# Model-Callable Web Crawler for Recursive Content Extraction and Mapping

Chinnasamy

# Table of Content

# 1. Introduction

## 1.1 Problem Statement

The rapid evolution of corporate web platforms, many of which utilize complex JavaScript frameworks, presents significant challenges in extracting structured and actionable information. Manual web content extraction is time-intensive and inconsistent, rendering traditional scraping approaches ineffective, especially for dynamic sites.

## 1.2 Objective of the Project

This project seeks to develop a robust, fully automated platform for recursively crawling corporate websites, extracting content from both static and dynamically rendered pages, and producing concise, AI-driven summaries. The solution should operate on demand, accept user-specified entry points, and output the data in a standardized, machine-readable format amenable to downstream processing.

## 1.3 Business Need for Recursive Web Scraping

Effective business intelligence, compliance monitoring, and branding oversight depend on timely, complete knowledge of available web content. Recursive web scraping enables organizations to:

- Continuously monitor entire sites for content and compliance changes.
- Automate maintenance of internal knowledge bases or training corpora for AI.
- Accelerate discovery of market intelligence and competitor strategy.

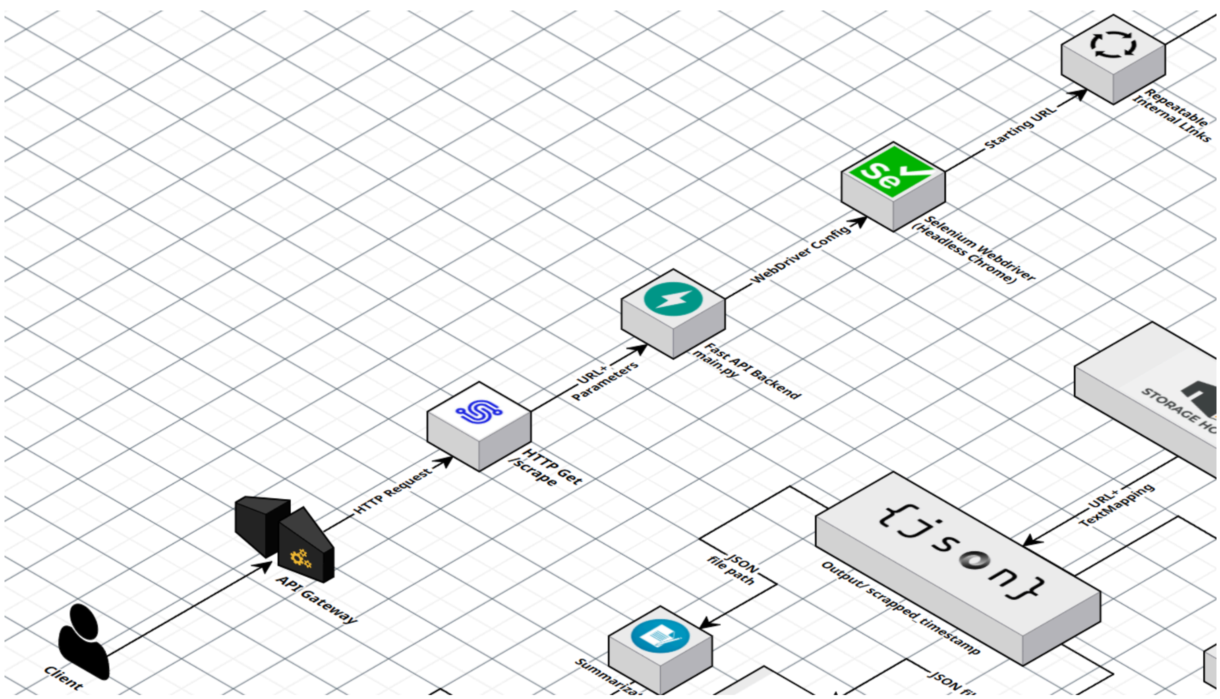
## 1.4 Overview of Model-Callable Programs (MCPs)

Model-Callable Programs (MCPs) encapsulate core business logic as programmatic APIs, making them accessible to both human operators and intelligent agents. In this solution, the MCP acts as a middleware service, exposing composable endpoints for web scraping, content summarization, and topic classification—thus bridging the gap between low-level data operations and advanced, model-driven workflows.

