ChatGPT

Revised Plan: Automated Used Car Listings Compilation

Objective and Criteria

The goal is to quickly gather a **large list of used car listings** that meet your specific criteria (e.g. make/model, price range, year, mileage, location around Fairfax, VA). You want to compile these listings within **1-2 days** so you can review them and potentially make offers. The solution should minimize cost (preferring free tools/DIY over paid services) and leverage advanced AI/automation to speed up implementation. In summary, we need an **automated web-scraping workflow** (potentially AI-assisted) to pull many relevant car listings *fast*.

Overview of the Approach

We will implement a **web-scraping based solution** to aggregate listings from multiple sources (online marketplaces and listing sites). The plan is to use **AI as a coding assistant** rather than a fully autonomous agent, for reliability and speed. Specifically, we'll let GPT-powered tools (like OpenAI's Codex/GPT-4 via VS Code or similar) help write the scraping scripts, but we will run the scrapers ourselves. This gives us the best of both worlds: rapid development with AI help, and direct control over the scraping process for efficiency.

The approach involves: - Identifying the **best websites to scrape** for used cars (to cover both dealer listings and private sales). - Choosing the right **tools/frameworks** for scraping (e.g. Python with requests/BeautifulSoup or Selenium, or using an existing scraper service) considering the no-cost constraint. - Using **AI assistance** (such as GitHub Copilot Chat or ChatGPT in VSCode) to quickly generate and refine the code for each site's scraper. - Running the scrapers to collect data into a consolidated list (e.g. a CSV file), within the short timeframe. - Ensuring we respect basic scraping etiquette (avoiding getting IP blocked by using delays, etc.) while still completing in time [1].

Throughout, the priority is **fast implementation**. That means we will favor methods that are straightforward and known to work (possibly reusing code from similar projects) over overly complex or experimental solutions. AI "agent" tools will be used in a **supporting role** rather than fully trusted to do everything autonomously (to avoid the risk of the agent going off-track and wasting time).

Tooling Options for Implementation

There are a few possible approaches, each with pros/cons. We'll compare them and then pick the optimal route:

- **1. Prebuilt Scrapers via Apify or Similar:** Apify offers ready-made scrapers ("actors") for many car listing sites (e.g. Cars.com, AutoTrader, CarGurus, etc.). For example, Apify's Autotrader actor can extract car descriptions, prices, mileage, seller info, etc., with no coding on your part [2]. Likewise,

they have a CarGurus scraper where you just input a zipcode and get results in minutes [3]. This is *fast to get running* and requires minimal development – you'd just configure and run the scrapers. However, using Apify for a large data pull might incur costs if you exceed the free tier. Since you prefer a no-cost solution, we'd have to stay within any free limits or use it sparingly. It's a good backup if coding hits a snag, but we should be mindful that heavy usage may require a paid plan.

- **2. Custom Python Scraping (with AI-assisted coding):** This approach involves writing our own scripts using Python libraries like `requests` + `BeautifulSoup` for HTML parsing, or `Selenium` for sites requiring dynamic interaction. The advantage is **full control and zero platform cost** – everything runs locally. Development can be accelerated with GPT-4/Codex helping to generate code for parsing pages. Many car sites are actually straightforward to scrape. For instance, **Cars.com** can be scraped with simple HTTP requests (no special headers needed, as the site doesn't aggressively block default Python user-agents [4]). We can leverage tutorials and existing code: ScrapeHero has a guide on scraping Cars.com with Python and BeautifulSoup [5], and there are open-source examples for CarGurus and others. By using these as a starting point, we can very quickly build custom scrapers. This method is likely the *most cost-effective and quick*, given your comfort with running code locally. We'll use AI in a supporting role – e.g., ask Copilot/ChatGPT to write a function to fetch and parse listings from page HTML – rather than having an AI agent do the browsing. This keeps the process deterministic and fast.

