Free cases -> treasury dropping cash into the economy
Steam drops cases into the cs economy, we should buy all of them up and then set the price

200 referral, bet both sides, take profits (profit-200)

🚩 Example Arbitrage Scenario:

- **BitSkins price**: $10.00 (external marketplace)
- **Steam Marketplace highest buy order**: $12.50
- **Steam fee**: ~15%
- **Profit** calculation:
    - After fees: $12.50 × (1 - 0.15) = **$10.63**
    - **Net profit** per skin: $10.63 - $10.00 = **$0.63 (6.3%)**

💡 Full Arbitrage Purchase Flow:

1. Identification of Arbitrage Opportunity

- Your arbitrage analyzer flags a price discrepancy:
    - Lower ask price on an external market (e.g., BitSkins)
    - Higher buy order price on Steam Marketplace
- You verify profitability after factoring in all fees.

---

2. Funding and Account Setup

- Ensure your external marketplace account (e.g., BitSkins) has sufficient funds or deposit funds if needed.

- Ensure your Steam account is active, unrestricted, and eligible to trade and sell items.

---

3. Purchasing on External Market (BitSkins)

- Log into BitSkins and search for the specific skin identified as profitable.
- Purchase the skin at the listed low price (e.g., $10.00).
- After successful purchase, the skin appears in your BitSkins inventory instantly.

---

4. Withdrawal to Your Steam Inventory

- Initiate withdrawal of the purchased skin to your Steam account:
    - Provide your Steam trade URL within BitSkins.
    - BitSkins generates a trade offer to your Steam account.
- Accept the incoming trade offer from BitSkins in your Steam account:
    - The skin is transferred immediately upon trade confirmation.

---

5. Listing or Selling Immediately on Steam

- Go to Steam Marketplace → Inventory:
    - Select the skin you just received.
    - Choose the option "Sell."
- You have two options:
    - **Instant Sell (to highest buy order)**:

        - Immediately sell to the current highest buy order (e.g., $12.50), ensuring immediate liquidity.

    - **Regular Listing** (alternative option):

        - Set a slightly higher ask price (above current highest buy order) if you prefer higher profits and can afford to wait for a sale.

- Confirm the listing:

    - Steam applies a standard ~15% marketplace fee.

---

6. Finalizing the Sale and Realizing Profit

- **Instant sell** scenario:

    - Your skin sells instantly at $12.50.

    - Steam deducts ~15% fee ($1.87), leaving $10.63 credited immediately to your Steam wallet.

- Net profit confirmed immediately: $10.63 (Steam wallet) - $10.00 (spent externally) = **$0.63**.

- Profit appears in your Steam wallet as Steam balance, which can be used for future arbitrage or other Steam purchases.

🚨 Key Considerations & Pitfalls:

- **Market Volatility:** Prices fluctuate quickly—execute swiftly to lock in profits.

- **Inventory Holds:** Ensure there are no trade or market holds on your Steam account, as delays can eliminate arbitrage opportunities.

- **Transaction Fees:** Always include external and Steam fees accurately in calculations.

- **Liquidity:** Make sure the Steam buy order is sufficiently large and active.

- **External Market Reliability:** Verify the external platform's reliability and withdrawal times.

https://github.com/Revadike/InternalSteamWebAPI/wiki
https://steamapis.com/
Gamerpay
skimport

CS:GO Marketplace Arbitrage Analyzer *(Python, C++, Flask, Steam APIs, pybind11)*

- Developed an arbitrage cdetection tool that identifies profitable trading opportunities for CS:GO skins by analyzing real-time bid/ask spreads and historical pricing data from Steam and third-party marketplaces.

- Engineered a high-performance C++ arbitrage detection engine integrated into a Python backend using pybind11, improving computational efficiency and enabling rapid analysis of 20,000+ marketplace items.

- Implemented Python modules for data retrieval, normalization, and storage, ensuring accurate parsing of marketplace API data while managing rate limits and avoiding request throttling.

- Built a local web dashboard using Flask and Bootstrap, providing real-time visual analytics including arbitrage alerts, pricing trends, and dynamic market insights via interactive JavaScript visualizations (Chart.js).

- Open-sourced the project on GitHub, complete with clear documentation, setup instructions, and automated tests to facilitate easy deployment and use by the community on macOS

CS:GO Skins and Cases Arbitrage System Roadmap

This document outlines a comprehensive roadmap for developing a CS:GO skins and cases arbitrage system. The system will retrieve market pricing data from the Steam Community Marketplace and optionally from third-party trading platforms. It will analyze bid/ask spreads and price trends to identify arbitrage opportunities. The implementation will use Python for data handling and a local web UI, with performance-critical logic in C++ (integrated via pybind11). The project is structured into incremental phases from initial planning through deployment.

Phase 1: Project Planning and Requirements

In this phase, we define the project scope, requirements, and technical stack. Clear goals and a solid plan will guide development.

Key Tasks:

- **Define Scope and Goals:** Determine which markets to include (Steam and optional third-party platforms) and what constitutes an arbitrage opportunity. For example, arbitrage may mean finding items with a significantly lower **ask (sell) price** than the **bid (buy) price** on the same market, or price differences between Steam and external markets.

- **Requirements Gathering:** List functional requirements (price retrieval, bid/ask spread calculation, historical price tracking, alerts for opportunities, user interface for visualization) and non-functional requirements (high update frequency, efficiency, cross-platform support, ease of use).

- **Choose Tech Stack:** Confirm the use of **Python** for data retrieval/preprocessing and web UI (due to its rich libraries and faster development) and **C++** for the core arbitrage logic (for speed). Plan integration via **pybind11** to combine Python and C++.

- **Environment & Tools:** Ensure the development environment supports macOS. Plan to use compilers available on macOS (like Clang) and tools like CMake or similar for building C++ components. Identify needed libraries (e.g., `requests` or `aiohttp` for Python HTTP calls, `pybind11` for integration, Flask or FastAPI for the web UI).

- **Repository Structure:** Design an initial project structure. For example:

    - `data_fetch/` for Python modules handling API calls,

    - `engine/` for the C++ arbitrage logic source,

    - `webapp/` for the Flask/FastAPI app and HTML/JS,

    - `tests/` for test cases,

    - plus configuration files for API keys, etc.

- **Success Criteria:** Establish how to measure success. For instance, the system should be able to track prices for a large number of items (e.g. all CS:GO skins) at a high frequency (e.g. several updates per minute) without being rate-limited, and display arbitrage opportunities clearly in the UI. Define acceptable performance (latency of data

updates, memory usage) and user experience (simple local setup, intuitive interface).

Early in this phase, a key decision is what data to focus on. We need **real-time market data** (current lowest sell price and highest buy order) and **historical trends** (price movement over time). We will prioritize the Steam Community Market as the primary data source, since all CS:GO items are traded there, and supplement with a third-party platform for additional price points. Planning also involves deciding how "arbitrage" is defined: whether purely intra-market (Steam buy vs. sell spread) or cross-market (Steam vs. external markets). These definitions guide the data requirements and algorithm design.

Finally, project planning includes outlining the subsequent phases (as detailed below) and setting up version control. A public repository (e.g. GitHub) will be initialized with a README, and continuous integration could be set up to run tests and perhaps build the C++ module on pushes. By the end of Phase 1, we should have a clear roadmap and all necessary accounts or API access prepared (Steam account for API usage, maybe accounts on third-party sites like BitSkins to obtain API keys, etc.).

BITSKINS API: 0d9b3aefe71c6b1395a772a8c7d82ce9c944ca90cc3227d9860df8104a682964

https://bitskins.com/docs/api#api-account-profile-me

Phase 2: Understanding the Steam and Third-Party Marketplace APIs

With requirements set, the next phase is researching how to get the required market data. We must understand Steam's Community Market endpoints and any third-party APIs to be used.

Steam Community Market API (public endpoints): The Steam Community Market does not offer an official comprehensive public API for market data; however, there are **undocumented HTTP endpoints** that return market data in JSON ([Steam API] - Get all prices in the format .json :: Help and Tips). Key endpoints and their uses include:

- **Price Overview:** `https://steamcommunity.com/market/priceoverview` – Provides a quick summary for a specific item's pricing. By specifying parameters like `appid=730` (CS:GO's app ID), `market_hash_name=<ItemName>`, `currency=<currency_code>`, this returns JSON with fields such as the lowest listing price, median price, and volume of recent sales ([Steam API] - Get all prices in the format .json :: Help and Tips). For example, `priceoverview` for a "Chroma 3 Case" returns its lowest sell price and total volume sold on Steam ([Steam API] - Get all prices in the format .json :: Help and Tips).

- **Price History:** `https://steamcommunity.com/market/pricehistory` – Returns historical sale data for a specific item. It requires the same parameters (`appid` and `market_hash_name`, plus a currency and country code) and returns a time-series of median sale prices and volumes. According to documentation, this includes daily median prices and volumes, and for recent weeks it provides more granular data (e.g., every 3 hours) (Learning Python – Project 3: Scrapping data from Steam's Community Market | Dr. Blake Porter). This is useful for analyzing **price movement over time**.

