

LLM Web Crawler Setup

Using Serper, ScrapingFish API if rendering JavaScript pages into HTML content (if needed) and Crawl4AI with a suitable open source LLM for 16GB VRAM graphics card create a modular and potent web crawling application that follows this criteria:

Building a web crawler that leverages Large Language Models (LLMs) or AI agents for lead generation involves combining several components: query generation, web search, AI-based filtering, and detailed web scraping. The goal is to efficiently find business and professional service providers (e.g. tax consultants, investment advisors, financial analysts) in specified regions, and extract rich profile data about them. Below we outline a modern approach using open-source, cost-effective tools and frameworks, and highlight proven methods and examples from 2024–2025.

1. LLM-Generated Search Queries (Region-Specific)

To begin, an LLM can expand a user's prompt (industry/domain + region) into a diverse set of search queries. For example, if the user is looking for tax advisory firms in the UAE, the LLM might generate queries like "tax consultants in Dubai", "corporate tax advisors UAE", "VAT compliance services Abu Dhabi", etc. This ensures broader coverage of relevant terms (including synonyms and local terminology) beyond the user's initial phrasing. You can achieve this by prompting a model like GPT-4 (or a local LLM such as Llama 2 for cost savings) to brainstorm search strings given the industry and geographic scope.

Tools/Frameworks: This step can be implemented with a direct LLM API call or via frameworks like LangChain, which allows you to create an LLM chain for query generation. For instance, a LangChain LLMChain could take the user input and output a list of refined search keywords. The key is to incorporate the geographic filter into the prompt so that the LLM includes region-specific terms (country, city names, etc.) in the queries. This LLM-driven approach ensures the crawler starts with highly relevant search terms, reducing

noise from the very beginning.

2. Automated Web Search for Leads

After generating search queries, the system needs to execute them on search engines or specialized lead sources to gather initial leads (candidate company names and websites). There are a few budget-friendly options to do this:

Search Engine APIs: Using an API avoids scraping HTML directly and often returns structured results. For example, the SerpApi service provides a JSON response for Google Search queries (essentially “the JSON representation of Google search results”). SerpApi is a paid service but offers a free trial/key with limited queries. Another option is Serper API, which is a Google Search/Places wrapper; it has a free tier (~2500 queries) and was used in a 2024 lead-gen project for querying Google Places. If using Serper, you can hit endpoints for Google Places to find businesses by query and location.

Google Places API / Maps: Especially for local business leads, the Google Places API can directly return businesses matching a search in a region (with fields like name, address, website, rating, etc.). In one real-world implementation (Mart Kempenaar, 2024), a “LeadFinder” agent queried Google Places via Serper’s API to get a list of businesses for a given niche and location. This approach is effective for geographically constrained searches – it yields companies in the area along with their addresses and sometimes websites.

3. LLM-Based Filtering of Search Results

Not all search results will be relevant leads (some may be irrelevant pages, generic articles, or low-quality entries). Here, an LLM can act as a filter or classifier, judging which results truly correspond to companies/consultants in the desired domain and region. There are a couple of ways to implement this filtering:

Snippet Evaluation: For each result, take the title and snippet (or meta-description) and prompt an LLM with something like: “Given the following snippet, is this likely a [tax consulting service] provider in [UAE] (yes/no)?” or “Classify this result as: 1) Relevant service provider, 2)

Unrelated.” The LLM’s responses can be used to drop irrelevant results. This prompt-based filtering harnesses the LLM’s understanding of context to save you from manually coding numerous rules.

Use open-source LLMs (or smaller models) if cost is a concern – the classification task may not require the largest models. Frameworks like LangChain simplify this step: one can create a custom Tool or chain that takes a URL, fetches content, and uses an LLM to evaluate relevance. In fact, advanced agent frameworks allow chaining these actions: e.g. an agent that, for each result, performs a “ReadContent” tool and then decides “Keep or discard” based on the LLM’s analysis.

4. Web Scraping and Data Extraction for Lead Details

Once relevant leads (company websites or profile pages) are identified, the next step is to scrape detailed information from each. The goal is to gather fields like Company Name, Logo, Country, Website, Social media links (LinkedIn, Facebook, Instagram), Services offered, Contact numbers, Emails, Office locations, Ratings, Key Individuals, Pricing, and Service Descriptions.

Data Outputs: The output of this scraping step should be structured data for each lead. A common format is JSON or CSV with fields as columns.

I want modular and ready to use code that adheres to the above instructions.

