Advanced Web Scraping Architectures: A Comprehensive Documentation of Paradigms, Tools, and Semantic Integration (2025)

---

1. Introduction to the Modern Data Extraction Landscape

The digital ecosystem of 2025 operates on data. From e-commerce pricing intelligence to academic research and financial modeling, the ability to programmatically harvest structured information from the World Wide Web is a foundational capability. However, the mechanisms by which this data is extracted have undergone a radical transformation. We are currently witnessing a paradigm shift from **syntactic** extraction—relied upon for two decades—to **semantic** extraction driven by Artificial Intelligence.

For years, the industry standard relied on "Classic" web scraping. This deterministic approach utilized HTTP requests to fetch raw HTML and parsing libraries to traverse the Document Object Model (DOM) using rigid selectors (CSS or XPath).[1] While computationally efficient, this model is inherently brittle; a single change in a website's layout or class naming convention can render a scraper obsolete overnight. Furthermore, the modern web has evolved into a dynamic landscape of Single Page Applications (SPAs) heavily reliant on JavaScript, rendering simple HTML fetchers ineffective without complex browser automation.[1]

The emergence of Large Language Models (LLMs) has introduced a new era of resilience and adaptability. By integrating models like GPT-4, Llama-3, and DeepSeek into the scraping pipeline, developers can now build systems that "understand" the content they traverse.[1] These systems do not merely look for a specific tag; they look for the *concept* of a price, a product description, or an author's name, regardless of the underlying markup.

This report serves as an exhaustive update to the technical documentation for web scraping architectures. It explicitly integrates specific tools, comparative statistics, pricing tables, and code examples provided in the core research documentation [1], while synthesizing broader market trends and academic case studies to provide a holistic view of the domain in 2025.

---

2. The Foundation: Classic Web Scraping Architectures

To understand the innovation of AI-driven scraping, one must first master the foundational architecture that still powers a significant portion of the internet's data collection pipelines. The "Classic" or "Traditional" approach is defined by its linear, synchronous nature and its reliance on structural pattern recognition.

2.1 The Request-Response Linear Pipeline

At its core, traditional web scraping involves fetching web pages via HTTP requests, parsing the returned HTML or XML content, and extracting structured data—such as text, links, or tables—for downstream analysis or storage.[1] This process programmatically mimics the actions of a web browser but is distinct in its ability to handle scale, vastly outperforming manual copy-paste operations.[1]

The architecture typically follows a strict linear pipeline:

1. **Request:** The client sends an HTTP GET request to a target URL.
2. **Response:** The server returns the raw HTML document.
3. **Parse:** The scraper parses this text into a navigable tree structure.
4. **Extract:** Specific nodes in the tree are targeted using selectors.
5. **Store:** Data is serialized (CSV/JSON) and saved.

2.1.1 Core Libraries: Requests and BeautifulSoup

In the Python ecosystem, which dominates the scraping landscape [2], two libraries form the bedrock of this architecture:

- **Requests:** This library handles the HTTP transport layer. It is responsible for sending requests and retrieving content. It allows for the customization of headers (e.g., User-Agents) to mimic legitimate browser traffic, which is a rudimentary form of anti-bot evasion.[1]
- **BeautifulSoup (bs4):** This library handles the parsing layer. It converts raw HTML text into a complex tree of Python objects. It provides methods for navigating the parse tree, allowing developers to search for tags, navigate to children or parents, and filter by attributes like class or ID.[1]

This combination is favored for its speed and minimal resource footprint. Because it does not require a Graphical User Interface (GUI) or a JavaScript execution engine (like a browser), it can process thousands of pages per minute on modest hardware.[1]

2.2 Implementation Strategy and Code Analysis

The workflow for developing a traditional scraper involves manual inspection. A developer

must open the target website, use browser Developer Tools (DevTools) to inspect the DOM, and identify unique signatures (selectors) for the data of interest.[1]

