MASTER'S THESIS 2025

PriceFinderAgent: An Agentic Approach to Web Scraping

David Pettersson, Ludvig Eskilsson

ISSN 1650-2884
LU-CS-EX: 2025-35

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2025-35

PriceFinderAgent: An Agentic Approach to Web Scraping

PrisAgenten: Ett agentbaserat system för webbaserad datainsamling

David Pettersson, Ludvig Eskilsson

PriceFinderAgent: An Agentic Approach to Web Scraping

David Pettersson
dawei.98@hotmail.com

Ludvig Eskilsson
ludvig.eskilsson@gmail.com

June 18, 2025

Master's thesis work carried out at Prisjakt Sverige AB.

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se
Richard Toth, richard.toth@prisjakt.nu

Examiner: Mathias Haage, mathias.haage@cs.lth.se

Abstract

Web scrapers are programs used to collect structured data from the public web, but traditional scrapers often break due to hard-coded logic and changes in site structure. Recent advances in generative Artifical Intelligence AI through large language models (LLMs) and large multimodal models (LMMs) offer new opportunities for building more robust alternatives. This thesis explores the use of AI to automate web scraping workflows. We focus on extracting phone subscription offers from three Swedish mobile carrier websites, a domain that requires both reasoning and interaction due to the complexity of contracts and layouts. We evaluate multiple AI-assisted strategies, including rule-based and agentic frameworks. Among these, a multi-agent LLM-based system using vision achieved the strongest results, reaching 90% precision, 100% recall, and 92% accuracy. The findings show that modern LLM agents can function as reliable scrapers for dynamic websites, provided they are guided by effective prompts and supported with appropriate navigation tools.

Keywords: Web scraping, AI agents, Prompt engineering, Price extraction

Acknowledgements

We would like to thank our academic supervisor at LTH, Pierre Nugues, for his guidance and support throughout this project. He has not only helped us improve the structure and clarity of our report, but also emphasized the importance of evaluating our system's performance in a measurable way. Thanks to his advice, we focused on benchmarking early on, which greatly improved how we presented our results.

We are also very grateful to our supervisor at Prisjakt, Richard Toth, who introduced us to the problem this thesis is based on. Without him, this project would not have happened. He has been welcoming from the start and made sure we had everything we needed to carry out the work – resources, data, and a great environment. He was also a valuable sounding board for discussing ideas and figuring out how to tackle the challenges we faced.

A big thank you also goes to everyone else at Prisjakt who took the time to help us during the project. Your input, feedback, and support were very appreciated.

Finally, we would like to thank Mathias Haage for acting as the examiner for this thesis.

Contents

Chapter 1
Introduction

Web scraping is a widely used technique for collecting publicly available data from websites. It plays a central role in applications such as price tracking, market research, and large-scale data aggregation. Traditional scrapers are typically hand-written scripts that rely on static HyperText Markup Language (HTML) structures or specific Document Object Model (DOM) selectors. These methods are fragile as minor layout changes or JavaScript refactors on the website often break the scripts logic. As websites increasingly use client-side rendering, asynchronous data loading, and interfaces with hidden elements and animations, scraping has become more complex and error-prone. Maintaining traditional scraping scripts is labor-intensive and requires ongoing manual adjustments (Ahluwalia and Wani, 2024).

This thesis explores the use of AI-assisted scraping in a specific and challenging domain: phone subscription contracts from mobile carrier websites. Unlike static product listings, phone contracts involve complex user flows including multi-step configuration wizards, bundled promotions, modal interfaces, and pricing that depends on selected hardware, service plans, and payment durations. Extracting structured pricing data from these pages requires both interaction and interpretation.

Recent advances in multimodal Large Language Models (LMMs) and AI agents have introduced the possibility of intelligent scraping agents that reason about page content and adapt to dynamic web interfaces. These models are capable of tool use, structured decision-making, and even code generation. This has sparked growing interest in using LLMs to automate scraping workflows.

However, using AI agents as scrapers still poses challenges when applied to complex real-world websites. In particular, navigation remains a frequent failure point. He et al. (2024) found that 44.4% of their agent's failures were due to getting stuck in navigation flows typically caused by misclicks, incorrect selections, or failure to detect next-step elements. Even top-performing systems still fall short of full reliability. For example, Müller and Žunič (2024) report state-of-the-art results with Browser Use, yet still achieve below 90% task success on the WebVoyager benchmark. Similarly, OpenAI (2025) reaches only 58.1% on the more challenging WebArena benchmark using their Computer-Using Agent (CUA). Beyond

navigation, broader limitations persist across LLM-based agents. Current systems can struggle with semantic understanding, grounding outputs in the UI, and reliably extracting structured data. These limitations frequently lead to hallucinations where the agent generates plausible-sounding but incorrect outputs that are not actually present on the page (Ahluwalia and Wani, 2024).

We propose PriceFinderAgent (Figure 1.1), an AI-powered web scraping agent that extracts subscription prices by interacting with dynamic web pages. The system combines a rule-based workflow with a multi-agent setup, where different AI agents and LLMs are assigned to handle specific stages of the scraping task. We evaluate the system on a subset of 10 offers from three live carrier websites – Telia, Telenor, and Halebop – each with a distinct layout, using ground truth data collected by manually inspecting the carrier sites.

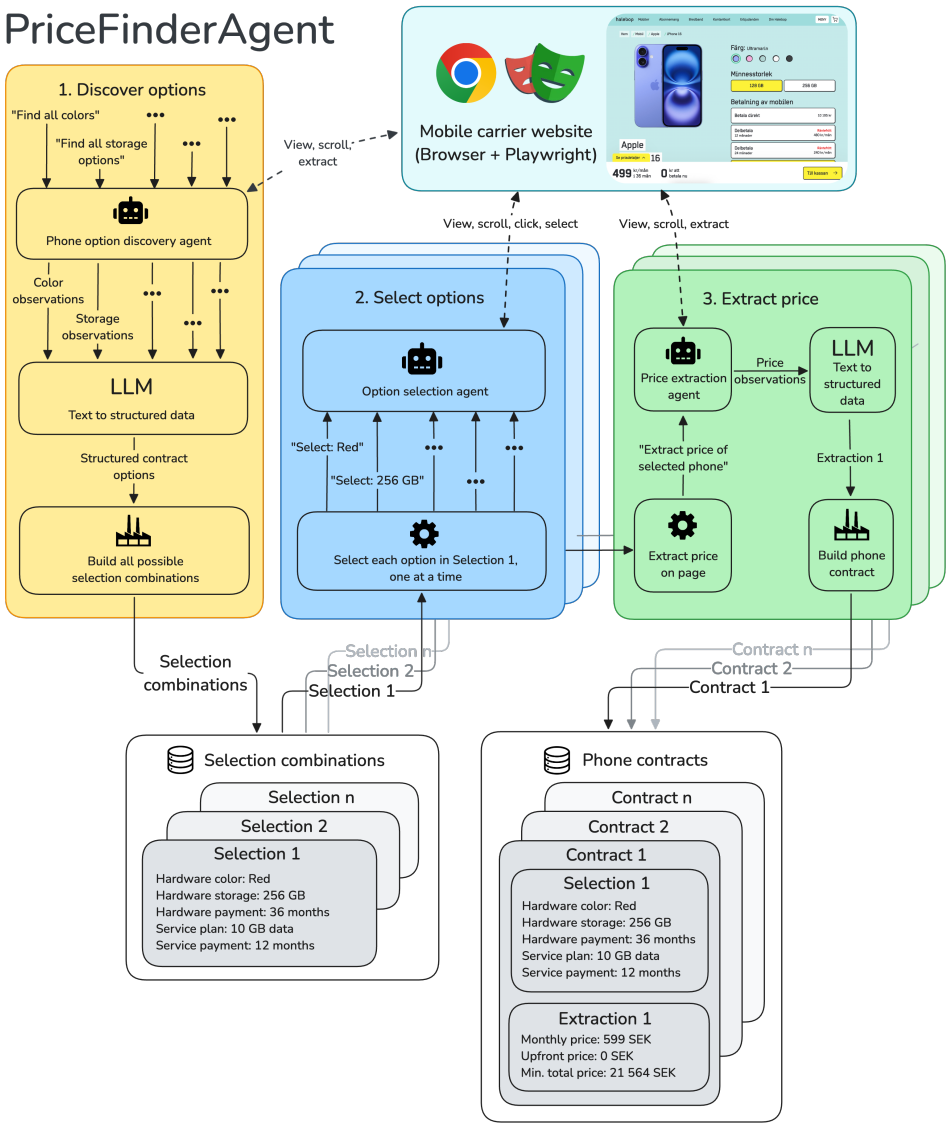


Figure 1.1: Overview of PriceFinderAgent, a multi-agent system for extracting all available phone contracts from mobile carrier websites. The standardized format of the contracts enables price comparison between carriers.

Our results show that PriceFinderAgent achieves strong overall performance, with 90% precision, 100% recall, and 92% accuracy when aggregating results across all three carriers. Notably, the agent reached perfect performance on Halebop, achieving 100% in all core metrics. This significantly outperforms the baseline performance of scraping with plain Browser Use, which achieved only 23% precision, 88% recall, and 47% accuracy under the same evaluation setup. While challenges remain, particularly around navigation complexity and cost, our results demonstrate that modern LLM agents can be made robust and reliable for real-world web scraping. These findings suggest a promising foundation for future systems that are both accurate and easier to maintain as websites evolve.

1.1 Study Context – Prisjakt and Swedish Mobile Carriers

Prisjakt is a leading price comparison platform in Sweden, with the primary objective of enabling consumers to identify the lowest available prices for products across different retailers (Prisjakt, 2025). Their service primarily relies on structured data feeds submitted directly by e-commerce stores, which allows for consistent and scalable price aggregation. However, in cases where retailers do not supply such data, either due to technical limitations or strategic business choices, Prisjakt sometimes employ manually written web scraping scripts to retrieve prices directly from online storefronts.

Among the many product categories on the Prisjakt website, mobile phones constitute one of the largest and most frequently visited. The importance of this category is further amplified by consumer purchasing behavior in Sweden, where it is common to acquire a mobile phone through a bundled contract with a mobile carrier rather than buying the device outright. These contracts typically package the hardware with a mobile data plan, spreading the cost across a fixed monthly fee.

This purchasing model introduces significant complexity when it comes to price comparison. Swedish mobile carriers such as Telia, Telenor, Tre, Halebop, and Hallon offer both mobile phones and associated service plans, typically bundled into mobile contracts. While these carriers offer similar services such as national coverage, phone models, and data packages, their pricing structures are deliberately obfuscated. Unlike static product listings, mobile contracts are often embedded within multi-step configuration flows, dynamically rendered interfaces, and conditional pricing schemes that depend on hardware selection, data allowances, and commitment duration.

This complexity is likely not on accident. According to the Swedish Quality Index (SKI) (2024) report on mobile services from 2024, customer satisfaction in this domain is most strongly influenced by perceived value for money. As a result, mobile carriers have strong financial incentives to avoid full price transparency, thereby discouraging direct comparisons. This strategic approach is reflected in their unwillingness to provide structured data feeds to third parties such as Prisjakt.

Despite these barriers, it is in both Prisjakt's and consumers' interest to enable price comparisons in this category. Transparent access to subscription pricing would empower users to make informed purchasing decisions and increase competitive pressure among carriers. Therefore, this thesis addresses the need for an automated and scalable solution to extract mobile contract pricing directly from carrier websites. This thesis was conducted at Pris-

jakt to explore whether mobile contract prices can be reliably extracted from dynamic and intentionally obfuscated carrier websites.

1.2 Research Questions

In order to evaluate the capabilities and limitations of AI-assisted web scraping, we aim to answer the following research questions:

RQ1: Is it possible for AI-assisted web scraping techniques to consistently extract accurate data from dynamic and visually complex websites?

Our goal is to investigate whether modern LLM-based agents can consistently extract accurate pricing data from dynamic and visually complex websites, achieving high precision comparable to traditional scraping approaches, with the benefit of being more flexible and potentially requiring less maintenance.

RQ2: Is it possible to generalize the AI-assisted web scraping technique across different dynamic websites without relying on site-specific rules?

This question addresses the core challenge of generalizability. We aim to determine whether a system like PriceFinderAgent can handle a wide variety of layouts, UI flows, and interaction models without relying on brittle, hardcoded logic specific to each target site.

1.3 Scientific Contribution

This thesis contributes to the field of AI-assisted web automation by demonstrating that multi-agent systems using LLMs can robustly extract complex pricing data from dynamic and interactive websites. Prior research has shown limitations in AI agents' ability to reliably navigate and extract structured data in real-world environments due to challenges like asynchronous content loading and obfuscated interfaces (He et al., 2024). Our work addresses this gap by evaluating and comparing multiple AI-assisted scraping strategies specifically tailored to the intricate domain of mobile phone subscription contracts.

1.4 Contribution Statement

We contributed equally to the initial research phase of this project. During the implementation stage, we developed two separate systems in parallel to explore different approaches. Once we had evaluated their performance, we aligned on a single direction and continued working together on the same system.

When writing this report, we divided the work by drafting different sections individually and then reviewing and proofreading each other's contributions to ensure clarity and consistency.

Generative AI tools were occasionally used throughout the project. In the report-writing process, they helped us improve phrasing, structure sentences more clearly, and refine the

overall flow of our text. During development, we used generative AI as a coding assistant to generate or modify code snippets based on instructions we provided.

Chapter 2
Related Work

To understand how our approach to AI-assisted web scraping builds upon and differs from existing systems, this chapter reviews the current landscape of research and tooling in the area of web agents and AI web scraping. We focus on established benchmarks and key frameworks such as WebVoyager, Browser Use, and Stagehand.

2.1 Agent Benchmarks

To evaluate the performance and capabilities of web agents, researchers have proposed several standardized benchmarking suites. Examples include WebArena by Zhou et al. (2023), Mind2Web by Deng et al. (2023), OSWorld by Xie et al. (2024), and WebVoyager by He et al. (2024). These benchmarks provide controlled environments that mimic real-world web pages. Within these environments agents are assigned high-level tasks that resemble common activities humans perform online, such as searching, navigating, or interacting with forms (Zhou et al., 2023; He et al., 2024).

The performance of an agent can be evaluated either by analyzing its action trajectory or by comparing the final output against the expected result. This enables standardized assessment of both task completion accuracy and interaction efficiency.

These benchmarks are now increasingly used by companies releasing web agent frameworks and models. For instance OpenAI uses benchmarks like WebArena to evaluate its computer-using agent named CUA (OpenAI, 2024).

2.2 WebVoyager

WebVoyager is a research project that introduces both an end-to-end web agent and a benchmark for evaluating such agents, developed using a multimodal large language model (LMM)

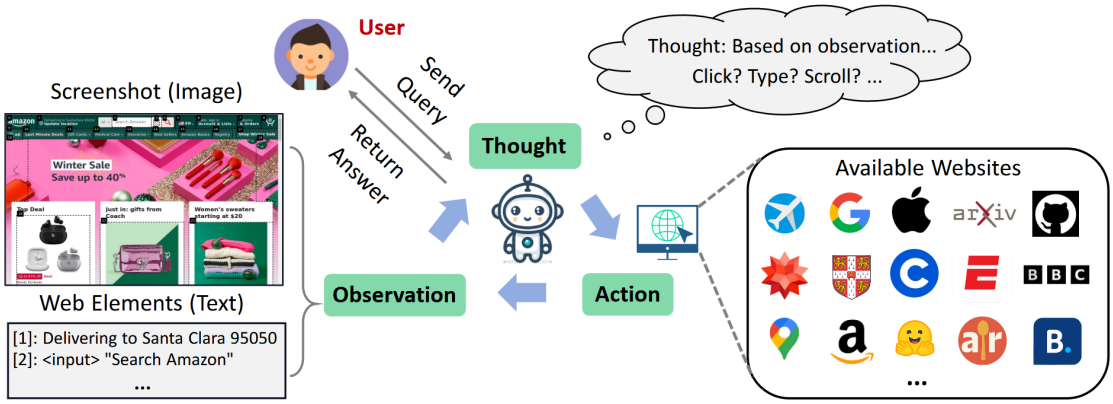


Figure 2.1: High level architecture of WebVoyager. The input to the model is composed of an screenshot with annotations and bounding boxes coupled with text describing the annotations (called Observation in the figure). The agent then follows a thought-action-observation pattern to move towards the end goal given by the user. Source: He et al. (2024).

(He et al., 2024). The agent can perform tasks such as navigating to Amazon and placing orders by processing both text and visual inputs, offering a fuller context of the webpage state. Unlike previous text-only agents evaluated on static snapshots, WebVoyager mimics human browsing by processing rendered webpages, marking interactive elements, and choosing actions (click, type, scroll). This method overcomes the limitations of earlier approaches that missed crucial visual cues.

WebVoyager's agentic framework is based upon time steps, where each time step includes a context. Formally they represent their agentic model with the following variables: $\mathcal{M}$ is the LLM model, ϵ is the Environment, $\mathcal{A}$ is action space, and $\mathcal{O}$ is the observation space. The context at time step t is then defined as

$$c_t = (o_1, a_1, ..., o_{t-1}, a_{t-1}, o_t, I)$$

and $a_t = M(c_t)$. So the current action is retrieved from giving the model the current context, with clipping if context is too long (He et al., 2024). This loop of observation, thought and action can be seen illustrated in Figure 2.1.