- **3. Autonomous AI Agents (Auto-GPT style or VSCode Agent Mode):** Another idea is to deploy an AI agent that you prompt with the goal and let it figure out how to browse sites and gather data. For example, an Auto-GPT instance could theoretically search for used car deals and attempt to scrape them itself. However, in practice this approach can be **hit-or-miss and time-consuming to fine-tune**. Autonomous agents sometimes get stuck or take inefficient actions without careful guidance. As an illustration, *Wired* magazine experimented with Auto-GPT for a task and found it often required human intervention and could go "haywire" on complex web tasks [6] [7]. Especially for something specific like multi-site web scraping, a purpose-built script is usually more reliable. Additionally, using the AI to directly parse lots of web content can be **expensive in API usage** – one engineer noted that trying to extract data from many pages via GPT-4 racked up over $24 in API costs in just two days [8]. Given our goal of no-cost (or low-cost), an autonomous agent that continuously calls GPT-4 on page content is not ideal. That said, there are some promising "agent in VSCode" tools (for example, ChatGPT extensions that can read files and suggest code). These are essentially an extension of approach #2: you still write code, but with the agent's help in your development environment. We will leverage those (which cost nothing extra aside from possibly some OpenAI token usage) to speed up coding, but **not rely on a fully automatic bot** to do the entire job unsupervised.

Decision: We will primarily pursue **Option 2: Custom Python Scraping with AI coding assistance**, as it best meets the quick implementation and no-cost requirements. We'll keep Apify in mind as a fallback for any site that proves too difficult or if we're running out of time (since Apify scrapers can be tried for free on a small scale quickly [3]). The autonomous agent approach (#3) will be limited to using AI to assist our coding (e.g. using Copilot in VSCode to generate functions), rather than letting an agent roam the web on its own. This way, we maintain control over time and output format, ensuring we get a usable list of cars within a day or two.

Selecting Data Sources and Strategy

Next, we identify which websites to scrape for a **comprehensive used car search**. We want to cover both **dealer listings** (for volume and breadth) and **private seller listings** (for potentially better deals and negotiation). Here are the sources we'll target and why:

- **CarGurus:** A major aggregator that lists used cars from dealerships nationwide (and some private sellers). CarGurus provides useful info like price vs. market value ("Good Deal" badges, etc.) and has a large inventory. We can search by zipcode and radius to cover our area. Others have successfully scraped massive amounts of data from CarGurus – for example, a Kaggle dataset of 3 million used car entries was built by crawling CarGurus in 2020 [9], so we know it's feasible. We'll use CarGurus to get a broad view of dealer offerings within, say, 100 miles of our zip code. *Method:* CarGurus might require either simulating a browser (Selenium) or finding a hidden API. There is precedent: one open-source scraper used Selenium to click through pages of results [10] [11]. We can attempt a direct GET request approach first; if the site uses static HTML page navigation with URL parameters, we can loop over page numbers easily. If not, Selenium will be our backup to automate clicking the "next page" button.

- **AutoTrader.com or Cars.com:** These are also big marketplaces for dealer listings (and some certified pre-owned). They have overlapping inventories with CarGurus, but might include different dealerships or listing details. **Autotrader** in particular allows filtering and is a long-established site. We may choose one of these to supplement CarGurus. **Cars.com** is known to be scraper-friendly – it doesn't block basic requests and you can get details like name, price, dealer info, etc., from both the listing and detail pages [5] [12]. Using Cars.com could be straightforward with a simple BeautifulSoup parser since no special anti-bot measures are in place [4]. **Autotrader**, on the other hand, might have some anti-scraping defenses (like requiring a modern user-agent or limiting results without login). However, Apify has a high-rated Autotrader scraper [2], indicating it's doable. To keep things quick, we might **pick Cars.com** over Autotrader in the first iteration, simply due to ease of scraping (as documented by ScrapeHero [4]). If time permits, we can add Autotrader for completeness (perhaps even using Apify's Autotrader actor via their API if we want to avoid coding it).