To make sure the code is tailored correctly, could you please clarify the following:

1. What programming language would you prefer for this project? (e.g., Python)
2. Do you plan to run this as a local script or as part of a web application or service?
3. Which open-source LLM would you prefer for filtering/classification (e.g., LLaMA 2, Mistral 7B, etc.)?
4. Will you be using your own API keys for Serper and ScrapingFish, or should I mock these for now?
5. Should the application support batching multiple industry-region pairs or focus on one at a time?

Once I have this information, I can generate the modular and ready-to-use code for you.

What programming language would you prefer for this project? Python

Do you plan to run this as a local script or as part of a web application or service? Locally

Which open-source LLM would you prefer for filtering/classification (e.g., LLaMA 2, Mistral 7B, etc.)?
Chose both a suitable LLM and a framework for calling and prompting the LLM based on my hardware constraints (16GB VRAM)

Will you be using your own API keys for Serper and ScrapingFish, or should I mock these for now? Mock these for now in a config/env like file

Should the application support batching multiple industry-region pairs or focus on one at a time? Focus on one at a time

Great. I'll prepare a modular and ready-to-use Python application for local use that integrates LLM-powered search query generation, Serper-based lead discovery, ScrapingFish for dynamic page rendering and scraping, and an open-source LLM-based filter and classifier. It will support one industry-region pair at a time and use mock API keys stored in a `.env` or config file. I'll also select an appropriate LLM (compatible with 16GB VRAM) and framework for local inference and prompt orchestration.

I'll let you know as soon as the code and structure are ready.

 Modular AI-Powered Web Crawler for Local Lead Generation



LLM-Powered Web Crawler for Regional Lead Generation

Overview: We propose a local Python application that finds business/professional service providers (e.g. tax consultants, investment advisors) in a specified region and extracts detailed profile data. The system is modular, with separate components for query generation, web search, result filtering, and data scraping. A lightweight open-source LLM (7B parameters, e.g. Mistral-7B or LLaMA2-7B) running on a 16GB VRAM GPU powers the query generation and filtering logic. The pipeline for one industry-region run is as follows:

1. LLM-Generated Search Queries

The first module uses a local **Large Language Model (LLM)** to brainstorm search engine queries tailored to the target industry and region. We load an open-source 7B model (such as **Mistral 7B** or **LLaMA 2 7B**) using an efficient runtime like `llama-cpp-python` or `vLLM`. These models can run on ~16GB of GPU memory in full precision, or even less with quantization. For example, using `llama-cpp-python` one could load a GGML/GGUF quantized model and enable GPU acceleration for speed:

```
python

from llama_cpp import Llama
# Load the 7B model locally (using GPU layers for speed if available)
llm = Llama(model_path="models/7B/llama-2-7b.gguf.q4_0.bin", n_gpu_layers=50)
```

Model Inference: We prompt the LLM to suggest multiple search queries based on the **industry** and **location**. For instance, if the user inputs *"tax advisory firms in UAE"*, we craft a prompt like:

```
css

"Generate a list of 5 Google search queries to find tax advisory companies in UAE."
```

The LLM's completion might return a list of queries (each on a new line or numbered). We then parse the output text into a Python list. In practice, the `Llama` API returns a completion object from which we extract the text. We can split the text by newlines or list markers to get individual query strings. For example:

```
python
```

```
prompt = f"List 5 search queries to find {industry} in {region}."
result = llm(prompt, max_tokens=100) # get LLM-generated text
queries = [q.strip("- ") for q in result["choices"][0]["text"].splitlines() if
q.strip()]
```

This yields queries such as:

- "Best tax consultants in UAE"
- "UAE tax advisory firms for businesses"
- "Dubai investment advisory services"

The LLM's ability to adapt to different inputs means no hardcoded queries; it uses its knowledge to cover various relevant search intents. We can further guide it via prompt engineering (e.g., "include city names or local terminology"). Using a local LLM in this zero-shot manner is feasible since we're leveraging its general knowledge without additional training.

Note: A 7B model is sufficient for generating short queries. It loads quickly and can be re-used for other tasks in the pipeline, which minimizes overhead. We orchestrate LLM calls either via a framework like **LangChain** or with simple wrapper functions. (LangChain provides tools for prompt templates and LLM integration, but a custom implementation gives more control in production.)

