A Framework for Generating a Sophisticated Crypto P&L Analysis Tool via Large Language Models

Section 1: Deconstruction of Core Technical Requirements

This document outlines a comprehensive set of specifications for generating a sophisticated, production-quality Python script for cryptocurrency portfolio analysis. The objective is to produce a detailed prompt for a Large Language Model (LLM) tasked with emulating a senior Python engineer. The final deliverable must be a single, self-contained Python file that provides a Profit & Loss (P&L) overview for a Bitvavo account, incorporating robust API interaction, accurate accounting logic, a user-friendly command-line interface (CLI), and an embedded test suite.

1.1 Interfacing with the Bitvavo API: A Protocol for Robust Data Retrieval

The script's foundation is its ability to communicate reliably with the Bitvavo exchange. This requires a multi-faceted approach encompassing secure authentication, precise endpoint usage, and resilient data retrieval strategies.

1.1.1 Authentication and Security

The script must interface with the Bitvavo API exclusively through the official python-bitvavo-api library. Security is paramount; therefore, API credentials (APIKEY and APISECRET) must not be hardcoded. The script must retrieve these keys from the operating system's environment variables using the os.environ.get() method. This practice prevents accidental exposure of sensitive credentials in source code. The script's user-facing documentation should clearly instruct users on how to set these variables and stress the importance of creating a read-only API key. Adhering to the principle of least privilege by using a read-only key minimizes potential damage if the key were ever compromised.

A critical, yet often overlooked, aspect of the Bitvavo API is its time-sensitive authentication mechanism. Each authenticated request requires a Bitvavo-Access-Timestamp, a Unix timestamp in milliseconds, and is only considered valid within a specific access_window (typically 10 seconds). A request arriving outside this window will be rejected with error code 304, indicating the request was not received in time. This can lead to intermittent, difficult-to-diagnose authentication failures caused by clock drift between the user's machine and Bitvavo's servers.

To engineer a resilient solution, the script must proactively mitigate this issue. Before making any authenticated calls, the client should first query the public /time endpoint to fetch the official server time. By comparing this server time to the local system time, the script can calculate a precise time offset. This offset must then be applied to the timestamp of every subsequent

authenticated request, ensuring all calls are perfectly synchronized with the server clock and eliminating a potential point of failure.

1.1.2 Required API Endpoints and Wrapper Methods

To construct a complete P&L overview, the script must retrieve three distinct types of data using specific methods from the python-bitvavo-api library. The following table specifies the necessary API interactions.

Table 1: Bitvavo API Endpoint Specification

Endpoint Path	python-bitvavo-api Method	HTTP Method	Key Parameters	Purpose in Script	Rate Limit Weight
/time	time()	GET	(none)	Clock synchronization to prevent authentication errors.	1
/balance	balance()	GET	symbol (optional)	Fetch current holdings of all assets for unrealized P&L.	5
/{market}/trades	trades()	GET	market, limit, tradeIdTo	Fetch all historical transactions for FIFO calculation.	5
/ticker/price	tickerPrice()	GET	market (optional)	Get current market prices to value holdings.	1

This structured specification ensures the correct data sources are used for each part of the calculation: balance() and tickerPrice() for the current state and unrealized P&L, and trades() for the historical data needed for realized P&L.

1.1.3 The Criticality of a Robust Pagination Strategy

A significant challenge in retrieving historical data is handling API pagination correctly. The trades endpoint returns a maximum of 1000 records per request. A naive implementation that simply calls the trades() method once will fail to retrieve a user's complete history if they have more than 1000 trades, leading to grossly inaccurate P&L calculations.
To guarantee the retrieval of all transactions, the script must implement a diligent pagination loop. The Bitvavo API facilitates cursor-based pagination using the tradeIdTo parameter. The correct procedure is as follows:

1. Make an initial request to the trades endpoint for a specific market without the tradeIdTo parameter to get the most recent batch of up to 1000 trades.