To identify interactive elements, WebVoyager uses a rule-based mechanism that examines element tags, attributes, and visual features to overlay bounding boxes and numerical labels on the page. This process referred to as Set-of-Mark Prompting (Yang et al., 2023) allows the agent to link visual cues directly to corresponding DOM elements, streamlining decision-making by highlighting key areas without parsing verbose HTML.

2.3 Browser Use

Browser Use is an open-source framework that has rapidly become one of the most widely adopted toolkits for building LLM-powered web agents, accumulating tens of thousands of GitHub stars within months of its release (Y Combinator, 2025). Similar to WebVoyager, it

enables multimodal AI models to interact with a live browser environment through tool calling, allowing tasks to be expressed in natural language and executed via browser automation. While both frameworks share the same core idea, Browser Use is more actively maintained and offers broader model support and tooling.

Browser Use is designed to operate from high-level task descriptions such as:

Go to Telia's website and find which storage options are available for iPhone 16

Given such a prompt, the agent autonomously decomposes the instruction into smaller subtasks, plans a sequence of steps, and performs browser actions – such as navigating to a website, clicking elements, and extracting information – until it determines that the task is complete.

The system is based on the idea that modern foundation models possess strong reasoning capabilities and broad world knowledge. When provided with an appropriate interface, these models can serve as effective agents for decision-making. Browser Use implements this by placing a LLM in control of a Playwright-driven browser session, using the model as a high-level reasoning and control layer.

Much like WebVoyager, Browser Use operates in a agent loop, more specifically in a *ReAct loop* (Reason Act) (Yao et al., 2023), where the agent observes, reasons, chooses an action, and receives updated context until the task is completed or a maximum number of agent steps has been reached.

The framework defines two core components for interaction: the observation space and the action space. At each step of the agent loop, the LLM receives a structured snapshot of the browser state, including the current URL, open tabs, the task description, a memory trace of prior actions, and a list of indexed interactive DOM elements. An example of this input is shown in Figure 2.2. The action space consists of callable functions such as 'click', 'type', 'scroll', and 'extract_text', which the LLM can invoke via function calls. These actions reference DOM elements by their assigned numeric indexes, allowing the model to specify which element to interact with based on the structured list it received in the input.

Based on the input, the LLM is instructed to respond at each step with a specific JavaScript Object Notation (JSON) object containing a high-level reasoning trace (e.g., progress evaluation, memory updates, next subgoal) and a list of actions to execute on the current state of the browser (Müller and Žunič, 2024). This structured interaction pattern allows pre-trained LLMs to function as effective browser agents.

Task: Extract the subscription price
Previous steps: Clicked plan selector
Current URL: https://telia.se/plans
Open Tabs: [1] Telia Start
Interactive Elements:
[33] `<div>`User form`</div>`
[35] `<button aria-label='Submit form'>`Submit`</button>`

Figure 2.2: Example input format provided to the LLM at each time step in Browser Use. Indentation reflects DOM hierarchy, and numeric indexes indicate interactable elements. Asterisks mark newly seen elements since the last step.

The DOM element abstractions used in the inputs and outputs are extracted using rule-based heuristics that identify visible, interactive elements – such as buttons, inputs, and links – while ignoring purely decorative or static content (Müller and Žunič, 2024). Each selected element is assigned a unique index and a minimal set of attributes. The index maps to a saved XPath that uniquely identifies the element in the DOM, allowing the LLM to reference elements during interaction. This representation is far more token-efficient than raw HTML, yet still preserves the essential structure needed for reasoning and control.

If vision is enabled, the agent also receives a screenshot of the current page alongside the text-based input. This allows the model to incorporate visual layout and styling into its reasoning, which can help resolve ambiguities not captured in the DOM-based representation. Notably, the same rule-based extraction used to identify interactive elements in the DOM is applied to the screenshot using a variant of Set-of-Mark prompting (Yang et al., 2023), marking areas of greater interest. Clickable elements are visually annotated with colorful bounding boxes and labeled with the same indices used in the textual description enabling the model to ground its decisions across both modalities.

2.4 Stagehand

Stagehand is an open-source project that aims to automate web interactions by extending Playwright with AI-driven methods (Browserbase, 2025b). It has four core functionalities: `observe`, `act`, `extract`, and `agent`.

`Observe`: Identifies candidate DOM elements for interaction. Given a description of the desired elements, it returns the corresponding selectors.

`Act`: Executes an action on a web element based on a given instruction. Returns details of the action performed.

`Extract`: Retrieves structured data from the page according to a specified instruction and schema, returning the information in the requested format.

`Agent`: Allows users to define high-level goals that are autonomously achieved through sequences of `observe`, `act`, and `extract` operations. Alternatively, agents can use specialized computer-use models that directly interact with the browser without relying on these primitives.

Unlike other AI-agent systems like Browser Use, Stagehand takes a modular approach to web automation by breaking tasks into smaller, independent actions instead of handling them as a single, complex task. This method can potentially improve reliability and makes it easier to identify and fix errors, as failures can be traced to specific steps rather than an entire process.

Stagehand operates in two stages when using `act`, `extract`, or `observe`. The first stage involves processing the DOM to identify relevant elements. It extracts all visible and interactable elements while discarding non-essential parts of the DOM. For each candidate element, an XPath is generated. The result is a list of candidate elements and their corresponding XPath selectors, which are then analyzed by a multimodal LLM.

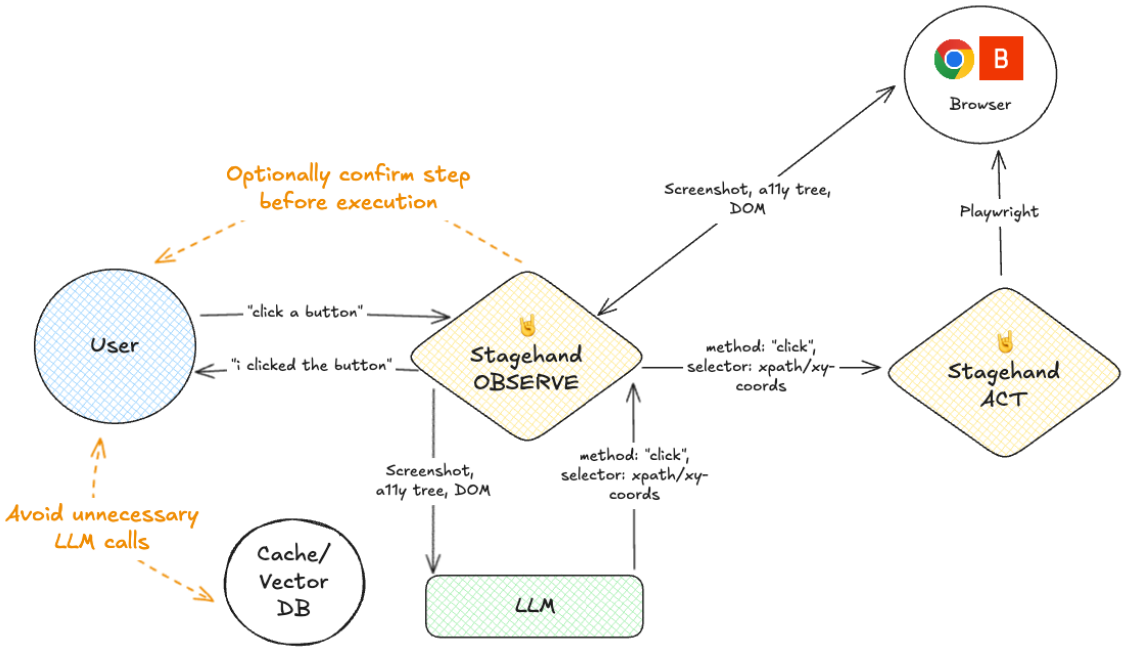


Figure 2.3: Stagehand's architecture for modular web automation using `observe` and `act`. The system first identifies elements with `observe` and confirms actions before executing them with `act`, supported by LLM reasoning and optional caching. Source: Browserbase (2025b)

In the second stage, the processed DOM is passed to the LLM for analysis. If the user requests an `act` operation, the model generates Playwright code to interact with the appropriate element based on the given instruction. If the model fails to locate a matching element, Stagehand employs a fallback mechanism using vision-based analysis. A screenshot of the page is provided to the model, allowing it to visually identify the target element and complete the requested action. An overview of Stagehand's architecture can be seen in Figure 2.3.

When using a computer-use model, Stagehand bypasses the `act`, `extract`, and `observe` methods entirely. Instead of generating a list of interactable DOM elements, then writing Playwright scripts to interact with these elements, the model interacts with the browser only through screenshots of the browser and clicking on screen coordinates, typing into input fields, and navigating the page visually (Browserbase, 2025a).

Agents in Stagehand are created by specifying a model provider (such as OpenAI or Anthropic), a model (e.g., computer-use-preview or claude-3-7-sonnet), and custom instructions. During execution, the agent maintains a structured memory of performed actions, evaluates outcomes, and continues planning iteratively until the goal is completed or a failure is reported.

2.5 Computer-Using Agents

Computer-using agents are an emerging class of AI systems developed by leading research labs such as Anthropic (2025), OpenAI (2025), and Google DeepMind (2025b). These agents are controlled via natural language. Much like humans they interact with applications or websites by perceiving screen content visually. These computer-using agents operate in a visual environment and output low-level actions (mouse movement, clicks, typing) by grounding their decisions in screen pixels.

They are built on top of multimodal foundation models with vision capabilities, such as GPT-4o (OpenAI, 2025) and Claude (Anthropic, 2025), which are able to process both text instructions and screen images. DeepMind's agent, Project Mariner, uses a variant of Gemini with native multimodal support (Google DeepMind, 2025b).

Although the internal details of these systems are not fully disclosed, their behavior aligns with principles observed in recent research. In particular, they extend foundation models with agentic capabilities: the capacity to observe, reason, and act in a loop. This process often follows a structured sequence, where reasoning informs action, and subsequent observations refine future reasoning, resembling frameworks such as ReAct (Yao et al., 2023).

The general pre-training of LLMs provides strong reasoning and understanding capabilities (Sager et al., 2025). But these new agents typically rely on models that have undergone additional adaptation. For example, OpenAI's Computer-Using Agent (CUA) is based on GPT-4o but further trained with reinforcement learning to improve its ability to interact with GUIs effectively (OpenAI, 2025). Anthropic's Claude agent was trained on a small set of desktop applications but generalized well to new environments, highlighting the strength of in-context learning and multimodal perception (Anthropic, 2025).

These agents have demonstrated strong generalization to unseen user interfaces and perform competitively on benchmarks such as OSWorld and WebArena.

Chapter 3
Background

3.1 The Structure of a Mobile Contract

Mobile carriers offer phone contracts that combine a device, a mobile service plan, and a price. Each carrier provides many different contract combinations, depending on the device configurations and services available for that device. In this thesis we have defined a mobile contract as the combination of a hardware component, a service plan component and a price component.

Carrier: A provider of mobile services that also sell devices to be used with their services

Mobile contract: The combination of a device, a service plan, payment durations, and prices

Hardware: A specific phone or device

Service plan: A mobile plan that enables data usage and access to a cell phone network

Price: A price split into multiple components

3.1.1 The Hardware Component

Multiple carriers may sell the same phone models (like iPhone 16 or Samsung Galaxy S25), but not always with the same configurations. For example, one carrier may only offer the 128 GB version in black, while another offers all color and storage options.

Devices are sold in different configurations. These configurations differ depending on color, storage capacity and payment duration.

Model: The device model name

Color: The color of the device

Storage capacity: The internal storage of the device (e.g., 128 GB, 256 GB, 512 GB)

Payment duration: The payment duration for which you agree to pay off the device (e.g., pay over 12 or 24 months)

3.1.2 The Service Plan Component

Carriers may offer different mobile service plans depending on which hardware you're buying. For instance, a certain plan might only be available when buying a specific phone. Services usually have two main attributes, which are a plan name, and a binding time or payment duration.

Plan name: The name of the service plan. Indicates how much data usage is included (e.g., 5 GB, 20 GB, Unlimited, Unlimited with Netflix and Disney+)

Payment duration: How long you are required to pay for the service plan once you commit to the contract (e.g., 12 months, 24 months)

3.1.3 The Price Component

Mobile carriers often display a wide range of prices and use different terminology to describe them. During the development of our system, we identified three core price components that were consistently present across all carriers:

Total upfront price: The initial amount the customer pays at the start of the contract.

Total monthly price: The recurring monthly charge, typically including both the hardware and the service plan.

Minimum total price: The lowest possible total amount a customer would pay over the full duration of the contract.

3.1.4 Example Mobile Contract

Table 3.1 shows an example of a mobile contract. This figure illustrates how a carrier combines hardware, service, and pricing into a mobile contract offer.

3.2 Theory

This section outlines the theoretical foundation for building a system that extracts data from dynamically rendered websites. It covers the limitations of traditional web scraping and introduces large multimodal models as a means to interpret and interact with web interfaces.

Carrier	Name: Telia
Hardware	Model: iPhone 16 Color: White Storage: 128 GB Payment Duration: 36 months
Service	Plan: Unlimited with Netflix, Max and Disney+ Payment Duration: 24 months
Price	Total upfront price: 0 SEK Total monthly price: 499 SEK Minimum total price: 13 896 SEK

Table 3.1: Example of a mobile contract with specific carrier, hardware, service, and pricing

3.2.1 Web Scraping

Web scraping is the automated extraction of information from websites, commonly used to collect publicly available data that is not offered on a structured format via an API (Proxyway, 2023). This process enables the transformation of human-readable web content, such as HTML pages, into machine-readable datasets. In this thesis, web scraping is employed to extract mobile contract options and pricing information for contracts from Swedish mobile carrier websites.

The most basic approach to web scraping is to simply download and parse the raw HTML of a website. However, as website architectures have become increasingly complex, relying on dynamic content, JavaScript execution, and user interactions, more sophisticated techniques are required to reliably extract the desired data.

Scraping Mobile Contracts

In this thesis, we will extract phone contract prices from mobile carriers product pages. This presents specific challenges compared to scraping static product pages such as those for clothing or electronics. On some mobile carrier websites the correct pricing information is not immediately visible but is instead revealed only after a series of user-driven selections. These include actions such as specifying whether the user is a new or existing customer, selecting whether to get a new phone number or transfer an existing one, and choosing specific contract or data plan configurations. Figure 3.1 illustrates how one Swedish carrier requires the user to either select `log in` or `continue as guest` before displaying any contract options or prices.

Unlike typical e-commerce pages where prices are embedded directly in the initial HTML on page load, these subscription flows require scrapers to emulate a full user session, e.g. interacting with the UI and waiting for asynchronous content to load. This adds complexity and increases the risk of scraping failure if any part of the interaction logic changes. Tools like Playwright can be used to handle these problems and will be introduced in Section 3.3.1.

Another aspect that makes phone contracts complex is the structure of the contracts themselves. The monthly price is typically affected by multiple parameters, including the

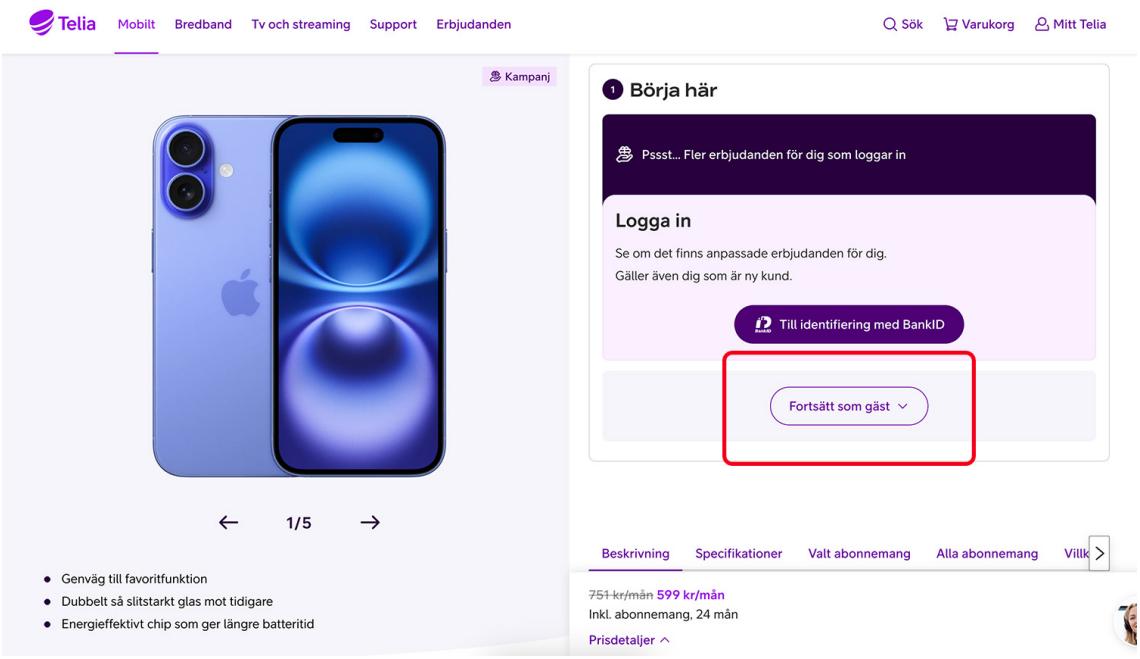


Figure 3.1: Screenshot from Swedish carrier Telia's product page for iPhone 16. They require the user to interact with the page before loading any contract options. Source: (Telia, 2025)

service plan, hardware storage, and payment duration. In addition, carriers frequently run special promotions that may include temporary discounts, extra data, or bundled gifts. These special offers often have time-based conditions. For example, a 30% discount for the first three months, which further complicates the extraction and normalization of pricing data.