2. Automated Web Search

The next module takes each LLM-generated query and runs a web search to collect candidate leads. We use the **Serper API** (a Google Search JSON API) to perform these searches programmatically. The application loads the Serper API key from a config (e.g. a `.env` file or central config) – for example, an environment variable `SERPER_API_KEY` can be set.

For each query string, we call Serper's search endpoint (e.g. via an HTTP GET request) and retrieve the JSON results. A pseudo-code example using the `requests` library:

```
python

import os, requests
api_key = os.getenv("SERPER_API_KEY")
```

```
headers = {"X-API-KEY": api_key}
params = {"q": query} # search query
response = requests.get("https://google.serper.dev/search", params=params,
headers=headers)
results_json = response.json()
```

Serper returns a JSON with fields like `"organic"` for organic results, `"places"` for map results (if `type='places'` is used), etc. We focus on organic search results. Each result item typically includes a title, URL, and snippet (the summary text) . For example, a portion of Serper's response might look like:

```
json

{
  "organic": [
    {
      "title": "ABC Tax Consultants - Dubai, UAE",
      "link": "https://abctaxuae.com/",
      "snippet": "ABC Tax Consultants offers financial and tax advisory services in Dubai..."},
    ...
  ]
}
```

From each query's results, we collect a list of candidate URLs along with their snippet and title. We can configure how many results to take per query (Serper's default is 10 results per query). These results are passed to the filtering stage next.

Note: Instead of raw Google Search, one could also leverage the **Google Places API** for business search. For example, a Places API query for "tax consultants in Dubai" could directly return business listings with names, addresses, and ratings. However, using Serper (Google Search) is simpler to integrate for our use-case and can find not only official websites but also directory listings or reviews that might contain leads. We can mix both approaches if needed (e.g., use Places for an initial list of known businesses, then use web search to get their websites).

3. LLM-Based Filtering of Results

Not all search results will be relevant (some might be irrelevant blogs, general articles, or out-of-region hits). To automatically filter these, we employ the **same LLM** (already loaded) to evaluate each result's snippet (and title) and decide if it likely corresponds to a target lead.

We prompt the LLM in a classification mode. For example, for each result, we feed a prompt like:

kotlin

Snippet: "ABC Tax Consultants offers financial and tax advisory services in Dubai..."

Question: Does **this** describe a tax advisory company located **in** UAE?

Answer (Yes or No):

Using a consistent template, the LLM will respond with a yes/no (or a brief affirmation/refusal). Because the model has strong comprehension abilities, it can use context words in the snippet (like location names or service keywords) to judge relevance. This is essentially **zero-shot classification**: the LLM uses its pre-trained knowledge to categorize text without additional training .

We implement a filtering function that calls the LLM for each snippet and parses the response. For example:

python

```
def is_relevant(snippet: str) -> bool:
    prompt = (f"Snippet: \"{snippet}\"\\n"
              f"Question: Is this snippet about a {industry} in {region}? Answer yes"
              "or no.")
    output = llm(prompt, max_tokens=10)
    answer = output["choices"][0]["text"].strip().lower()
    return "yes" in answer
```


If the LLM responds with "yes" (or equivalent), we mark the result as a valid lead; if "no", we discard it. This dramatically narrows down to the real leads.

Efficiency: Since the LLM is running locally, this step is fast for short texts. We can also batch prompts if using an engine like vLLM which supports batched generation . The key benefit is using the same loaded model for both query generation and filtering, which **reduces infrastructure complexity** (we don't need a separate classifier model) . The modular `LeadFilter` class simply reuses the LLM instance, or, for separation of concerns, it can accept text and return a boolean (internally calling a shared LLM resource).

4. Web Scraping and Detail Extraction

For each URL deemed relevant, the crawler fetches the web page and extracts structured contact/profile information. This is handled by a **Scraper** module. To robustly retrieve pages (including those requiring JavaScript), we use the **ScrapingFish API** as a rendering service. ScrapingFish allows us to get the full HTML of a page by simply providing the URL and our API key, taking care of headless browser rendering, proxies, and anti-bot measures

We store the ScrapingFish API key in configuration (e.g. `SCRAPINGFISH_API_KEY` in the `.env`). For each lead URL, we call the API, for example:

```
python

api_key = os.getenv("SCRAPINGFISH_API_KEY")
scrape_url = f"https://scraping.narf.ai/api/v1/?api_key={api_key}&url={target_url}"
html_response = requests.get(scrape_url)
page_html = html_response.text # full HTML content
```

This yields the raw HTML of the page , which we then parse using **BeautifulSoup**:

```
python

from bs4 import BeautifulSoup
soup = BeautifulSoup(page_html, "html.parser")
```

