Costronaut

Browser Automation Tool — Technical Plan & Implementation Guide

1. Executive Summary

Costronaut is a browser automation tool designed to programmatically control web browsers, execute actions on web pages, and enable AI-driven automation workflows. This document outlines three distinct architectural approaches for building Costronaut, each with unique strengths and trade-offs.

2. Overview of Approaches

1. **Playwright + Chrome Extension Harness:** Industry-standard automation library with extension-based enhancements for real browser control
2. **NW.js + Custom Harness:** Desktop application framework with embedded Chromium, offering deep DOM access and native OS integration
3. **Alternative Approaches:** Chrome DevTools Protocol (CDP), Puppeteer, Selenium WebDriver, and hybrid architectures

2.1 Quick Comparison

Criteria	Playwright + Ext	NW.js	CDP Direct
Setup Complexity	Low	Medium	High
Cross-Browser	Yes (Chrome, FF, Safari)	Chromium only	Chromium only
Real User Sessions	With extension	Yes	Limited
Native OS Access	No	Yes	No
Bot Detection Risk	Medium	Low	High
Best For	Testing, scraping	Desktop apps	Low-level control

3. Approach 1: Playwright + Chrome Extension Harness

3.1 Architecture Overview

This approach combines Playwright's robust automation capabilities with a Chrome extension that acts as a bridge between automated scripts and real browser sessions. The extension enables features like capturing user interactions, accessing authenticated sessions, and providing visual feedback.

3.2 Step-by-Step Implementation

Phase 1: Project Setup (Days 1-2)

1. **Initialize the project structure**

```
mkdir costronaut-playwright && cd costronaut-playwright
npm init -y
mkdir -p src/{core,extension,agent} tests
```

2. **Install core dependencies**

```
npm install playwright playwright-extra puppeteer-extra-plugin-stealth
npm install typescript @types/node ts-node --save-dev
npm install ws express dotenv zod
```

3. **Configure TypeScript** Create tsconfig.json with strict mode, ES2022 target, and NodeNext module resolution.

4. **Install Playwright browsers**

```
npx playwright install chromium firefox webkit
```

Phase 2: Core Automation Engine (Days 3-7)

1. **Create the BrowserController class (src/core/BrowserController.ts)**
   - Implement browser launch with configurable options (headless, viewport, proxy)
   - Add context management for isolated sessions with persistent storage
   - Integrate playwright-extra with stealth plugin to avoid bot detection
   - Implement page lifecycle hooks (beforeNavigate, afterLoad, onError)
2. **Build the ActionExecutor class (src/core/ActionExecutor.ts)**

- Define action types: click, type, scroll, wait, screenshot, extract
- Implement smart element location (CSS, XPath, text content, ARIA labels)
- Add automatic retry logic with exponential backoff
- Create action recording/playback functionality

Phase 3: Chrome Extension Harness (Days 8-14)

1. **Create extension manifest (src/extension/manifest.json)** Use Manifest V3 with service worker. Request permissions for activeTab, scripting, storage, webNavigation, and debugger APIs.
2. **Build the background service worker (src/extension/background.js)**
   - Establish WebSocket connection to Costronaut server
   - Handle incoming commands and route to appropriate handlers
   - Manage tab lifecycle and session persistence
   - Implement heartbeat mechanism for connection health
3. **Create content script (src/extension/content.js)**
   - DOM manipulation and element interaction
   - Event capture and forwarding (clicks, inputs, navigation)
   - Visual highlighting of target elements
   - Screenshot and DOM snapshot capabilities

Phase 4: AI Agent Integration (Days 15-21)

1. **Create the AgentController class (src/agent/AgentController.ts)**
   - Page state extraction (DOM tree, visible text, interactive elements)
   - Screenshot annotation with element bounding boxes
   - Action planning interface for LLM integration
   - Execution feedback loop with success/failure detection

4. Approach 2: NW.js + Custom Harness

4.1 Architecture Overview

NW.js (formerly node-webkit) provides a desktop application shell with embedded Chromium and full Node.js integration. This approach gives you complete control over both the browser environment and the host system, enabling features like file system access, native notifications, and custom window management.

4.2 Step-by-Step Implementation

Phase 1: Project Setup (Days 1-3)

1. **Initialize the NW.js project**

```
mkdir costronaut-nwjs && cd costronaut-nwjs
npm init -y
npm install nw@sdk --save-dev
```

2. **Configure package.json for NW.js** Add required NW.js fields: main (entry HTML file), window configuration, node-remote for Node.js access in web context, chromium-args for debugging flags.