3.2.2 Modern Web Architecture

Modern websites are increasingly built using component-based JavaScript frameworks like React, Vue, and Next.js (Docs, 2025). These frameworks promote modular, reusable user interface components and allow developers to build highly interactive Single-Page Applications (SPAs) (Kothapalli, 2021). While this architecture improves user experience and maintainability, it introduces complexity for web scraping that rely on static HTML structures. To understand the challenges these modern sites present to automated data extraction, it is important to examine their rendering models, data-loading behavior, and internal structure.

Rendering Models: CSR and SSR

Early websites were primarily statically rendered, meaning the server stored a complete HTML file that was served exactly the same for each client that requested it. As websites became more interactive, static rendering became inefficient, especially for dynamic content like user-specific data or real-time updates. Regenerating static pages for every change was impractical, and to address this, two new rendering strategies were adopted:

Server-Side Rendering (SSR): After receiving a request, the server renders HTML and sends it to the client. While more scraping-friendly, SSR applications may still in-

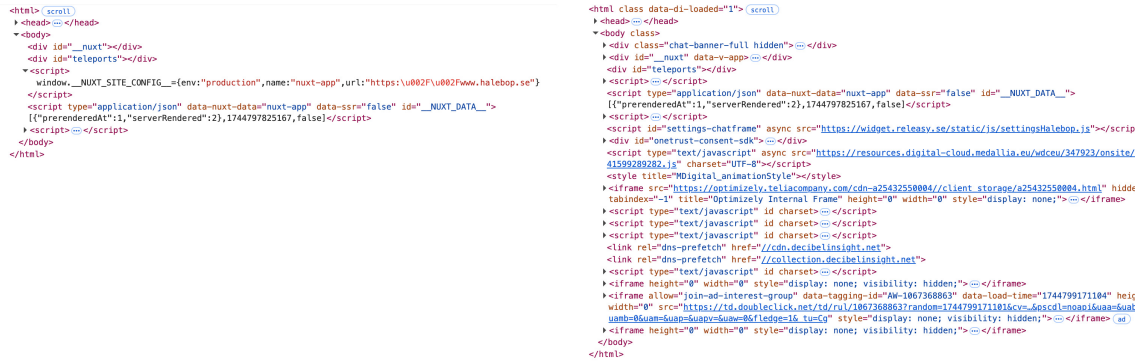


Figure 3.2: HTML from Swedish mobile carrier Halebop using CSR. Left: initial server response. Right: rendered webpage after Javascript has been run and web content has been fetched from the server. Source: (Halebop, 2025)

clude client-side enhancements that load additional data after page load (Whitaker, 2023).

Client-Side Rendering (CSR): The server returns a minimal HTML shell. JavaScript running in the browser then fetches and renders the content. This approach reduces server response time and server load but delays the visibility of data and complicates scraping, as most content is not available in the initial HTML. Figure 3.2 shows an example of a website using CSR. The left part of the image shows the initial HTML sent from the server. The right part of the image shows the HTML once all JavaScript has been executed and web elements have been fetched from the server.

Asynchronous Data Loading and Interactions

Regardless of whether CSR or SSR is used, many modern websites rely heavily on asynchronous data fetching to load content dynamically. Data is often retrieved using JavaScript (e.g., via `fetch()` or `XMLHttpRequest`) in response to user actions like selecting a product variant, scrolling, or clicking a button.

This interaction-dependent behavior creates additional barriers for scrapers. Key information may not be available on page load and will only appear after certain UI elements are triggered. A scraper must therefore simulate user actions and wait for asynchronous data responses to fully extract all relevant content (Whitaker, 2023).

DOM Structure and Selectors

The DOM is the browser's internal representation of a webpage. It structures the page content as a tree of nodes, where each node represents an element such as a `button`, `span`, or `div` (W3Schools, nd).

Selectors can be used to extract specific elements from the DOM. CSS selectors target elements based on class names, IDs, and element types. Figure 3.3 shows an example of a CSS selector for a price element on a mobile carrier website. In the figure, Chrome DevTools

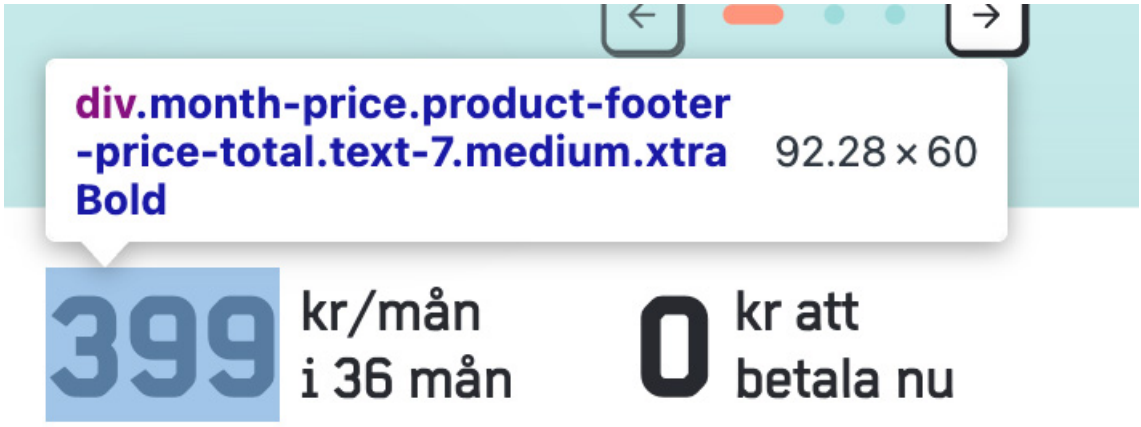


Figure 3.3: A CSS selector for a price element on a Swedish mobile carriers website. (Halebop, 2025)

provide us with a suggested CSS selector for the price element, but it is possible to select it with other CSS selectors as well.

XPath expressions allow for more precise navigation using a path-like syntax. In some cases, it can be difficult to find a CSS selector that maps to only one element. In these cases, you may be required to use an XPath to target the element instead. The drawback of XPaths is that they are fragile, any change in the website's layout or class names can invalidate the selectors and thus break the scraping script.

3.2.3 Large Language Models (LLMs)

Large Language Models (LLMs) are a class of deep learning models trained on massive amounts of text. They specialize in understanding and generating human language, and are best understood as autoregressive models that predict the next token in a sequence based on prior context. This stochastic process involves estimating a distribution over possible next tokens and selecting or sampling the most likely one. LLMs typically contain billions of parameters that capture linguistic structure, usage patterns, and factual knowledge.

Most LLMs are built on the transformer architecture, introduced by Vaswani et al. (2023) in 2017. A prominent example is the Generative Pre-trained Transformer (GPT) family, such as GPT-4o (OpenAI, 2024b), which uses a decoder-only variant optimized for autoregressive generation. This design, combined with large-scale pretraining, enables GPT models to produce coherent, contextually relevant outputs and to perform tasks with strong generalization – often in zero-shot or few-shot settings – without task-specific training.

LLMs have become foundational in both research and industry, shifting Natural Language Processing (NLP) from task-specific models to general-purpose systems that can be adapted with minimal supervision.

Transformer Architecture

Traditional feedforward neural networks operate on fixed-size input vectors and lack mechanisms for handling sequential or variable-length data, making them unsuitable for natural

language.

Early solutions included Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber, 1997). They process inputs token by token while maintaining an internal state across steps. Tokens represent chunks of text, such as words or subwords. While effective to some extent, these models suffer from limited parallelism and difficulty capturing long-range dependencies (Vaswani et al., 2023).

The transformer architecture overcomes these limitations by using self-attention instead of recurrence. Input text is tokenized, embedded, and enriched with positional encodings before being passed through layers of multi-head self-attention and feedforward networks. In self-attention, each token dynamically weighs its relationship to all other tokens using query, key, and value vectors. This enables efficient parallel processing and better modeling of global context. Transformers are highly scalable and form the basis of most modern LLMs, particularly in decoder-only models like GPT.

Decoder Architecture

In the original transformer design, the model is split into an encoder and a decoder. The encoder processes the entire input sequence and builds contextualized representations. The decoder then generates the output sequence token by token using both its own prior outputs and the encoder's representations (Vaswani et al., 2023). This encoder-decoder structure is well-suited for tasks like machine translation where the input and output sequences differ.

In contrast, decoder-only architectures omit the encoder entirely and rely solely on masked self-attention. Each decoder layer computes attention over the preceding tokens while masking future positions to prevent information leakage. Since there is no encoder output to attend to, the model depends entirely on the previously generated tokens and internal representations. This design aligns naturally with causal language modeling where the training objective is to predict the next token given a prefix (Chalvatzaki et al., 2023). It also enables efficient scaling and deployment as seen in large models like GPT. The simplicity of decoder-only transformers has contributed to their widespread adoption for generative tasks such as text completion, dialogue, and instruction following.

Multimodal Large Language Models (LMMs)

While some LLMs operate solely on text, recent advancements have led to the development of multimodal large language models (LMMs). These models can process and reason over multiple types of input, such as text, images, audio, and video. For instance, OpenAI's model GPT-4V and many newer models can analyze images alongside textual prompts, enabling richer and more flexible interactions (Yin et al., 2024).

Multimodal LLMs are particularly promising for tasks where textual information alone is insufficient, such as interpreting web page layouts, visual elements, or screenshots of dynamic interfaces. This capability becomes crucial when information is rendered visually rather than being directly encoded in the HTML.

Context Windows

A context window defines the maximum number of tokens an LLM can consider in a single input. The size of the context window directly affects how much input data a model can

Model	Context window (tokens)
OpenAI GPT-4o	128,000
Anthropic Claude 3.7 Sonnet	200,000
Google Gemini 2.5 Pro	1,000,000
xAI Grok 3	1,000,000
DeepSeek V3	128,000

Table 3.2: Context window sizes of leading large language models (as of 2025). Sources: OpenAI (2024a); Anthropic (2025); Google DeepMind (2025a); xAI (2025); DeepSeek (2025).

process at once. Table 3.2 shows the context window sizes of several current state-of-the-art LLMs.

Larger context windows enable models to reason over longer documents or more detailed inputs. However, they also increase computational costs and may introduce latency due to the quadratic complexity of the attention mechanism (Zhu et al., 2024).

3.2.4 Agent Observation Modalities

An important design choice for computer-using agents is the format of the observation space, which may consist of a textual representation of the interface (such as HTML or DOM), a raw image, or a combination of both.

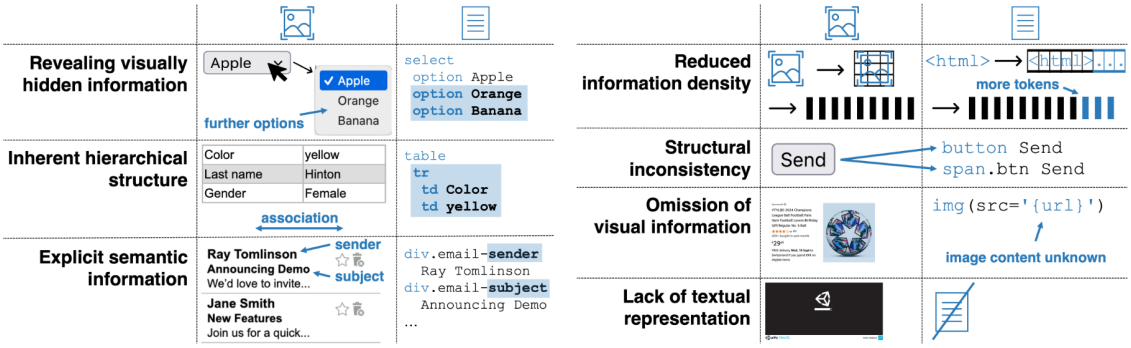
Earlier systems often relied on textual inputs which offer semantic clarity and perform well on structured, predictable benchmarks. However, in real-world scenarios HTML tends to be verbose and inconsistent across websites. This degrades the agent performance. To address this problem, modern agents increasingly adopt visual or bi-modal inputs (Sager et al., 2025). Vision-based inputs allow agents to perceive the interface as a user would, making them more robust in unstructured and dynamic environments. A comparison of textual versus visual input modalities can be seen in Figure 3.4.

3.3 Tools

Although the modern web has grown more complex and harder to scrape, developer tools have also evolved and are now offering more powerful and sophisticated solutions for web scraping. This section presents the main tools used in the implementation of our system. It includes technologies for browser automation and frameworks for building AI agents.

3.3.1 Playwright

The difficulty in scraping modern websites stems not just from whether they use CSR or SSR but primarily from the fact that content is often dynamically loaded or interaction-dependent. Key information may be missing from the initial HTML response, as it is loaded asynchronously after the page has rendered. In many cases this content only appears following



(a) Advantages (b) Disadvantages

Figure 3.4: Illustration of the advantages (a) and disadvantages (b) of textual screen representations compared to visual ones. Textual inputs, such as HTML code, offer benefits like access to hidden information, semantic structure, and explicit labels. However, they suffer from issues like verbosity, structural inconsistency, and loss of visual content (e.g., images, layout). Visual representations preserve what the user actually sees, making them more robust in real-world environments. Adapted from Sager et al. (2025).

user interactions such as clicking buttons or selecting options. As a result, scrapers must not only wait for the content to load but also simulate user behavior to access the required data (Whitaker, 2023).

To handle these challenges, browser automation tools have become essential. Playwright is one such framework, originally developed by Microsoft for end-to-end testing of modern web applications (Microsoft, 2025). It allows developers to automate real browser sessions and provides a powerful API for interacting with the page. This includes actions such as clicking elements, filling out forms, scrolling, waiting for network responses, and extracting dynamically rendered text. While Playwright was designed for testing, it has been widely adopted for web scraping tasks due to its reliability and flexibility in simulating real user behavior.

A common example of interaction-based content is a product configuration page, where available details or pricing only appear after the user selects specific options. On a (fictional) site like `examplecarrier.com/iphone-16`, a user must first select a phone color before related information is shown. This behavior is often implemented using JavaScript-controlled dropdown menus. A simplified version of such a dropdown might look like the following:

```
<select id="color-select">
  <option value="">Choose color</option>
  <option value="black">Black</option>
  <option value="blue">Blue</option>
  <option value="gold">Gold</option>
</select>
```

Listing 3.1: Example dropdown menu for phone color in HTML

To simulate this interaction and select the `blue` option using Playwright, the following script can be used:

```
const { chromium } = require('playwright');

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();
  await
   ↪ page.goto('https://www.examplecarrier.com/iphone-16');

  // Select "blue" from the color dropdown
  await page.selectOption('#color-select', 'blue');

  await browser.close();
})();
```

Listing 3.2: Playwright script to select a phone color from dropdown

This interaction not only sets the desired color on the page, but may also trigger dynamic updates such as image previews, stock status, or price changes. By scripting such behavior, Playwright enables scrapers to fully engage with interactive interfaces and extract the data that would otherwise remain hidden.

Chapter 4
Evaluation Method

To assess the effectiveness of our system, PriceFinderAgent, we designed a structured evaluation process that evolved alongside its development. This chapter describes how we measured its accuracy in extracting mobile contract data from real-world carrier websites, and explains the reasoning behind our evaluation design choices.

Our evaluation began with the development of scraping scripts for seven major Swedish mobile carriers, from which we created a large ground truth dataset of 1476 unique mobile contract offers. We then manually validated a subset to confirm the accuracy of our scraped data. To better understand relationships between different contract attributes, we performed a correlation analysis across all offers. Based on these insights, and to keep the benchmarking process both focused and realistic, we selected three representative carrier websites that differ in layout and interaction complexity.

Finally, we built a benchmarking framework to evaluate our system's performance in two core areas: identifying selectable options on a product page, and extracting accurate pricing information. As the project progressed, we narrowed the evaluation scope to a smaller set of manually collected offers to accommodate changes in data requirements and to enable a faster and more flexible benchmarking process.

Each section of this chapter details a specific stage of this evaluation design process, from initial data collection and validation to benchmarking and scope reduction.

4.1 Data Collection via Playwright Scripts

We initially surveyed the product page for iPhone 16 on seven Swedish mobile carriers that bundle phones with service plans: Comviq, Halebop, Hallon, Tele2, Telenor, Telia, and Tre. To build a gold standard dataset, we wrote manual Playwright scraping scripts for each site.

These scripts emulated user interactions such as clicking buttons and selecting options. Figure 4.1 shows the scraping loop. Each script iterated through combinations of options (hardware color, hardware storage, service plan, payment durations) and extracted price data.