Data Extraction: We aim to gather fields like **Company Name**, **Website URL**, **Social Media links**, **Email**, **Phone number**, **Services offered**, **Locations**, **Ratings**, etc., if available on the page. The extraction logic can be implemented with a combination of HTML parsing and regex:

- **Company Name:** Often in a `<title>` tag or an `<h1>` header. For company websites, the `<title>` text or header text is a good proxy. For directory listings, the listing's name field can be targeted (e.g., a specific CSS class or the first `<h1>` on the page). For example, on a YellowPages listing, one might do `name = soup.select_one("h1.business-name").text`. In general, we can take `soup.title.text` as the name and refine if needed.
- **Website URL:** If the page itself is the company's site, the URL is already known. But if the page is a profile on a directory (like a Yelp page for the business), it might contain an external link to the business's official website. In that case, we search for anchor tags that likely represent an external site (e.g., anchor text "Website" or an icon link). If found, that href can be recorded as the "Website". Otherwise, we use the current page URL as the website.
- **Social Media Links:** We scan all `<a>` tags for URLs containing known social domains (like Facebook, Twitter, LinkedIn, Instagram). For example, we can collect any anchor whose href contains `"facebook.com"`, `"twitter.com"`, `"linkedin.com"`, etc.
. This quickly yields a list of social profile URLs if the company has linked them on their site:

```
python

social_links = []
for a in soup.find_all('a', href=True):
    href = a['href']
    if any(domain in href for domain in ["facebook.com", "linkedin.com",
    "twitter.com", "instagram.com"]):
        social_links.append(href)
```

The result might be, e.g., `["https://www.facebook.com/ABCTaxUAE", "https://linkedin.com/in/abc-tax-consultants"]`.

- **Email Addresses:** Many sites include contact emails as `mailto:` links. We find `<a>` tags with `href` starting with "mailto:" and extract the address part.
For example:

```
python
```

```
emails = set()
for a in soup.find_all('a', href=True):
    if a['href'].startswith("mailto:"):
        email = a['href'][7:] # strip "mailto:"
        emails.add(email)
```

This collects any direct email links (e.g., `info@abctaxuae.com`). In addition, we can search the text for plain email patterns using regex (e.g. `r'[\w\.-]+@[\w\.-]+'`). This covers cases where email is written as text or slightly obfuscated. (However, many modern sites use forms instead of listing emails, so results may vary.)

- **Phone Number:** We extract phone numbers by searching the page text for digit patterns. A regex can match international and local phone formats. For instance, one approach is to use a broad pattern like:

```
python
```

```
phone_pattern = re.compile(r"[+]*(?\\d{1,4})?[-\\s\\.0-9]{7,}")
phones = phone_pattern.findall(soup.get_text())
```

and then filter out obviously short or malformed hits. In the code snippet below, a regex finds sequences of digits (optionally starting with +country code) and we consider those with length > 7 or so as potential phone numbers:

```
python
```

```
text = soup.get_text()
raw_phones = re.findall(r"[+]*([\\{0,1\\}\\d{1,4}[\\]}{0,1}[-\\s\\.0-9]+", text)
phones = [p.strip() for p in raw_phones if len(p.strip()) > 6]
```

This might yield entries like `" +971 4 123 4567"` or `"04-1234567"`. We could further normalize them if needed.

- **Services Offered:** To get a description of services, we look for textual sections that list what the company does. Commonly, a site might have a bullet list or a paragraph under a heading like "Our Services" or "What we do". We can search for headings containing "Services" (e.g., `soup.find(text=re.compile("Services"))`) and then grab the next sibling elements (like a `` of services). Alternatively, some sites have a summary paragraph on the homepage – we might take a snippet of the page text around keywords like "provides" or "specialized in". This part may require site-specific tuning. As a generic approach, extracting the first few sentences from the About section (if identifiable) or meta description content could serve as a summary of services.
- **Locations/Address:** If the business has a listed address, it could be anywhere on the page. We search for typical address patterns or keywords (e.g., city names, postal codes). For example, finding an `<address>` tag or any text containing the region name (like "Dubai" or "UAE") near other address clues (digits, street, etc.). On directory sites, the address might be in a specific tag (e.g., a ``). We can attempt a regex for common address formats or simply capture the line containing the region name. For a basic implementation, we might do:

```
python

address = None
for text_line in soup.stripped_strings:
    if region in text_line:
        address = text_line
        break
```

This is a heuristic; a more robust solution might parse microdata (`` etc.) if present.