The following code example, derived from the primary documentation, illustrates the standard implementation of this pattern. It demonstrates fetching a page, checking for successful transmission, and iterating through HTML elements to extract text.

```
import requests
from bs4 import BeautifulSoup

# Define the target URL
url = "https://example.com"

# Define Headers to mimic a browser (Crucial for avoiding basic blocks)
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"
}

# 1. Fetch the content
resp = requests.get(url, headers=headers)

# 2. Handle Errors
if resp.status_code!= 200:
    raise Exception(f"Failed to fetch {url}: {resp.status_code}")

# 3. Parse the HTML
soup = BeautifulSoup(resp.content, "html.parser") # 'lxml' is faster if installed

# 4. Extraction Logic
# Example: Extracting all <h2> headings
for h2 in soup.find_all("h2"):
    print("Heading:", h2.get_text(strip=True))

# Example: Extracting content from a specific container
content_div = soup.find("div", class_="main-content")
if content_div:
    for p in content_div.find_all("p"):
        print("Paragraph:", p.get_text(strip=True))
```

This code highlights the imperative nature of the approach. The developer must explicitly tell the script *where* to look (div.main-content) and *what* to look for (p tags). If the website administrator decides to rename main-content to article-body, this script will fail silently or raise an error, requiring manual intervention to update the code.

2.3 Limitations of the Classic Model

While efficient for static, stable websites, the classic model faces significant challenges in the modern web environment. The documentation identifies three primary limitations:

1. **Fragility to Layout Changes:** The logic is tightly coupled to the visual presentation layer. Reliance on specific CSS classes or IDs makes the scraper brittle. As noted in the comparative analysis, this method is "fragile to layout changes," and maintenance costs increase linearly with the number of target sites.[1]
2. **Inability to Handle Dynamic Content:** Libraries like requests only retrieve the initial HTML response sent by the server. They do not execute JavaScript. Consequently, for sites built with React, Angular, or Vue.js—where content is loaded asynchronously via API calls after the initial load—the traditional scraper often sees only an empty shell (e.g.,).
3. **Complex Data Handling:** Extracting deeply nested, inconsistent, or semi-structured data (e.g., tables that vary in column count across pages) requires complex conditional logic (if/else blocks) that is difficult to write and harder to maintain.[1]

---

3. The Paradigm Shift: LLM-Integrated and AI-Powered Scraping

To overcome the brittleness of selector-based scraping, the industry has adopted **LLM-Integrated Scraping**. This architecture introduces a semantic reasoning layer, effectively decoupling the extraction logic from the HTML structure. Instead of instructing the scraper to "get the text inside the div with class 'price'," the developer instructs an AI model to "extract the product price from this text."

3.1 Architecture of LLM-Integrated Pipelines

The integration of Large Language Models (LLMs) fundamentally alters the scraping pipeline. It shifts the burden of parsing from the developer (writing selectors) to the AI (interpreting tokens).

3.1.1 The Hybrid Workflow (Fetch + LLM Post-Process)

The most common implementation in 2025 is a hybrid approach that leverages the speed of traditional tools for data retrieval and the intelligence of LLMs for data parsing.

The Workflow:

1. **Scrape/Fetch:** Use requests or a headless browser (like Playwright) to retrieve the page content.
2. **Clean/Prune:** Remove non-essential HTML tags (scripts, styles, navigation bars) to reduce token usage.

3. **Prompt:** Pass the cleaned HTML or text to an LLM (e.g., GPT-4o, Gemini, DeepSeek) with a natural language instruction.
4. **Structure:** The LLM returns a structured object (JSON) based on the prompt.