2. If the returned list of trades is not empty, store these trades.
3. Identify the id of the *last* trade in the received batch.
4. Enter a loop that continues as long as the API returns data. In each iteration, make a new request to the trades endpoint, this time passing the ID from the previous step as the tradeIdTo parameter.
5. The loop terminates when the API returns an empty list, signifying that all historical trades for that market have been fetched.

This logic must be encapsulated in a dedicated function, such as get_all_trades(market), to ensure every asset's full transaction history is processed.

1.1.4 Proactive Rate Limit Management

The Bitvavo API imposes a weighted rate limit of 1000 points per minute per API key. Exceeding this limit results in a temporary API key ban, identified by error code 105. The pagination loop, especially when processing multiple assets with extensive trade histories, can easily trigger this limit.

A senior-level engineering approach is not to simply react to a 429 "Too Many Requests" error, but to prevent it entirely. The python-bitvavo-api library provides the getRemainingLimit() method, which returns the number of remaining weight points. The script must integrate this check into its data retrieval functions. Before each API call within the pagination loop, the script should query the remaining limit. If the limit falls below a conservative threshold (e.g., 10 points, accounting for the weight of the upcoming call), the script must pause execution (e.g., using time.sleep()) for a short duration to allow the rate limit counter to reset. This proactive management ensures the script runs smoothly without interruption, regardless of the user's trading volume.

1.2 The First-In, First-Out (FIFO) P&L Algorithm: An Accounting Specification

The core of the P&L calculation relies on the First-In, First-Out (FIFO) accounting method. This method requires meticulous tracking of purchase lots to accurately determine the cost basis for each sale.

1.2.1 Core FIFO Logic

The FIFO principle mandates that when an asset is sold, the cost basis is derived from the oldest-held units of that asset. To implement this efficiently, the script must maintain a separate queue of purchase lots for each cryptocurrency. The ideal data structure for this is collections.deque, as its append() (for buys) and popleft() (for sells) operations have an average time complexity of O(1), making it highly performant.

Each element within the deque will represent a single purchase lot and should be stored as a dataclass or namedtuple for clarity and immutability. This PurchaseLot object must contain at least three fields: quantity (as a Decimal for precision), price (as a Decimal), and timestamp.

1.2.2 Handling Partial Fills and Lot Depletion

Cryptocurrency transactions are inherently fractional, which adds complexity to the FIFO

calculation. A single sale can consume several purchase lots or only a fraction of a single lot. For instance, consider a sale of 1.5 BTC when the purchase deque contains [PurchaseLot(quantity=1.0,...), PurchaseLot(quantity=2.0,...)]. The algorithm must:

1. Process the sale against the first lot in the deque (popleft). Since the sale quantity (1.5) is greater than the lot quantity (1.0), this entire lot is consumed. The realized P&L for this portion is calculated.
2. The remaining sale quantity (0.5 BTC) is then processed against the next lot in the deque.
3. This second lot (2.0 BTC) is only partially consumed. The algorithm must not simply discard it. Instead, it must *update* this lot's quantity in-place, reducing it by 0.5 BTC to 1.5 BTC.
4. This updated lot is then placed back at the *front* of the deque (appendleft) to be the next available lot for future sales.

This logic of iterating through the purchase deque, fully or partially depleting lots, and correctly modifying the last-touched lot is critical for maintaining an accurate cost basis over time.

1.2.3 Differentiating Realized and Unrealized P&L

A comprehensive P&L overview must distinguish between realized gains (profits locked in from sales) and unrealized gains (on-paper profits of current holdings).

- **Realized P&L** is calculated *only* at the moment of a sale. It is the total proceeds from the sale minus the summed cost basis of the FIFO-determined purchase lots that were consumed.
- **Unrealized P&L** is a snapshot of the current portfolio value. It is calculated for all remaining lots in the purchase deques. For each lot, the unrealized P&L is (Current Market Price - Purchase Price) * Remaining Quantity.

The script must implement this logic in two distinct functions: one for calculating realized P&L by processing the full trade history, and another for calculating the unrealized P&L of the final holdings. The final report presented to the user must clearly display both metrics for each asset.

1.3 Designing a User-Centric Command-Line Interface (CLI)

The script must be controlled via a clean and intuitive CLI built with Python's standard argparse module. The interface should be simple, focusing on core functionality like filtering and accessing embedded documentation.

