ChatGPT

Multichain Wallet Scanner Architecture

Overview

This system allows users to scan multiple blockchain networks for addresses with activity by generating or providing cryptographic keys. It consists of a **Next.js frontend** for user interaction and a **FastAPI backend** for heavy lifting. The workflow is: - **User Input:** The frontend offers a mode selector for **hexadecimal range input** or **random key generation**. - **Key Generation:** The backend expands short hex inputs into full 256-bit (64 hex chars) keys or generates random keys on the fly. These keys are converted into hierarchical deterministic (HD) master keys (xprv/tprv) and their extended public key formats (xpub/ypub/zpub for mainnet, tpub/upub for testnet) [1]. - **Address Derivation & Scan:** Child addresses are derived from the extended keys for each supported blockchain (e.g. Bitcoin, Ethereum). The backend asynchronously queries external APIs (Infura, Tatum, Blockchair, etc.) to check each address for transaction history or balances. Only addresses with any activity (transactions or non-zero balance) are collected as results. - **Real-time Updates:** As scanning progresses, results are streamed back to the frontend in real-time (via WebSockets or Server-Sent Events) so the user sees live status updates and any “hits” (active addresses found). - **Logging:** All scan jobs and findings are logged to a MongoDB database in real-time. Metadata like input type (range or random), source chain, keys derived, addresses found, and timestamps are stored for monitoring and future analysis [2].

This architecture ensures a responsive UI and a scalable backend that can handle potentially large key scans without blocking, using fully asynchronous processing.

Frontend: Mode Selection and Input

On the frontend (built with Next.js + React), the user can choose the scan mode via a dropdown or button group: - **Hex Range Input Mode:** Allows input of a hex pattern or range. For example, the user might enter a short hex like “`222`” or “`aaa`” as a prefix. The frontend will send this to the backend as the start of a range or pattern. The backend will expand it to a full 64-character hex key (padding or repeating as needed) before scanning [3]. Alternatively, the user can provide explicit start and end hex values to define a range to scan (inclusive). The UI will have two input fields for start and end when "Range" mode is selected. - **Random Key Mode:** No hex input is needed. The user simply initiates scanning, and the backend will generate random 64-character hex private keys on the fly. - **Specific Key Mode:** (Optional) The user can input a specific 64-hex private key or an existing extended public key (xpub/ypub/etc). This mode directly uses the provided key for scanning (useful for rescanning known keys).

The frontend includes a **“Start Scan”** button to kick off the process and a **“Stop”** button to cancel an ongoing scan. When the scan starts, the UI may display a modal or a scrollable output area where results will appear in real-time. Key info (like the hex key or xpub being scanned) could be highlighted for clarity.

Additionally, a small status display or progress bar shows how many addresses have been scanned and how many with activity have been found (e.g. "**10,000 scanned / 6 found**"). This gives immediate feedback on progress.