The documentation provides a practical demonstration of this hybrid architecture using the OpenAI API. This code snippet illustrates how the parsing logic is replaced by a prompt :

```
import requests
from bs4 import BeautifulSoup
from openai import OpenAI # Requires API key

# Initialize the LLM client
client = OpenAI()
url = "https://example.com"

# 1. Fetch and Parse (Traditional Step)
soup = BeautifulSoup(requests.get(url).content, 'html.parser')

# 2. Pre-processing: Extract text and truncate to manage token limits
text = soup.get_text()[:4000]

# 3. LLM Semantic Extraction
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=
)

# 4. Output
print(response.choices.message.content)
```

This architecture is described as "Hybrid" because it retains the linear request pattern but outsources the complexity of extraction. It scales well for complex layouts where defining selectors is difficult, and it is robust against minor UI changes. However, it introduces new constraints: **Latency** (waiting for the LLM to generate tokens) and **Cost** (paying per token).[1]

3.2 Advanced Open-Source LLM Scrapers

To standardize the interaction between crawlers and LLMs, several open-source libraries have emerged. These tools handle the "glue code" required to fetch pages, manage browser contexts, and format data for AI consumption.

3.2.1 Crawl4AI: The Open-Source Standard

Identified in the documentation as the top open-source, LLM-friendly crawler (boasting 56k+ GitHub stars), **Crawl4AI** is specifically architected for RAG (Retrieval-Augmented Generation)

and AI pipelines.[1]

- **Core Philosophy:** It prioritizes the generation of clean Markdown rather than raw HTML. LLMs process Markdown much more efficiently than HTML, consuming fewer tokens to understand the document structure.[3]
- **Key Features:** It utilizes an asynchronous architecture built on Playwright, allowing for concurrent browsing. It supports Docker deployment and includes hooks for custom browser interactions (e.g., scrolling, clicking).
- **Extraction Strategy:** It supports both traditional CSS/XPath extraction and LLM-based extraction strategies.[4]

Practical Code Example (Async Implementation):

```
from crawl4ai import AsyncWebCrawler

async with AsyncWebCrawler() as crawler:
    # The 'arun' method handles the browser lifecycle asynchronously
    result = await crawler.arun("https://example.com")

    # The result object contains 'markdown' specifically optimized for LLMs
    print(result.markdown)
```

Source: Primary documentation [1]

3.2.2 llm-scraper (TypeScript)

For the JavaScript/TypeScript ecosystem, the documentation highlights **llm-scraper**. This library is designed to extract structured data from arbitrary webpages using LLMs while enforcing strict type safety.[1]

- **Mechanism:** It leverages the "Function Calling" (or Tool Use) capabilities of models like GPT-4. By defining a schema using Zod (a schema declaration library), the developer ensures that the LLM's output conforms exactly to the expected JSON structure (e.g., ensuring a price is a number, not a string).[5]
- **Browser Integration:** It uses Playwright under the hood to render pages, making it capable of handling dynamic JavaScript content.[6]

3.2.3 Scrapy-LLM (Python Middleware)

For enterprise-grade scraping projects that require high throughput, **Scrapy** remains the dominant framework. **Scrapy-LLM** is a middleware extension that integrates AI capabilities directly into the Scrapy pipeline.[1]

- **Architecture:** It allows developers to define a Pydantic model (a data validation class). The middleware intercepts the response, sends the content to an LLM, and populates the

Pydantic model with the extracted data.[7]

- **Advantage:** This blends the scalability and concurrency of Scrapy (which can handle millions of requests) with the adaptability of LLMs for the parsing step.

---

4. Comparative Analysis: Tools, Metrics, and Economics

The decision to adopt AI-powered scraping over traditional methods involves a complex trade-off between engineering effort, operational cost, and data quality. The updated documentation provides specific tables and metrics to guide this decision.

4.1 Tool Ecosystem Comparison (2025 Matrix)

The market has bifurcated into low-code tools for business users and high-code libraries for engineers. The following table, reconstructed from the primary documentation, summarizes the key players [1]:

Tool	Type	Pricing Model	Key Features	Target Audience
Thunderbit	Chrome Extension	Freemium (Free 6 pages/mo)	2-click AI scrape to Excel; GPT/Claude/DeepSeek support; Subpage crawling; Templates for Amazon/Zillow.	Non-technical (Marketing/Sales)
Browse AI	No-code Platform	Paid ($19+/mo)	Visual robot training; Handles JS/Dynamic content; Native Zapier integration.	Business Analysts

