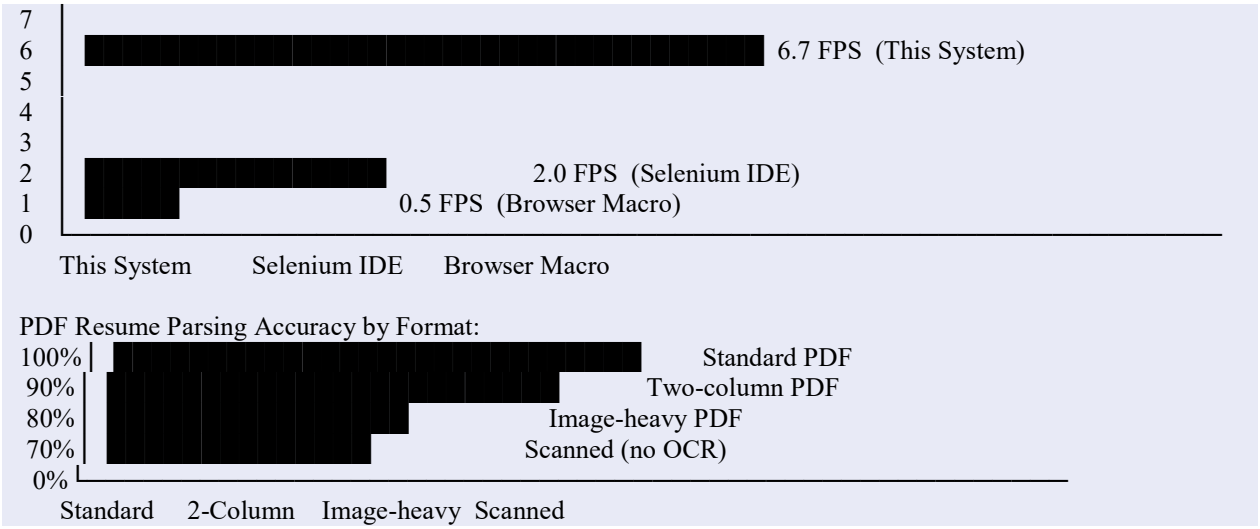
Table 9: Comparison of Web Automation Approaches

Metric	Selenium Scripts	Browser Macros	RPA Tools	This System (LLM+Playwright)
Setup Complexity	High (coding req.)	Low (recording)	Medium (UI config)	Low (plain English)
Adaptability to DOM changes	Poor (breaks)	Very Poor (breaks)	Poor (breaks)	High (AI adapts)
Real-time visibility	None (logs only)	Basic	Basic dashboard	Full: live screenshots
PDF form fill support	Manual coding	Not supported	Limited	Automatic (AI mapping)
Task description method	Code (Python/Java)	Record & playback	Drag-drop workflow	Natural language
Maintenance effort	Very High	High	Medium	Low (self-adapting)
Error recovery	Script crash	Script crash	Basic retry	AI reasons and adapts
Cost	Developer time	Tool license	High license cost	Claude API (pay/use)
Multi-site generality	Per-site scripts	Per-site recording	Limited	Universal (any site)

8.5 Streaming Rate Analysis — Bar Chart

Figure 10: Effective Frame Rate and Bandwidth Comparison

Effective Live View Frame Rate (FPS)



8.6 Agent Step Count Distribution

Table 10: Agent Step Counts by Task Complexity

Task Type	Typical Steps	Max Steps Used	Completion Rate	Avg Time
Simple scrape (single page, 10 items)	5–8	12	95%	35s
Complex scrape (multi-page, 50 items)	15–20	24	78%	180s
Simple form fill (5–10 fields)	7–12	18	92%	45s
Complex form fill (20+ fields, dropdowns)	15–22	25	71%	120s
Demo form fill (built-in test)	6–9	11	100%	38s