## 2. System Architecture and Design

### 2.1 High-Level Architecture Diagram

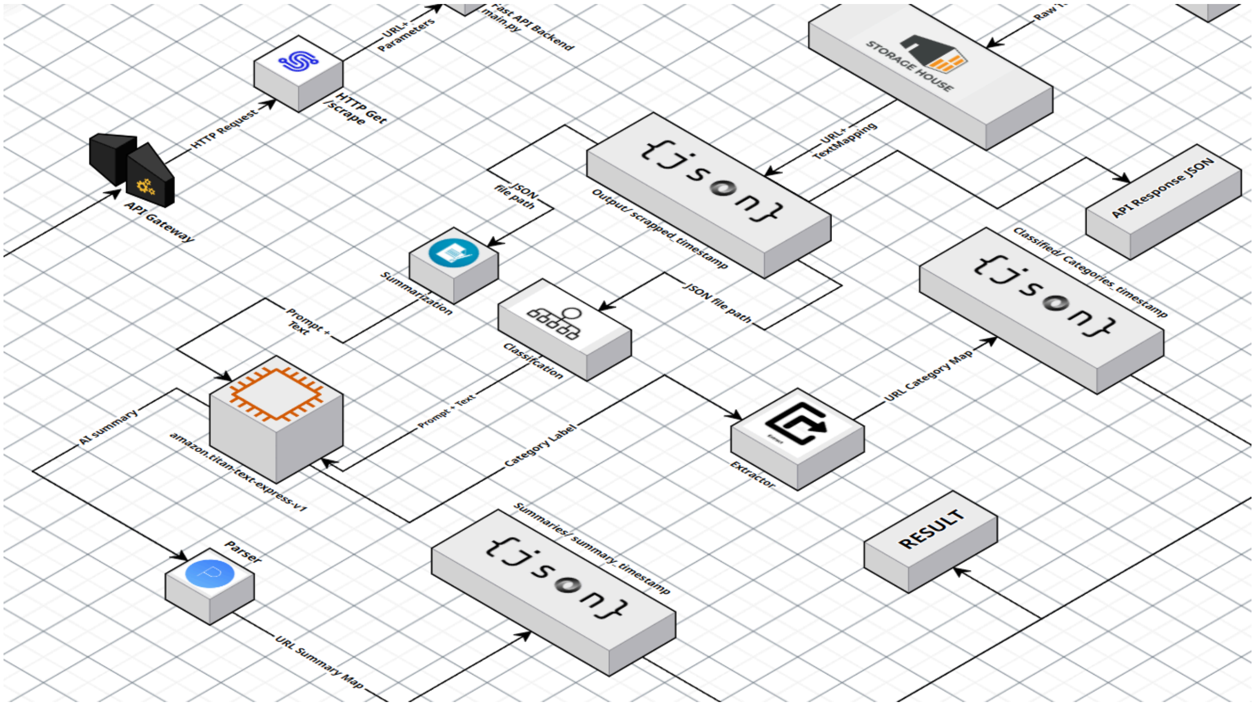
- **Modular & Scalable** – Independent components allow easy upgrades and scaling.
- **Auditable** – Timestamped JSON at each stage supports debugging and compliance.
- **AI-Enhanced** – Amazon Bedrock LLM delivers advanced summarization and classification.
- **Interoperable** – Standardized JSON integrates with analytics and business systems.
- **Resilient** – Decoupled design aids recovery, maintenance, and adaptability.




*Client Request to Internal Link Extraction via API Gateway and FastAPI Backend*

**MCP Scrapping Summary Flow**

- **Client Request** – User sends an HTTP GET to /scrape with the target URL.
- **API Gateway** – Authenticates, validates, and forwards the request.
- **FastAPI Backend** – Parses parameters (URL, depth) and triggers scraping.
- **Selenium WebDriver** – Launches headless browser, fully renders pages (incl. JavaScript).
- **Recursive Crawl** – Follows internal links up to the set depth, extracting cleaned text.