ScrapingBee	API	Paid ($49+/mo)	Headless browser management; Proxy rotation; Anti-bot evasion; 1000 free credits.	Developers (SaaS)
ZenRows	API	Paid (Custom)	Universal scraper API; Focus on anti-detection (98% success rate); Scalable extraction.	Enterprise/High-Volume
Crawl4AI	Open-Source	Free (Self-Hosted)	LLM-ready Markdown output; Docker support; Playwright integration; Hooks for customization.	Engineers/Data Scientists

Deep Dive on Key Tools:

- **Thunderbit:** This tool represents the democratization of scraping. It features an "AI Suggest Columns" capability that analyzes a page visually and proposes data fields (e.g., "Price," "Rating") without user input.[8] It employs a "Waterfall" method for reliability, automatically retrying failed scrapes with different strategies to achieve a claimed 99% success rate.[8]
- **ZenRows:** This tool differentiates itself through "Anti-Detection." It focuses on bypassing sophisticated bot protections (like Cloudflare or Akamai) by mimicking human behavior and utilizing premium residential proxies.[9]

4.2 Architectural Comparison: Traditional vs. LLM-Integrated

The following comparative analysis highlights the operational differences between the two dominant architectures.[1]

Metric / Aspect	Traditional (BS4 + Requests)	LLM-Integrated (Crawl4AI / llm-scraper)
Speed	**Fast:** Limited only by network latency and CPU parsing speed.	**Slower:** Incurs overhead from headless browser rendering and LLM inference latency.
Accuracy (Dynamic Sites)	**Poor:** Often fails on JS-heavy sites (60% accuracy).	**Excellent:** Renders JS and understands context (90%+ accuracy).
Resilience	**Fragile:** Breaks if HTML class names or layout changes.	**Robust:** Semantic understanding adapts to layout shifts.
Cost	**Low:** Minimal compute; effectively free.	**Higher:** API fees (~$0.01/request) and higher compute for rendering.
Complexity	**High Setup:** Requires manual inspection and selector coding.	**Low Setup:** Prompt/Schema-based; less code to write.
Scalability	**High:** Can scrape millions of pages efficiently.	**Moderate:** Constrained by API rate limits and token costs.

Insight: The "Cost" metric in LLM scraping is nuanced. While the *marginal* cost per page is higher due to API tokens, the *Total Cost of Ownership (TCO)* may be lower for LLM scraping because it drastically reduces the engineering hours required to maintain and fix broken selectors. As noted in the snippets, LLM scrapers can reduce maintenance effort by up to 70%.[10]

---

5. Case Study: Durghotona GPT and Hybrid Frameworks

To illustrate the practical application of these architectures, we examine **Durghotona GPT**, a framework documented in recent academic literature for automating road accident data collection.[11]

5.1 Problem and Solution

- **Context:** Collecting road accident data in Bangladesh for safety analysis is traditionally manual, relying on reading newspapers and logging details. This process is slow, error-prone, and lacks coverage.
- **Objective:** Automate the extraction of structured accident data (27 variables including location, vehicle type, casualties) from national dailies like *Prothom Alo* and *Dhaka Tribune*.

5.2 Architectural Implementation

The researchers adopted a **Hybrid Architecture** that perfectly aligns with the best practices identified in this report:

1. **Data Collection (Scraping Layer):** They utilized web scraping techniques to harvest raw news articles. This corresponds to the "Fetch" phase, ensuring a continuous flow of up-to-date unstructured text.
2. **Data Processing (Semantic Layer):** The raw text was processed using LLMs. The study evaluated multiple models:
    - **GPT-4:** The proprietary state-of-the-art model.
    - **Llama-3:** An open-source model.
    - **GPT-3.5:** A cost-effective legacy model.

5.3 Key Findings and Performance Statistics

The study provided critical benchmarks comparing open-source and proprietary models for extraction tasks:

- **Llama-3 Performance:** The open-source Llama-3 model achieved an accuracy of **89%**, performing comparably to the much more expensive GPT-4.[13]
- **Implication:** This validates the viability of local, open-source LLMs for high-volume scraping tasks. It suggests that cost need not be a barrier to entry; organizations can self-host models like Llama-3 to achieve near-SOTA performance without incurring per-token API fees.