- **Craigslist (Private Sellers):** Craigslist is a prime source for local private-party car sales (and some independent used car lots). Deals here can often be negotiated, fitting your "make an offer" goal. There is a **Craigslist cars/trucks category** for each region (e.g., Washington DC area). People have scraped Craigslist at scale in the past – one Kaggle dataset scraped ~423k car listings from Craigslist across the US [13]. We won't need anywhere near that volume if we focus on our region and criteria, which keeps it manageable. *Method:* Craigslist is mostly static HTML and can be scraped with requests + BeautifulSoup fairly easily. We'll construct a search URL with filters (e.g., by price range, specific keywords or make/model, etc., depending on your criteria) and then paginate through. We should be cautious with rate limiting on Craigslist to avoid IP blocks (insert short random delays between requests and not scrape too many pages per minute [1]). If multiple Craigslist regions are relevant (e.g., DC, Richmond, Baltimore, etc.), we can repeat for each region's site and combine results. **Note:** Craigslist removed official RSS feeds for searches, so we'll parse the HTML directly rather than relying on RSS or API [14]. Each listing's essential info (title, price, location, posting date) can be pulled from the results page, and we'll also capture the URL so you can click through for contact details (we likely won't attempt to scrape emails/phone from Craigslist listings, as those

might be protected behind captchas or email relay links – instead, you'll use the listing link to get that info manually for any car you're interested in).

- **Other Sources (Optional):** If time allows or if you have specific sources in mind, we could include others:

- *Facebook Marketplace:* Huge volume of private listings, but **not scrape-friendly**. It requires login and Facebook actively tries to prevent scraping. This is likely not feasible within 1-2 days (and violates their terms), so we'll skip it.
- *CarMax or Carvana:* These are nationwide dealer-specific sites. They have a lot of inventory but since they are single-seller (you buy directly from CarMax/Carvana), negotiating might not be possible. Also, their inventories might already be included on CarGurus/Autotrader in some form. We can deprioritize these to save time.
- *Carfax Used Car Listings:* Carfax has a search tool that aggregates dealer listings with verified Carfax reports. It could be useful if you wanted only cars with clean history, etc. Apify has a Carfax scraper [15], but again, including this means more to implement. Given the time crunch, it's probably unnecessary if CarGurus and Cars.com cover the bases.
- *Others:* eBay Motors, TrueCar, Edmunds used car listings, etc., all exist but each additional site adds complexity. We'll stick to the big three: **CarGurus, Cars.com, Craigslist** for the initial plan. These will provide plenty of data to work with in a short time.

By focusing on the above sources, we capture a wide net: **dealership listings (CarGurus/Cars.com)** for volume and comparisons, plus **local private sales (Craigslist)** for potentially better prices. Each source will be handled separately (with its own scraper code or tool), and we'll merge the results as needed.

Implementation Steps and Timeline

Given the tight 1-2 day timeline, we'll break down the plan into concrete steps that can be executed rapidly. We assume you have a development environment ready on your local machine (or an "agent mode" workspace in VSCode) with Python installed. Here's the game plan:

Day 1: Setup and Prototyping

1. Environment Setup: Ensure Python 3 is available on your desktop. Install essential libraries: - `requests` (for HTTP requests) and `beautifulsoup4` (for HTML parsing) – these will handle most static pages [16]. - `selenium` and a browser driver (like ChromeDriver) – in case we need to simulate a browser for sites like CarGurus. If using Selenium, also install `webdriver-manager` for easy driver setup. - `pandas` (optional) – to organize results into DataFrame/CSV easily.

AI assist: You can have ChatGPT/Copilot write a quick requirements list or pip install script for these. Example: *"Write a pip install command for requests, beautifulsoup4, lxml, selenium, webdriver-manager, pandas."*

2. Define Search Criteria: Clearly specify your search filters so we use them in our scrapers: - e.g. **Location**: Fairfax, VA – we'll use a central ZIP (like 22030) and a radius (maybe 50 or 100 miles) for CarGurus/Cars.com queries. - **Budget**: (Example) $5,000 to $15,000 – use this range to filter out cars outside your price range. - **Type**: if you have particular makes/models or body type (say you only want SUVs, or specifically looking for

a Toyota Camry), note that. Otherwise, we search “all makes” within criteria to cast a wide net. - **Year/ Mileage**: any minimum year or max mileage? Include if relevant. - We will apply these filters on each site if possible (most sites allow filtering by price, year, etc. either via URL parameters or parsing out unwanted results).