*Modular Pipeline for JSON Storage, AI Summarization, Classification, and Data Aggregation in Automated Web Content Processing*

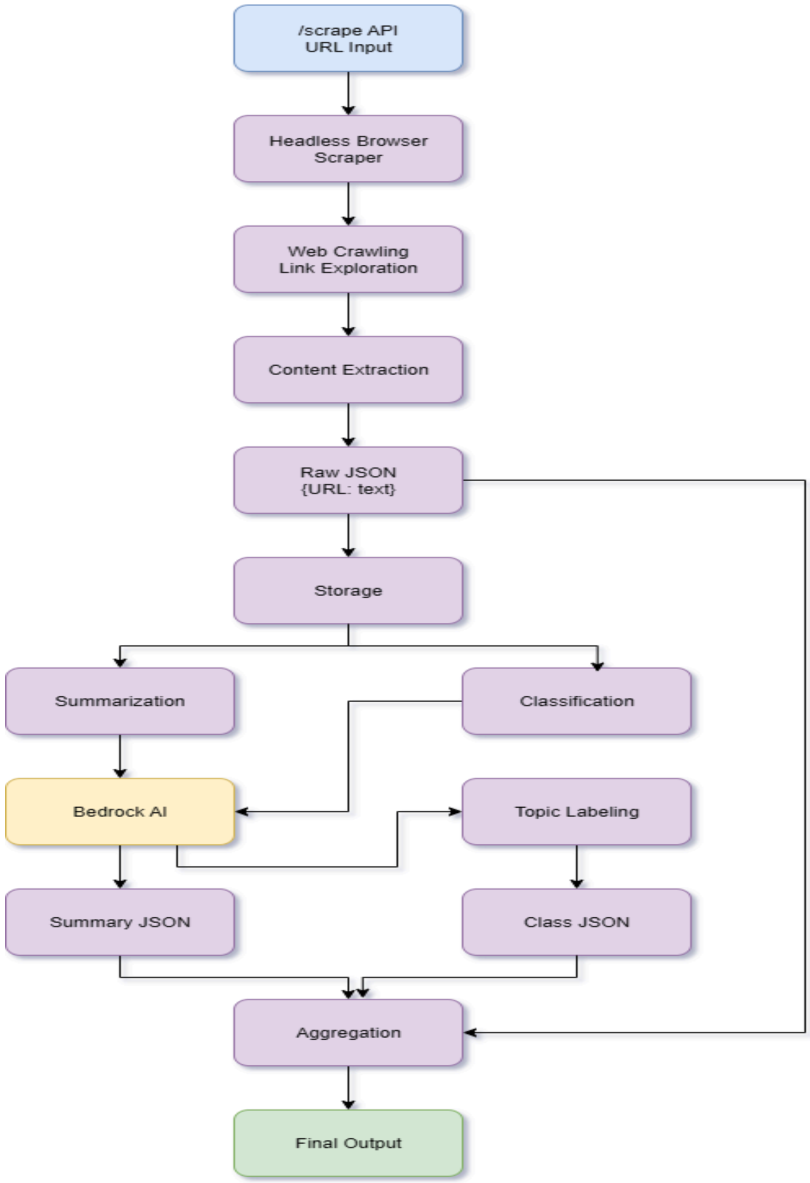
**MCP Post-Scraping Processing Workflow**

- **Central Storage** – Raw content and intermediate artifacts stored for persistence and sharing.
- **JSON Generation** – Crawler output saved as structured URL→content mappings.
- **Summarization** – Amazon Bedrock LLM generates concise summaries per page.
- **Classification** – Pages labeled with semantic categories (e.g., “About Us,” “Products”).
- **Parsing & Extraction** – Validate, normalize, and retrieve targeted information (entities, stats).
- **Aggregation** – Merge text, summaries, categories, and extracted data into a unified JSON.
- **Final Delivery** – Persist enriched JSON for export, reporting, or downstream AI use.