- **Market Search (Bulk Data):** `https://steamcommunity.com/market/search/render` – Allows retrieving data for multiple market listings in one request. By searching for all CS:GO items (`query=appid:730`) and using parameters like `start=0`, `count=100`, and `norender=1`, we get a JSON listing of up to 100 items per page with their current market stats ([Steam API] - Get all prices in the format .json :: Help and Tips). We can iterate through pages by increasing the `start` value in increments of 100 until all items are retrieved ([Steam API] - Get all prices in the format .json :: Help and Tips). Each item entry typically contains its name, perhaps the current lowest price (`sell_price` or `sell_price_text`), and other info (like sale volume). This is an efficient way to get a snapshot of many items' prices at once.

- **Order Book (Buy/Sell Orders):** Steam's web interface shows the buy and sell order book (the highest buy order and lowest sell listing, etc.) on item pages, but this isn't provided by the simple endpoints above. There is an **`itemordershistogram`** endpoint which returns the order book for a given item (Buy and Sell orders of an item in Steam API/JSON - Stack Overflow). It requires an `item_nameid` (an internal ID for the item on the market) along with locale and currency parameters, and returns detailed JSON about active buy orders and sell listings (including the highest buy price and lowest sell price). The challenge is that `item_nameid` is not readily available without scraping the item's listing page or using a third-party service (Buy and Sell orders of an item in Steam API/JSON - Stack Overflow). One must fetch the HTML of `market/listings/730/<ItemName>` and parse out the `item_nameid` (which is embedded in a script tag calling `Market_LoadOrderSpread`) (Buy and Sell orders of an item in Steam API/JSON - Stack Overflow). Once you have the `item_nameid`, you can call `itemordershistogram` to get up-to-date bid/ask data. This is vital for precise arbitrage calculations because it directly provides the highest bid and lowest ask in real-time.

- **Steam Web API (private):** Steam has an official Web API (requiring an API key), but it mostly covers inventories, player data, etc., and **does not expose market pricing** directly. If we have a Steam account and login session, we might use authenticated requests (with cookies or OAuth) to access certain data (like our own buy orders or inventory values), but for public market data the above endpoints are the primary

method. We will focus on these **public (undocumented) endpoints**, since they are accessible with just HTTP requests. We must note that since these are not official, we should be prepared for potential changes or the need to adjust our approach if Valve modifies their interface.

Third-Party CS:GO Marketplace APIs: To enhance data coverage, we consider integrating third-party trading platforms. The aim is to compare Steam's prices with external market prices (where items can often be traded for real money) to spot cross-market arbitrage. Popular CS:GO skin marketplaces include BitSkins, Buff.163, Skinport, DMarket, etc. For this roadmap, we'll consider **BitSkins** as an example (it's well-known and has an API), and mention others as optional integrations:

- **BitSkins API:** BitSkins provides a RESTful API (requires a free API key and a secret for authentication). Key endpoints useful for price data include:
    - `getAllItemPrices` – returns BitSkins' entire price database for CS:GO items (GitHub - alvinl/bitskins: Unofficial BitSkins API client). This is a bulk endpoint that can provide current prices for all items on BitSkins in one call (making it very powerful for cross-market analysis).
    - `getMarketData` – returns pricing data for up to 250 specified item names in one request (GitHub - alvinl/bitskins: Unofficial BitSkins API client). We can use this to query a targeted list of items (e.g., items that we found promising on Steam) for their current BitSkins lowest listing price, number of listings, etc.
    - Additionally, BitSkins has endpoints for recent sales, buy orders, etc., but for our purposes the main interest is the current sell prices (and possibly buy offers on BitSkins, if any). We will need to authenticate each call (BitSkins uses an HMAC signature with the secret and a time-based code).
- **Other Platforms (Optional):** Depending on how comprehensive we want the system, we can integrate other price sources:
    - *Buff.163:* A large Chinese marketplace with often lower prices. It has no official English API; integration would involve either using their unofficial API (if any) or relying on an aggregator.
    - *Skinport, DMarket, CS.Money:* These have their own APIs (some public, some on request). For instance, Skinport provides an API for market data if you have an account. We might skip direct integration due to complexity and instead use aggregated services.

    - *Aggregators:* Services like **SteamApis.com** or **PriceEmpire** aggregate market data from multiple sources including Steam. SteamApis is paid (but affordable) and returns data in JSON (php - Steam Market API? - Stack Overflow). There are also free APIs like **SkinScout** that provide pricing for all CS:GO items across many platforms (Buff, Skinport, BitSkins, etc.) with a single API token. Using an aggregator could simplify multi-market coverage (one API instead of many), though it introduces an external dependency. For a public open-source tool, we might keep this optional.

- **API Fields of Interest:** For each data source, identify the key fields:
    - Steam priceoverview: `lowest_price`, `median_price`, `volume`.
    - Steam pricehistory: list of `[timestamp, price, volume]` points (we will parse to get historical trends).
    - Steam itemordershistogram: contains arrays or summary of buy orders (price and quantity) and sell orders. Critically, it includes the highest buy order price and lowest sell order price at that moment (often labeled as `highest_buy_order` and `lowest_sell_order` in the JSON).
    - BitSkins: The price data includes the lowest listing price on BitSkins for each item, the quantity available, etc. We might get fields like `price` (or `lowest_price`) and maybe recent sale prices. BitSkins' `getAllItemPrices` likely returns a list of items with fields for recent price, last sale price, etc.
    - Other APIs: If using SkinScout or others, they might return a JSON with Steam price and other market prices in one response, which could directly give us the price difference.

Rate Limiting Considerations: Each API has limitations we must abide by:

- **Steam:** The undocumented endpoints have no official rate limit numbers published, but community experience suggests caution. For example, one developer reported that making ~200 rapid requests to the `priceoverview` endpoint led to an IP block by Valve ([Steam API] - Get all prices in the format .json :: Help and Tips). The market search endpoint (`search/render`) and listing pages are also subject to throttling. Scraping too fast can result in HTTP 429 (Too Many Requests) or temporary bans. One guideline from experience is to limit item listing page fetches to ~5 per minute (Buy and Sell orders of an item in Steam API/JSON - Stack Overflow) if scraping `item_nameid`. However, others found that setting an appropriate `Referer` header (to mimic a normal browser request to the market listing) can significantly increase the allowed request rate,

e.g., fetching hundreds of items with only a 5-second delay each without hitting 429 (Buy and Sell orders of an item in Steam API/JSON - Stack Overflow). We will need to empirically determine a safe frequency – likely spacing requests by a few seconds, and batching where possible (using the `search/render` for bulk data to reduce calls).

- **BitSkins:** The API documentation indicates certain limits (often a few hundred calls per hour, and too many rapid calls might get your key temporarily disabled). BitSkins also has a **rate limit on their price endpoints** (not explicitly stated in the snippet above, but typically they ask for no more than e.g. 1 call per second or so). Because BitSkins allows bulk data (all prices at once), we might only need to call that infrequently (maybe once every few minutes) to update external prices.

- **Other APIs:** If we use an aggregator like SkinScout, we'd need an API key and they might have limits (maybe one call per minute for all data, which is usually sufficient given they aggregate per minute updates (CS2 Skins Pricing API - Pricempire.com)). SteamApis (paid) might allow more frequent queries depending on plan. We will ensure to respect any terms of service and include delays in our code to stay below limits.

Technical Decisions in Phase 2:

- We decide to start with **Steam's own data** for a baseline system. This means using `search/render` to get a broad list of items and their prices on Steam, and possibly `priceoverview` for quick checks on specific items. For precise bid/ask spreads, we plan to utilize `itemordershistogram` with caution (maybe only for targeted items that we suspect have arbitrage, to avoid excessive scraping).

- For **third-party integration**, BitSkins will be our initial external source because of its comprehensive API. We will design our system such that adding another source later is modular (e.g., a generic interface for an "ExternalMarket" so we could plug in others or switch to an aggregator).

- We confirm that we'll use **JSON over HTTP** for data retrieval (no need for WebSockets from Steam since none exist; BitSkins does have a socket for live feed, but we can start with polling REST endpoints).

- We note that some Steam data may require a logged-in session (pricehistory and itemordershistogram endpoints often work better with a Steam login cookie to bypass any age restrictions or CAPTCHA). In development, we might use our own account cookie for this. If so, we'll clearly document that the user may need to provide their Steam session cookies or API key (if Valve ever provides one for market) to unlock full functionality. However, priceoverview and search endpoints usually work without login.

By the end of Phase 2, we will have collected sample API responses, documented the needed parameters and data parsing logic for each endpoint, and possibly written a small **proof-of-concept script** to fetch data for a couple of items to verify access (e.g., get the current Steam price and BitSkins price for a known item to ensure we can compare them). We will also finalize which external APIs to include initially and obtain the necessary API keys.