9. System Requirements

9.1 Functional Requirements

Table 11: Functional Requirements

Requirement	Description	Priority
Prompt-Driven Scraping	Extract structured data from any URL via natural language prompt	High
AI Form Filling	Parse PDF resume and autonomously fill any web form	High
Real-Time SocketIO Streaming	Live screenshot + color-coded logs during automation	High
Seven-Tool Browser API	click/fill/select/navigate/scroll/extract_data/done	High
Flask-Login Authentication	Session-based auth with PBKDF2 password hashing	High

Requirement	Description	Priority
Excel Export	Scraped data → formatted .xlsx downloadable file	High
PDF Resume Parsing	pymupdf4llm → LLM field extraction → JSON structure	High
Task History	Log all automation sessions with status + download links	Medium
Demo Form	Built-in test form at /form-filler/demo-form	Medium
Stop Task	Terminate running automation mid-execution cleanly	Medium

9.2 Hardware Requirements

Table 12: Hardware Requirements

Component	Minimum	Recommended
Processor	Intel Core i3 / AMD Ryzen 3	Intel Core i5 / AMD Ryzen 5 or higher
RAM	8 GB (Playwright headless needs ~400MB)	16 GB for concurrent sessions
Storage	2 GB free (browser + exports)	5 GB SSD for faster browser launch
GPU	Not required	Not required (no ML inference locally)
Network	Stable broadband (Claude API calls)	≥ 25 Mbps for SocketIO streaming
OS	Windows 10 / Ubuntu 20.04 / macOS 12	Windows 11 / Ubuntu 22.04 / macOS 14

10.1 Conclusion

This paper has presented a Prompt-Driven AI Web Automation system that successfully demonstrates the practical deployment of LLM-powered browser agents for real-world web automation tasks. The system achieves its core objective: enabling non-technical users to automate data extraction and form filling tasks using natural language descriptions, with full real-time transparency into the AI agent's decision-making process.

The four-layer architecture — Presentation (Bootstrap 5), Application (Flask 3.1), Intelligence (Claude AI), and Automation (Playwright) — provides clean separation of concerns that proved effective in practice. The ReAct-inspired agent loop, where Claude AI reasons about page state before issuing structured tool_use calls, delivers adaptive automation robust to website layout variations without requiring hardcoded selectors.

Key quantitative achievements: 120ms page load, 45ms SocketIO connection, 6.7 FPS live screenshot streaming, 1.2-second PDF parsing, 45-second 10-field form fill, and 35-second single-page scraping. All 10 unit tests and 8 integration tests pass, with 100% authentication security test compliance. The system demonstrates that practical, deployable AI-powered web automation is achievable with commodity hardware, open-source frameworks, and pay-per-use LLM API pricing.

10.2 Limitations

- Requires Anthropic API key with usage-based billing — cost scales with automation volume.
- CAPTCHA, anti-bot protection, and JavaScript-heavy SPAs may resist automated interaction.
- MAX_AGENT_STEPS = 25 limits complex multi-page tasks; very long workflows may truncate.
- Current architecture supports single concurrent automation session per Flask worker.
- Scanned PDF resumes without OCR text layer may yield incomplete field extraction.

10.3 Future Scope

International Journal of Information Technology & Computer Engineering

ISSN 2347–3657

Volume 14, Issue 2, 2026

- Multi-LLM Provider Support: GPT-4, Gemini, and local Ollama as alternative LLM backends.
- Multi-Tab Automation: Parallel browser contexts for tasks spanning multiple websites.
- Voice Commands: Speech-to-text task description for hands-free automation.
- CI/CD Integration: Scheduled automation tasks via pipeline triggers.
- Browser Extension: Initiate automation tasks directly from the active browser tab.
- Advanced Error Recovery: SHAP/LIME-style explanation of failure states for debugging.
- Cloud Deployment: Docker containerization + AWS/GCP scaling for enterprise usage.
- REST API Gateway: Programmatic task triggering for external system integration.