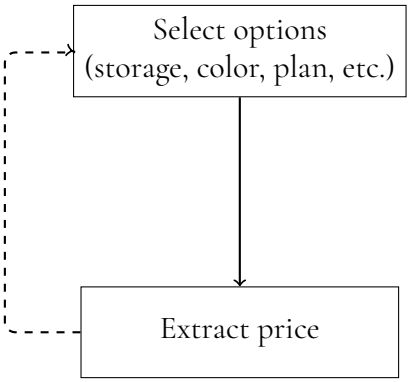


Figure 4.1: Scraping loop: selecting options and extracting price

Table 4.1: Attributes extracted per contract using Playwright scripts

Field	Description
Carrier	Mobile carrier providing the plan
Hardware name	Phone model name
Hardware color	Device color
Hardware storage (GB)	Storage size in GB
Hardware payment duration (months)	Duration of hardware payments
Hardware monthly price (SEK)	Monthly hardware price
Hardware upfront price (SEK)	Upfront hardware price
Data plan name	Name of the mobile data plan
Data plan allowance (GB)	Included data volume
Data plan payment duration (months)	Duration of the data plan
Data plan monthly price (SEK)	Monthly cost of the data plan
Increased allowance (GB)	Extra promotional data volume
Increased duration (months)	Months the promotion applies
Special payment duration (months)	Discount duration in months
Special monthly price (SEK)	Discounted monthly price

In total, we extracted data for 1476 unique mobile contracts. Each contract included 16 attributes, described in Table 4.1.

4.2 Data Validation

To ensure accuracy, we manually validated 20 randomly selected contracts from each carrier, totaling 140 samples. Table 4.2 summarizes the results. All 140 were correctly extracted by the Playwright scraping scripts, yielding 100% sample accuracy.

4.3 Data Analysis

To understand how different parts of a mobile contract relate to each other, we analyzed the structure of our collected data. Specifically, we computed a feature correlation matrix over

Table 4.2: Scraped phone plan offers from seven different carriers. The total number of offers per carrier, how many we validated manually and the accuracy of our samples based on manual validation.

Carrier	Number of Offers	Validated samples	Correct samples	Sample accuracy (%)
Comviq	120	20	20	100
Halebop	176	20	20	100
Hallon	28	20	20	100
Tele2	600	20	20	100
Telenor	222	20	20	100
Telia	210	20	20	100
Tre	120	20	20	100
Total	**1476**	**140**	**140**	**100**

all 1476 contracts, as shown in Figure 4.2. This allowed us to explore how attributes such as hardware storage size, hardware payment duration, and data allowance correlate with pricing.

Each value in the matrix represents the Pearson correlation coefficient between a pair of attributes, indicating the degree to which they vary together. A positive value signifies a direct relationship, while a negative value implies an inverse relationship. Key observations from the matrix include:

Service plan gigabytes vs. Service plan monthly price ($r = 0.82$): More data correlates with higher monthly price.

Hardware upfront price vs. Hardware payment duration ($r = -0.76$): Shorter contract leads to higher upfront price.

Hardware upfront price vs. Hardware monthly price ($r = -0.60$): Paying more upfront lowers the monthly price.

Hardware storage capacity vs. Hardware monthly price ($r = 0.24$): Larger storage leads to higher monthly price.

Note that the matrix includes only a subset of the 16 extracted attributes, specifically the most important numerical fields that are consistently available across all contracts. Less critical fields, such as promotional data and temporary discounts, were omitted from the figure for brevity.

4.4 Benchmark Scope and Website Selection

To maintain a high-quality yet manageable benchmark, we deliberately reduced the evaluation scope and selected a focused subset of mobile carrier websites. While our initial dataset covered 1476 contract combinations across seven carriers, benchmarking all of them proved impractical due to time and cost constraints. Instead, we chose to evaluate a sample of 30 contracts, 10 from each of three carefully selected carriers: Telia, Telenor, and Halebop.

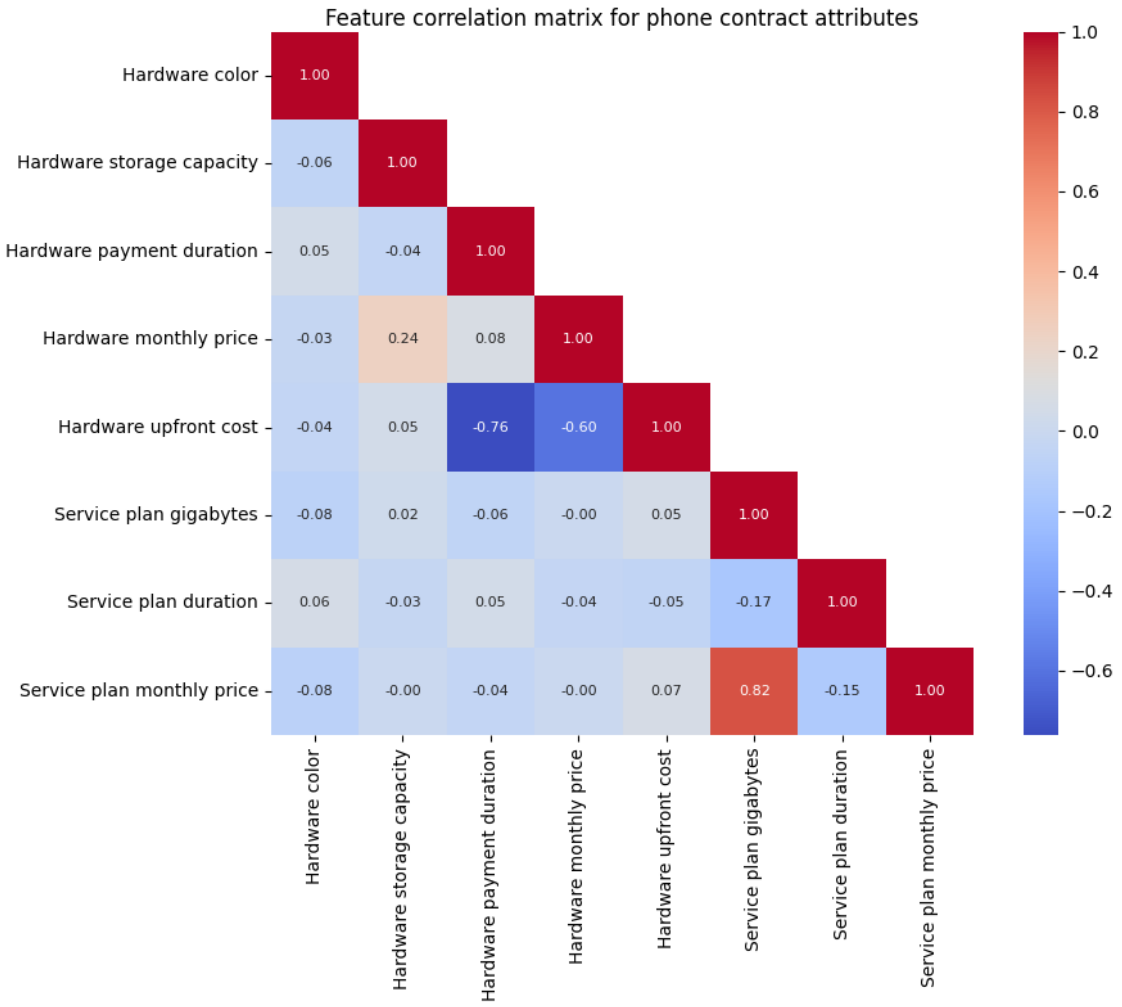


Figure 4.2: Correlation matrix of key features in 1476 mobile phone contracts. Red colors indicate positive correlation, while blue tones indicate negative correlation.

These three websites were selected based on the diversity of their user interface structures and interaction flows. In particular, they differed in how configuration options are presented and how pricing is accessed. This diversity allowed us to evaluate how well the system generalizes to websites it has not seen before, by simulating a range of real-world layouts and interaction patterns without benchmarking every carrier. Table 4.3 summarizes the selection flows and layout characteristics for each site.

The decision to switch from Playwright-based scraping to manual data collection for these 30 benchmark samples was also driven by evolving needs. As our understanding of relevant pricing attributes matured, we chose to include additional fields such as the total minimum price of the contract. Incorporating these changes into the existing scraping scripts would have been time-intensive, so for a small number of samples, manual collection proved faster. This reduced benchmark provided a realistic and meaningful basis for assessing the agent's capabilities under varied real-world conditions.

Table 4.3: Flow comparison on iPhone 16 product pages across the three carriers chosen for benchmarking.

Aspect	Telia	Telenor	Halebop
Color	Swatch on LP	Swatch on LP	Swatch on LP
Storage	Radio buttons on LP	Radio buttons on LP	Radio buttons on LP
Service plan	Radio buttons inside wizard	Radio buttons inside accordion	Radio buttons on LP
Service payment duration	Dropdown inside wizard	Toggle button on LP	Fixed at 24 months
Hardware payment duration	Fixed at 36 months	Radio buttons inside accordion	Radio buttons on LP
Pricing visibility	Modal view	Visible on LP	Modal view

Note: LP = Landing page

Table 4.4: Definitions of classification metrics used to evaluate the scraping agent.

Metric	Definition
True positive (TP)	Data was extracted and was correct
False positive (FP)	Data was extracted, but was incorrect
False negative (FN)	Data was not extracted, but it should have been
True negative (TN)	Data was not extracted, and it did not exist

4.5 Benchmark Design and Evaluation Metrics

We designed a benchmarking framework to evaluate two core abilities of our system: identifying which configuration options are available on a product page (option discovery) and extracting the correct prices for a specific contract setup (price extraction).

For both tasks, we manually collected ground truth values and compared them to the agent's output using standard classification metrics: true positive (TP), false positive (FP), false negative (FN), and true negative (TN). These are defined in Table 4.4. From these, we calculate precision, recall, accuracy, and F1-scores using the following formulae:

$$\textbf{Precision} = \frac{TP}{TP + FP}, \quad \textbf{Recall} = \frac{TP}{TP + FN},$$

$$\textbf{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}, \quad \textbf{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Precision measures how often the values extracted by the agent were correct, while **recall** indicates how many of the expected values were successfully found. **Accuracy** accounts for all correct outcomes, including both true positives and true negatives.

The **F1 score** combines precision and recall into a single metric by taking their harmonic mean, and is particularly useful when there is an uneven distribution of false positives and false negatives. Together, these metrics provide a comprehensive view of system performance across correctness, completeness, and robustness.

4.5.1 Option Discovery Benchmark

Our option discovery benchmark evaluated whether the agent could identify all selectable configuration options on a product page, such as hardware color, storage size, contract duration, and service plan.

For each page, the correct set of options was manually recorded. The agent's output was then compared to this ground truth using the classification metrics explained in Table 4.4. The results indicate how comprehensively and accurately the agent mapped the configuration space, a critical step for enabling downstream price extraction.

4.5.2 Price Extraction Benchmark

The price extraction benchmark was based on a dataset of 30 manually verified mobile contract configurations, with 10 contracts from each of the three selected carriers: Telia, Telenor, and Halebop. These samples represented a diverse range of pricing structures and contract setups.

The agent was given a target contract configuration and instructed to navigate the product page, select the correct options, and extract all relevant price components such as monthly price and upfront payment. The extracted results were compared to the ground truth using the metrics in Table 4.4.

We chose a task-wise evaluation approach for benchmarking price extraction, rather than a step-wise one. As illustrated in Figure 4.3, the agent must successfully complete a sequence of dependent steps: selecting the correct variant, navigating the interface, and reading the appropriate price fields. The evaluation, however, does not check each step individually. Instead, it only verifies whether the final extracted output is correct. This means that any error in the sequence, whether caused by a miss-selection or a misread value, results in the entire attempt being marked as incorrect.

A step-wise evaluation approach, where each individual action is assessed separately, could have been beneficial for debugging and understanding where the agent fails. It would have allowed us to isolate and quantify errors at specific points in the pipeline. However, due to time constraints and the added complexity of annotating and evaluating intermediate steps, we opted for a simpler task-wise approach. While less granular, this method still captures the end-to-end reliability of the system and reflects realistic usage where only the final result matters.

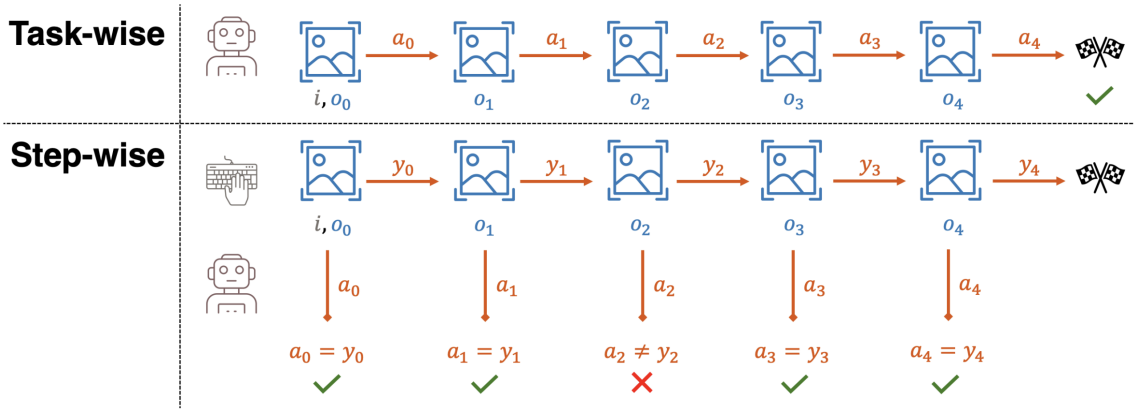


Figure 4.3: Comparison between task-wise and step-wise evaluation of AI agents. In task-wise evaluation (top), only the final result is assessed, meaning errors at any point in the sequence lead to failure. In step-wise evaluation (bottom), intermediate actions are evaluated individually, allowing partial credit. Our price extraction benchmark uses the task-wise strategy. Source: (Sager et al., 2025).

Chapter 5
Scraper Implementations

The goal of this project was to create a system that could autonomously extract all contract and price combinations from previously unseen carrier websites. Given the diverse and frequently changing structures of these websites, the system needed to generalize well without relying on site-specific rules or prior knowledge.

To address this challenge, we explored two AI-assisted scraping frameworks: Browser Use, a single-agent approach that attempts to complete tasks end-to-end, and Stagehand, which decomposes complex tasks into smaller, well-defined actions executed sequentially. To fully understand the strengths and limitations of each framework, we implemented and evaluated three distinct approaches:

- Single-agent scraping with Browser Use (Version 1)
- Rule-based AI-assisted scraping with Stagehand (abandoned)
- Multi-agent scraping with Stagehand and CUA (Versions 2–6)

Table 5.1 provides a summary of each scraper version developed throughout the project, including the underlying framework, the models used, and a brief description of the key improvements introduced in each iteration. The second approach, rule-based AI-assisted scraping with Stagehand, was discontinued before benchmarking and is therefore not included as a standalone version in the summary table.

Version	Framework	Model	Description
Version 1	Browser Use	GPT-4o	Fully autonomous single-agent scraper.
Version 2	Stagehand	CUA	Multi-agent scraper using visual agents driven by CUA. Built on groundwork from the rule-based Stagehand attempt.
Version 3	Stagehand	CUA	Extraction schema simplification, went from 8 to 3 attributes.
Version 4	Stagehand	CUA	New extraction attributes.
Version 5	Stagehand	CUA + GPT-4o	Improved navigation for dropdowns using a GPT-4o based agent.
Version 6	Stagehand	CUA + GPT-4o	Final version with hybrid agent delegation, increased viewport and prompt improvements.

Table 5.1: Summary of scraper versions.

This chapter provides a detailed overview of each implementation approach and explains the rationale behind our design decisions. At the end of the chapter, we also present a detailed description of the changes made in each benchmarked version listed in Table 5.1.

5.1 Single-Agent Scraping with Browser Use

Our initial implementation employed Browser Use to create a fully autonomous scraping agent. We hypothesized that a single, capable agent using a powerful foundation model could adapt to various website structures and complete the task of selecting contract options and extracting the price in one go. Figure 5.1 outlines this system.

5.1.1 Agent Prompt and Execution

The agent's workflow involved selecting contract options – such as hardware color, storage capacity, and service plan – and extracting the corresponding pricing information. It operated based on detailed instructions provided in a structured prompt, which included:

- A short introduction to the task
- Contract options configurations for the run
- Definitions of each attribute the agent is supposed to extract
- Step-by-step instructions specific to the target website

The full prompt given to the agent can be found in Appendix A.1. Our agent was implemented using the Browser Use project, with OpenAI's GPT-4o model serving as the underlying engine. Since Browser Use supports structured output, the agent was configured to return results in a fixed object format.

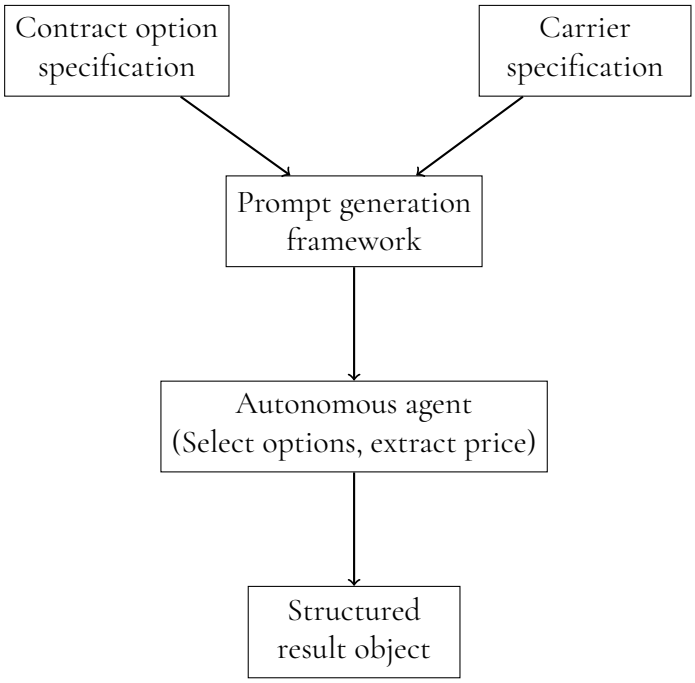


Figure 5.1: Overview of the autonomous scraping system. The prompt generation framework combines the contract option specification and carrier specifications to create a structured task prompt for the agent. The agent executes the task and returns a structured result.