This step is mostly manual specification by you. Once decided, we can incorporate the filters into the scraping code.

3. Scraper for CarGurus: Start with CarGurus since it’s a big source. - **Approach:** Attempt to use requests to fetch results. We need to find the URL pattern for CarGurus search results. Often, CarGurus uses a query with many parameters (as seen in a snippet of a scraper code) with a structure like `inventorylisting/viewDetailsFilterViewInventoryListing.action?...zip=<ZIP>&distance=<X>&page=<N>` [17]. We’ll search the HTML of the CarGurus site manually (or use your browser’s developer tools) to see if changing the page is just adding `page=2` or similar in the URL. - If a **simple GET with page parameters** works, we can loop pages 1..N and parse. If not, fall back to Selenium: - Using Selenium, have it load the initial search page (for our zip, radius, and filters). Then repeatedly click the “Next” button, as the GitHub gist did [11], collecting page HTML each time. - **Parsing:** For each page, parse the HTML to extract listing elements. Based on an existing CarGurus scraper, each car’s info is in a `<div class="cg-dealFinder-result-wrap">` or similar [18]. Within that: - Find the listing title (which usually contains year, make, model). - Price (span or div with price). - Mileage and possibly location (often listed in the snippet). - Deal rating or “market price” info (CarGurus provides text like “$1,200 below market” – could be nice to grab). - URL of the listing detail page (so you can click it later) – likely the `<a>` around the title. - Seller info: CarGurus often lists the dealer name and maybe a rating. We can capture dealer name (or for private, it might say “Private Seller”). - We’ll have GPT assist by feeding it a sample HTML snippet (or describing it) and asking for code to parse out these fields. For example, *“Using BeautifulSoup, how do I extract the title, price, mileage, and listing URL from each result card on a CarGurus search results page?”* The model might output code using `soup.find_all` with the appropriate classes (like it did for Cars.com in the ScrapeHero tutorial [19] [20]). - **Testing:** Run the CarGurus scraper on a small scale (e.g., first 2 pages) and print a few results to verify it’s pulling correct data. We’ll likely get ~20 results per page; ensure fields are coming through correctly (e.g., price as a number, etc.). - Adjust as needed. If we face any blocks (like CarGurus requiring a modern user-agent string), we can set headers in `requests` to imitate a browser.

4. Scraper for Cars.com: This one should be quicker thanks to available examples. - **Approach:** Use the Cars.com search URL with query params. For instance, Cars.com allows filtering by make/model, distance, zip, price range, etc. We can construct a URL (using their search form to get the exact query string). - We’ll use `requests.get()` to fetch the HTML. As ScrapeHero notes, Cars.com doesn’t block the Python default user agent [4], but to be safe we can include a simple header. - **Parsing:** Based on the tutorial, each listing on Cars.com’s results is contained in a `<div class="vehicle-details">` block [21]. Within that: - Title is in an `<h2 class="title">` [22]. - Price in a `<span class="primary-price">` [23]. - Mileage in a `<span class="mileage">` (if available) [24]. - Dealer name and possibly location are also in the block (the tutorial mentions dealer details are present in the snippet). - Listing URL in an `<a href>` (need to prepend "https://cars.com" if it’s relative) [20]. - GPT can easily generate code once we provide these class names. (The ScrapeHero article even outlines code for find_all and find on these classes, which we can adapt [25].) - **Testing:** Run for maybe the first 1-2 pages (Cars.com typically paginates results, and you can often append something like `&page=2` to the URL or it might be `?page=2` depending on their format). Verify output similarly to CarGurus.