Backend API Endpoints and Job Initiation

The FastAPI backend provides endpoints to control the scanner: - **POST** `/start_scan`**:** Initiates a new scan job. The request can include form data for `mode` (random or range or specific). In range mode, `start` and `end` hex values are provided; in specific mode, an `input` (hex key or xpub) is provided; in random mode, no additional input is needed. The handler logic: 1. If a scan is already running, respond with an error/status to prevent overlapping jobs. 2. Determine the key source: - For `mode=random`: generate a new random 64-hex key (256-bit) [4] [5]. - For `mode=range`: take the `start` and `end` hex, convert them from hex string to integers, and ensure `start <= end`. Initialize the scan at the start value (converted back to 64-char hex) [6]. (The range will be handled either by repeated calls or streaming, explained later.) - For `mode=specific`: check the `input`. If it's a hex string (<=64 chars), use it as the key; otherwise assume it's an extended key (like an xpub) [7]. 3. If a raw hex key was determined, the backend derives the full set of extended keys. This uses an HD key derivation function that produces all relevant extended private keys (xprv, yprv, zprv, etc.) and extended public keys (xpub, ypub, zpub, tpub, upub) from the 32-byte seed [1]. If the provided hex was shorter than 64 chars, it's zero-padded on the left to 64 chars [3] (ensuring it represents a 256-bit value). If the hex is invalid or yields an invalid key (e.g. 0 or >= curve order), the scan is aborted with an "invalid key" status. 4. Determine the primary extended public key to scan. For Bitcoin-like chains, the first available xpub/ypub/zpub is used (preferring the standard format – e.g., xpub for Bitcoin mainnet) [8] [9]. For Ethereum, an `xpub` is still derived but will be used to generate Ethereum addresses (since Ethereum uses the same elliptic curve, we can repurpose the xpub for deriving ETH addresses). 5. Instantiate a **Scanner** object with appropriate parameters (max address gap, concurrency level, follow depth, chain type) [10]. The scanner is the core worker that will derive addresses and check them. 6. Log the scan initiation in MongoDB: insert a record into a `keys` (or `scans`) collection with details like timestamp (`started_at`), input type/mode, the chain being scanned, the hex or xpub, and the derived extended keys [11]. (This record contains the master key info and acts as an audit of the scan) [2]. 7. Launch the scan job asynchronously. This is done by scheduling an `asyncio` task for the scanner (e.g. `scan_task = asyncio.create_task(scanner.scan_xpub(xpub))`) [12]. The `start_scan` endpoint immediately returns a JSON response indicating the scan has started along with some key info (like the hex and derived addresses of index 0) for display [13] [14]. Importantly, the HTTP response is not held open – the actual scanning continues in background. - **POST** `/stop_scan`**:** Allows the user to stop an ongoing scan. This will signal the scanner to stop (setting a flag) and await the task cancellation [15]. The endpoint responds with a simple "stopped" status. The front-end can call this when the user clicks a "Stop" button. - **GET** `/stats`**:** Returns current statistics about the scan (in-memory stats from the Scanner instance) as JSON [16]. This can include fields like `addresses_scanned`, `active_addresses_found`, `with_balance_found`, etc., updated in real-time. For example, `addresses_scanned` counts how many addresses have been checked so far [17] and `active_addresses` how many had transactions (tx_count > 0) [18]. The frontend can poll this if WebSockets are not used, to update progress. - **GET** `/metrics`**:** (Optional) Provides aggregated metrics, potentially from the database, like total addresses scanned with >1 tx, >2 tx, etc., and historical usage [19]. This is more for analytics and not critical to the scanning itself. - **GET** `/range_stream` **and** `/random_stream`**:** These endpoints (if using Server-Sent Events) stream a continuous sequence of generated keys. In the provided design, `/range_stream` takes `start` and `end` hex params and

iterates from start to end, for each value deriving keys and returning data as an event stream [20] [21]. Similarly, `/random_stream` generates an indefinite sequence of random keys (or a fixed count) and streams results [22] [23]. Each event contains the same JSON data that a single scan would return (key info, addresses, etc.). These SSE endpoints provide an alternative to the explicit `start_scan` for continuous scanning modes. If using pure WebSocket, these might not be needed – instead the WebSocket connection can handle continuous data.

Frontend initiation: When the user clicks "Start", the frontend will call the appropriate endpoint based on mode: - For *range* and *random* modes, the Next.js app can establish a streaming connection. In a pure WebSocket approach, the client could send a message like `{action: "start_range", start: <hex>, end: <hex>}` or `{action: "start_random"}` to a WebSocket endpoint, and the server will respond by sending a stream of results. If using SSE (as in the current design), the frontend opens an `EventSource` to `/range_stream` or `/random_stream` [24] [25]. - For a single *specific* key scan, the frontend can simply POST to `/start_scan` and then listen on the WebSocket or poll for results.

The key point is that the **job initiation is separate from result retrieval** – the POST starts the process, and then results are pushed asynchronously to the client.