5.1.2 Carrier Context and Framework

As part of our development process, we experimented with incorporating carrier-specific context to improve the agent's reliability. This included hints such as the location of pricing elements and basic navigation instructions. Inspired by OpenAI (2024), we designed a simple framework that dynamically generated prompts based on the current carrier.

This setup was not intended to be part of the final, generalizable system. Instead, it served as a temporary aid to better understand the agent's behavior during early iterations.

5.1.3 Limitations of Single-Agent Scraping

We eventually abandoned the single-agent Browser Use approach due to poor performance. While the system handled simple tasks reasonably well, our use case required precise, multi-step interactions to configure contract options and extract prices. In this setting, the agent struggled to reliably complete tasks and our benchmarks confirmed that its accuracy was too low for practical use. Rather than continuing to iterate on this architecture, we moved to a more modular setup.

The core issue was navigation. Browser Use attempts to complete tasks in a single continuous run without checkpoints or persistent state validation. This means that if the agent clicks the wrong button, selects an incorrect option, or navigates to the wrong page, the resulting state may silently diverge from the intended one – and those errors cascade through the rest of the session. For example, selecting the wrong color or service plan early in the

flow, without realizing or correcting it, leads to extracting the price for an unintended configuration. The agent does attempt to verify its current state against the task goal and would sometimes catch mistakes. But many times it also incorrectly concluded that the state was correct.

These navigation failures were often caused by limitations in Browser Use's DOM abstraction. The agent receives a textual representation of the page's structure as seen in Figure 2.2 along with an annotated screenshot. While this works for elements that are well-labeled in the DOM (e.g., buttons with clear aria-labels), many websites used non-standard structures. For example, some color swatches were implemented as `<div>` elements with only a background-color style and no inner text. In the Browser Use text representation, these were represented as empty `div` tags with no style information, making it impossible for the model to recognize them as color options and which color they represented. Screenshots helped somewhat but were unreliable in practice, especially on dynamic or visually complex interfaces.

5.2 Rule-based Scraping with Stagehand

To explore a more structured alternative to fully autonomous agents, we implemented what we call *a rule-based AI-assisted scraper* using the Stagehand framework. The goal of this approach was to have finer control over scraping workflows, while still leveraging the reasoning capabilities of LLMs. We also spent considerable time making this scraping system *self-healing* by implementing action caching.

5.2.1 System Design and Workflow

Our rule-based scraper operated as a pipeline of well-defined phases. Each phase performed a focused task and passed structured information to the next stage. While the Browser Use implementation ran end-to-end in a single agent loop, the rule-based Stagehand setup favored clarity and robustness by dividing scraping into smaller steps.

For this implementation, we used Playwright for page interaction, Stagehand for DOM analysis, and GPT-4o for decision making.

Phase 1: Discover Options

The first phase aimed to extract all configuration options on a product page (e.g., hardware color, hardware storage size, payment duration, service plan) using Stagehand's `extract` method, which returned structured data from the DOM. Inspired by the fuzzy matching in Zhou et al. (2023), this output was then passed to GPT-4o, which categorized the extracted options into six groups: hardware color, hardware storage, hardware payment duration, service plan, service plan payment duration, and other.

From the labeled options, we generated all possible selection combinations. Each selection combination represented a unique mobile contract configuration defined by one option from each category. This allowed us to define a clear loop of what combinations needed to be selected. Each combination served as the basis for a scraping session in the next phase.

Phase 2: Select Options

For each selection combination, we performed two steps: selection and extraction. Using Stagehand's `observe` method, we identified selectors for each option in the option combination. Then, with `act`, we simulated clicks to select the desired configuration. This sequence mimicked a user manually selecting each combination.

Phase 3: Extract Price

Once the correct variant was selected, we used Stagehand's `extract` again to retrieve all visible prices, each labeled and categorized via structured output. The list was then passed to GPT-4o, which identified the relevant values for monthly hardware price, upfront hardware price, monthly service plan price, and any additional price components. These values were saved in a structured results object, or contract, as we call it.

5.2.2 Self-Healing Caching Mechanism

To increase efficiency and reduce the number of LLM-calls being made, we developed a caching mechanism that allowed our rule-based scraper to remember how to interact with known page states. Specifically, whenever the an AI action generated a working Playwright command (e.g., selecting the phone color Red), we stored the action in a cache indexed by a hash of the current URL and the specific action context.

The next time the scraper encountered the same page and needed to perform the same action, it first checked the cache. If a match was found, it executed the previously successful Playwright snippet directly, skipping the LLM entirely.

Our caching mechanism also accounted for change and failure. If the cached action no longer succeeded, perhaps due to a page layout change, the script would gracefully fall back to the AI model to regenerate the Playwright code, execute the new action, and update the cache accordingly. This created a feedback loop that allowed the scraper to self-heal in response to website changes, without manual intervention.

This mechanism worked well in practice and significantly reduced the number of model calls needed during scraping runs.

5.2.3 Limitations of Rule-Based Scraping

Unfortunately, we never had the opportunity to benchmark our rule-based scraper, as it struggled with navigating more complex websites such as Telia and Telenor, which rely on multi-step wizards for contract option selection. While the scraper worked reliably on simpler sites like Halebop, the fully rule-based approach proved too rigid and lacked the flexibility needed to generalize across more intricate site structures.

As a result, we decided to pivot towards a hybrid approach that combines the strengths of our two first implementations. Specifically, we explored a version of our rule-based Stagehand implementation that leveraged agents for small, focused tasks, what we call *multi-agent scraping* using Stagehand and OpenAI's Computer-Using Agent (CUA).

5.3 Multi-Agent Scraping with Stagehand and CUA

During the development of our rule-based Stagehand scraper, OpenAI released its latest computer-using model, `computer-use-preview`. This enabled us to evolve our system by integrating agent-based reasoning into our existing rule-based framework.

The core idea was to maintain the precision and predictability of the rule-based system while introducing a layer of flexibility through agents capable of reasoning about user interfaces. In our original rule-based Stagehand implementation, a prompt such as *"extract the price"* assumed that the price was already visible on the page. This assumption worked for simpler websites like Halebop, where all information is visible on page at all times, but failed for more complex carrier websites such as Telia and Telenor, where users must navigate interactive wizards to reveal pricing information.

In this hybrid version, we still issue the same command, *"extract the price"*, but assign it to an agent. This agent executes the task in multiple steps. It first evaluates whether the price is currently visible and, if not, decides which interactions (such as clicks or selections) are necessary to make it visible. This process adds contextual reasoning that the earlier system lacked.

To reduce the risk of errors and hallucinations, we designed the system to assign only small and well-defined tasks to each agent. Instead of attempting to solve complex objectives with a single agent, the system decomposes the scraping workflow into a sequence of focused actions. Each of these actions is handled by an agent with a specific and limited scope.

This implementation, which became our final and most successful version after several iterations, represents a hybrid between the fully autonomous Browser Use agent and the deterministic rule-based Stagehand scraper. It uses OpenAI's `computer-use-preview` model to coordinate a set of purpose-driven agents, each responsible for a small part of the overall scraping task. An overview of our final version, which we have named PriceFinderAgent, can be seen in Figure 5.2.

5.3.1 Agent Roles and Workflow

Similar to our previous rule-based Stagehand implementation, this scraping pipeline is divided into three main phases: discover options, select options, and extract prices. Each phase is handled by a dedicated agent designed to fulfill a specific role.

Phase 1: Discover Options

As in our original rule-based Stagehand implementation, the process begins with option discovery. In this version, however, the task is delegated to a dedicated **option discovery agent** that navigates the product page to identify configurable options. Rather than attempting the entire task in a single pass, the agent executes five clearly defined sub-tasks:

1. Find all available colors
2. Find all storage size options

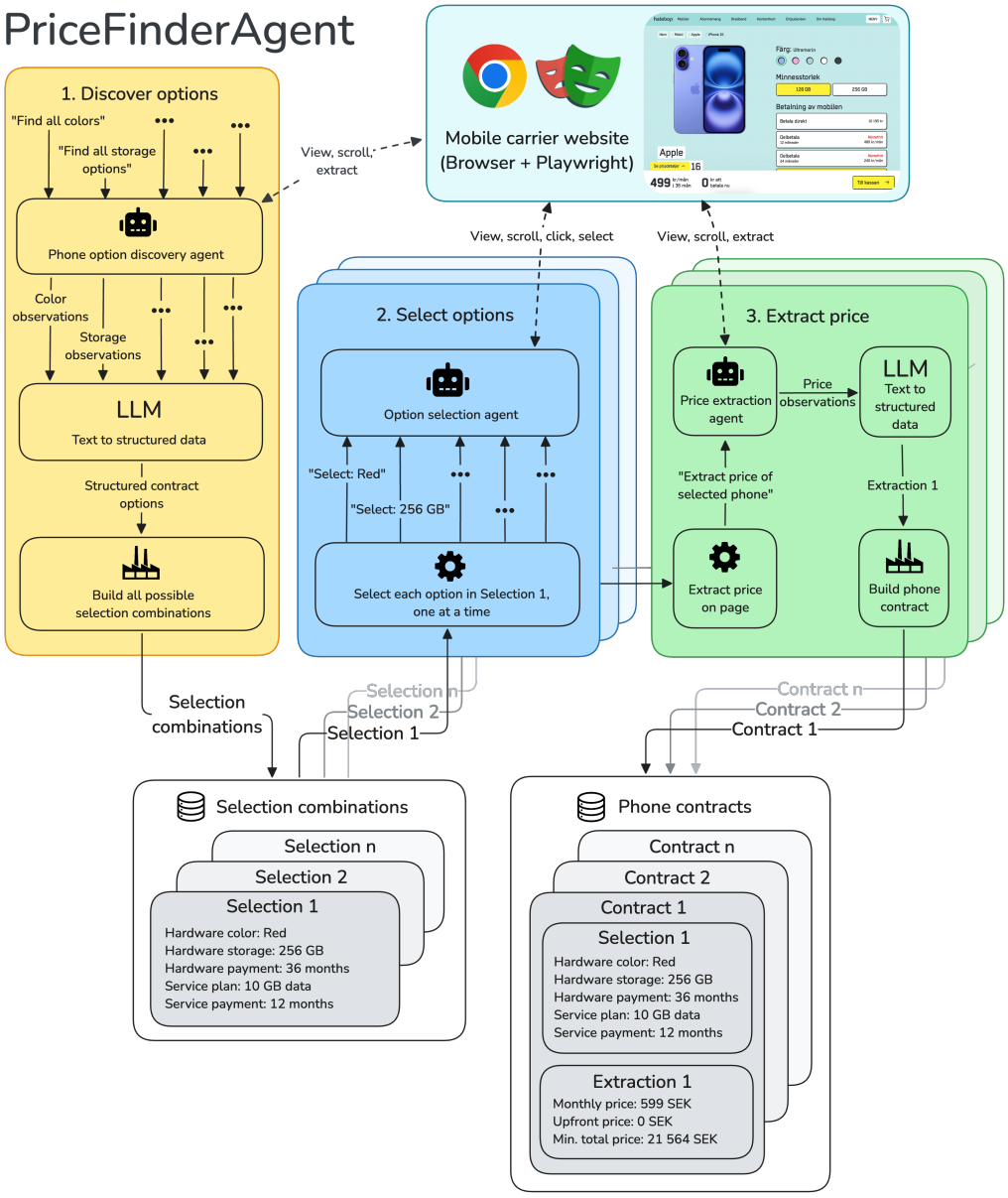


Figure 5.2: Overview of PriceFinderAgent, a multi-agent system for extracting all available phone contracts from a mobile carriers website. The standardized format of the contracts enables price comparison between carriers.

3. Find all hardware payment durations

4. Find all available service plans

5. Find all service plan payment durations

Each sub-task is performed independently, with a fresh page load for every run to ensure a clean and consistent starting state. Prompts given to the agent in this phase can be found in Appendix B.1. The agent returns results in natural language, such as `"I found the colors`

`gray, black, and white"`. These responses are then passed to a secondary LLM, GPT-4o, which parses the text and extracts structured values (e.g., color names) based on a predefined schema.

The result of these operations are five arrays, each representing the discovered options for one of the five categories. For instance, the array for colors might be `["gray", "black", "white"]`.

Using the discovered options, we then programmatically construct a set of unique selection combinations. Each selection combination contains one value from each of the five configurable categories: hardware color, hardware storage, service plan, hardware payment duration, and service plan payment duration.

An overview of this step within the broader system is illustrated in Figure 5.2, where the generated selection combinations are shown feeding into the subsequent selection and extraction phases.

Phase 2: Select Options

To interact with the website and select a given configuration, we create a **variant selection agent**. This agent is guided by a system prompt that provides both the current page context and explanations of domain-specific terms such as service plan, hardware payment duration, and so on. For each variant, the agent performs a step-by-step selection of its components, for instance:

1. Select the color `Red`.
2. Select the storage option `256 GB`.
3. Select the hardware payment duration `36 months`.
4. Select the service plan `10 GB`.
5. Select the service plan payment duration `12 months`.

Prompts given to the agent in this phase can be found in Appendix B.2. If the agent fails to select one of the options, due to page state inconsistencies or unexpected UI behavior, it reloads the product page and retries the full selection sequence from the beginning. This retry process is limited to a maximum of three attempts. The number three was not chosen for any theoretical reason but was selected as a practical and reasonable balance between robustness and efficiency. Should all three attempts fail, the variant is marked as unselectable, and the pipeline proceeds to the next selection combination.

Phase 3: Extract Price

Once the correct variant is selected, we launch a **price extraction agent**. Its system prompt, which can be read in its entirety in Appendix B.3, specifies the currently selected configuration and clearly instructs the agent not to alter any option values. It also informs the agent that price information may be located behind dropdowns, modals, or within the checkout process.

The agent then explores the page to find all relevant price information and returns a detailed natural language summary. This summary often contains rich but unstructured information, such as:

> *The monthly payment for the phone is 599 SEK, there is no upfront cost. The minimum total price is 21 564 SEK.*

This response is then passed to a different LLM that maps the data into a structured result object. The final output is a structured representation of a phone contract, as can be seen in the box labeled "Contracts" in Figure 5.2.

5.3.2 Iterative Improvements and Learnings

As shown in Table 5.1, we developed six different versions of our scraping system over the course of this project. These versions represent a progression from a single-agent baseline to a robust and more modular multi-agent architecture. This section details the rationale behind each version, the improvements introduced, and the lessons learned along the way.

Version 1: Single-Agent Scraper

Our first implementation employed the Browser Use framework to create a fully autonomous single-agent scraper. This version, described in Section 5.1, served as a baseline for performance. While conceptually appealing due to its simplicity, this system struggled with navigation and reliable price extraction. These shortcomings became clear during benchmarking, which led us to shift our focus toward a more controllable and modular architecture.

Version 2: First Multi-Agent Scraper

Developed in parallel with Version 1, this iteration built upon the rule-based architecture introduced in Section 5.2, but replaced some of the rigid logic with CUA-powered agents. While the core structure – option discovery, selection, and extraction – remained intact, these tasks were now delegated to specialized agents, resulting in better flexibility. Benchmarking results revealed a clear performance advantage over the single-agent approach, justifying further investment in this direction. All subsequent versions evolved from this foundation.

Version 3: Fewer Extraction Attributes

In early versions, our system attempted to extract up to eight pricing attributes per variant, including hardware monthly price, hardware upfront price, plan monthly price, and more nuanced elements such as “special offers” or “data increase duration". However, during evaluation, we observed that the agent occasionally hallucinated values – sometimes fabricating prices entirely or mixing up which price corresponded to which attribute. In other cases, the agent correctly extracted some fields while inventing others.

We hypothesized that this behavior stemmed from cognitive overload: the agent was being asked to interpret too much at once in a high-ambiguity environment. To mitigate this, we simplified the extraction schema and reduced the number of target attributes to just three of our original eight: hardware monthly price, hardware upfront price, and service

plan monthly price. By narrowing the agent's focus, we made the task more concrete and less error-prone.

Version 4: Different Extraction Attributes

Although the simplified attributes in Version 3 worked better than previous versions, we recognized that not all carriers exposed the same price components in a consistent format. To improve generalizability, we surveyed our seven original carrier websites and identified three pricing attributes that were universally available: *total monthly price*, *total upfront price*, and *minimum total price*. These values represent the most standardized and comparable pricing information across carriers.

We therefore modified the extraction schema to focus exclusively on these three attributes. This change increased cross-carrier comparability and retained enough information to support both total cost comparisons and monthly affordability analysis.

Version 5: Handling Dropdown Selection Issues

While earlier iterations focused primarily on improving data extraction accuracy, we found that many failures were actually rooted in the option selection phase – particularly on websites like Telia, where dropdown menus are used for option selection. These dropdowns posed a challenge for our CUA-driven agents, as the contents of expanded dropdowns were excluded from the screenshots used for reasoning. As a result, the agent could not see the selectable elements in the dropdowns and would therefore fail to interact with the element entirely.