5. Scraper for Craigslist: This will cover private sellers. - **Approach:** Identify the Craigslist site for the region. For Fairfax, the relevant Craigslist might be the Washington, D.C. area craigslist (`washingtondc.craigslist.org`) under the category "cars+trucks – by owner" (since dealer postings on Craigslist might be less priority if we already have dealer listings elsewhere, and private owner sales are what we want for offers). We can actually scrape both subcategories: "by owner" and "by dealer" on Craigslist if desired. - Construct the search URL: We can include parameters for price min/max, maybe keywords or specific makes. Craigslist search URLs look like: `https://<region>.craigslist.org/search/cto?query=<keywords>&min_price=X&max_price=Y&auto_year_min=...&auto_year_max=...`. If you have no specific keyword, just leaving `query` blank will show all cars in that category filtered by the other params. - Craigslist paginates by 120 results per page by adding `?s=120`, `?s=240` etc., to the URL (the `s` parameter is the index of the first result to display). We will loop increasing by 120 until no more results. - **Parsing:** Each result is in an `<li class="result-row">` item. Within: - Title text is in `<a class="result-title">`. - Price in `<span class="result-price">`. - Location (if listed) in `<span class="result-hood">` (small text indicating the neighborhood in parentheses). - Post date in `<time class="result-date">` (could ignore or keep as reference). - URL is the href of the `<a result-title>`. - Use BeautifulSoup to find all result rows and extract these details. GPT can help write this loop. (It's a relatively simple structure; many StackOverflow examples exist for Craigslist scraping that the AI might know [26].) - **Testing:** Run for one page (e.g., first 120 results) and check output. Craigslist might have a **lot** of results if price range is broad; be prepared that even 1000+ listings could show up. We might later need to narrow criteria (or just handle pagination carefully).

6. Basic Data Handling: As we get each scraper working, have it **save results to a file** incrementally: - For simplicity and quick turnaround, saving each source to a CSV (or JSON) as we go is fine. For example, `cargurus_results.csv`, `carscom_results.csv`, `craigslist_results.csv`. - Include columns like: Source, Title (Year Make Model), Price, Mileage, Location/Dealer, URL, and maybe a couple of special fields (e.g., CarGurus Deal Rating, Craigslist post date). - We will likely use pandas for this: accumulate data in a list of dicts, then `pd.DataFrame` -> `to_csv`. - Ensuring encoding and formatting are handled (pandas will take care of commas in text, etc., in CSV).

By the end of Day 1, ideally, we have **working code for each source** that can scrape a few pages and store output. This is the prototyping phase to iron out any issues (like needing to log something in, or adjusting to unexpected HTML structures). If one site proves too troublesome by end of day, we can pivot (for example, if CarGurus were completely difficult, perhaps switch to Autotrader or skip it). But based on similar projects, Cars.com and Craigslist are usually smooth, and CarGurus is doable especially with Selenium if needed.

Throughout this prototyping, we lean on the AI assistant: use it to troubleshoot parsing, quickly find the right CSS selectors, and accelerate writing loops and file handling. Because we're comfortable coding, this should significantly cut down dev time, as GPT-4 can generate 80-90% of the boilerplate on demand.

Day 2: Full Data Extraction and Integration

7. Run Full Scrapes: With tested scripts, we'll execute the scrapers to get the complete data: - **CarGurus:** Run through pages until we have gathered all results within the radius and criteria. We need to decide how deep to go – possibly CarGurus limits to 100 pages in their UI. If each page is ~20 cars, 100 pages = ~2000 cars, which is plenty. (If they have more, we might refine filters rather than trying to get everything

unlimited, because evaluating too many is impractical anyway.) We can stop when the scraper either hits an empty page or when a pre-set max page is reached. - **Cars.com:** Similar process. Cars.com might have slightly fewer per page but also can run into hundreds of pages if broad search. We can set a reasonable cutoff or use the site's count (sometimes the HTML or pagination shows total results). - **Craigslist:** Run through all pages of results. Craigslist often shows total count at top ("X results"). We can loop `s=0,120,240,...` until no more results are returned or until a max (if your criteria yields thousands, maybe stop at a certain point if it becomes unwieldy). - It's wise to **add delays** between requests/pages to not bombard the servers [1]. E.g., a `time.sleep(2 or 3 seconds)` in each loop iteration (possibly a bit longer for Craigslist since they are sensitive). This will slow the scraping slightly but given we have many hours in a day, it's fine. - Monitor the runs. If any error or ban is encountered (HTTP 429 or captcha page), we might slow down further or use a different IP (if on a home connection, probably fine at moderate pace).