A common requirement is to allow users to filter the report for a specific list of assets, such as BTC,ETH,ADA. The most robust and pythonic way to parse such an argument is not with nargs, which can be clumsy, but by defining a custom type function for the argument. A simple lambda function, type=lambda s: [item.strip().upper() for item in s.split(',')], will instantly convert the comma-separated input string into a clean list of uppercase asset symbols, handling whitespace and ensuring consistent casing.

The following table outlines the required CLI arguments.

Table 2: CLI Argument Design

Argument	Flag(s)	argparse dest	argparse action/type	Help String	Example Usage
Assets Filter	--assets	assets	type=lambda s: [i.strip().upper()	Filter report for a	script.py --assets

Argument	Flag(s)	argparse dest	argparse action/type	Help String	Example Usage
			for i in s.split(',')]	comma-separated list of assets (e.g., BTC,ETH).	BTC,ETH
Run Tests	--run-tests	run_tests	action='store_true'	Run the embedded unit test suite instead of the P&L report.	script.py --run-tests
Show README	--show-readme	show_readme	action='store_true'	Display the embedded README.md content and exit.	script.py --show-readme
Show Requirements	--show-requirements	show_reqs	action='store_true'	Display the embedded requirements.txt content and exit.	script.py --show-reqs

1.4 Ensuring Code Integrity with an Embedded, Self-Contained Test Suite

The constraint of a single-file deliverable requires an unconventional but elegant approach to testing and documentation.

1.4.1 The Single-File Architecture

To embed a test suite within the main script file, the script's entry point (if __name__ == "__main__":) must act as a dispatcher. It will first parse the command-line arguments. If the --run-tests flag is present, it will dynamically import the pytest library and execute it against the script file itself using pytest.main(['-v', __file__]). If the flag is absent, it will proceed with the main P&L reporting logic.
Similarly, the content for the README.md and requirements.txt files will be stored as large, triple-quoted multiline string constants within the script. If the --show-readme or --show-requirements flags are used, the script will print the corresponding constant and exit.

1.4.2 Mocking Strategy with pytest-mock

Unit tests must be deterministic and isolated from external dependencies; they must not make live API calls. The pytest-mock plugin is required for this purpose. The correct mocking strategy is to patch the methods on the Bitvavo client object itself (e.g., bitvavo.trades, bitvavo.balance), not the lower-level requests library calls. This makes the tests resilient to changes in the underlying python-bitvavo-api library's implementation. The tests must use mock data that

reflects the real API response structure to validate the FIFO and P&L logic against known inputs and expected outputs.

Table 3: Mock Data Schema for Unit Tests

Mocked Method	Return Data Structure	Key Fields	Example Value
bitvavo.time()	dict	time	{'time': 1672531200000}
bitvavo.balance()	list[dict]	symbol, available, inOrder	``
bitvavo.trades()	list[dict]	id, timestamp, side, amount, price, fee	[{'id': '...', 'timestamp':..., 'side': 'buy',...}]
bitvavo.tickerPrice()	list[dict]	market, price	``

1.4.3 Custom Exception Handling

The official python-bitvavo-api library does not appear to expose a rich hierarchy of custom exceptions, often returning a generic error. However, the API's JSON error responses contain valuable details, including a specific errorCode and message. To provide superior user feedback, the script should define its own small hierarchy of custom exceptions (e.g., BitvavoAPIException as a base class, with subclasses like InvalidAPIKeyError and InsufficientFundsError). All API-calling functions must wrap their calls in a try...except block. In the except block, the code should parse the error details from the underlying exception and re-raise one of the more specific, informative custom exceptions. This try-parse-reraise pattern translates cryptic API errors into actionable feedback for the user.