- **Ratings:** On some pages (especially directories or Google places pages), there may be a star rating or review count. If we suspect the page has a rating (for example, Yelp or Google Maps listing), we search for numerical patterns like "4.5 out of 5" or star symbols. For instance, on a Google place JSON from Serper, a rating might appear in the `knowledgeGraph` or `places` field, but since we are scraping HTML, we might find something like `4.5`. We can search for substrings like `/5` or the word "Rating". If found, we extract the preceding number. In a simpler approach, any number between 1.0 and 5.0 that is followed by `/5` could be the rating. We record it if available.

All the extracted pieces are assembled into a structured profile (e.g., a Python dict for that lead):

```
python

lead_info = {
    "name": company_name,
    "website": website_url,
    "social_links": social_links,
    "email": list(emails)[0] if emails else None,
    "phone": phones[0] if phones else None,
    "services": services_summary,
    "address": address,
    "rating": rating
}
```

Each field is optional – if something isn't found, we leave it as `None` or empty. The scraper module can be designed to handle missing data gracefully. For example, not all company websites will list a rating (that's more common on third-party listings), and some might omit an email in favor of a contact form.

After processing all leads, we will have a list of dictionaries containing the structured data for each identified provider.

5. Output to Structured JSON/CSV

Finally, the collected leads are saved in a `leads/` directory in both JSON and CSV formats. We ensure the output is structured for easy consumption:

- **JSON Output:** We can dump the list of lead dictionaries to a JSON file. For example:

```
python

import json
with open("leads/tax_advisors_UAE.json", "w") as f:
    json.dump(leads_list, f, indent=2)
```

This creates a JSON array where each element is a detailed profile.

- **CSV Output:** Using the same data, we create a tabular CSV (with headers like Name, Website, Email, Phone, etc.). This can be done with Python's `csv` module or using `pandas` for convenience:

```
python

import pandas as pd
df = pd.DataFrame(leads_list)
df.to_csv("leads/tax_advisors_UAE.csv", index=False)
```

The CSV will have one row per lead, which is useful for reviewing in Excel or feeding into other systems.

By storing results in a timestamped or name-based file (for instance including the industry and region in the filename), repeated runs for different inputs can generate separate lead files.

Modular Design and Configuration

The application is organized into modular components, each encapsulated in a Python class or function. This modular approach makes it easy to maintain, test, and extend each part independently:

- **LLM Query Generator (module)** – Handles loading the LLM model and generating search queries. Its configuration (model path, model type, prompt template) can be set in a config file. The model is loaded once at startup to avoid re-loading for each query. We use a local model to avoid external API calls, keeping costs low and speed high. The module interface might be `generate_queries(industry, region, num_queries) -> List[str]`.
- **Search Module** – Wraps the Serper API usage. It might have a function like `search(query) -> List[Result]` where `Result` contains title, URL, snippet. API keys for Serper are loaded from environment or a config file (we include a `.env` with placeholders like `SERPAPI_API_KEY="YOUR_KEY_HERE"` and similarly `SCRAPINGFISH_API_KEY="YOUR_KEY_HERE"` for ScrapingFish). In development, we can mock this module by reading from a saved JSON (for testing without hitting the actual API). The search module could also support switching between Google search (Serper) and Google Places search depending on a config flag.

- **Filtering Module** – Implements `filter_results(results) -> List[Result]` using the LLM to classify each result's relevance. It reuses the LLM from the Query Generator (passed in or accessed via a common LLM wrapper). This module can be optimized to batch process results for efficiency if needed. The prompt template for classification (which includes the industry/region context) is configurable.
- **Scraper Module** – Handles retrieving the HTML (via ScrapingFish) and parsing the content. It might expose a method `scrape_profile(url) -> dict` that returns the extracted fields for that URL. Internally it uses BeautifulSoup and regex as described. This module is also a good place to add any site-specific logic if certain target sites need special handling (e.g., different parsing for different known domains). By default, it uses generic strategies. The API key for ScrapingFish is also read from config. (If ScrapingFish is not available, one could swap in an alternative like Playwright for local rendering or requests for simple pages, so keeping this abstracted is useful.)
- **Orchestrator/Main** – A central script (or CLI entry point) that ties everything together. It reads the user inputs (industry and region, possibly via command-line arguments or a simple input prompt), initializes the modules, and runs the pipeline:
 1. Generate queries
 2. For each query, do search
 3. Filter results
 4. Scrape each filtered result
 5. Aggregate and save outputs.