11. Sustainable Development Goals

Table 13: SDG Alignment

SDG	Goal	System Contribution
SDG 4	Quality Education	Demonstrates LLM + browser automation integration; teaches software engineering patterns (strategy, abstract class, layered architecture); NLP-based interface lowers programming barrier for students
SDG 9	Industry, Innovation & Infrastructure	Novel LLM-driven automation pattern replacing brittle scripts; open-source stack deployable by any organization; democratizes enterprise-grade automation for SMEs
SDG 10	Reduced Inequalities	Natural language interface removes programming barrier; enables non-technical workers (HR, admin, data entry) to automate repetitive tasks; levels automation access globally

11.1 Quantified Productivity Impact

Figure 11: Time Saved per Task Type — AI Automation vs Manual

```
Time per Task (minutes) — Manual vs AI Automation:

Web Data Collection (10 items)
Manual:                                    30 min
AI Automation:                             0.6 min
Savings: 98% ──────────────────────────────

Resume Form Fill (10-field job portal)
Manual:                                    15 min
AI Automation:                             0.75 min
Savings: 95% ──────────────────────────────

For a team of 10 performing 20 such tasks/day:
Manual:      10 × 20 × 22.5min = 4,500 min/day = 75 hours/day
AI System:   10 × 20 × 0.67min =   134 min/day =  2.2 hrs/day
SAVINGS: ~73 hours/day (~97% reduction in manual web task time)
```

References

[1] A. Ronacher, "Flask: A Python Microframework," Pallets Projects, 2024.
[2] Microsoft, "Playwright: Fast and Reliable End-to-End Testing for Modern Web Apps," 2024.
[3] Anthropic, "Claude API Documentation: Tool Use (Function Calling)," 2024.
[4] M. Grinberg, "Flask-SocketIO Documentation," 2024.
[5] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," ICLR, 2023.
[6] S. Zhou et al., "WebArena: A Realistic Web Environment for Building Autonomous Agents," ICLR, 2024.
[7] X. Deng et al., "Mind2Web: Towards a Generalist Agent for the Web," NeurIPS, 2024.
[8] R. Nakano et al., "WebGPT: Browser-Assisted QA with Human Feedback," arXiv:2112.09332, 2022.

[9] I. Gur et al., "A Real-World WebAgent with Planning, Long Context, and Program Synthesis," ICML, 2024.
[10] B. Zheng et al., "GPT-4V(ision) is a Generalist Web Agent if Grounded," arXiv:2401.01614, 2024.
[11] H. Furuta et al., "Multimodal Web Navigation with Instruction-Finetuned Foundation Models," AAAI, 2024.
[12] J. Kim et al., "Language Models as Automated Form Fillers: Challenges and Opportunities," ACL Workshop, 2023.
[13] Anthropic, "Claude 3.5 Sonnet: Technical Report," Anthropic Research, 2024.
[14] M. Grinberg, "Flask Web Development," 2nd ed. O'Reilly Media, 2018.
[15] W. McKinney, "pandas: Powerful Python Data Analysis Toolkit," pandas Dev Team, 2024.
[16] E. Gazoni and C. Clark, "openpyxl: A Python Library to Read/Write Excel 2010 files," 2024.
[17] Artifex, "PyMuPDF Documentation," 2024.
[18] M. Otto and J. Thornton, "Bootstrap 5.3: The Most Popular HTML/CSS/JS Library," 2024.
[19] SQLite Consortium, "SQLite Documentation," 2024.
[20] Pallets Projects, "Werkzeug: The Comprehensive WSGI Web Application Library," 2024.
[21] Pallets Projects, "Jinja2: Template Engine for Python," 2024.
[22] M. Baroni, "Flask-Login: User Session Management for Flask," 2024.
[23] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," ICLR, 2015.
[24] T. Brown et al., "Language Models are Few-Shot Learners," NeurIPS, 2020.
[25] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in LLMs," NeurIPS, 2022.
[26] T. Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," NeurIPS, 2023.
[27] A. Patel et al., "Web Scraping in the Age of AI: Techniques and Challenges," IEEE Access, 12, 2024.
[28] L. Wang et al., "A Survey on LLM Based Autonomous Agents," Frontiers of Computer Science, 18(6), 2024.
[29] Y. Shen et al., "HuggingGPT: Solving AI Tasks with ChatGPT," NeurIPS, 2023.
[30] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," NeurIPS, 2020.