8. Combine/Merge Data (optional): After obtaining the CSVs, we can merge them for your convenience: - At a minimum, ensure they have a common format for key fields (price numeric, etc.). We can concatenate them and add a "Source" column (e.g., "CarGurus", "Cars.com", "Craigslist") to know where each came from. - If you prefer separate lists by source, that's okay too. But it might be useful to have one master file to sort/filter all in one place. For example, you could sort by price across all sources to see cheapest deals first, or remove duplicates if the same car appears on multiple sites (though that's relatively rare across these particular sources, except maybe a dealer listing on both CarGurus and Cars.com – if VIN is present we could deduplicate by VIN). - This step can be done with pandas in a few lines, or even just manually later in Excel. Since time is limited, we'll consider it optional – the main goal is to *get the data*. Merging can be a quick follow-up if everything else is successful.

9. Initial Analysis (very brief): With the compiled list, do a quick sanity check: - Are all entries within the desired criteria? (If some out-of-range entries slipped through due to filter limitations, we might drop them now. For instance, CarGurus might not allow a price filter unless logged in; if so we might see some cars outside budget – we can post-filter those by price in the data.) - Remove any obvious junk or duplicates. - We might also flag particularly good deals using the data (e.g., CarGurus's own rating like "Great Deal" could be an indicator – those might deserve attention). Since CarGurus provides a "deal rating" or how much below market price, we can highlight those if present [27]. - However, detailed analysis is not the main focus – you will likely do the evaluation manually. The key is that you have a **comprehensive list** ready to go.

10. (Optional) Advanced Filtering or AI Insight: If time permits and you want to experiment further, we could use AI to help analyze the scraped data: - For example, load the CSV into a Python notebook and ask GPT (if using something like Code Interpreter or Pandas AI) to identify the top 10 deals or any anomalies. This is purely optional and for the "advanced workflow" experimentation. It might rank cars by price vs. age/mileage to find best value, etc. But be cautious: this could eat into time. The safest play is to get the raw listings first, then do any analysis separately if time remains.

By the end of Day 2, we expect to have **a compiled dataset of used car listings matching your criteria**, likely numbering in the hundreds or low thousands (depending on how broad the search was). This will be delivered as a CSV file (or multiple files) that you can open in Excel or any tool, sort and filter, and start picking promising cars to contact sellers.

Throughout Day 2, if we encounter issues (like a site blocking our scraping mid-way), we have contingency: - Use the Apify actors for that site as a quick backup (since we know they exist and can output data fast) [3]. For example, if CarGurus via our script is troublesome, we could run Apify's CarGurus Zipcode Scraper with

our filters and get the data via their platform, then download as CSV. The Apify actor boasts getting data in minutes for CarGurus [3], and Autotrader as well has an actor with no limitations on filtering [2]. - Alternatively, switch to another site: if Cars.com was not giving enough, try Autotrader or Carfax quickly with a similar script or Apify. Since you're open to trying things, this flexible approach ensures we get *something* useful even if a particular source doesn't pan out.

Ensuring a Quick and Successful Execution

To maximize the chances of finishing this in one or two days, here are some best practices and comparisons drawn from similar projects:

- **Leverage Existing Code and Templates:** We're not starting from scratch conceptually – many have done web scrapers for cars. For instance, a developer scraped a Colombian car site in a Medium article and shared the process [28], ScrapeHero published complete code for Cars.com [29], and Kaggle community scripts exist for CarGurus. Using these as reference greatly accelerates development. We'll use AI to quickly search our codebase for these patterns and implement them. This is much faster than an autonomous agent trying to learn the site structure by trial and error. Essentially, we're using **AI to code with human strategy**: we decide the plan (which sites, which fields) and the AI writes the boilerplate. This avoids the open-ended wandering that an autonomous agent might do.