## 2.2 Components and Responsibilities

- 1. API Server (main.py)
- Handles incoming requests, triggers scraping, and coordinates summarization and classification workflows.
- 2. Scraper Module (scraper.py)
- Uses Selenium to crawl internal links and extract cleaned text, outputting a JSON map of URLs to raw content.
- 3. Summarization Module (summarize.py)
- Invokes LLMs (e.g., Amazon Bedrock) to generate concise summaries from raw page content.
- 4. Classification Module (classify.py)
- Assigns categories to each page based on content, enhancing interpretability and structure.
- 5. Storage Management (/output, /summaries, /classified)
- Organizes all intermediate and final JSON artifacts with versioned filenames for traceability.
- 6. Utility Functions (utils.py)
- Supports parsing, logging, JSON handling, and shared helpers across modules.

## 2.3 Flow of Data through the System




*End-to-End Workflow of the MCP System*

- **Scraping Request Reception:** The system receives a homepage URL via the /scrape endpoint, initiating the pipeline.
- **Recursive Crawling & Content Extraction:** A headless browser explores internal links and extracts fully rendered, cleaned text from each page.
- **Raw Text Persistence:** Extracted content is stored as a JSON map of URLs to raw text for downstream processing.
- **AI-Based Summarization:** The summarizer invokes Amazon Bedrock to generate concise summaries per URL, output as a JSON mapping.
- **Topic Classification:** Each page is assigned a relevant thematic label based on its content and structure.
- **Aggregation & Delivery:** Raw text, summaries, and classifications are merged into a unified JSON artifact, ready for API response or export.

### 2.4 Interaction Between MCP and LLM Agent

The MCP server exposes endpoints for scraping and summarization, callable by LLM agents over HTTP. This enables an agent-initiated pipeline: it can instruct the MCP to scrape a site, summarizes the output, and use the resulting data for higher-level reasoning.

## 3. Tools and Technologies Used

### 3.1 Python Environment & Package Setup

- Python 3.10+ as the base runtime
- Virtual environment handled by venv
- Required packages: selenium, fastapi, uvicorn, bs4, pydantic, boto3 (for Bedrock)

command to install the packages: ***pip install fastapi pydantic selenium beautifulsoup4***

### 3.2 FastAPI for Server Implementation

- Supports high-performance async processing for concurrent I/O-bound tasks.
- Enforces type-safe data validation using Pydantic models.
- Auto-generates interactive API documentation via Swagger and ReDoc.
- Provides built-in support for OAuth2, JWT, and API key authentication.
- Enables modular, scalable deployment with ASGI servers like Uvicorn.

FastAPI underpins the server layer, offering asynchronous request handling and automatic endpoint documentation:

```
# FastAPI for building the web API
from fastapi import FastAPI, Query
app = FastAPI() # Initialize FastAPI app
```

### 3.3 Selenium for Headless Browser Automation

Selenium operates Chrome in headless mode, supporting complete rendering of modern, JavaScript-intensive web pages

```
# Initialize a headless Chrome WebDriver
def init_driver():
    options = Options()
    options.headless = True  # Run browser in headless mode (no UI)
    options.add_argument("--disable-gpu") # Disable GPU hardware
acceleration (for compatibility)
    options.add_argument("--no-sandbox") # Bypass OS security model
(needed for some environments)
    options.add_argument("--disable-dev-shm-usage") # Avoid /dev/shm
usage (useful in Docker/limited environments)
    return webdriver.Chrome(options=options)
```

### 3.4 Amazon Bedrock for Summarization

- Used for calling generative LLMs (Titan)
- Invoked via boto3's Bedrock Runtime client
- Handles large context summarization with configurable prompt

## 4. MCP Server Implementation

### 4.1 Role and Purpose of the MCP Server

- The MCP server governs the entire pipeline for scraping, summarization, and classification.
- It presents a unified, extensible interface for all processing tasks.
- The design abstracts browser automation and model invocation complexity.

- It enables secure, maintainable, and modular operations.
- Agents and users interact through a clear, centralized API gateway.