Phase 3: Implementing Data Retrieval in Python

With knowledge of the APIs, we proceed to implement the data retrieval layer in Python. This will involve writing scripts or modules to query the Steam market and third-party APIs at regular intervals and collect the data needed for arbitrage detection.

Key Tasks:

- **API Client Implementation:** Develop Python functions or classes to call each required endpoint:
    - A function `get_steam_overview(item_name)` that constructs the `priceoverview` URL and returns parsed JSON (lowest price, median, volume).
    - A function `get_steam_history(item_name)` for `pricehistory` (returns a list of (time, price, volume) points).
    - A function `search_steam_market(start=0)` for `search/render` that returns a batch of item results (so we can loop `start=0,100,200,...` to get all items).
    - If doing buy order scraping: a function `get_item_orders(item_nameid)` that calls `itemordershistogram` and returns highest buy and lowest sell prices.
    - BitSkins client: e.g., a function `get_bitskins_prices()` that calls `getAllItemPrices` and returns a dictionary of item -> price, or `get_bitskins_market_data(list_of_items)` for targeted queries.
- **Handling Authentication/Keys:** For Steam, if we require login for some endpoints, use the `requests` session with cookies (possibly loaded from a config where the user can input their Steam login cookies). For BitSkins, implement the required authentication (generate the signature with the secret and pass the API key and signature in the URL, according to BitSkins API docs).
- **Data Parsing:** Parse the JSON responses into Python data structures. For example:

    - Steam `priceoverview` returns prices as strings (e.g., `"$5.34"`). We need to strip the currency symbol and convert to a float (taking care of localization – e.g., some currencies use commas).

    - `search/render` returns a JSON with a list of results under a key like `"results"`. Each result may have fields like `"name"`, `"sell_price"` (in cents) or `"sell_price_text"` (human-readable). We will parse those into a standardized form (e.g., convert prices to floats in a common currency, USD).

    - BitSkins price data likely comes as numeric values (already in USD). We ensure to map item names exactly between Steam and BitSkins (they both use the same item market hash names for CS:GO items).

- **Rate Limiting & Throttling:** Implement safe request pacing. For example:

    - Use Python's `time.sleep()` or `asyncio` to pause between calls. We might introduce a short delay (like 0.5 to 1 second) between sequential Steam requests to avoid hitting limits. If using `search/render` to get 100 items at a time, this significantly reduces total calls (e.g., ~225 requests to cover ~22,500 CS:GO market listings, which can be spread over time).

    - For continuous tracking, consider using a scheduling loop that updates different portions of data in a round-robin fashion. For instance, one cycle could update Steam prices, the next cycle update BitSkins prices, then repeat.

    - If using `itemordershistogram` for many items, be extremely careful: perhaps fetch order book data only for top N items that look profitable from the overview data to confirm the arbitrage spread.

- **High-Frequency Tracking Approach:** To track prices at the highest frequency possible, we might use multithreading or asynchronous IO:

    - Python's `asyncio` with `aiohttp` could allow parallel fetching of many item pages. However, given potential rate limits, we may not actually want true parallelism against Steam (it could backfire by spamming too quickly). Instead, we might limit concurrency (e.g., fetch 5 items concurrently, then wait).

    - We can also spawn separate threads or processes: one thread continuously pulls Steam data, another pulls BitSkins, so they operate in parallel. We must ensure thread-safe data structures if they share memory for storing results.

- **Example Implementation Snippet (Steam Data Fetch):**

```
import requests, time

def get_price_overview(item_name):
    url = ("https://steamcommunity.com/market/priceoverview/?"
           f"appid=730&country=US&currency=1&market_hash_name={item_name}")
    resp = requests.get(url)
    if resp.status_code != 200:
        return None  # handle errors or retries
    data = resp.json()
    # Example of data:
{"success":true,"lowest_price":"$5.34","volume":"1234","median_price":"$5.50"}
    if not data.get("success"):
        return None
    # Parse fields:
    price_str = data.get("lowest_price") or data.get("median_price")
    lowest_price = float(price_str.replace('$','')) if price_str else None
    volume = int(data["volume"].replace(',', '')) if data.get("volume") else None
    median_price = float(data["median_price"].replace('$','')) if data.get("median_price") else
None
    return {"lowest_price": lowest_price, "median_price": median_price, "volume": volume}

# Usage
item = "AK-47 | Redline (Field-Tested)"
overview = get_price_overview(item)
print(f"{item} - Lowest Price: ${overview['lowest_price']}, Volume: {overview['volume']}")
```

This snippet demonstrates basic retrieval and parsing for one item. In practice, we will integrate this into a loop or a scheduler.

- **Bulk Fetch with Search:** Using the `search/render` endpoint to fetch multiple items in one go:

```
def fetch_page(start=0, count=100):
    url = (f"https://steamcommunity.com/market/search/render/?appid=730&norender=1"
           f"&count={count}&start={start}")
    resp = requests.get(url)
    data = resp.json()
    return data

# Example: get first page of results
page0 = fetch_page(0)
total_items = page0["total_count"]
results = page0["results"]
```

```
for item in results:
    name = item["hash_name"]
    price_text = item["sell_price_text"]  # e.g. "$5.34"
    # parse price_text similar to above
```

We would loop over `start` from 0 to `total_count` in steps of 100 to get all items (python - How to fetch total market value of csgo market - Stack Overflow). As one Stack Overflow example noted, we can iterate and aggregate results until we've covered all pages (python - How to fetch total market value of csgo market - Stack Overflow). We will likely store these results incrementally to avoid memory issues with one giant list, and break or sleep if needed between pages.

- **External Data Fetch:** For BitSkins, using their API might look like:

```
import time, hashlib, hmac
import requests

API_KEY = "<your_bitskins_key>"
SECRET = "<your_bitskins_secret>"

def get_all_bitskins_prices():
    # BitSkins requires a signature which is HMAC of secret and a nonce (e.g., timestamp)
    nonce = int(time.time())
    # signature = HMAC_SHA256(secret, nonce) as hex
    sig = hmac.new(SECRET.encode(), str(nonce).encode(), hashlib.sha256).hexdigest()
    url = (f"https://bitskins.com/api/v1/get_all_item_prices/?api_key={API_KEY}"
           f"&app_id=730&code={nonce}&signature={sig}")
    resp = requests.get(url)
    data = resp.json()
    # data likely has a structure with a list of items and prices
    return data.get("prices")
```

We'd adjust according to BitSkins' actual parameter names (this is an illustrative snippet). The result might be a list of items with fields like `"market_hash_name"` and `"price"`. Similar functions would exist for other endpoints if needed.

- **Error Handling:** Implement robust error handling and logging. If a request fails (non-200 or exception), log the error, perhaps retry after a delay. If Steam starts returning captcha or blocked (HTTP 429), back off and reduce frequency. We may use exponential backoff for retries. Also handle JSON decoding errors (Steam sometimes returns HTML on error instead of JSON).

Data Update Loop: Finally, design how data retrieval runs continuously. We could create a loop like:

```
while True:
  update_steam_data()   # fetch latest Steam prices (maybe split into multiple smaller calls)
  update_bitskins_data()  # fetch external prices
  # Mark timestamp of update
  time.sleep(refresh_interval)
```

- The `refresh_interval` might be a few seconds to a minute depending on how aggressive we can be without bans. We might dynamically adjust it if we hit rate limits (e.g., slow down if errors occur). We should also provide a way to stop the loop cleanly (in case the user wants to exit the program).

Performance considerations: Python is relatively slow for very high-frequency tasks, but by leveraging bulk endpoints and keeping network I/O as the bottleneck, we mitigate CPU concerns. The use of C++ comes later for computation – here in data fetching, network latency dominates. We will, however, be careful about memory (if storing data for thousands of items) and use efficient structures (like dictionaries or pandas DataFrames) to hold data. If needed, we might use numpy arrays for large numeric data like price history to speed up computations, but plain Python structures should suffice initially.

By the end of Phase 3, we expect to have a Python module that can retrieve and update pricing data from Steam and BitSkins. We should be able to run it for a short time and see it collecting data (perhaps printing some sample arbitrage candidates to the console as a test). This sets the stage for storing and analyzing the data.

Phase 4: Designing Data Storage and Processing Layer

Now that we can fetch data, we need to store it in a way that allows efficient processing, analysis, and retrieval for our UI. This phase focuses on how to maintain current market state and historical data, and perform any preprocessing needed before passing data to the arbitrage engine.

Key Tasks:

- **Choose a Data Storage Solution:** Decide between in-memory storage vs persistent database:
    - For simplicity and speed, an in-memory approach (using Python data structures) is appealing. For example, maintain a dictionary for current prices: `steam_data[item_name] = {"sell": price, "buy": buy_price, "volume": vol, ...}` and similarly `bitskins_data[item_name] =`

`{"price": price, "quantity": qty, ...}`.