- **Avoiding Pitfalls:** We will heed web scraping etiquette to prevent getting blocked or wasting time:

- Insert small random delays between page requests (as one Reddit scraping guide suggests, ~7 seconds plus a random jitter between pages [1] can keep you under the radar).
- If a site uses a lot of JavaScript for loading results (e.g., infinite scroll), use Selenium or another headless browser to let it render fully. But use it only where needed (since Selenium is slower); for Cars.com and Craigslist, pure requests are fine and much faster.

- Respect data usage – we're gathering data for personal use (finding a car), which is generally fine. We won't scrape personal info beyond what's public (and we'll stay clear of anything like scraping phone numbers directly to be safe on legality).

- **Comparison to Alternate Workflows:**

- If we *had* gone with an Auto-GPT agent approach, we'd likely spend a lot of time supervising it anyway. Users have noted that Auto-GPT can incur a lot of API calls without delivering a clean result, so our hands-on, AI-assisted coding approach is more direct for this task [8].
- Using Apify alone could have been quickest to start (just clicking "Try for free" on a Cars.com or Autotrader scraper [3]), but then merging multiple sources and possibly hitting free tier limits could slow us down. Our plan still allows using Apify as a **complementary tool** if needed rather than the main method, which gives flexibility without locking us into one platform.

- Coding everything manually without AI would certainly be slower. By using GPT-4 (via VSCode or ChatGPT web) as a coding co-pilot, we speed up writing parsers, and can debug faster with its suggestions. This is a lightweight way to incorporate AI – essentially using it as an intelligent IDE

assistant. It costs nothing if you already have access to GPT (just the subscription or some API credits you might already be using for this chat), and it will not significantly impact the timeline negatively.

- **Time Management:** We will aim to finish the core scraping by early Day 2 so that there's buffer time. If something is incomplete, we can prioritize the source that yields the most unique value. For instance, **Craigslist might have fewer results if your criteria are narrow, but those results could be very valuable (private sellers)**, so it's worth including. **CarGurus/Cars.com will have a lot of overlapping dealer cars**; even if one is partially done, you'll still have many cars to look at. Thus, focusing on completing one source fully before moving to the next is wise (rather than doing all in parallel and possibly none finished). Given the plan, Cars.com and Craigslist scrapers should be relatively quick to complete, whereas CarGurus might take a bit more tweaking – so we might finish Cars.com and Craigslist early, guaranteeing we have those lists, then use remaining time for CarGurus.

- **No-Cost Considerations:** All tools we're using are free:

- Python libraries are open-source.
- AI assistance is through tools you presumably have (ChatGPT or Copilot). If using ChatGPT Plus, the cost is fixed monthly, not per use. If using the OpenAI API via an IDE extension, the cost in tokens for writing some code is negligible for this scale.
- We won't use any paid proxy or API. If, for some reason, a site aggressively blocks your IP and we desperately need a workaround, a free proxy or a Tor routing could be attempted, but likely not necessary for our moderate use.
- Apify's free tier could be used in a pinch without spending money, as long as our data volume is within their limits (they often give a few hundred CPU-seconds free, which might cover a few hundred pages of scraping). But we will try to avoid needing that by doing it locally.

Conclusion and Revised Plan Summary

In conclusion, the revised plan is to **rapidly implement a multi-site web scraper with AI-assisted coding**, targeting key used car listing sources, in order to generate a large, consolidated list of vehicles matching your criteria within a day or two. This approach emphasizes quick turnaround, no added cost, and leveraging advanced workflows (AI agents in a supportive role) without the risk of losing time to fully autonomous but unreliable processes.