## 4.2 Endpoint Design (/scrape)

The central scraping endpoint follows RESTful conventions

```python
# API endpoint to start scraping
@app.get("/scrape")
def scrape(url: HttpUrl, limit: int = Query(25, le=100)):
    # Start crawling from the given URL
    scraped_data = crawl_site(str(url), limit=limit)

    # Convert all keys to string (for JSON serialization)
    stringified_data = {str(k): v for k, v in scraped_data.items()}

    # Save scraped data to output file
    os.makedirs("output", exist_ok=True)
    output_path = f"output/scraped_{int(time.time())}.json"
    with open(output_path, "w", encoding="utf-8") as f:
        json.dump(stringified_data, f, indent=2, ensure_ascii=False)

    # Return response with status and output file info
    return {
        "status": "success",
        "total_pages": len(stringified_data),
        "output_file": output_path
    }
```

*FastAPI /scrape Endpoint Using Query Parameters for URL and Limit*

**Response body**


```json
{
  "status": "success",
  "total_pages": 25,
  "output_file": "output/scraped_1754565316.json"
}
```


*JSON Response from the /scrape Endpoint Indicating Success and Output File Path*

### 4.3 Recursive Crawling with Selenium

The recursive crawler implements a breadth-first or depth-limited exploration strategy, systematically extracting page content and queuing internal links

```
# Crawl the website starting from start_url, up to a page limit
def crawl_site(start_url, limit=25):
    driver = init_driver()  # Start the browser
    to_visit = [start_url]  # Queue of URLs to visit
    scraped = {}            # Dictionary to store scraped data

    while to_visit and len(scraped) < limit:
        url = to_visit.pop(0)
        if url in visited_urls:
            continue  # Skip already visited URLs
        visited_urls.add(url)

        content, links = scrape_page(driver, url)  # Scrape the page
        scraped[url] = content
        # Add new links to the queue, excluding already visited
        to_visit.extend(links - visited_urls)

    driver.quit()  # Close the browser
    return scraped
```

*Recursive Website Crawling Logic with URL Deduplication anknkd Page Limit*

### 4.4 JSON Output Structure

- Produces a standardized JSON output for easy integration, processing, and traceability.

## 5 Summarization Pipeline

### 5.1 Extracting and Loading JSON Content

The ingestion of raw scraping output from persistent storage into the processing pipeline for subsequent summarization and analysis.

Source of Data

- The scraper outputs JSON files located under the /output directory.
- Files follow a timestamped naming convention, e.g., scraped_YYYYMMDD_HHMMSS.json.
- Each JSON file contains a mapping of URLs to their extracted textual content.

```
# --- Load scraped content ---
with open(INPUT_FILE, "r", encoding="utf-8") as f:
    scraped_data = json.load(f)
```

*Code Snippet for Loading JSON Data from a File Using UTF-8 Encoding*

Kasadara

### 5.2 Summarizing Each Page with Bedrock

The summarizer iterates over all extracted documents, invoking Bedrock for each:

```
prompt = f"Summarize the following webpage content:\n\n{text[:4000]}"
body = json.dumps({
    "inputText": prompt
})

response = bedrock.invoke_model(
    body=body,
    modelId=MODEL_ID,
    accept="application/json",
    contentType="application/json"
)

response_body = json.loads(response["body"].read())
return response_body.get("results", [{}])[0].get("outputText", "").strip()
```

*Programmatic Workflow for Invoking an LLM Model via AWS Bedrock API with JSON Payload*

- Summaries are saved to /summaries.

### 5.3 Handling Model Invocation and Errors

- Model API calls are wrapped in exception handling to prevent pipeline failure due to service or network errors.
- Transient errors invoke retry logic with exponential backoff to improve success without overwhelming services.
- Input data is validated and sanitized before calls to avoid malformed or excessive prompt errors.
- Detailed error logging captures contextual information to support efficient troubleshooting and monitoring.
- The pipeline ensures graceful degradation by continuing processing on failure and supports checkpointing for recovery.