    - However, for historical analysis and persistence (so that data isn't lost on restart), a lightweight database is useful. A good choice is **SQLite**, as it is file-based (no separate server needed) and Python has built-in support. We can create tables like `prices` with columns (`timestamp`, `item_name`, `steam_sell`, `steam_buy`, `bitskins_price`, ...). Each update cycle can insert new rows for changed prices or periodically log snapshots.

    - Alternatively, we can use **pandas DataFrames** to accumulate historical data and then write to disk (CSV or pickle) for persistence (Learning Python – Project 3: Scrapping data from Steam's Community Market | Dr. Blake Porter) (Learning Python – Project 3: Scrapping data from Steam's Community Market | Dr. Blake Porter). For example, maintain a DataFrame of recent price points for each item to compute trends.

    - The system should be kept simple for a local app, so a full-blown database might be optional. We could start with writing periodic JSON or CSV dumps of data (for backup) and keeping live data in memory.

- **Data Model Definition:** Define what information we store for each item:

    - **Static info:** item name, perhaps item category or type if needed (not strictly necessary for price, but maybe for UI grouping).

    - **Steam Market data:** latest lowest sell price, latest highest buy price (if we retrieve it), volume of recent sales (from priceoverview or history), maybe an indicator of price trend (could be computed).

    - **External Market data:** latest price on BitSkins (lowest listing there), volume or quantity available externally, etc.

    - **Derived metrics:** We might compute the spread percentage (`(buy_price - sell_price)/sell_price * 100`) for Steam, and cross-market spread (`Steam price - BitSkins price`). Also, track price changes over time (e.g., difference from an hour ago).

    - **Timestamps:** store when the data was last updated to display freshness and to allow time-based analysis.

- **Processing and Transformation:** Prepare the data for arbitrage logic:

    - Convert all prices to a common currency (likely USD). Steam's `currency` parameter can be set to get USD prices (currency=1 for USD (Buy and Sell orders of an item in Steam API/JSON - Stack Overflow)), so we should consistently use that for ease. If needed, allow configuring currency.

    - Account for fees in data or in the engine? We might not adjust prices for fees until the arbitrage calculation phase, to keep raw data intact. But we should record fee rates (e.g., Steam 15%) in a config.

    - Compute simple indicators: For instance, calculate a rolling average or recent price change for each item to identify *trends*. If the roadmap includes "price movement over time", we may implement logic to detect if an item's price has increased X% in the last Y hours, etc. This could be done by comparing recent entries in the price history.

    - Filter out items we don't care about (maybe extremely low-value items or high-volume cases with minimal margins) to reduce noise in arbitrage detection. This can be a configurable filter (e.g., ignore items under $0.10 or with volume below a certain threshold).

- **Data Structures:** Implement classes or use dataframes to manage data:

    - We might have a class `MarketDataStore` that encapsulates the data structures and provides methods to update them. For example, `store.update_steam(item_name, sell_price, buy_price, volume)` would update the current data and also append to a history log.

    - If using SQLite, define schema and use SQLAlchemy or direct SQL to insert and query. Ensure primary keys (like item_name plus timestamp) and consider indices on item_name for fast lookups.

    - If using pandas, maintain a DataFrame for current state and maybe a dictionary of DataFrames for history per item (but that could be heavy if thousands of items; better to keep one global structure).

- **Example Schema (if using SQLite):** We could have:

    - Table `item` (`id`, `name`, `type`, ...).

    - Table `steam_prices` (`item_id`, `timestamp`, `sell_price`, `buy_price`, `volume`).

    - Table `bitskins_prices` (`item_id`, `timestamp`, `price`, `quantity`).

    - Table `arbitrage_events` (to log when an arbitrage opp was found, optional).

- This normalization separates current vs historical data. Alternatively, a single table combining steam and external prices by timestamp might be used for simplicity.

- **Performance Considerations:**

    - If we track a lot of historical data, the database/file can grow large. We might limit history to a rolling window (e.g., last 7 days) or sample less frequently for older data (maybe store every hour instead of every minute after a day).

    - Querying: The arbitrage engine will need the **latest data** mostly. So we should ensure quick access to the latest price of each item. If using a DB, that might mean querying by max timestamp per item, which could be slow for thousands of items each time. Instead, maintain a separate in-memory cache of latest prices that is updated on each fetch cycle. The DB then is just for persistence and possibly for the UI to query history for charts.

    - Memory: Thousands of items with a few fields each is fine in Python memory. For example, 20,000 items with maybe a dozen fields is trivial (a few MB). History of each item might be larger, but if we store one point per hour for 20k items, that's 480k entries per day, which in a database or file is okay for local use if handled properly.

- **Integrating with Arbitrage Detection:** Decide how the C++ engine will consume data. If the engine expects data in a certain format (like arrays of numbers), we may do a transformation when calling it. Possibly prepare Python lists or numpy arrays of relevant values (e.g., parallel lists: one of Steam buy prices, one of Steam sell prices, one of BitSkins prices, aligned by item index) to pass to C++. This preparation is part of processing.

Example Data Access: If we want to get the latest Steam and BitSkins price for an item for use in arbitrage logic:

```
 # Using in-memory dicts:
steam_price = steam_data.get(item_name, {}).get("sell_price")
steam_buy = steam_data.get(item_name, {}).get("buy_price")
bitskins_price = bitskins_data.get(item_name, {}).get("price")
# If using DataFrame:
latest = current_df[current_df['item_name']==item_name]
steam_price = float(latest['steam_sell'])
```

- We ensure these accesses are efficient (dictionary lookups are O(1), DataFrame filtering is O(n) but we could set item_name as index for O(1) by label).

- **Ensuring Data Freshness:** The data store should track when each item was last updated from each source. If the UI requests a piece of data that hasn't been updated recently, we might trigger an update on the fly (for example, if a user clicks on an item's detail page, we could fetch a fresh pricehistory for that item).

By the end of Phase 4, we will have a module or component that the rest of the system can query for market data. We should be able to do queries like "give me all current prices" or "give me history of item X" easily. This will be tested by perhaps printing out the top 5 items with largest price changes or highest spreads, computed entirely in Python at this stage. We will also have decided on the interface between this layer and the C++ arbitrage engine (what data structure or format we will hand over for processing).

Phase 5: Developing the C++ Arbitrage Detection Engine

In this phase, we implement the core logic that identifies arbitrage opportunities. This engine will be written in C++ for performance, as it may need to compute on large lists of items rapidly and possibly perform frequent updates.

Key Tasks:

- **Define Arbitrage Criteria Clearly:** Decide what constitutes an arbitrage opportunity in our context:

    - **Steam Intra-market Arbitrage:** Identify items where the highest buy order price (bid) is significantly above the lowest sell listing price (ask). A trivial arbitrage is if bid > ask, one could buy the item at the ask and immediately sell to the buy order, pocketing the difference. However, due to Steam's 15% transaction fee on sales, the condition for profit is actually: **(highest_bid * 0.85) > ask_price** (meaning after selling, you get more than what you paid). We might include a small profit margin threshold (to avoid very tiny gains).

    - **Cross-market Arbitrage:** Compare Steam's prices with BitSkins (or others). Two scenarios:

        - Steam price higher, external lower: Buy item on BitSkins (for real money) at price P_ext, then sell on Steam at current buy order price P_steam_buy. Profit if **P_steam_buy * 0.85 > P_ext * (1 + ext_fee)** (ext_fee is external market fee, e.g., BitSkins takes ~5%). This yields Steam wallet profit, which is only useful within Steam unless one is okay

with that.

        - External price higher, Steam lower: Buy on Steam at ask price P_steam_sell, then sell on BitSkins for P_ext. Profit if **P_ext * (1 - ext_fee) > P_steam_sell** (Steam has fee when selling, but here we are removing item from Steam to sell externally, so Steam fee applies on buying? Actually, when buying on Steam you pay the ask price which includes fees to seller, but you as buyer just pay exactly that price). In this scenario, profit would be real money profit on BitSkins after their fee.

        - In practice, scenario (1) is common: people find items cheap on BitSkins to sell on Steam for Steam credit (though that only makes sense if you want Steam funds). Scenario (2) is how a trader could turn Steam items into cash by finding items that are undervalued on Steam relative to cash markets.

    - We will implement detection for both types, but the system can allow the user to focus on one or the other.

    - **Trend-based Opportunities:** Additionally, consider opportunities from price movement. For example, if an item's price has been rising fast (perhaps due to an update or popularity surge), a trader might want to buy before it rises further. While not classical arbitrage, our system could flag items with unusual price increases or volume spikes (a form of market inefficiency exploitation). This could be a later enhancement. Initially, we focus on direct price discrepancies at a single time.

- **C++ Data Structures:** Represent the data in C++ in a convenient form. Perhaps we pass vectors of prices into C++:

    - A `std::vector<double> steam_buy_prices;`

    - A `std::vector<double> steam_sell_prices;`

    - A `std::vector<double> bitskins_prices;`

    - And a corresponding `std::vector<std::string> item_names;` to identify items.