The success of Durghotona GPT demonstrates that the hybrid approach—combining traditional crawlers for retrieval with LLMs for understanding—is not just a theoretical concept but a proven methodology for solving complex, real-world data extraction problems.

---

6. Cost Optimization and Engineering Strategies

One of the primary concerns with LLM-based scraping is the cost of tokens. A naive implementation that feeds raw HTML to GPT-4 can be prohibitively expensive. The documentation and research snippets provide several strategies for optimization.

6.1 The "Pruning" Strategy

Raw HTML contains significant noise: <script> tags, huge Base64 images, inline CSS styles, and navigation boilerplates. This content consumes tokens but provides no semantic value for extraction.

Technique: Use a PruningContentFilter (as seen in Crawl4AI) or a simple BeautifulSoup script to strip these tags before the LLM step.

- **Impact:** This can reduce token usage by **70-90%**.[15]
- **Format Conversion:** Converting HTML to Markdown is another powerful optimization. Markdown is denser and more token-efficient. A 270k token HTML page can be reduced to ~11k tokens in Markdown, a 96% reduction that directly correlates to cost savings and faster processing.[16]

6.2 The DeepSeek Factor

The emergence of the **DeepSeek** model family has disrupted the economics of AI scraping. DeepSeek V3 offers performance comparable to top-tier models at a fraction of the cost.

- **Economy Mode:** The documentation notes the use of DeepSeek's economy features and context caching. By caching the system prompt (the instructions on *how* to extract), users only pay for the changing context (the page content).[17]
- **Thinking Mode:** For complex extraction logic (e.g., "Calculate the total price including tax based on the table rules"), DeepSeek's "Thinking Mode" (Chain of Thought) improves accuracy by allowing the model to reason before outputting the final JSON.[17]

6.3 Local LLM Inference

For maximum cost efficiency, running models locally is the ultimate solution. Tools like **Ollama** allow developers to run Llama-3 locally on their own hardware.

Code Example: Local Llama-3 Scraper:
The following script demonstrates using Ollama to process scraped content without any cloud API costs 18:

```
import requests
import json

# Define the local Ollama API endpoint
```

```
OLLAMA_URL = "http://localhost:11434/api/generate"

def process_with_local_llm(text_content):
    prompt = f"Extract product details as JSON: {text_content}"

    payload = {
        "model": "llama3",
        "prompt": prompt,
        "stream": False,
        "format": "json"
    }

    response = requests.post(OLLAMA_URL, json=payload)
    return response.json()['response']

# Usage
clean_text = "Product: Gaming Mouse. Price: $49.99"
data = process_with_local_llm(clean_text)
print(data)
```

This approach shifts the cost from Opex (API fees) to Capex (hardware/GPU), which is preferable for high-volume, continuous scraping operations.

---

7. Advanced Architectures: The Shift to Agentic Systems

As we move deeper into 2025, the frontier of scraping is shifting from **Extraction** (parsing data) to **Navigation** (browsing the web). This gives rise to **Agentic Scrapers**.

7.1 The Agentic Feedback Loop

Traditional scrapers follow a pre-defined script. Agentic scrapers use a feedback loop to navigate autonomously:

1. **Observe:** The agent views the page (via DOM or Screenshot).
2. **Reason:** The agent decides the next action (e.g., "Click the 'Next' button," "Close the popup").
3. **Act:** The agent executes the action.
4. **Verify:** The agent checks if the data loaded.

This allows scrapers to handle complex scenarios like:

- **Auto-Pagination:** Detecting different pagination styles (infinite scroll vs. numbered

buttons) without custom code.[1]
- **Authentication:** Navigating login screens and handling multi-factor authentication flows.
- **Self-Healing:** If a selector changes, the agent "looks" for the new element based on visual cues rather than crashing.