Section 2: Conclusions

The framework detailed above provides an exhaustive blueprint for prompting an LLM to generate a high-quality, senior-level engineering artifact. By specifying not just *what* to build, but *how* and *why*, the prompt guides the model toward a solution that is secure, robust, accurate, and user-friendly. Key technical directives—such as proactive clock synchronization, diligent pagination, rate-limit management, precise FIFO logic for fractional trades, and a self-contained testing architecture—elevate the expected output from a simple script to a production-grade tool. This level of detail is essential for leveraging LLMs to create complex software that correctly handles the nuances and potential failure points of real-world systems.

Works cited

1. python-bitvavo-api 1.1.0 - PyPI, https://pypi.org/project/python-bitvavo-api/1.1.0/ 2. python-bitvavo-api - PyPI, https://pypi.org/project/python-bitvavo-api/ 3. Python - Hiding the API Key in Environment Variables - Network Direction, https://networkdirection.net/python/resources/env-variable/ 4. API Keys, Authentication, Environment Variable, and Sending SMS in Python - Medium, https://medium.com/@saifulj1234/api-keys-authentication-environment-variable-and-sending-sms-in-python-434dc96a3aec 5. Python wrapper for the Bitvavo API - GitHub, https://github.com/bitvavo/python-bitvavo-api 6. Authentication | Welcome to Bitvavo docs,

https://docs.bitvavo.com/docs/authentication/ 7. API error messages - Bitvavo Help Center, https://support.bitvavo.com/hc/en-us/articles/10122205636753-API-error-messages 8. API structure | Welcome to Bitvavo docs, https://docs.bitvavo.com/docs/rest-overview/ 9. Get account balance | Welcome to Bitvavo docs, http://docs.bitvavo.com/docs/rest-api/get-account-balance/ 10. Get trades | Welcome to Bitvavo docs, http://docs.bitvavo.com/docs/rest-api/get-trades/ 11. Get ticker prices | Welcome to Bitvavo docs, http://docs.bitvavo.com/docs/rest-api/get-ticker-prices/ 12. Releases · bitvavo/python-bitvavo-api - GitHub, https://github.com/bitvavo/python-bitvavo-api/releases 13. Bitvavo API (2.1.2), https://wikibitimg.fx994.com/attach/2020/10/938355101/WBE938355101_39902.pdf 14. Handle errors | Welcome to Bitvavo docs, https://docs.bitvavo.com/docs/errors/ 15. FIFO (First-In-First-Out) approach in Programming - GeeksforGeeks, https://www.geeksforgeeks.org/dsa/fifo-first-in-first-out-approach-in-programming/ 16. Calculating Profit Using FIFO / LIFO / AVERAGES (CRYPTOCURRENCIES), https://community.tableau.com/s/question/0D54T00000C6AcDSAV/calculating-profit-using-fifo-lifo-averages-cryptocurrencies 17. Python Class for LiFo, FiFo and AVCO Accounting of bonds - Github-Gist, https://gist.github.com/mdriesch/da65371d512791f8ca6d2b3827cef38c 18. FIFO class in python library? - Stack Overflow, https://stackoverflow.com/questions/62978186/fifo-class-in-python-library 19. Argparse - Python - mkaz.blog, https://mkaz.blog/working-with-python/argparse 20. argparse — Parser for command-line options, arguments and subcommands — Python 3.13.5 documentation, https://docs.python.org/3/library/argparse.html 21. Python's argparse and lists - Tuxevaras Blog, https://www.tuxevara.de/2015/01/pythons-argparse-and-lists/ 22. argparse action or type for comma-separated list - Stack Overflow, https://stackoverflow.com/questions/52132076/argparse-action-or-type-for-comma-separated-list 23. pytest-mock Tutorial: A Beginner's Guide to Mocking in Python - DataCamp, https://www.datacamp.com/tutorial/pytest-mock 24. Testing APIs with PyTest: How to Effectively Use Mocks in Python - CodiLime, https://codilime.com/blog/testing-apis-with-pytest-mocks-in-python/ 25. Python unit testing with Pytest and Mock | by Brendan Fortuner - Medium, https://medium.com/@bfortuner/python-unit-testing-with-pytest-and-mock-197499c4623c 26. how to mock the response from a library api in pytest - Stack Overflow, https://stackoverflow.com/questions/72748698/how-to-mock-the-response-from-a-library-api-in-pytest