To address this issue, we introduced a secondary option selection agent powered by GPT-4o. Unlike the original CUA agent, which relied on screen coordinates and visual cues, the GPT-4o agent interacted with the page programmatically using Playwright. Specifically, it accessed the Chrome accessibility tree to locate and manipulate DOM elements directly. This gave the agent a robust way to identify and interact with dropdowns – even if their contents were not visible in a screenshot.

The GPT-4o agent was invoked specifically for selection steps involving dropdown menus. It was explicitly instructed to use Playwright's `selectOption` method when handling these elements, which significantly improved reliability and precision. The underlying approach – using Playwright in combination with the Chrome accessibility tree – is described in more detail in Section 2.4.

Version 6: Fine-Tuning Prompts Selective Agent Use

In the final version, we conducted detailed manual evaluations of the scraping pipeline to identify weak spots in agent reasoning. By watching the system run through its benchmark, we noted specific behaviors that led to failure or confusion. These observations informed targeted prompt refinements for each agent role. We clarified task scopes and explicitly warned against common pitfalls. These refinements improved overall reliability and marked the final stage of our scraper development. The final prompts used in this version can be found in Appendix B.

In addition to prompt improvements, we also refined our use of the dual-agent option selection strategy introduced in Version 5. While Version 5 used a hybrid setup – invoking

both the CUA and GPT-4o agents – for all three carriers (Telia, Telenor, and Halebop), we observed that only Telia required this. Telenor and Halebop do not rely on dropdown menus for option selection, and the CUA agent performs well on both. In fact, the CUA agent generally achieved better performance when dropdowns were not involved. Therefore, in Version 6, we restricted the use of the GPT-4o option selection agent to Telia only, where dropdowns made it necessary.

Although we did not implement a fallback mechanism in which the system automatically switches to the GPT-4o agent if the CUA agent fails, this could be added quite easily. Due to time constraints, we opted to enforce the agent split manually in this version. This also allowed us to clearly isolate the impact of each agent on the benchmark results.

Chapter 6
Results

In this chapter, we present the results from our two benchmarks: *option discovery* and *price extraction*. We evaluated only a single scraping approach for option discovery because the initial implementation yielded satisfactory results. In contrast, we tested multiple scraping strategies for price extraction to explore performance differences. The results show how accurately each system extracted the required information relative to the ground truth. We executed each test once.

6.1 Option Discovery Results

By option discovery, we mean the process of identifying the available options associated with mobile contracts, such as hardware storage sizes, hardware colors, or service plans. We conducted a full evaluation across three carriers – Telia, Telenor, and Halebop – and extracted five distinct option groups per carrier. Each group contains a varying number of options depending on the carrier, and together, they form a contract combination. Section 5.3.1 describes the implementation details of our option discovery.

Table 6.1 shows the performance of our option discovery agent using precision and recall as evaluation metrics. Precision measures how many of the extracted options were correct, while recall indicates how many relevant options the agent successfully discovered. This task particularly requires high recall because if the agent misses options, it may fail to generate accurate contracts.

As shown in Table 6.1, the agent achieved perfect extraction performance for both Telenor and Halebop. For Telia, however, the agent missed three out of eight service plan options, which led to a lower recall for that specific option group. These service plans were missed because there existed multiple different plans with almost identical names. Our agent thus grouped them together under the same service plan. We believe that tuning the prompts even more will solve this issue, but leave that for future work.

Table 6.1: Option extraction performance across carriers. Format: Precision / Recall.

Attribute	Telia	Telenor	Halebop
Hardware color options	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Hardware storage options	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Hardware payment durations	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Service plan options	1.00 / 0.63	1.00 / 1.00	1.00 / 1.00
Service plan payment durations	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Total	**1.00 / 0.86**	**1.00 / 1.00**	**1.00 / 1.00**

6.2 Price Extraction Results

We evaluated six different iterations of our price extraction pipeline, each varying in key aspects such as the scraping framework, the underlying language model, and the set of targeted extraction attributes.

Versions 1 and 2 serve as our baseline systems. We used Version 1 to evaluate our single-agent scraper built with Browser Use and Version 2 to evaluate our multi-agent scraper built with Stagehand and CUA. Both versions target a total of eight attributes, as detailed in Tables 6.3 and 6.4.

In Versions 3 and 4, we refined the attribute set by reducing the number of extracted data points from eight to three. For Version 4 specifically, we selected three entirely different attributes because we discovered that these were more consistently available across all carriers.

With Versions 5 and 6, we focused on improving how the agent selects elements and interacts with the page. To achieve this, we enhanced the prompt design and refined the agent's interaction handling mechanisms.

Chapter 5 provides detailed implementation descriptions, and Table 5.1 presents a high-level comparison of all six versions.

6.2.1 Evaluation Metrics

As defined earlier in Section 4.5, we use standard classification metrics to evaluate each scraper version. With the actual results data at hand, we highlight two metrics as particularly relevant: **precision** and **accuracy**.

Precision measures how often the scraper produces a correct value. This is especially important in our case, where the most common failure mode involves false positives – extracting incorrect values or values that do not exist.

Accuracy offers additional context by counting both correct extractions and correct omissions. This makes it useful in cases where many fields are irrelevant or empty.

In practice, we consistently observed that the scraper produced zero or very few false negatives across all carriers. This means the scraper almost never missed values that were actually present, resulting in a recall of approximately 1.0. Although this outcome is desirable,

it also inflates the F1 score and reduces its informativeness, since F1 is the harmonic mean of precision and recall. For this reason, we primarily rely on precision as our main evaluation metric and use accuracy to provide additional context.

6.2.2 Results Overview

Table 6.2 summarizes the performance of all six scraper versions. For consistency, we report metrics for exactly three attributes per version. In Versions 1 to 3, we selected *hardware monthly price*, *hardware upfront price*, and *plan monthly price* as the evaluation attributes. Although Versions 1 and 2 extracted a total of eight attributes (see Tables 6.3 and 6.4), we excluded the remaining ones from the overview table. We made this decision to ensure comparability across versions, and we chose the three selected attributes because they represent the minimum required to meaningfully compare subscription prices.

In Versions 4 to 6, we targeted a different set of three aggregated attributes: *minimum total price*, *total upfront price*, and *total monthly price*. Although these fields differ in content, we maintained a consistent table structure – each version contributes exactly three attributes to the TP/FP/TN/FN calculations.

This overview provides a high-level comparison of precision, recall, F1 score, and accuracy across carriers and versions. The numbers reveal a clear trend: each new version improves upon the previous one, especially in terms of precision, while Halebop consistently yields the best results due to its simpler page structure.

6.2.3 Baseline Performance: Version 1 and 2

Tables 6.3 and 6.4 present the detailed per-attribute results for Versions 1 and 2, which we treat as our baseline systems. As previously noted, both versions extracted eight fields in total, but we emphasize three business-critical attributes – *hardware monthly price*, *hardware upfront price*, and *plan monthly price*. For clarity, we display these attributes above a dashed line in each table.

At a glance, the total performance across carriers in Version 1 may appear similar. However, when we isolate the business-critical attributes, we observe clear differences (see Table 6.3 and Table 6.2). Halebop stands out with higher precision and accuracy, while Telenor and Telia fall behind. Telenor shows the lowest overall precision, F1 score, and accuracy, whereas Telia records the lowest recall at 0.67.

Halebop consistently achieves the best results in both versions. Its static layout, with all options clearly visible on the product page, enables agents to extract data more accurately. In contrast, Telia and Telenor perform worse due to their use of interactive UI elements. In Version 1, the agent selected the correct contract configuration for Telia in 0% of cases, for Telenor in 20%, and for Halebop in 80%. These results indicate that most failures stemmed from the agent reaching the wrong context rather than incorrectly parsing visible prices.

Version 2 demonstrates clear improvements over Version 1, especially for the business-critical attributes. Precision for these fields increases from 0.23 to 0.52, while accuracy improves from 0.47 to 0.64. We attribute this improvement to better navigation and selection behavior. The CUA-based agent handled interactive elements – such as color swatches and wizard flows – more effectively than the Browser Use approach in Version 1.

Table 6.2: Scraping performance comparison across six scraper versions

Version	Carrier	TP	FP	TN	FN	Precision	Recall	F1	Acc
Version 1	Telia	4	14	10	2	0.22	0.67	0.33	0.47
– Browser Use	Telenor	2	21	8	0	0.09	1.00	0.16	0.32
– GPT-4o	Halebop	8	12	10	0	0.40	1.00	0.57	0.60
– 8 parameters	**Total**	**14**	**47**	**28**	**2**	**0.23**	**0.88**	**0.35**	**0.47**
Version 2	Telia	5	15	10	0	0.25	1.00	0.40	0.50
– Stagehand	Telenor	12	12	6	0	0.50	1.00	0.67	0.60
– CUA	Halebop	16	4	9	1	0.80	0.94	0.86	0.83
– 8 parameters	**Total**	**33**	**31**	**25**	**1**	**0.52**	**0.97**	**0.64**	**0.64**
Version 3	Telia	7	13	9	1	0.35	0.88	0.50	0.53
– Stagehand	Telenor	12	8	8	2	0.60	0.86	0.71	0.67
– CUA	Halebop	17	3	10	0	0.85	1.00	0.92	0.90
– 3 parameters	**Total**	**36**	**24**	**27**	**3**	**0.60**	**0.92**	**0.71**	**0.70**
Version 4	Telia	6	14	10	0	0.30	1.00	0.46	0.53
– Stagehand	Telenor	21	9	0	0	0.70	1.00	0.82	0.70
– CUA	Halebop	22	1	7	0	0.96	1.00	0.98	0.97
– 3 parameters	**Total**	**49**	**24**	**17**	**0**	**0.67**	**1.00**	**0.80**	**0.73**
Version 5	Telia	14	6	9	1	0.70	0.93	0.80	0.77
– Stagehand	Telenor	23	7	0	0	0.77	1.00	0.87	0.77
– CUA + GPT-4o	Halebop	20	3	7	0	0.87	1.00	0.93	0.90
– 3 parameters	**Total**	**57**	**16**	**16**	**1**	**0.78**	**0.98**	**0.86**	**0.82**
Version 6	Telia	16	4	10	0	0.80	1.00	0.89	0.87
– Stagehand	Telenor	27	3	0	0	0.90	1.00	0.95	0.90
– CUA + GPT-4o	Halebop	23	0	7	0	1.00	1.00	1.00	1.00
– 3 parameters	**Total**	**66**	**7**	**17**	**0**	**0.90**	**1.00**	**0.95**	**0.92**

Among the core attributes across Versions 1 and 2, *hardware upfront price* achieved the highest accuracy because the value is often zero and clearly labeled. *Plan monthly price* generally proved more reliable than *hardware monthly price*, but Telia and Telenor frequently mixed the base price with promotional variants, which led the scraper to extract discounted values instead. These deviations from the ground truth made it more difficult to extract the correct price, even when the agent reached the intended context. *Hardware monthly price* remained the most error-prone attribute due to inconsistent layout structures and mixed formatting of units.

In summary, Version 2 significantly improved performance on Telenor and Halebop but continued to struggle with Telia. Telia's structure still posed interaction challenges, and the agent often failed to interpret complex pricing layouts or correctly map visual data into the output format, especially when promotions or bundled offers appeared.

6.2.4 Simplified Extraction: Versions 3 and 4

We designed Versions 3 and 4 to simplify the price extraction schema and reduce confusion caused by promotional prices and non-essential attributes. Tables 6.5 and 6.6 present their

Table 6.3: Version 1 detailed breakdown: Precision and Accuracy per attribute across carriers (Browser Use + GPT-4o). Format: Precision / Accuracy. The most important metrics are above the dashed line.

Attribute	Telia	Telenor	Halebop
Hardware monthly price	0.13 / 0.10	0.00 / 0.20	0.50 / 0.60
Hardware upfront price	– / 1.00	0.00 / 0.60	0.00 / 0.80
Plan monthly price	0.30 / 0.30	0.18 / 0.18	0.40 / 0.40
Plan monthly price special	0.00 / 0.00	0.00 / 0.09	0.00 / 0.10
Plan payment special	0.50 / 0.50	0.67 / 0.64	0.00 / 0.10
Plan increased data	0.50 / 0.90	– / 1.00	– / 1.00
Plan increased data duration	0.50 / 0.90	– / 1.00	– / 1.00
Availability	0.90 / 0.90	0.91 / 0.91	1.00 / 1.00
Total	**0.38 / 0.58**	**0.35 / 0.58**	**0.38 / 0.63**

Table 6.4: Version 2 detailed breakdown: Precision and Accuracy per attribute across carriers (Stagehand + CUA). Format: Precision / Accuracy. The most important metrics are above the dashed line.

Attribute	Telia	Telenor	Halebop
Hardware monthly price	0.10 / 0.10	0.00 / 0.20	0.67 / 0.70
Hardware upfront price	– / 1.00	0.33 / 0.60	1.00 / 0.90
Plan monthly price	0.40 / 0.40	1.00 / 1.00	0.90 / 0.90
Plan monthly price special	0.29 / 0.50	0.40 / 0.40	0.00 / 0.40
Plan payment special	0.43 / 0.60	0.80 / 0.80	0.00 / 0.40
Plan increased data	– / 1.00	– / 1.00	0.00 / 0.90
Plan increased data duration	– / 1.00	– / 1.00	0.00 / 0.90
Availability	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Total	**0.45 / 0.70**	**0.63 / 0.75**	**0.59 / 0.76**

results.

As we described in Section 5.3.2, Version 3 retained the original schema but limited the output to three high-value attributes: *hardware monthly price*, *hardware upfront price*, and *plan monthly price*. In contrast, Version 4 introduced a new schema with three different attributes – *minimum total price*, *total upfront price*, and *total monthly price* – which we chose because websites often displayed them more clearly and used them to summarize final pricing.

For both versions, we used the CUA agents with Stagehand, building on earlier findings that this setup improved navigation (see Section 5.3.2).

When we compared these results to those from Version 2, we found that both Versions 3 and 4 achieved higher precision and accuracy. This improvement resulted primarily from a reduction in false positives and an increase in true positives. Among the two, Version 4 performed slightly better overall. These results support our hypothesis that simplifying the extraction schema – by either reducing the number of attributes or focusing on elements that are more visible on the page – reduces ambiguity and improves output quality.

Despite these gains, Telia continued to be challenging. During manual inspection, we observed that the agent consistently failed to select options from dropdown menus, even

Table 6.5: Version 3 detailed breakdown: Precision and Accuracy per attribute across carriers (Stagehand + CUA with simpler scraping object). Format: Precision / Accuracy. The most important metrics are above the dashed line.

Attribute	Telia	Telenor	Halebop
Hardware monthly price	0.40 / 0.40	0.00 / 0.60	0.71 / 0.80
Hardware upfront price	– / 0.90	0.83 / 0.70	1.00 / 1.00
Plan monthly price	0.30 / 0.30	0.70 / 0.70	0.90 / 0.90
Total	**0.35 / 0.53**	**0.60 / 0.67**	**0.85 / 0.90**

Table 6.6: Version 4 detailed breakdown: Precision and Accuracy per attribute across carriers (Version 4 – Aggregated price metrics). Format: Precision / Accuracy.

Attribute	Telia	Telenor	Halebop
Minimum total price	0.30 / 0.30	0.60 / 0.60	1.00 / 1.00
Total upfront price	– / 1.00	0.80 / 0.80	1.00 / 1.00
Total monthly price	0.30 / 0.30	0.70 / 0.70	0.90 / 0.90
Total	**0.30 / 0.53**	**0.70 / 0.70**	**0.96 / 0.97**

though it handled all other UI elements correctly. This failure alone caused nearly half of the scrapes on Telia to return incorrect results. On Telenor, the agent frequently selected the wrong plan because the plans had similar names.

6.2.5 Navigation and Prompt Improvements: Versions 5 and 6

Tables 6.7 and 6.8 present the results for Versions 5 and 6. In these versions, we focused on resolving remaining interaction issues and tuning agent assignments per carrier based on performance observations from earlier runs.

In Version 5, we addressed a specific navigation failure on Telia related to the selection of the service plan payment duration. Previous agents had used the CUA model across all steps, but it failed to interact reliably with dropdown elements. To solve this, we assigned a GPT-4o-based agent to handle that particular step across all carriers. This single change led to a significant performance boost, with Telia's precision and accuracy rising to match Telenor's.

In Version 6, we further refined the setup. After observing that the GPT-4o agent performed poorly on Telenor – where the service plan payment duration relied on a toggle button instead of a dropdown – we reverted to using the CUA agent exclusively for both Telenor and Halebop. This change significantly improved precision and accuracy. For Telia, we retained the hybrid setup, using GPT-4o for dropdown interactions and CUA for all other steps. Halebop, which never required dropdown interaction, continued using only CUA but still benefited from improved prompt phrasing in Version 6.

All carriers showed improved performance in Version 6. Telenor reached 90% precision with perfect recall, and Halebop achieved perfect scores across all metrics. Telia also reached

Table 6.7: Version 5 detailed breakdown: Precision and Accuracy per attribute across carriers (Version 5 – Aggregated price metrics). Format: Precision / Accuracy.