7.2 The "Waterfall" Reliability Pattern

Thunderbit employs a "Waterfall" architecture to ensure reliability. This pattern involves attempting extraction with the cheapest/fastest method first and escalating only upon failure:

1. **Tier 1:** Standard Selector Extraction (Fast, Free).
2. **Tier 2:** Heuristic/Pattern Extraction (Fast).
3. **Tier 3:** AI/LLM Extraction (Slower, Costly).

This ensures that the expensive AI resources are only used when necessary, optimizing the cost-performance ratio.[8]

---

8. Conclusion and Strategic Recommendations

The web scraping landscape of 2025 is defined by diversity. The "one-size-fits-all" approach of the past has been replaced by a spectrum of tools tailored to specific needs.

Strategic Recommendations:

1. **For Enterprise Production:** Adopt a **Hybrid Architecture**. Use Crawl4AI or Scrapy to handle the crawling and high-volume retrieval. Use local LLMs (Llama-3) or cost-optimized APIs (DeepSeek) to parse unstructured or dynamic data. This balances the robustness of AI with the economic viability of traditional code.[1]
2. **For Quick Data Needs:** Leverage no-code tools like **Thunderbit** or **Browse AI**. The ability to use "AI Suggest Columns" democratizes access to data, allowing non-technical teams to bypass the engineering bottleneck.[19]
3. **For Complex/Hostile Environments:** Utilize specialized APIs like **ZenRows** or **ScrapingBee**. The cost of these tools is justified by their ability to handle proxy rotation, headless browsing, and anti-bot evasion, which are non-trivial to build in-house.[9]

In summary, while the *mechanics* of HTTP requests remain unchanged, the *intelligence* applied to the response has evolved. The future of web scraping is not just about fetching bytes; it is about semantic understanding and autonomous navigation, enabled by the integration of Large Language Models.

---

9. Appendix: Detailed Code Implementation

9.1 Robust Hybrid Scraper with Retry Logic

The following Python code synthesizes the patterns discussed in this report. It implements a robust scraper that attempts traditional extraction first and falls back to an LLM if the data is missing or structured incorrectly.

```
import requests
from bs4 import BeautifulSoup
from openai import OpenAI
import json

# Configuration
API_KEY = "your_api_key"
client = OpenAI(api_key=API_KEY)

def intelligent_scrape(url):
    """
    Hybrid scraper: Tries BS4 first, falls back to LLM.
    """
    try:
        # 1. Fetch Content
        headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)"}
        response = requests.get(url, headers=headers, timeout=10)
        response.raise_for_status()

        # 2. Parse HTML
        soup = BeautifulSoup(response.content, 'html.parser')

        # 3. Attempt Classic Extraction (Fast & Free)
        # This part requires prior knowledge of the site structure
        title = soup.find('h1', class_='product-title')
        price = soup.find('span', class_='price')

        if title and price:
            print("Classic Extraction Success")
            return {
                "title": title.get_text(strip=True),
                "price": price.get_text(strip=True),
                "method": "classic"
            }
```

```
        # 4. Fallback to LLM (Robust but Costly)
        print("Classic failed. Engaging LLM...")

        # Pruning: Remove noise to save tokens
        for tag in soup(['script', 'style', 'nav', 'footer']):
            tag.decompose()

        # Truncate to fit context window
        clean_text = soup.get_text()[:3000]

        completion = client.chat.completions.create(
            model="gpt-4o-mini",
            messages=,
            response_format={"type": "json_object"}
        )

        data = json.loads(completion.choices.message.content)
        data['method'] = "ai_fallback"
        return data

    except Exception as e:
        return {"error": str(e)}

# Execution Stub
if __name__ == "__main__":
    result = intelligent_scrape("https://example.com/product/123")
    print(json.dumps(result, indent=2))
```