    - Alternatively, pass an array of structures, e.g., `struct ItemPrice { double steam_sell; double steam_buy; double ext_price; };` and a vector of those, aligned with a vector of names or an array inside structure.

    - We might also include volume or other fields if needed for decision (e.g., skip if volume is too low or buy order quantity is zero).

- **Arbitrage Algorithm Implementation:** Iterate over each item's data and check for conditions:

    - Compute Steam spread: `double steam_net_profit = steamBuyPrice * 0.85 - steamSellPrice;` If `steam_net_profit > 0`, record an arbitrage opportunity. We can also compute percentage profit: `(steamBuyPrice*0.85 / steamSellPrice - 1)*100%`.

    - Compute cross-market differences: `double steam_to_ext = steamBuyPrice * 0.85 - extPrice * (1 + extFee);` and `double ext_to_steam = extPrice * (1 - extFee) - steamSellPrice;`. If either is > 0, we have an opportunity (also compute percent gain relative to cost).

    - The algorithm is O(N) for N items per check, which is fine even if N is 20k or more (20k operations in C++ is negligible, and even if we do this every few seconds, it's fine).

    - If we want to be fancy, we could sort items by potential profit to present the top opportunities first, or maintain a max-heap of top K profits as we iterate.

    - We should also consider only listing meaningful opportunities – e.g., require profit > some cents or > some percentage to avoid noise. This threshold can be configurable (maybe default to at least $0.10 profit or 5%).

Output of Engine: The C++ engine could produce a list of arbitrage opportunities. Define a struct for output, e.g.:

```
 struct ArbitrageOpportunity {
    std::string item;
    std::string type; // "Steam" or "Cross"
    double buy_price;
    double sell_price;
    double profit;
    double profit_percent;
};
```

- The engine can return a `std::vector<ArbitrageOpportunity>` (which pybind11 can convert to a Python list of dicts or custom objects). For Steam internal arbitrage,

`type` might be "Steam Market gap"; for cross, maybe "Buy on X sell on Y".

- **Incorporate Historical/Trend Logic (if any):** If we want to include trend-based signals, the engine might also look at the price history (which would be more complex, possibly requiring time-series analysis). A simple approach: check if current price is significantly lower than the moving average over last day – that could indicate a dip that might correct (buy low now). Or if volume spiked 10x, maybe a sudden demand change. These are interesting but can be added later. We mention them as future improvement in documentation, and keep initial engine focused on direct price differences.

- **Code Snippet (C++ logic example):**

```
// Example C++ pseudo-code for arbitrage detection
struct ItemData {
   std::string name;
   double steamSell;
   double steamBuy;
   double extSell;
};

std::vector<ArbitrageOpportunity> findArbitrage(const std::vector<ItemData>& items,
                                    double steamFee=0.15, double extFee=0.05) {
   std::vector<ArbitrageOpportunity> opportunities;
   opportunities.reserve(items.size());
   for (const auto& item : items) {
      double netSteamProfit = 0.0;
      if (item.steamBuy > 0 && item.steamSell > 0) {
         netSteamProfit = item.steamBuy * (1 - steamFee) - item.steamSell;
         if (netSteamProfit > 0.0) {
            double percent = netSteamProfit / item.steamSell * 100.0;
            opportunities.push_back({"Steam", item.name, item.steamSell, item.steamBuy,
netSteamProfit, percent});
         }
      }
      if (item.extSell > 0) {
         // Steam -> External
         if (item.extSell * (1 - extFee) > 0 && item.steamSell > 0) {
            double netExtProfit = item.extSell * (1 - extFee) - item.steamSell;
            if (netExtProfit > 0.0) {
               double percent = netExtProfit / item.steamSell * 100.0;
               opportunities.push_back({"Steam->External", item.name, item.steamSell,
item.extSell, netExtProfit, percent});
            }
```

```
            }
            // External -> Steam
            if (item.steamBuy > 0) {
                double netExtProfit2 = item.steamBuy * (1 - steamFee) - item.extSell;
                if (netExtProfit2 > 0.0) {
                    double percent = netExtProfit2 / item.extSell * 100.0;
                    opportunities.push_back({"External->Steam", item.name, item.extSell,
item.steamBuy, netExtProfit2, percent});
                }
            }
        }
    }
    // Optionally sort opportunities by profit or percent:
    std::sort(opportunities.begin(), opportunities.end(),
              [](auto& a, auto& b){ return a.profit_percent > b.profit_percent; });
    return opportunities;
}
```

This pseudo-code shows the kind of calculations we'll do. The actual implementation will handle corner cases (e.g., missing data fields, or extremely low prices where fees make it irrelevant).

- **Performance and Optimization:** The above loop is very fast in C++. If we had, say, 20,000 items, that loop might take a few microseconds to a millisecond. So performance is not a concern for the computation itself. The reason to use C++ is if we later incorporate more complex analysis or simply to ensure that even if we scale to more frequent or more complex calculations (like Monte Carlo simulations or portfolio optimizations), we have headroom.
    - We will ensure to avoid unnecessary memory allocations in the loop. Using `reserve` on vectors as shown and reusing structures helps.
    - If needed, we can parallelize the loop using OpenMP or C++17 parallel algorithms since each item check is independent. But given the speed, it's probably overkill.
- **Testing the Engine:** We will create test cases for the C++ logic. For instance, feed a small array of ItemData with known prices where we can manually compute expected outcomes, and verify the function returns those opportunities. This can be done by writing a C++ test or by invoking the compiled module from Python in a test script (once integrated via pybind11).
- **Integration Points:** The engine will eventually be called from Python. At this phase, we can also design the interface for that call: likely a function `find_arbitrage(items)`

that Python can call with a list/dict of items. We should ensure the engine doesn't hold global state (just computes on input and returns output) to keep it side-effect-free and easy to use.

By the end of Phase 5, we will have a C++ implementation of the arbitrage detection logic, tested in isolation. However, it won't be usable by the Python code yet; that's handled in the next phase.

Phase 6: Integrating Python and C++ Components

With the Python data layer and the C++ engine developed, we now integrate them using pybind11 so that Python code can invoke C++ functions seamlessly. This phase covers building the C++ code into a Python module and ensuring data can flow between Python and C++ efficiently.

Key Tasks:

- **Set up pybind11 Build System:** We can use **pybind11** to create a Python extension module in C++.

Create a C++ source file (e.g., `arb_module.cpp`) that includes pybind11 headers and wraps our functions. For example:

```
 #include <pybind11/pybind11.h>
#include <pybind11/stl.h>
struct ArbitrageOpportunity { std::string type, item; double buy_price, sell_price, profit,
profit_percent; };
std::vector<ArbitrageOpportunity> find_arbitrage_py(const std::vector<double>& steam_buy,
                                        const std::vector<double>& steam_sell,
                                        const std::vector<double>& ext_price,
                                        const std::vector<std::string>& names) {
   // ... call the internal findArbitrage logic, constructing ItemData from the vectors ...
}
PYBIND11_MODULE(arbitrage, m) {
   pybind11::class_<ArbitrageOpportunity>(m, "ArbitrageOpportunity")
      .def_readonly("type", &ArbitrageOpportunity::type)
      .def_readonly("item", &ArbitrageOpportunity::item)
      .def_readonly("buy_price", &ArbitrageOpportunity::buy_price)
      .def_readonly("sell_price", &ArbitrageOpportunity::sell_price)
      .def_readonly("profit", &ArbitrageOpportunity::profit)
      .def_readonly("profit_percent", &ArbitrageOpportunity::profit_percent);
   m.def("find_arbitrage", &find_arbitrage_py, "Find arbitrage opportunities");
```

}

- This snippet exposes a `find_arbitrage` function to Python and a custom class for opportunities. We also tell pybind11 how to convert `std::vector<ArbitrageOpportunity>` to a Python list of `ArbitrageOpportunity` objects.

- Alternatively, we could skip making a class and just return a list of Python dicts. Pybind11 can automatically convert a vector of `std::tuple` or `std::dict` if we construct those in C++. But defining a structure as above is cleaner for typed access.

- We need to include `<pybind11/stl.h>` to automatically handle standard containers like `std::vector` and `<pybind11/complex.h>` if we had complex types, etc. For our use, `stl.h` is enough to handle strings and vectors.

- **Compilation:** Create a build configuration:

Option 1: Use a simple **setup.py** with pybind11's helper. Pybind11 provides a `pybind11.setup_helpers` that can be used to easily compile the module. For example:

```
 # setup.py
import pybind11, setuptools
from pybind11.setup_helpers import build_ext, Pybind11Extension
ext_modules = [
    Pybind11Extension("arbitrage", ["arb_module.cpp"], cxx_std=17),
]
setuptools.setup(
    name="arbitrage",
    ext_modules=ext_modules,
    cmdclass={"build_ext": build_ext}
)
```

- Running `python setup.py build` will compile the C++ code into a `.so` (or `.dll` on Windows, `.dylib` on Mac) which can be imported in Python.