Attribute	Telia	Telenor	Halebop
Minimum total price	0.70 / 0.70	0.60 / 0.60	0.90 / 0.90
Total upfront price	– / 0.90	0.90 / 0.90	0.67 / 0.90
Total monthly price	0.70 / 0.70	0.80 / 0.80	0.90 / 0.90
Total	**0.70 / 0.77**	**0.77 / 0.77**	**0.87 / 0.90**

Table 6.8: Version 6 detailed breakdown: Precision and Accuracy per attribute across carriers (Version 6 – Aggregated price metrics). Format: Precision / Accuracy.

Attribute	Telia	Telenor	Halebop
Minimum total price	0.80 / 0.80	0.90 / 0.90	1.00 / 1.00
Total upfront price	– / 1.00	0.90 / 0.90	1.00 / 1.00
Total monthly price	0.80 / 0.80	0.90 / 0.90	1.00 / 1.00
Total	**0.80 / 0.87**	**0.90 / 0.90**	**1.00 / 1.00**

its highest performance to date. These results underscore the importance of prompt design and robust navigation capabilities in achieving reliable extraction performance.

6.3 Cross-Version Comparison

Across Versions 1 to 6, we consistently observe improvements in both precision and accuracy, as illustrated in Figure 6.1. Recall has never posed a major challenge and has remained close to 1.0 throughout the versions, as shown in Figure 6.2. Among the three carriers, Halebop proved to be the easiest to handle, reaching a precision of 0.8 as early as Version 2.

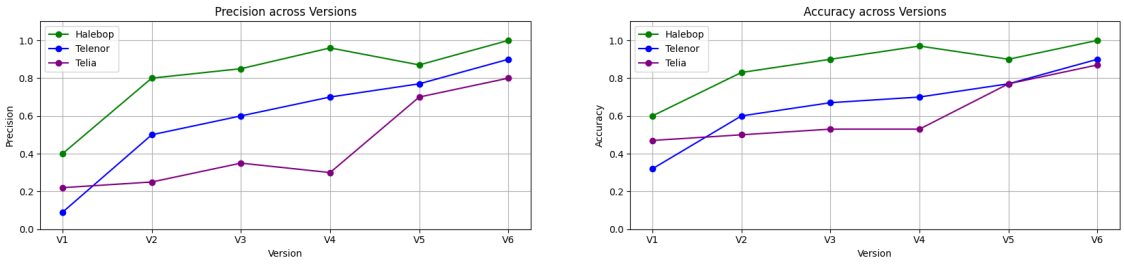


Figure 6.1: Precision (left) and Accuracy (right) across five scraper versions. Halebop shows high and stable results. Telenor improves steadily. Telia lags behind until Version 5.

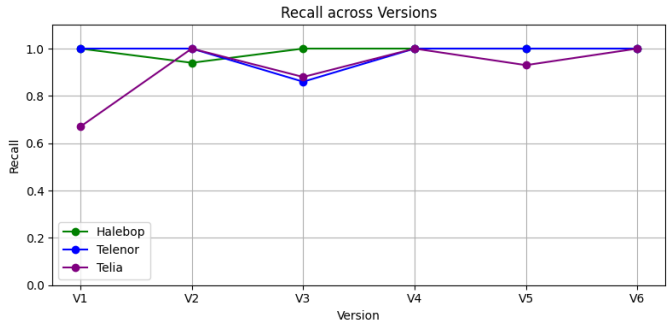


Figure 6.2: Recall across five scraper versions. Recall remains high across most carriers and versions, indicating few missed extractions.

6.4 Telemetry

We recorded telemetry data for Version 6 to better understand the resource demands of the final system. The results, shown in Figure 6.3, include average agent steps, inference time, and token usage per contract, broken down by carrier.

Telia had the highest resource usage across all metrics, followed by Telenor. Halebop required the fewest steps and the least time and tokens per scrape. These differences align with each carrier's UI complexity: Halebop presents most information up front, while Telia involves deeper navigation and interactive elements such as dropdowns and wizards.

Figure 6.4 shows a breakdown of agent steps per subtask. The service plan selection step on Telia stands out as particularly costly which makes sense given that it involved navigating a multi-step wizard. In contrast, simpler tasks like selecting storage required minimal effort across all carriers.

Token usage and cost are detailed in Table 6.9. Telia consumed nearly 129,000 tokens per scrape, translating to an average cost of $0.40. Halebop was the most efficient, at only $0.15. Most token usage comes from CUA input tokens, with a minor contribution from GPT-4o in Telia's dropdown step.

Finally, Figure 6.5 shows a strong linear relationship between step count, token usage, and inference time. This confirms that more complex interactions – often driven by UI design – directly impact scraping cost and latency.

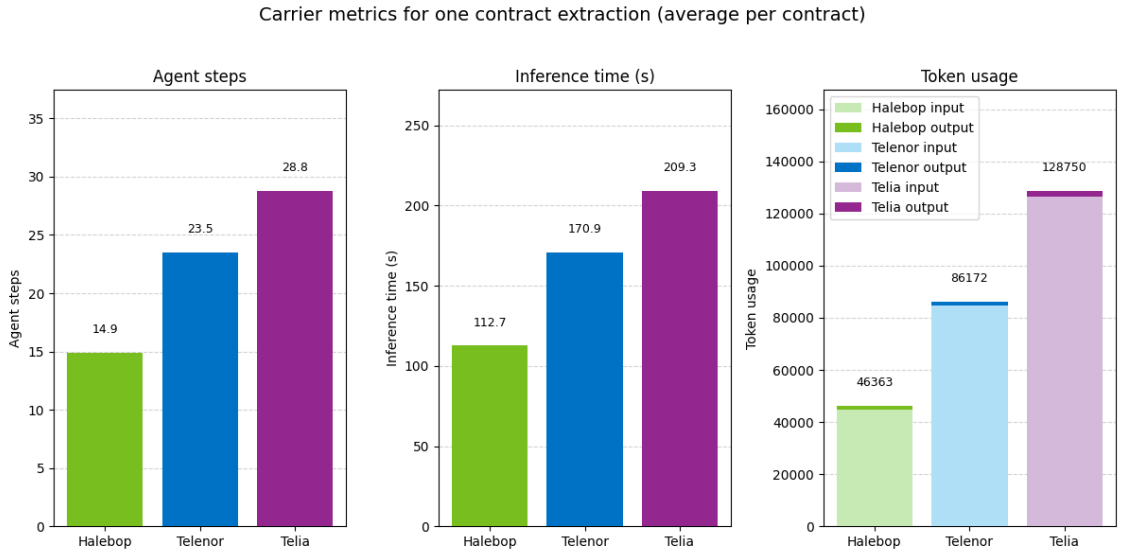


Figure 6.3: Comparison of average agent steps, inference time, and token usage across mobile carriers.

Table 6.9: Average token usage and cost per extraction for each carrier. Halebop shows the lowest cost and token usage overall, with most costs driven by CUA input tokens.

Token / Cost Type (avg)	Halebop	Telenor	Telia
CUA input tokens	44,913	84,582	120,285
CUA output tokens	1,450	1,590	2,299
GPT-4o input tokens	0	0	6,126
GPT-4o output tokens	0	0	40
Total tokens	**46,364**	**86,173**	**128,750**
CUA cost ($)	0.1521	0.2728	0.3884
GPT-4o cost ($)	0.0000	0.0000	0.0157
Total cost ($)	**0.1521**	**0.2728**	**0.4042**

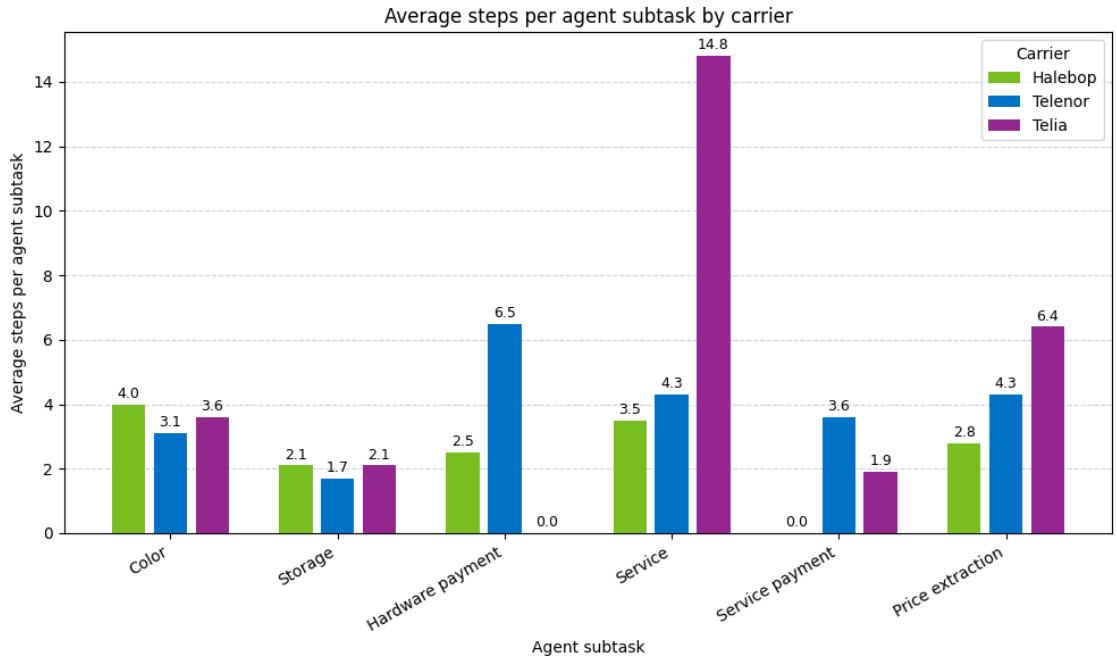


Figure 6.4: Average number of agent steps required to complete each subtask, grouped by mobile carrier. The chart highlights that Telia requires significantly more agent steps during the service plan selection stage.

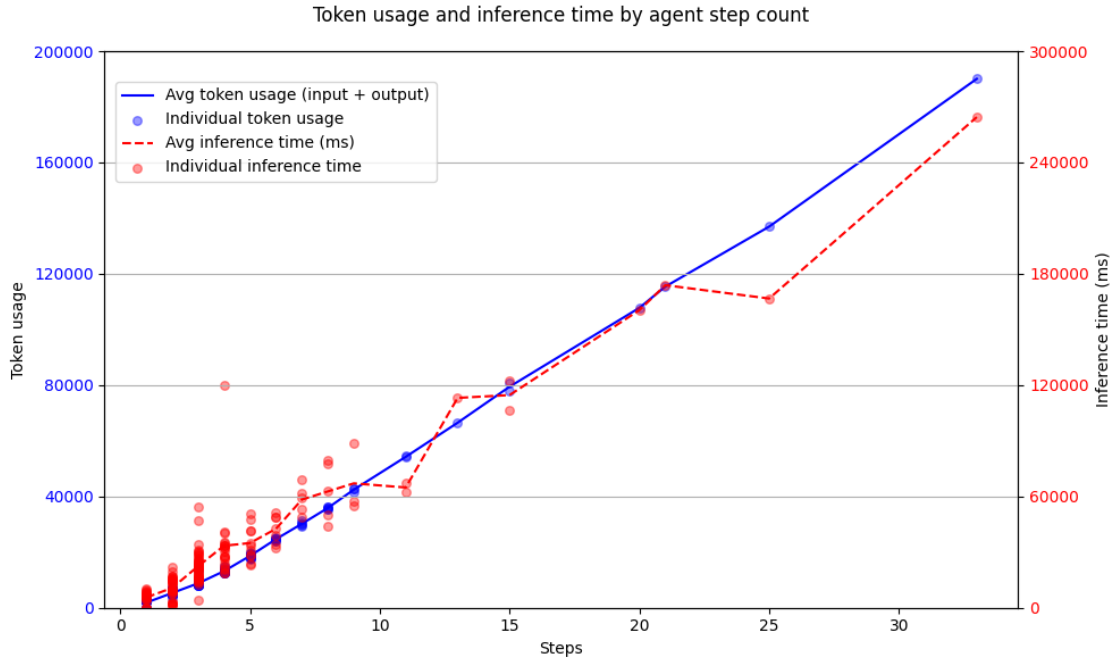


Figure 6.5: The relationship between token usage step count as well as inference time and step count. Both follow a linear trend.

Chapter 7
Conclusion

This thesis set out to explore the capabilities and limitations of AI-assisted web scraping through the development and evaluation of *PriceFinderAgent*, a multi-agent system for extracting mobile contract data from dynamic websites. Our investigation was guided by two main research questions:

RQ1: Is it possible for AI-assisted web scraping techniques to consistently extract accurate data from dynamic and visually complex websites?

RQ2: Is it possible to generalize the AI-assisted web scraping technique across different dynamic websites without relying on site-specific rules?

The results from our benchmark evaluations suggest that the answer to RQ1 is affirmative: our final system achieved 90% precision and 100% recall across three live carrier websites. These findings demonstrate that modern LLM-based agents can consistently extract accurate pricing data even from visually rich and highly interactive interfaces.

In addressing RQ2, we found that the same agentic architecture was able to handle structurally different websites – Telia, Telenor, and Halebop – without using any site-specific scraping logic. While one site-specific tailoring was required due to UI differences (dropdown handling on Telia), the overall system design remained general. This indicates a promising level of generalizability, though further evaluation across a broader range of websites would strengthen this conclusion.

The remainder of this chapter reflects on the limitations of our approach, suggests avenues for future work, and summarizes key insights.

7.1 Limitations

While the system showed strong improvements over baseline scraping methods, several limitations affect the extent to which RQ1 and RQ2 can be fully confirmed. First, the evaluation

was conducted on only three websites with ten test runs each. This small sample introduces uncertainty and limits the generalizability of our results. Although we achieved 100% recall and 90% precision in the final version, broader evaluation is necessary to confirm consistent accuracy (RQ1) on other sites.

We must also consider potential overfitting. Prompt templates and system behaviors were iteratively tuned using the same three websites throughout development. While no site-specific code was written, we did introduce carrier-specific agent configurations – e.g., using a GPT-4o agent for Telia's dropdown menus, but not for Telenor or Halebop. This indicates that some site-aware adaptations were required, which partly challenges the claim of complete generalizability (RQ2).

Another limitation is the high computational cost. A single contract extraction took between 112 and 209 seconds and cost up to $0.40 in LLM usage, making large-scale, frequent scraping costly. These constraints impact the practical feasibility of "consistent" scraping for RQ1 – even if accuracy is high, the cost and speed may prohibit scalable application. Finally, certain UI elements – such as modals and dropdowns – were harder to navigate uniformly, requiring manual fallback logic or adjusted prompts.

7.2 Future Work

To strengthen claims around RQ1 and RQ2, future work should broaden the evaluation to include a larger and more diverse set of mobile carrier websites. This would allow us to better test whether high accuracy is consistently achievable and whether the current multi-agent design can handle new layouts and interaction flows without modification.

We also propose a combined evaluation of both option discovery and price extraction in one benchmark run. This would more realistically reflect an end-to-end usage scenario and reveal how early-stage errors propagate through the pipeline.

Reducing cost is a critical next step for practical deployment. Future versions could implement action caching to lower the number of tokens per run. Testing other computer-using foundation models (e.g., Claude or open-source options) could also yield improvements in speed, cost-efficiency, or generalization.

In addition, we recommend adding support for dynamic fallback behavior. If one agent fails to complete a task (e.g., dropdown selection), the system should automatically retry with an alternative model or strategy. We only implemented this behavior manually for Telia in the final version; generalizing it would improve robustness and adaptability.

Finally, concurrent execution of agent tasks would significantly increase throughput and enable the system to operate at production scale. Supporting such concurrency would also allow for more data to be collected in less time, enabling richer performance insights.

7.3 Final Remarks

Despite its limitations, *PriceFinderAgent* demonstrates the viability of AI-assisted web scraping using LLM-based agents. With 90% precision and 100% recall, we find strong evidence that AI agents can consistently extract accurate data from dynamic websites (RQ1), at least within a defined evaluation scope. The system also handled three visually and structurally

different sites using a unified framework, showing that generalization without hardcoded site-specific logic is possible (RQ2), though further validation is needed.

This work contributes to the growing field of agentic AI for web automation and offers a practical proof of concept. With continued improvements in model capabilities and system design, we believe AI-driven scraping agents like PriceFinderAgent will play a role in making structured data extraction more robust, scalable, and maintenance-free.

References

Ahluwalia, A. and Wani, S. (2024). Leveraging large language models for web scraping.

Anthropic (2025). Claude 3.5 family overview. https://www.anthropic.com/news/claude-3-5-family. Accessed: 2025-04-28.

Anthropic (2025). Computer use (beta). https://docs.anthropic.com/en/docs/agents-and-tools/computer-use. Accessed: 2025-04-16.

Browserbase (2025a). Stagehand. https://github.com/browserbase/stagehand. GitHub repository, accessed 2025-04-28.

Browserbase (2025b). Stagehand, the ai browser automation framework. Accessed: 2025-04-28.

Chalvatzaki, G., Younes, A., Nandha, D., Le, A., Ribeiro, L. F. R., and Gurevych, I. (2023). Learning to reason over scene graphs: A case study of finetuning gpt-2 into a robot language model for grounded task planning.

DeepSeek (2025). Deepseek-v3: Expanding multimodal horizons. https://deepseek.com/research/deepseek-v3. Accessed: 2025-04-28.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. (2023). Mind2web: Towards a generalist agent for the web.