Optionally, it can log progress and timing for each step. All configuration (API keys, model path, output directory, etc.) can be defined in a config file or as environment variables, making it easy to adjust without changing code. For example, a `config.ini` or YAML could specify:

yaml

```
model_path: "./models/mistral-7b-q4.bin"
num_queries: 5
results_per_query: 10
output_dir: "./leads"
SERPER_API_KEY: "<your-key>"
SCRAPINGFISH_API_KEY: "<your-key>"
```

The code would load this config at startup. In our case, using a `.env` for secrets and either command-line args or a config file for other settings is a practical approach.

API Key placeholders: In the provided `.env.example` (or README documentation), we include placeholders:

```
ini

SERPER_API_KEY = "YOUR_SERPER_API_KEY_HERE"
SCRAPINGFISH_API_KEY = "YOUR_SCRAPINGFISH_API_KEY_HERE"
```

Users must fill these with their actual keys. The application will typically use `python-dotenv` to load these into environment variables at runtime, or instruct the user to set them in their OS environment. The documentation clearly notes where to obtain these keys (Serper.dev account for Google Search API; ScrapingFish for the scraping API) and how to plug them in.

Performance considerations: We chose a 7B model to ensure fast inference on a local GPU. Loading a larger model (13B, 70B) might improve quality marginally, but would be slower and consume more memory. The modular design allows swapping the model if needed (for instance, if someone has a more powerful GPU, they could configure a 13B model path in the config). Similarly, if running on CPU only, one could switch to a 4-bit quantized model and still get acceptable performance. The search and scrape steps are I/O-bound (network calls), so we could make those asynchronous or parallel to speed up the pipeline. For example, after generating all queries, we can fire off searches in parallel using `asyncio` or threading, and likewise scrape multiple pages concurrently. This would significantly cut down total runtime when dealing with many results. The modular structure facilitates this – e.g., the Orchestrator can use a thread pool for the scraper module calls.

Extensibility: Each component can be extended or replaced:

- We can add another filtering rule (e.g., a simple keyword filter before the LLM filter to cut obvious junk).
- We can integrate a different search API if needed (Bing API, etc.) by writing a new Search module with the same interface.
- Additional data fields can be extracted in the Scraper by adding functions (for instance, grabbing the business hours if present, or the employee count, etc., depending on the sites).

- If needed, one could integrate a small database or CSV deduplication if running the tool repeatedly (to avoid collecting the same lead twice).

Throughout the development, emphasis is on **clarity and modularity**. Each part of the code has a single responsibility and can be tested on its own (for example, one can unit test the email extraction with sample HTML input, or the LLM filter with some example snippets). By organizing the code in classes (`LLMQueryGenerator` , `GoogleSearchClient` , `LeadFilter` , `LeadScraper` , etc.), and using a central configuration, the solution becomes maintainable and easy to configure.

Finally, we document usage in a README: how to install dependencies (e.g. `pip install llama-cpp-python bs4 requests python-dotenv` etc.), how to place the model file, how to set the API keys, and how to run the script. The user just needs to run something like:

```
arduino
```

```
python run.py --industry "Tax Consultants" --region "UAE"
```

and the tool will produce `leads/tax_consultants_UAE.json` and `.csv` with the results.

Conclusion: This LLM-powered web crawler automates lead generation by combining the strength of language models in understanding context (for query generation and result vetting) with traditional web scraping techniques. The design is modular and configurable, focusing on one industry-region at a time for clarity. By running entirely locally (LLM inference on GPU and API calls for search/scraping), it ensures data can be gathered efficiently without exposing the process to external LLM services, and it can be adapted or scaled as needed for different domains.

Sources:

- Mistral 7B model hardware requirements and use of quantization
- Example of using `llama-cpp-python` to load a local LLM model and generate text
- LangChain Serper API wrapper showing setting `SERPER_API_KEY` in environment
- Serper API output JSON structure (title, link, snippet) for Google search results
- Advantages of using the same LLM for multiple tasks (query generation and classification)

- ScrapingFish API usage example via curl (passing API key and target URL)
- BeautifulSoup scraping for emails via mailto links
- BeautifulSoup scraping for social media links by filtering anchor hrefs
- Regex-based phone number extraction from page text (Scrapfly example)
- Recommendation to use LangChain for prototyping but custom code for production for flexibility

     Sources