- Option 2: Use **CMake**. We can have a CMakeLists.txt that finds pybind11 and builds the module. This is useful if the project grows. For a simple case, setup.py might be quicker. However, since we want to ensure it's easy for users, using pip to build might be okay. We should document the build steps clearly.

    - We must ensure to compile for macOS properly (matching the Python version's architecture). On macOS, if using default system Python or Homebrew Python, just running the above will likely use clang and produce the correct output. We should test on a Mac environment.

- **Data Transfer Efficiency:** Passing data from Python to C++:

    - Using standard types (lists of floats, list of strings) is straightforward, but for large lists the conversion overhead is something to consider. Pybind11 will convert Python lists to `std::vector` by copying each element. If we have thousands of items, copying two or three lists of floats (for prices) is trivial (say 20k floats -> negligible overhead). If it were millions, we might worry, but here it's fine. If needed, we can accept numpy arrays in C++ (pybind11 can map them without copy via `<pybind11/numpy.h>`). For now, vectors are fine.

    - Another approach: do the filtering in Python and only send the subset of items that meet a rough criteria to C++ for detailed check. But since C++ check is so fast, we might just send everything every time.

    - Memory management: pybind11 handles reference counting and creation of Python objects for us. We just need to ensure we return things in a way that Python can own them (i.e., no static global structures that might cause issues).

- **Integration in Code Flow:** Modify the Python data layer to call the C++ function:

After each data update cycle, or on demand when the UI requests to show opportunities, call something like:

```
 import arbitrage  # the pybind11-compiled module
# Prepare data lists
names = []
steam_sell = []
steam_buy = []
ext_prices = []
for item, info in steam_data.items():
    names.append(item)
    steam_sell.append(info.get("sell_price", 0.0))
    steam_buy.append(info.get("buy_price", 0.0))
    ext_price = external_data.get(item, {}).get("price", 0.0)
    ext_prices.append(ext_price)
opportunities = arbitrage.find_arbitrage(steam_buy, steam_sell, ext_prices, names)
# 'opportunities' is a list of ArbitrageOpportunity objects (from pybind11)
```

    -
    - We might wrap the above logic in a Python function `detect_opportunities()` to keep things clean. That function prepares the data and returns a Python-friendly list (perhaps converting the pybind11 objects to Python dicts or DataFrame for easier manipulation).

- **Testing Integration:** Once compiled, we test calling the C++ function from Python with known inputs. For example, create a small dummy dataset in Python where we know one should be arbitrage (like steam_buy=10, steam_sell=5 -> definitely arbitrage) and verify the output object shows that item with correct profit calculation.

- **Cross-Platform Issues:** Ensure that pybind11 and the C++ code compile on macOS. MacOS might require special flags (e.g., `-mmacosx-version-min=10.14` or similar for compatibility) and to produce a `.dylib` with correct naming. Pybind11's build helpers usually handle it. Test the build on Mac early (perhaps using GitHub Actions CI on macOS).

    - Also, if we intend the repo to be public and easy, we might even pre-build the module for macOS (and possibly Windows/Linux) and include wheels or at least instructions like "pip install ." to build from source.

- **Code Maintenance:** Document the C++ code and create a clear separation: the pure logic (which could be in a separate `.cpp` file without pybind11 includes) and the pybind11 glue (in another `.cpp`). This way, the logic can be unit-tested in C++ or even reused elsewhere. The glue is only for Python integration.

- **Example pybind11 usage snippet (for demonstration in documentation):**

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
using namespace pybind11::literals;  // for _a literal if needed

// Suppose we have a function defined elsewhere:
std::vector<ArbitrageOpportunity> findArbitrage(const std::vector<ItemData>& items);

// We create a pybind11 wrapper that accepts Python lists and returns Python list of dicts
py::list find_arbitrage_py(py::list names, py::list steam_buy, py::list steam_sell, py::list ext_price)
{
    size_t n = py::len(names);
    std::vector<ItemData> items;
    items.reserve(n);
    for (size_t i = 0; i < n; ++i) {
        ItemData d;
```

```
        d.name = names[i].cast<std::string>();
        d.steamBuy = steam_buy[i].cast<double>();
        d.steamSell = steam_sell[i].cast<double>();
        d.extSell = ext_price[i].cast<double>();
        items.push_back(d);
    }
    auto result = findArbitrage(items);
    py::list pyList;
    for (auto& opp : result) {
        py::dict opp_dict;
        opp_dict["item"] = opp.item;
        opp_dict["type"] = opp.type;
        opp_dict["buy_price"] = opp.buy_price;
        opp_dict["sell_price"] = opp.sell_price;
        opp_dict["profit"] = opp.profit;
        opp_dict["profit_percent"] = opp.profit_percent;
        pyList.append(opp_dict);
    }
    return pyList;
}

PYBIND11_MODULE(arbitrage, m) {
    m.doc() = "Arbitrage detection module";
    m.def("find_arbitrage", &find_arbitrage_py, "Find arbitrage opportunities");
}
```

In practice, we might not manually iterate Python lists like that since `std::vector` binding can be automatic. But this demonstrates how we convert data and return results as Python dicts for convenience. We would choose an approach and implement accordingly.

- **Memory and Performance:** We should test how fast the integration call is. Converting 20k items to C++ and back each time is likely fine (maybe a few milliseconds overhead). If it becomes a bottleneck, we can optimize by filtering items (only pass items above a certain price, for example) or by using numpy arrays for direct memory access. But given this is a local analytics tool, a few milliseconds is nothing noticeable to the user, especially if we update the UI every few seconds at most.

By the end of Phase 6, the Python code will be able to get arbitrage results from the C++ engine. We should be able to run the system (data fetch + engine) and log outputs like "Item X: buy at $Y on Steam, sell at $Z on BitSkins, profit = $N". This integrated functionality is the core of the application.

Phase 7: Building a Localhost Web UI (Flask/FastAPI)

With the backend in place, we create a user interface to present the market analytics and arbitrage opportunities. The UI will run on a local web server and allow users to view data in their browser.

Key Tasks:

- **Select Web Framework:** We choose Flask or FastAPI for the web server. Both are suitable; Flask is minimal and straightforward for serving HTML pages, while FastAPI is powerful for building APIs (and could serve a JavaScript front-end). We'll proceed with **Flask** for simplicity, using it to render templates and serve a few endpoints.

- **Design the User Interface:** Determine what pages and data visualizations are needed:

    - A **dashboard page** showing a summary of the market and top arbitrage opportunities. This could include a table of the top N arbitrage opportunities with columns (Item, Opportunity Type, Buy Price, Sell Price, Profit, Profit%) and maybe color-coding high profits. It could also show some key stats like how many items are being tracked, last update time, etc.

    - A page or modal for **item details**: if a user clicks on a particular item, show its price history chart over time, and maybe current order book (if we have that data). We can plot historical prices using a JavaScript chart library (like Chart.js) or serve a pre-rendered image of a chart (matplotlib) if needed. But interactive JS charts on a webpage would be nicer.

    - A section for **market trends**: perhaps charts for overall market indices (e.g., total market value of all items over time if we computed that) or top gainers/losers in price.

    - **Controls**: if needed, allow the user to input settings (like filter thresholds, which external markets to include, refresh frequency) either via a config file or a simple form on the page.

- **Implement Flask Routes:**

`GET /` -> render the main dashboard template. In the Flask view function, we gather data from the Python data store and engine:

```
 @app.route('/')
def dashboard():
    opps = detect_opportunities()  # function that calls arbitrage engine and returns a list/dict
```