Docs, M. W. (2025). Introduction to frameworks and libraries - learn web development. Accessed: 2025-04-03.

Google DeepMind (2025a). Gemini 1.5 and 2.5 models. https://deepmind.google/technologies/gemini/. Accessed: 2025-04-28.

Google DeepMind (2025b). Project mariner. https://deepmind.google/technologies/project-mariner/. Accessed: 2025-04-16.

Halebop (2025). Apple iPhone 16 – Halebop. Accessed: 2025-05-01.

He, H., Yao, W., Ma, K., Yu, W., Dai, Y., Zhang, H., Lan, Z., and Yu, D. (2024). Webvoyager: Building an end-to-end web agent with large multimodal models.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Kothapalli, M. (2021). The evolution of component-based architecture in front-end development. *The Journal of Scientific and Engineering Research*, 8:261–264.

Microsoft (2025). Installation. `https://playwright.dev/docs/intro`. Accessed: 2025-04-03.

Müller, M. and Žunič, G. (2024). Browser use: Enable ai to control your browser.

OpenAI (2024). Cua evaluation extra information. `https://cdn.openai.com/cua/CUA_eval_extra_information.pdf`. Accessed: 2025-03-26.

OpenAI (2024a). Gpt-4o: Openai's latest model. `https://openai.com/index/gpt-4o`. Accessed: 2025-04-28.

OpenAI (2024b). What is artificial intelligence? `https://openai.com/index/hello-gpt-4o/`. Accessed: 2025-05-08.

OpenAI (2025). Computer-using agent: Introducing a universal interface for ai to interact with the digital world.

Prisjakt (2025). Om prisjakt. `https://www.prisjakt.nu/information/om-prisjakt`. Accessed: 2025-05-14.

Proxyway (2023). Web scraping for beginners: A guide 2025. `https://proxyway.com/guides/what-is-web-scraping`. Accessed: 2025-04-03.

Sager, P. J., Meyer, B., Yan, P., von Wartburg-Kottler, R., Etaiwi, L., Enayati, A., Nobel, G., Abdulkadir, A., Grewe, B. F., and Stadelmann, T. (2025). Ai agents for computer use: A review of instruction-based computer control, gui automation, and operator assistants.

Swedish Quality Index (SKI) (2024). Ski mobil 2024. `https://www.kvalitetsindex.se/wp-content/uploads/2024/10/ski-mobil-2024-a.pdf`. Accessed: 2025-05-14.

Telia (2025). Apple iPhone 16 – Telia. Accessed: 2025-05-01.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

W3Schools (n.d.). Javascript html dom. `https://www.w3schools.com/js/js_htmldom.asp`. Accessed: 2025-04-03.

Whitaker, E. (2023). Scraping websites with client-side rendering. Accessed: 2025-04-03.

xAI (2025). Introducing grok-3. `https://x.ai/blog/grok-3`. Accessed: 2025-04-28.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. (2024). Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments.

Y Combinator (2025). Browser use. `https://www.ycombinator.com/companies/browser-use`. Accessed: 2025-05-02.

Yang, J., Zhang, H., Li, F., Zou, X., Li, C., and Gao, J. (2023). Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). React: Synergizing reasoning and acting in language models.

Yin, S., Fu, C., Zhao, S., Li, K., Sun, X., Xu, T., and Chen, E. (2024). A survey on multimodal large language models. *National Science Review*, 11(12).

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Bisk, Y., Fried, D., Alon, U., et al. (2023). Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

Zhu, Q., Duan, J., Chen, C., Liu, S., Li, X., Feng, G., Lv, X., Cao, H., Chuanfu, X., Zhang, X., Lin, D., and Yang, C. (2024). Sampleattention: Near-lossless acceleration of long context llm inference with adaptive structured sparse attention.

Appendices

Appendix A

Single-Agent Browser Use Implementation

A.1 Agent Prompt

```
Your ultimate task is: You are given a webpage with phone details. Your task is to extract the
     pricing information for the specified phone configuration. The webpage may present data in
     various formats, so focus on accurately identifying the key pricing details, including any
     discounts or special offers.

Fixed Configuration:
 - carrier: str = Telenor
 - product: str = iPhone 16
 - color: str = Teal
 - storage_gb: int = 128
 - data_plan_name: str = Obegransat Plus
 - data_plan_gb: int = None Note: this is the data included without any double surf campaigns or
      similar.
 - plan_payment_months: int = 0
 - hardware_payment_months: int = 36

The rest of the fields related to pricing are for you to fill in.

- plan_increased_data_gb Optional[int]: The increased data plan in gigabytes. Could be double
     data during a promotional period.
- plan_increased_data_months Optional[int]: The number of months the increased data plan is
     applicable.
- plan_monthly_price_special_sek: Optional[int] = The monthly price for the subscription plan
     with during a specific duration, None if no special pricing (eg. pricing of phone
     subscription is not tied to specific months). The special price can be a special offer or a
      campaign price, or it can be a higher price that is bound to end after a certain period of
      months.
- plan_payment_special_months: Optional[int] = The duration of the special pricing, None if no
     speicla pricing. The special pricing can have a certain period, eg. first 4 months, or
     month 1-12.
- plan_monthly_price_sek: int = The regular monthly price for the subscription plan. This price
     is the price that will be payed indefinately.
- hardware_monthly_price_sek: int = The monthly price for the phone hardware, paid until the
     phone is paid off.
- hardware_upfront_cost_sek: int = Often only used when hardware_payment_months is 0 (eg. paid
     upfront). The upfront cost for the phone hardware, can be 0.
- availability: bool = True if the phone is available, False if it is out of stock or the
     configuration is not possible to select.

Here are you specific steps to follow:

1. Accept cookies and close any popups that may appear.
2. On the landing page that you are currently on you should be able to perform the following
     actions:
3. Select the specified color for the phone: Teal
4. Select the specified hardware storage options for the phone: 128 GB
5. Select the specified hardware payment plan duration for the phone: 36 months.
6. Select the specified data plan for the phone: The name of the plan is ObegrÃ¤nsat Plus, which
      includes unlimited data.
7. Select the specified subscription plan duration for the phone: 0 months. The plan binding
     time selection is done with a switch. When the switch is active the plan is binded for 24
     months, and when switch is inactive the plan is binded for 0 months.
8. WHEN ALL OF THE ABOVE STEPS ARE COMPLETED, PLEASE EXTRACT THE PRICE INFORMATION PREVIOUSLY
     STATED
9. The pricing details for hardware and subscription plans are on the LANDING PAGE.

Finally, return the information in the specified format.. If you achieved your ultimate task,
     stop everything and use the done action in the next step to complete the task. If not,
     continue as usual.
```

Figure A.1: User prompt for Browser Use agent in system Version 1.

Appendix B

Multi-Agent Stagehand CUA Implementation

B.1 Option Discovery Agent Prompts

Here are an excerpt of prompts used for CUA in PriceFinderAgent, specifically during the option discovery phase of the system.

```
You are a helpful assistant that can use a web browser.

You are currently on the page \${carrier.url} which is a product page for a phone with a data
    plan.

Your purpose is to discover what different options are available for a phone and phone data plan
    .

Extract the specific options for the phone and data plan you are told to extract.

Please try to keep the option names in its original language. Don't translate them.

If you encounter a cookie popup, always accept the cookies.

Do not ask ANY follow up questions, the user will trust your judgement.
```

Figure B.1: System prompt used for the options discovery agent. The agent receives additional instructions via user prompts, which are shown in subsequent figures.

```
    What different colors are available for the phone? Extract them and tell them to me.
```

Figure B.2: User prompt for extracting color options.

```
What different storage options are available for the phone?

There may be different storage options depending on the color of the phone, so you may have to
     check each color option to find all storage options.
```

Figure B.3: User prompt for extracting storage options. Some options may only appear for certain colors, so the agent was expected to iterate through all color variants to find the full set.

```
What different payment binding times are available for the phone hardware on this page?

The payment binding time is the time you need to commit to the phone hardware.

Normally it's a number of months or direct payment.

Here are a few examples of hardware binding options:
- Direktbetalning
- 12 mÃěnader

You MAY have to look around the page or "go to the next step" to find these options.
```

Figure B.4: User prompt for extracting hardware payment duration options. The agent is instructed to look for all available payment binding times, including direct payment and monthly installment plans. Since these options might not be immediately visible, the agent is also encouraged to explore the interface if necessary.

```
What different data plans/abonnemang/surfmÃďngd are available for the phone?

Normally you can choose between a number of GB per month or unlimited.

Here is an example of data plans:
- 5 GB
- 10 GB
- 200 GB
- Unlimited
- Unlimited with Netflix
```

Figure B.5: User prompt for extracting data plan options. Since data plans are named differently across websites we give the agent some examples.

```
How long do we need to commit to the data plan for this phone? Do we have different options or
    is there no commitment?

Look at the page and try to figure out what the plan binding time/times are.

If there are different options, please list them.
```

Figure B.6: User prompt for extracting data plan payment duration options. Some websites have no binding period at all, others enforce a fixed duration (e.g., 24 months), while some allow the user to choose from multiple durations (e.g., 0åĂŞ36 months).

B.2 Option Selection Agent Prompts

```
You are a helpful assistant that can use a web browser.

Currently, you are on a mobile carrier's website on a product page for a phone.

You are going to help select options for a contract.

Do not ask ANY follow-up questions, the user will trust your judgement.

You are free to navigate as you wish, you do not need to ask for permission.

You cannot log in anywhere, so do not even try.

If you see an option to continue as guest, it is wise to do so because it will reveal more
    options.
```

Figure B.7: System prompt used for the Variant Selection Agent. This prompt sets the agentåĂŹs role and high-level behavior when selecting product and contract options.

```
Find the color option \${JSON.stringify(contractSelection.hardwareColor, null, 2)} and select it
    .

The color option is usually found high up on the page.

You may have to click every color option to find the correct one.

If the color is already selected, stop immediately and consider the selection complete.

If the color is not selected, select it then stop immediately and consider the selection
    complete.
```

Figure B.8: User prompt for selecting a specific phone color.

```
The hardware storage option refers to the storage capacity of the phone.

Select the hardware storage option \${JSON.stringify(contractSelection.hardwareStorage, null, 2)
    }

This selection is usually found high up on the page.

If the option is already selected, stop immediately and consider the selection complete.
```

Figure B.9: User prompt for selecting the hardware storage option.

```
The hardware payment duration option refers to the length of the payment plan for the actual
    phone/hardware (0 means direct payment or "Betala direkt" in swedish).

Select the hardware payment duration option \${JSON.stringify(contractSelection.
    hardwarePaymentDuration, null, 2)} by selecting it and verifying that the option is
    selected.

Once you have selected the hardware payment duration option, stop immediately and consider the
    selection complete.

If the option is already selected and you are certain that it is selected, stop immediately and
    consider the selection complete.
```

Figure B.10: User prompt for selecting the hardware payment duration.

```
Find the service plan option \${JSON.stringify(contractSelection.servicePlanName, null, 2)} and
    select it.

- The service plan name option refers to the name of the data plan sold with the phone.

- To find the service plan option, SOMETIMES (but not always) you have to click a button "VÃd̈lj
    abonnemang" or "FortsÃd̈tt som gÃd̈st" to reveal the service plan options. But sometimes its
    selectable under a section called "Abonnemang" or "VÃd̈lj abonnemang".

- Sometimes you have to expand the service plan section to see all the options.

- Don't bother looking in the product details section of the page. You want to find the service
    plan section section.

- If the service plan is already selected, stop immediately and consider the selection complete.
     Remember that the service plan names can be very similar, for there might be both a "
    ObegrÃd̈nsad surf" and a "ObegrÃd̈nsad surf bas" option. So make sure you select the correct
    one.

- If it is not selected, SELECT IT then STOP IMMEDIATELY and consider the selection COMPLETE.
```

Figure B.11: User prompt for selecting a data service plan. The agent is given rules and hints for how such plans are typically found and identified.

```
You have selected the service plan \${JSON.stringify(contractSelection.servicePlanName, null, 2)
    }.

Now your task is to select the service plan payment duration option \${JSON.stringify(
    contractSelection.servicePlanPaymentDuration, null, 2)} for that service plan.

This is usually done by clicking a toggle button or selecting a dropdown option.

Remember that the service plan payment duration or "binding time" shall not be confused with the
     hardware payment duration. I want you to select the service plan payment duration option,
    not the hardware payment duration option, leave the hardware payment duration option alone.

Once you have selected the service plan payment duration option, stop immediately and consider
    the selection complete.

If the option is already selected, stop immediately and consider the selection complete.
```

Figure B.12: User prompt for selecting the payment duration of a selected service plan. The agent must distinguish it from hardware duration and stop when the correct value is selected.

B.3 Price Extraction Agent Prompts

```
Your purpose is to extract detailed price information about the phone and data plan CURRENTLY
    SELECTED on the page. Do not change any product selections.

Extract the following price information from the page and report it back to the user:
- Minimum TOTAL price: aka "minsta totalpris", "minsta totalkostnad". Can usually be found under
     detailed price information.
- TOTAL upfront price: The total upfront price that has to be paid today. 0 if no upfront
    payment is required.
- TOTAL monthly price: The total monthly price that has to be paid each month. 0 if no monthly
    payment is required.

ALL PRICES CAN BE FOUND ON THE PAGE. DO NOT MAKE ANY CALCULATIONS YOURSELF, ONLY REPORT THE
    PRICES YOU SEE.

Do not ask ANY follow up questions, the user will trust your judgement.
Do not navigate to any other page, find the price information on the current page.
If you see a button to show price details, that could be a good place to start.
```

Figure B.13: System prompt for the Price Extraction Agent. Guides the agent to extract total, upfront, and monthly prices directly from the page without any user interaction or additional navigation.

```
Extract the price information for the phone and data plan CURRENTLY SELECTED on the page.
```

Figure B.14: User prompt for triggering contract price extraction.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2025-06-04

EXAMENSARBETE PriceFinderAgent: An Agentic Approach to Web Scraping
STUDENTER David Pettersson, Ludvig Eskilsson
HANDLEDARE Pierre Nugues (LTH), Richard Toth (Prisjakt)
EXAMINATOR Mathias Haage (LTH)

Vilket abonnemang är billigast? Låt våra AI-agenter hitta svaret åt dig

POPULÄRVETENSKAPLIG SAMMANFATTNING **David Pettersson, Ludvig Eskilsson**

Vi har tagit fram ett AI-system som själv kan utforska och tolka mobiloperatörers hemsidor för att hitta abonnemangspriser. Till skillnad från traditionella lösningar ser modellen på sidan visuellt, inte genom kod. Resultatet är en flexibel och träffsäker metod som klarar komplexa gränssnitt och olika hemsidor.

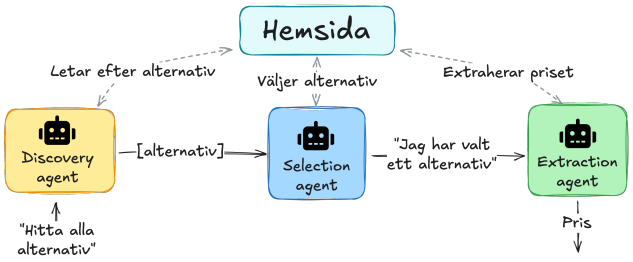
Har du någonsin försökt jämföra mobilabonnemang och märkt hur krångligt det kan vara? Priserna är ofta gömda bakom val av lagringsutrymme, bindningstid och surfmängd. Att klicka sig fram manuellt på flera hemsidor är både tidskrävande och frustrerande.

För att automatisera detta kan man använda så kallade webbscrapers, program som samlar in information från hemsidor. Men det är lättare sagt än gjort eftersom många webbsidor idag är interaktiva och förändras beroende på hur man använder dem. Man kan inte bara ladda ner sidkoden och analysera den, utan måste även klicka på knappar, välja alternativ och navigera på sidan för att pris ska laddas in. Traditionella webbscrapers kan simulera sådana handlingar, men slutar ofta fungera så fort något ändras. De kräver därför mycket underhåll och ett separat program måste skrivas för varje enskild hemsida.

I vårt examensarbete har vi byggt ett AI-baserat system som själv kan klicka runt, läsa innehåll och extrahera priser från mobiloperatörers hemsidor. Ungefär som en människa skulle göra.

Nyckeln till detta är en så kallad multiagentarkitektur. Systemet löser uppgiften i tre steg: först identifieras vad som går att välja (discovery), sedan görs valen (selection), och till sist samlas priset in (extraction). Varje steg hanteras av en specialiserad AI-agent som fått en tydlig uppgift. Till exempel att hitta färgalternativ eller läsa av en prissumma.

Agenterna bygger på en modell från OpenAI som är specialtränad för att navigera webbsidor. Modellen får enbart en skärmdump av sidan som visuellt underlag, tillsammans med en text som beskriver uppgiften. Den tolkar alltså innehållet precis som en människa skulle göra. Genom att se sidan, inte läsa dess kod.



Vi testade systemet på tre olika operatörer: Telia, Telenor och Halebop. Det uppnådde 92% träffsäkerhet, jämfört med 47% för en generisk AI-agent utan domänspecifik anpassning. Det visar att AI kan användas för att bygga mer robusta och flexibla scrapers, även i miljöer som tidigare varit svåra att automatisera.