Plan Recap: - Use **Python with BeautifulSoup/Selenium**, guided by GPT-4, to build scrapers for CarGurus (for dealer listings, broad coverage), Cars.com (dealer listings, easy scraping), and Craigslist (local private listings). - Each scraper will be developed and tested with AI help, using known patterns from similar projects (ensuring we verify and cite any borrowed logic). - Run the scrapers with proper delays and gather results into structured files (CSV). - Merge and filter the data as needed, aiming to deliver a comprehensive list of candidates. - Throughout, make decisions to keep things on track for the 1-2 day timeline, using backup options (like Apify actors or alternate sites) if any primary approach fails. - Ensure the output preserves all important info (price, mileage, location, etc.) and provides links so you can follow up and make offers on the vehicles of interest.

By following this plan, you'll efficiently utilize AI to do the heavy lifting in coding, while maintaining control over the scraping process to get quick and reliable results. The end result will be a robust set of used car

listings you can immediately start evaluating – and the process itself will be a great experiment in advanced AI-assisted workflows, all achieved at essentially no extra cost except a bit of your time and creativity.

References and Similar Examples

- ScrapeHero Tutorial – *Web Scraping Cars.com Using Python* (demonstrates how to parse Cars.com with requests/BS4) [4] [25].
- Apify Marketplace – Prebuilt scrapers for used car sites (e.g., Autotrader, CarGurus) showing the scope of data you can extract [2] [3].
- Kaggle Used Cars Dataset by Austin Reese – indicates Craigslist data can be scraped and was updated periodically [13].
- Kaggle US Used Cars (CarGurus) – 3 million listings scraped via a custom crawler [9].
- Reddit r/DataHoarder advice – on scraping etiquette (use delays, save pages offline first) [1].
- Eduardo Blancas Blog – *Using GPT-4 for web scraping* experiment (highlights cost of letting GPT parse lots of content directly) [8].

---

[1] Web Scraping Wisdom for Auto Website Project : r/DataHoarder
https://www.reddit.com/r/DataHoarder/comments/1bsdes8/web_scraping_wisdom_for_auto_website_project/

[2] [3] CarGurus Zipcode Search Scraper · Apify
https://apify.com/tri_angle/cargurus-zipcode-search-scraper

[4] [5] [12] [16] [19] [20] [21] [22] [23] [24] [25] [29] How to Scrape Cars.com Using Python
https://www.scrapehero.com/scrape-car-data-from-cars-com/

[6] [7] Chatbots are becoming GetStuffDoneBots, and I'm all for it
https://link.wired.com/public/32785221

[8] Using GPT-4o for web scraping Eduardo Blancas
https://blancas.io/blog/ai-web-scraper/

[9] GitHub - Foroughmo/Used-Car-Price-Prediciton: Price Prediciton Using Explainable Boosting Machines
https://github.com/Foroughmo/Used-Car-Price-Prediciton

[10] [11] [17] [18] [27] cargurus.com scraper · GitHub
https://gist.github.com/yuangaonyc/357ea1ecb86455a0618655fafff34c3f

[13] Exploring and Analyzing Used Car Data Set | by Irtasam Ali Wains | The Startup | Medium
https://medium.com/swlh/exploring-and-analyzing-used-car-data-set-2e2bf1f24d52

[14] Does anyone know how to set up a rss feed for craigslist? Thank you
https://www.reddit.com/r/craigslist/comments/l75mal/does_anyone_know_how_to_set_up_a_rss_feed_for/

[15] Carfax.com Scraper - Vehicle History Data Extractor - Apify
https://apify.com/lexis-solutions/carfax-com

[26] an evolution story of python script to a web scraper to an API driven ...
https://forum.leasehackr.com/t/multi-brand-new-used-car-inventory-search-an-evolution-story-of-python-script-to-a-web-scraper-to-an-api-driven-webapp/268129?page=7

[28] How I built a web scraper in Python to get car prices - Medium
https://medium.com/analytics-vidhya/scraping-car-prices-using-python-97086c30cd65