```
top_opps = sorted(opps, key=lambda o: o['profit_percent'], reverse=True)[:20]
last_update = data_store.last_update_time
return render_template('dashboard.html', opportunities=top_opps, last_update=last_update)
```

    - The template `dashboard.html` would loop over `opportunities` and display them in a table. It might also include some script to periodically refresh part of the page (using AJAX).

    - `GET /item/<name>` -> show detail for a given item. The view will fetch that item's history from the database or data store and pass it to the template. Possibly, it could also call the Steam API for live order book on demand (if we want to show current buy/sell orders graph).

    - `GET /api/opportunities` -> (optional) provide a JSON API for the opportunities. This can be used by a front-end script to refresh data without full page reload. For example, we could have a small JavaScript snippet on the dashboard that polls this endpoint every 30 seconds and updates the table dynamically.

    - `GET /api/history/<name>` -> returns historical price data for an item in JSON (to plot on client side).

    - Static files (CSS, JS, images) – set up Flask to serve these or use a CDN for libraries (like Bootstrap CSS and Chart.js).

- **UI Design and Visualization:**

    - Use **Bootstrap 5** (a popular CSS framework) to make the site look clean without much custom CSS. This gives us responsive tables, navbars, etc., easily.

    - **Tables:** Present arbitrage opportunities in a table. Perhaps highlight positive profits in green, or sort by profit%. Could allow clicking column headers to sort (Bootstrap or a small JS library like DataTables can make the table sortable and searchable).

    - **Charts:** Use Chart.js or similar to plot price history. For instance, on the item detail page, include a canvas for a line chart of price over time (with time on X axis and price on Y). We feed it the data from `/api/history/<name>` endpoint via an AJAX call in JavaScript. Alternatively, we can embed the data in the page using a script tag generated by Flask (pre-populating a JavaScript array with the data).

    - **Analytics displays:** Perhaps include a chart of "market index" – e.g., sum of prices of a basket of popular items over time – just as a fun metric. Or a bar chart

of the number of arbitrage opportunities in different categories.

    - Ensure the UI is not too cluttered: start with the essential info (the arbitrage table and maybe one chart).

- **Update Mechanism:** Decide how the page updates with new data:
    - Easiest is to have the user manually refresh the page to see updated data. But a better UX is auto-refresh:
        - We can use meta refresh (HTML) to reload the whole page every X seconds, but that's flickery.
        - Instead, use AJAX requests. For example, use JavaScript `setInterval` to call our `/api/opportunities` and then update the table DOM.
        - Or use WebSocket (Flask can work with SocketIO) to push updates, but that's more complex and probably unnecessary for this scope.
    - Given this is a local tool, polling every 30s or 60s is fine. We can calibrate this with the data refresh frequency (if our backend fetch loop runs every 10s, maybe update UI every 10s or 30s).
- **Concurrency:** Running Flask while the data fetching loop is running in the background means we need to ensure they don't conflict:
    - One approach: Run the data fetch loop in a **separate thread** (or as a background asyncio task) started when the Flask app starts. The global data store is then updated in that thread. Flask route handlers (in the main thread) will read from the data store. This requires using thread-safe data structures or simple locking. We can use Python's `threading` and perhaps a Lock around updates to ensure consistency when reading/writing shared data. If the update interval is short, maybe just minor risk of a read catching half-updated data. Using a lock or copying data for reading can solve it.
    - Alternatively, run the data fetch on-demand in each request (not ideal for real-time, as that would make the page load slow and not continuous).
    - Or use an in-memory database like SQLite in WAL mode that both threads can access safely.

The simplest: Start a thread:

```
 from threading import Thread
def background_fetch():
    while True:
        update_all_data()
        time.sleep(10)
Thread(target=background_fetch, daemon=True).start()
app.run(...)
```

- The daemon thread will update data continuously while Flask serves requests.

- **Example Template Snippet (dashboard.html):**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>CS:GO Arbitrage Dashboard</title>
  <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css">
</head>
<body class="p-3">
  <h1>CS:GO Market Arbitrage Opportunities</h1>
  <p>Last update: {{ last_update }}.</p>
  <table class="table table-striped table-hover" id="opps-table">
    <thead>
      <tr>
        <th>Item</th><th>Type</th><th>Buy Price ($)</th><th>Sell Price ($)</th><th>Profit
($)</th><th>Profit (%)</th>
      </tr>
    </thead>
    <tbody>
      {% for opp in opportunities %}
      <tr>
        <td><a href="{{ url_for('item_detail', name=opp.item) }}">{{ opp.item }}</a></td>
        <td>{{ opp.type }}</td>
        <td>{{ "%.2f"|format(opp.buy_price) }}</td>
        <td>{{ "%.2f"|format(opp.sell_price) }}</td>
        <td class="{% if opp.profit > 0 %}text-success{% else %}text-danger{% endif %}">{{
"%.2f"|format(opp.profit) }}</td>
        <td>{{ "%.1f"|format(opp.profit_percent) }}%</td>
      </tr>
      {% endfor %}
```

```
  </tbody>
 </table>
 <script>
  // Simple auto-refresh using fetch API:
  setInterval(function(){
   fetch("/api/opportunities").then(response => response.json()).then(data => {
    // Update table rows (for brevity, not fully implemented here)
    // You would loop through data and update the DOM accordingly.
   });
  }, 30000);
 </script>
</body>
</html>
```

This shows how we might render the initial table and set up periodic refresh. We'd flesh out the JS to properly update each row or rebuild the table.

- **Visualization for item detail:** On `/item/<name>`, we might include:

```
<canvas id="priceChart" width="600" height="400"></canvas>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
 fetch("/api/history/{{ item_name }}").then(res => res.json()).then(data => {
  const ctx = document.getElementById('priceChart').getContext('2d');
  new Chart(ctx, {
   type: 'line',
   data: {
    labels: data.timestamps,  // array of time labels
    datasets: [{
     label: 'Price ($)',
     data: data.prices,
     borderColor: 'rgba(75,192,192,1)',
     fill: false
    }]
   },
   options: {
    scales: {
     x: { type: 'time', time: { unit: 'day' } },
     y: { beginAtZero: false }
    }
   }
  });
 });
```

</script>

We assume `/api/history/<name>` returns a JSON with `timestamps` (in a format Chart.js can parse, maybe ISO strings) and `prices`.

- **Web Design Considerations:** Ensure the site is mobile-friendly by using Bootstrap's responsiveness (tables will scroll on small screens). Use intuitive formatting (currency with 2 decimals, percentages with 1 decimal). Possibly use abbreviations for thousand separators if needed (like 1,200 becomes 1.2k) for volume if we display it.

- **User instructions:** Since this is local, in the README we'll instruct the user to run the Flask app (e.g., by running a script `run_app.py` that starts Flask after starting background threads). Then they open `http://localhost:5000` in their browser to view the dashboard.

- **Testing the UI:** Manually test in a browser that pages load and data appears. We should also test edge cases like no opportunities (the table should say "None" gracefully) or extremely large values (to see if formatting holds). Use sample data if needed to simulate scenarios.

By the end of Phase 7, we have a functioning web interface that displays arbitrage opportunities and possibly other analytics. The data is updated in near real-time on the page. This makes the system user-friendly, allowing interactive exploration of the market data our system gathers.

Phase 8: Testing, Deployment, and Continuous Improvement

In the final phase, we rigorously test the system, prepare it for release, and outline future improvements.

Testing:

- **Unit Testing:** Write unit tests for individual components:

    - Python API client functions (using dummy or cached responses to verify parsing logic). For example, test that a known JSON from Steam is parsed into the correct numerical values.

    - The C++ arbitrage function (could be tested from Python after integration: give it a controlled dataset and check the output).

    - Utility functions (like any price conversion or data filtering logic).

- **Integration Testing:** Run the whole pipeline in a controlled scenario:

    - Possibly create a simulation mode where instead of hitting real APIs (which can be slow or variable), we use recorded data. For example, store a snapshot of Steam market data in a JSON file and have the data retrieval read from that. Then see if the engine picks up expected opportunities.

    - Test the multi-threading (if using it): ensure that data updates while the web page is being accessed do not cause crashes or inconsistencies. This might involve stress testing by rapid refreshes and simultaneous data fetch.

- **Performance Testing:** Although this is a local tool, ensure that it runs efficiently:

    - Time how long a full cycle takes (data fetch + processing). If it's too slow relative to desired frequency, identify bottlenecks. Perhaps scanning 22k items on Steam might take a while (especially if rate-limited to, say, 5 pages per minute, that would take ~45 minutes for all items – in which case we adjust strategy to focus on top items, etc.). We might simulate higher loads or adjust frequencies to find a good balance between coverage and update speed.

    - Memory usage: track memory while running for a long time to catch any leaks (especially since we use C++ and Python interplay, ensure no memory is leaking in the pybind usage – typically it's fine if we don't allocate globally).

- **User Acceptance Testing:** If possible, have a beta tester or the project owner run it on their Mac to see if setup is easy and everything works as expected in a real environment.

Deployment:

- **Documentation:** Finalize README.md with clear setup and usage instructions:

    - Prerequisites: e.g., “Install Python 3.x, C++ compiler, and ideally create a virtual environment.”

    - Steps to install: possibly `pip install -r requirements.txt` to get Flask, requests, pybind11, etc. Then build the C++ extension. We could simplify this by making the package pip-installable (so `pip install .` runs the setup.py to compile and install the module automatically).

    - How to run: e.g., `python run_app.py` or `flask run` depending on how we set it up.

    - How to configure: Explain where to put API keys (maybe in a config file or environment variables). For example, instruct: "Copy `config_example.py` to `config.py` and put your BitSkins API key and secret there, as well as Steam currency code, etc."

    - Troubleshooting: Note common issues like "If compilation fails on Mac, ensure Command Line Tools are installed (run `xcode-select --install`)." If the user doesn't want to compile, we might provide a pre-compiled wheel for macOS if possible.

    - Security note: Since this is local and uses your API keys, advise to keep them safe and not expose the running app to the internet.

- **Publishing the Repository:** Push to GitHub (or similar). Ensure to include a permissive license if open source (MIT or Apache, etc.) as appropriate.

    - Include the link to BitSkins API documentation and any references in the README if it helps users.

    - Possibly include some screenshots of the UI in the repo to showcase it.