### 5.4 Output and Storage of Summaries

Final summaries are mapped by URL and stored in consolidated JSON files:

## 6 Topic Classification System

### 6.1 Purpose of Classification

` The topic classification system semantically organizes extracted web content by assigning relevant categories to each crawled page. This enhances data usability and interpretability by enabling:

- Efficient navigation and filtering for end-users and downstream applications.
- Deeper contextual understanding to support business intelligence and thematic analysis.
- Structured data alignment for integration with search engines, knowledge graphs, and AI pipelines.

### 6.2 Mapping URLs to Categories

Classification assigns each URL a relevant category representing the page's theme or function. This is achieved through a combination of:

- Heuristic analysis of URL patterns (e.g., "/about" → "About Us")
- Keyword-based content analysis
- Reuse of prior classification metadata for consistency

This hybrid approach ensures accurate, scalable categorization across varied web structures.

```python
# --- Define system prompt for classification ---
def build_prompt(page_text):
    return f"""
Classify the following web page into one of these categories:

- About Us
- Contact
- Team
- Research
- Services
- Tests
- Blog
- Other

Only respond with the category name.

PAGE TEXT:
{page_text[:3500]}
"""
```

*Constructing a Prompt to Classify Webpage Text into Predefined Content Categories*

## 6.3 Classification Logic and Model

Classification Techniques

- **Rule-Based:** Fast, interpretable keyword or URL pattern matching.
- **ML-Based:** Supervised models using text features for category prediction.
- **Hybrid:** Combines rules and ML for scalable, accurate labeling.

Integrated into the MCP pipeline with robust fallback handling.

```python
# --- Classify each page ---
for url, text in data.items():
    if not text.strip():
        continue

    prompt = build_prompt(text)

    try:
        print(f"[CLASSIFYING] {url}")
        response = bedrock.invoke_model(
            modelId=model_id,
            contentType="application/json",
            accept="application/json",
            body=json.dumps({
                "inputText": prompt,
                "textGenerationConfig": {
                    "temperature": 0.5,
                    "maxTokenCount": 100,
                    "stopSequences": [],
                    "topP": 0.9
                }
            })
        )

        result = json.loads(response["body"].read())
        categories[url] = result["results"][0]["outputText"].strip()

    except Exception as e:
        categories[url] = f"Error classifying: {str(e)}"
```

*Systematically Classify Webpage Content into Established Categories*

### 6.4 Final Output Format

```
# --- Save output ---
os.makedirs("classified", exist_ok=True)
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
output_path = f"classified/categories_{timestamp}.json"

with open(output_path, "w", encoding="utf-8") as f:
    json.dump(categories, f, indent=2, ensure_ascii=False)

print(f"\n Classification results saved to: {output_path}")
```

*Code Snippet for Timestamped Export of Classification Results to Structured JSON Files*

*Structured Mapping of Company Webpage URLs to Their Corresponding Classification Labels in JSON Format*

## 7 Testing and Results

### 7.1 Test Website:

All system components were validated on publicly available website, selected for its representative structure and dynamic content.

### 7.2 Result Snapshots

*User Interface of a Web Scraping API Endpoint Parameter Input and Execution Controls*

*API Response Dashboard Displaying Web Scraping Request Parameters, JSON Output, and HTTP Headers*

### 7.3 Performance Observations

- End-to-end runtime varies with site structure and backend model throughput.
- The solution demonstrated high extraction reliability and accurate summarization for public content.
- Further tuning may be necessary for sites with login restrictions or rate-limiting.

## 8 Conclusion

The implemented MCP system offers a fully automated, agent-callable framework for large-scale web content extraction, intelligent summarization, and structured classification. Leveraging FastAPI, Selenium, and Bedrock, it ensures modularity, scalability, and seamless integration with downstream AI workflows.

Key Components:

- main.py: FastAPI endpoints (e.g., /scrape)
- crawl_site: Recursive crawling and link extraction logic
- summarize.py: Bedrock-powered content summarization
- classify.py: URL and content-based topic classification
- export.py: Aggregation and export of final structured results