Phase 2: Core Application Shell (Days 4-8)

1. **Create the main window (src/index.html)**
   - Split-pane layout: navigation panel + webview container
   - DevTools integration panel for debugging
   - Action log and status display area
2. **Implement the WindowManager class (src/core/WindowManager.ts)**
   - Multi-window orchestration with parent-child relationships
   - Window positioning, resizing, and state persistence
   - System tray integration for background operation

Phase 3: Automation Harness (Days 9-15)

1. **Create the InjectionManager (src/harness/InjectionManager.ts)**
   - Script injection into webview contexts using executeScript

- CSS injection for visual feedback (element highlighting)
- MutationObserver setup for DOM change detection

2. **Build the ActionBridge (src/harness/ActionBridge.ts)**
   - Bidirectional communication between Node.js and webview
   - Command serialization and result passing
   - Error boundary handling for crashed pages

5. Approach 3: Alternative Methods

5.1 Chrome DevTools Protocol (CDP) Direct

CDP provides low-level access to Chrome's debugging interface. This approach offers maximum control but requires more implementation effort.

Implementation Steps

1. **Launch Chrome with remote debugging enabled**

```
chrome --remote-debugging-port=9222 --user-data-dir=/tmp/chrome-debug
```

2. **Connect via WebSocket to CDP endpoint**
   - Fetch available targets from http://localhost:9222/json
   - Establish WebSocket connection to target's webSocketDebuggerUrl
3. **Implement CDP domain handlers**
   - Page domain: Navigation, screenshots, lifecycle events
   - DOM domain: Node queries, attribute manipulation
   - Input domain: Mouse, keyboard, touch events
   - Runtime domain: JavaScript execution

5.2 Puppeteer-Based Approach

Puppeteer is Google's official high-level API over CDP. It's simpler than raw CDP and battle-tested for Chrome automation.

1. **Install Puppeteer with extras**

```
npm install puppeteer puppeteer-extra puppeteer-extra-plugin-stealth
```

2. **Connect to existing browser (for real sessions)**
   - Use puppeteer.connect() with browserWSEndpoint
   - Access existing pages and sessions

5.3 Selenium WebDriver

Selenium provides cross-browser compatibility through the WebDriver protocol. Best for scenarios requiring Firefox or Safari support.

1. **Set up Selenium with Node.js bindings**

```
npm install selenium-webdriver chromedriver geckodriver
```

6. Recommended Implementation Strategy

6.1 Phased Development Plan

1. **Phase 1 (Weeks 1-2): Start with Playwright approach** Playwright offers the fastest path to a working prototype with excellent documentation and community support. Build the core automation engine and test against real websites.
2. **Phase 2 (Weeks 3-4): Add Chrome extension harness** The extension enables real browser session control, which is critical for workflows requiring authentication or avoiding bot detection.
3. **Phase 3 (Weeks 5-6): Evaluate NW.js for desktop version** If desktop distribution and native OS integration are requirements, port the core engine to NW.js. The automation logic can be shared between approaches.
4. **Phase 4 (Weeks 7-8): AI agent integration and polish** Implement the agent layer that can plan and execute multi-step tasks. Add visual feedback, error recovery, and user-facing documentation.

6.2 Key Success Factors

- Robust element location: Websites change frequently. Use multiple selector strategies with automatic fallbacks.
- Bot detection mitigation: Use stealth plugins, realistic timing, and consider extension-based approaches for sensitive sites.
- Session persistence: Save and restore browser state (cookies, local storage) between runs.
- Error recovery: Implement retry logic, screenshot capture on failure, and graceful degradation.
- Observability: Log all actions with timestamps, maintain execution traces for debugging.

6.3 Directory Structure

```
costronaut/
+-- packages/
|   +-- core/              # Shared automation engine
|   |   +-- src/
|   |   |   +-- actions/   # Click, type, scroll, etc.
|   |   |   +-- locators/  # Element finding strategies
|   |   |   +-- session/   # Browser session management
|   |   |   +-- agent/     # AI integration layer
|   |   +-- package.json
|   +-- playwright-driver/ # Playwright implementation
|   +-- nwjs-app/          # NW.js desktop application
|   +-- chrome-extension/  # Browser extension harness
+-- tests/
+-- docs/
+-- package.json           # Monorepo root
```