- **Packaging (Optional):** If targeting more users, we might publish it on PyPI for easy install. But because of the C++ component, building wheels for multiple platforms is extra work. Initially, focusing on macOS support as required, we can at least provide the source. A Homebrew formula could even be written to install it, but that's beyond scope unless specifically desired.

Continuous Improvement:

- Now that the MVP is done, consider enhancements for future versions:

    - **Additional Marketplaces:** Integrate more third-party platforms (Buff.163, DMarket, etc.) either through their APIs or via the SkinScout aggregator. This would provide more arbitrage paths (e.g., Buff vs BitSkins vs Steam triangular arbitrage). We designed the system with a modular data fetch, so adding another source is feasible by extending Phase 3 and 4 components.

    - **Real-Time Data Streams:** Instead of polling, use event-driven updates where possible. For example, if BitSkins provides a websocket for new listings or sales, connect to it to get instant updates on price changes. Similarly, one could

imagine using Steam's unofficial websocket (some have used Steam client APIs or firehose data, though that's complex) – likely not available, so polling is fine for Steam.

- **Machine Learning and Analytics:** Leverage the collected historical data to predict or highlight trends. For instance, build a model to predict if a price spike is temporary or will continue, helping to decide if an arbitrage is worth pursuing. Or cluster items by their price movement patterns. These would be separate analysis modules feeding into the UI (like "Trending Now" items).

- **User Alerts and Notifications:** Add functionality for the user to set alerts (e.g., notify if an arbitrage profit exceeds $1 or 10%). The app could then push a desktop notification or simply highlight those in the UI.

- **Better UI/UX:** Possibly create a richer single-page application using a framework like React or Vue for a more dynamic interface. This could communicate with our Flask (or convert to FastAPI) backend via REST or GraphQL. While overkill for now, it might be nice if the project grows.

- **Windows/Linux support:** Though we focused on macOS, ensure the code is portable. Pybind11 and C++17 are cross-platform, and Python code is portable. We might test on Windows/Linux and add any required instructions (like how to compile with MSVC on Windows). This expands the user base.

- **Optimization:** If the Steam data fetch for all items is too slow due to rate limits, implement strategies like prioritizing high-value items. For example, maintain a list of items above a certain price or volatility and update them more frequently, while updating cheaper, stable items less frequently. This way the most likely profitable items are always fresh.

- **Robustness:** Add more robust error recovery. If Steam temporarily blocks us, perhaps auto-switch to a secondary IP (if possible) or just wait and resume. Log these events so the user knows (maybe display "Data temporarily throttled" in UI if it happens).

- **Testing and CI:** Set up continuous integration to run tests on each commit, and possibly build the module for macOS to catch any compilation issues early.

- **Feedback Loop:** Since the repository is public, gather user feedback (via issues) for new features or bug fixes, and iterate.

Finally, ensure that running the app for long periods is stable. We might run it for several hours and monitor memory/cpu to ensure there are no leaks or runaway processes.

Final Deployment Checklist:

- All secrets/keys are in config and not hard-coded (and gitignored if necessary).
- Tests are passing.
- The user can install and run with minimal steps (ideally a one-command installation or a very short sequence).
- The documentation is clear on how to use the tool and what each component does.

Upon completing Phase 8, we will have a polished, publicly available arbitrage analysis tool. Users on macOS can clone the repository, build it, and within minutes have a local dashboard of CS:GO market opportunities. This concludes the roadmap, with a system that is functional and a foundation laid for future enhancements.

Overall, this phased approach ensures that we build the system iteratively, validating each part (data collection, processing, engine, UI) before moving to the next, which reduces complexity and results in a robust final product. The combination of Python and C++ gives us both flexibility and performance, and the use of a web UI makes the insights accessible and user-friendly.

Sources

Citations


steamcommunity.com
[Steam API] - Get all prices in the format .json :: Help and Tips
community market has no real open usable api.


steamcommunity.com
[Steam API] - Get all prices in the format .json :: Help and Tips
I created a website on which you can go and see how much all of the items in your inventory, the price of each item moved through - http://steamcommunity.com/market/priceoverview/?currency=5&country=us&appid=730&market_hash_name=Chroma%203%20Case&format=json

blakeporterneuro.com

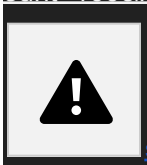
Learning Python – Project 3: Scrapping data from Steam's Community Market | Dr. Blake Porter

Yes, the pricehistory request will give you the median sale price and volume (number of items sold) every day. For closer time points (within the last few weeks) it will give more temporal details (eg median sale price and volume every 3 hours).

steamcommunity.com

[Steam API] - Get all prices in the format .json :: Help and Tips

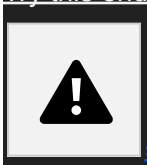
https://steamcommunity.com/market/search/render/?query=appid%3A730&start=0&count=100&norender=1



stackoverflow.com

Buy and Sell orders of an item in Steam API/JSON - Stack Overflow

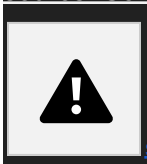
Try this endpoint: `https://steamcommunity.com/market/itemordershistogram`



stackoverflow.com

Buy and Sell orders of an item in Steam API/JSON - Stack Overflow

Just for example parameters, my `country`, `currency`, and `language` are `US`, `1`(USD), and `english`. `two_factor` can always be set to `0`, and `item_nameid` is what defines which item you're looking up. There's not an easy way to find it, but you can scrape it from here `https://steamcommunity.com/market/listings/<appid>/<market_hash_name>` for each item you need. You can find it passed as an argument to the `Market_LoadOrderSpread` function in the page source. Scraping the ids from this page though is rate limited to ~5 requests per minute.



stackoverflow.com

Buy and Sell orders of an item in Steam API/JSON - Stack Overflow

way to find it, but you can scrape it from here `https://steamcommunity.com/market/listings/<appid>/<market_hash_name>` for each item you need. You can find it passed as an argument to the `Market_LoadOrderSpread` function in the page source. Scraping the ids from this page though is rate limited to ~5 requests per minute.

github.com

GitHub - alvinl/bitskins: Unofficial BitSkins API client

Name Description Reference getAccountBalance() Returns your account balance. Link getAllItemPrices() Returns the entire price database used by Bitskins. Link getMarketData(options) Returns basic pricing data for up to 250 `market_hash_name`'s that are currently on sale. Link getAccountInventory(options) Returns your account's available inventory on Steam, Bitskins and pending withdrawals. Link getInventoryOnSale(options) Returns your inventory currently on sale.

github.com

GitHub - alvinl/bitskins: Unofficial BitSkins API client

getAllItemPrices() Returns the entire price database used by Bitskins. Link getMarketData(options) Returns basic pricing data for up to 250 `market_hash_name`'s that are currently on sale. Link getAccountInventory(options) Returns your account's available inventory on Steam, Bitskins and pending withdrawals. Link getInventoryOnSale(options) Returns your inventory currently on sale.

stackoverflow.com

php - Steam Market API? - Stack Overflow

1

steamcommunity.com

[Steam API] - Get all prices in the format .json :: Help and Tips

But it's not so good + my website is blocked by ip, after about 200 requests to this link.

stackoverflow.com

Buy and Sell orders of an item in Steam API/JSON - Stack Overflow

I found out a bug in my code. For using this endpoint it's crucial to use the `Referer` header, and in my code, I was using `Refer` header, so just by adding two letters and setting the `Referer` header to the value `https://steamcommunity.com/market/listings/${appId}/${itemName}` I was able to fetch with cooldown for 5-sec hundreds of unique items without catching `429` So sharing this info here, because it took me weeks to find out what was wrong

pricempire.com

CS2 Skins Pricing API - Pricempire.com

CS2 Skins Pricing API - Pricempire.com Get regular price updates across all supported marketplaces with reliable data delivery. 1-minute price updates; Current market prices; 24/7 data collection ...

stackoverflow.com

python - How to fetch total market value of csgo market - Stack Overflow

data = requests.get("https://steamcommunity.com/market/search/render/?search _descriptions=0&sort_column=name&sort_dir=desc&appid=730&norender=1&count=100&st art=0") json_data = json.loads(data.text) print(json_data)

blakeporterneuro.com

Learning Python – Project 3: Scrapping data from Steam's Community Market | Dr. Blake Porter

Don't forget to let the program (and Steam's servers) take a quick nap with time.sleep() again! Once all the items' data have been collected, we can pickle the allItemsPD as a .pkl file so we don't need to do this arduous data collection again (unless you need updated price information). Pickle will save your data in the Pandas dataframe format so when you load it again, all your data is still a dataframe.

blakeporterneuro.com

Learning Python – Project 3: Scrapping data from Steam's Community Market | Dr. Blake Porter

allItemsPD.to_pickle(gameID+'PriceData.pkl');

stackoverflow.com

Buy and Sell orders of an item in Steam API/JSON - Stack Overflow

Just for example parameters, my `country`, `currency`, and `language` are `US`, `1`(USD), and