Works cited

1. web scrapping (1).pdf
2. Web Scraping Statistics & Trends You Need to Know in 2025 - Scrapingdog, accessed December 3, 2025, https://www.scrapingdog.com/blog/web-scraping-statistics-and-trends/
3. Home - Crawl4AI Documentation (v0.7.x), accessed December 3, 2025, https://docs.crawl4ai.com/
4. The 6 best Firecrawl alternatives in 2025 - Roundproxies, accessed December 3, 2025, https://roundproxies.com/blog/best-firecrawl-alternatives/
5. AI Dev Tips #12: AI LLM Website Scraper review | by Chris St. John - Medium, accessed December 3, 2025, https://medium.com/ai-dev-tips/ai-dev-tips-12-ai-llm-website-scraper-review-33125515aa9c
6. How to Use llm-scraper for AI-Powered Web Scraping - Bright Data, accessed

December 3, 2025, https://brightdata.com/blog/ai/web-scraping-with-llm-scraper
7. Scrapy-LLM - Fully automated AI based web scraping. - GitHub, accessed December 3, 2025, https://github.com/Blacksuan19/scrapy-llm
8. Thunderbit Uncovered: An AI Scraper Deep Dive vs. ScraperAPI - Skywork.ai, accessed December 3, 2025, https://skywork.ai/skypage/en/Thunderbit-Uncovered-An-AI-Scraper-Deep-Dive-vs.-ScraperAPI/1972899408514838528
9. Best API-Based Web Scrapers (2025): Complete Developer Guide - Aloa, accessed December 3, 2025, https://aloa.co/ai/comparisons/ai-scraper-comparison/best-api-based-scrapers
10. LLM Web Scraping: How AI Models are Replacing Traditional Scrapers in 2025, accessed December 3, 2025, https://scrapegraphai.com/blog/llm-web-scraping
11. Durghotona GPT: A Web Scraping and Large Language Model Based Framework to Generate Road Accident Dataset Automatically in Bangladesh | Request PDF - ResearchGate, accessed December 3, 2025, https://www.researchgate.net/publication/392564737_Durghotona_GPT_A_Web_Scraping_and_Large_Language_Model_Based_Framework_to_Generate_Road_Accident_Dataset_Automatically_in_Bangladesh
12. Durghotona GPT: A Web Scraping and Large Language Model Based Framework to Generate Road Accident Dataset Automatically in Bangladesh - ResearchGate, accessed December 3, 2025, https://www.researchgate.net/publication/391329613_Durghotona_GPT_A_Web_Scraping_and_Large_Language_Model_Based_Framework_to_Generate_Road_Accident_Dataset_Automatically_in_Bangladesh
13. Durghotona GPT: A Web Scraping and Large Language Model Based Framework to Generate Road Accident Dataset Automatically in Bangl - arXiv, accessed December 3, 2025, https://arxiv.org/pdf/2504.21025
14. [2504.21025] Durghotona GPT: A Web Scraping and Large Language Model Based Framework to Generate Road Accident Dataset Automatically in Bangladesh - arXiv, accessed December 3, 2025, https://arxiv.org/abs/2504.21025
15. Web Scraping for $2/Day: Build a Cheap, Powerful Bot with DeepSeek V3 + Python | by Kevin Meneses González | Data Engineer Things, accessed December 3, 2025, https://blog.dataengineerthings.org/web-scraping-for-2-day-build-a-cheap-powerful-bot-with-deepseek-v3-python-546572c97617
16. Use LLaMA 3 and Python to extract structured data from websites like Amazon, leveraging LLM-powered parsing for resilient, AI-driven web scraping. - GitHub, accessed December 3, 2025, https://github.com/luminati-io/llama-3-web-scraping
17. Your First API Call | DeepSeek API Docs, accessed December 3, 2025, https://api-docs.deepseek.com/
18. Web Scraping with LLaMA 3: Turn Any Website into Structured JSON (2025 Guide), accessed December 3, 2025, https://brightdata.com/blog/web-data/web-scraping-with-llama-3
19. Best 15 Web Page Scrapers You Should Know About in 2025 - Thunderbit,

accessed December 3, 2025, https://thunderbit.com/blog/best-web-page-scraper
20. How can I use Deepseek for automated web scraping?, accessed December 3, 2025, https://webscraping.ai/faq/scraping-with-deepseek/how-can-i-use-deepseek-for-automated-web-scraping