Key Generation and HD Derivation

Once a hex seed is determined (from user input or random generation), the backend converts it into an HD master key: - **Hex to HD Key:** The 64-char hex string (256-bit) is interpreted as a big integer. If the integer is 0 or not in the valid range for secp256k1 (greater than curve order), it's rejected [26]. Otherwise, it is used as the private key for the master node. The system computes the corresponding public key (by multiplying the secp256k1 generator point by the private key) [26] [27]. In BIP32, a master key also has a chain code – in this implementation a dummy chain code (32 bytes of 0x00) is used if not deriving from a mnemonic seed [28]. - **Extended Key Formats:** Using the private key, the system constructs extended keys in multiple formats: - **xprv/xpub:** Mainnet Bitcoin extended private/public key (BIP32, P2PKH standard) [29]. - **yprv/ypub and zprv/zpub:** Extended keys for Bitcoin segwit accounts (BIP49/BIP84 – ypub for P2SH-wrapped segwit, zpub for native segwit) [30]. These use different version bytes but represent the same underlying key material. - **tprv/tpub and uprv/upub:** Testnet equivalents of x/y/z, used if scanning on test networks [31]. - The code maps each prefix to the correct version bytes and constructs the Base58Check-encoded string for each [32]. All extended keys are derived at once for completeness. - **Address Calculation:** The master public key (index 0) is also used to compute a default address for convenience: - For Bitcoin, the P2PKH address (legacy) is computed by hashing the compressed pubkey (SHA-256 then RIPEMD-160) and adding the network byte 0x00 + checksum [33] [34]. This gives a standard Bitcoin address (base58) for the master key. - For Ethereum, the address is computed by taking the Keccak-256 hash of the uncompressed public key and using the last 20 bytes (40 hex chars) with `0x` prefix [35] [36]. These initial addresses (e.g., `btc_address` and `eth_address`) are returned in the initial response for quick reference [33] [35]. In a range or random scan scenario, each generated key will have its first address precomputed like this so the user immediately sees if that root address had activity. - **HD Child Derivation:** The scanner will derive child addresses from the extended **public** key for each index incrementally. We use non-hardened child derivation (so that we can derive from the xpub alone, without needing the private key). In practice, the scanner creates an `XpubContext` object from the chosen xpub/ypub/zpub and target chain [37] [38]. This context knows how to derive sequential addresses: - **Bitcoin Paths:** It infers the address type from the xpub prefix (e.g., ypub -> P2SH-SegWit, zpub -> Bech32 SegWit, xpub or tpub -> P2PKH) [39]. For each child index, it performs the

elliptic curve point derivation (using HMAC-SHA512 with the chain code and index) to get the next public key point [40] [41]. Then it encodes that point into the appropriate address format (base58 for P2PKH/P2SH or bech32 for SegWit) [42] [43]. - **Ethereum Paths:** The same xpub is used; the context treats it as an Ethereum-type (if `chain=="eth"` was set) so that `AddressIndex(n)` will return an Ethereum address [44] [45]. Internally it computes the uncompressed pubkey for index *n* and hashes it for the address [45] [46]. (Ethereum uses a derivation path like m/44'/60'/0'/0/n for accounts – here we implicitly treat the xpub as m/0/n for simplicity, which is acceptable for scanning all possible addresses). - The result is that for any given key, the system can derive a sequence of addresses on each chain. In summary, the backend takes the input, **derives the master HD key and various extended formats, and prepares to generate child addresses for scanning across relevant blockchains** [47].

Asynchronous Scanning and Address Monitoring

The core scanning logic is handled by the `Scanner` class running asynchronously. Key points of the scanning process and architecture: - **Concurrent Workers:** The scanner uses an internal task queue and multiple worker coroutines to fetch address data in parallel. For example, it might launch 16 worker tasks (default) to consume addresses from a queue [48] [49]. This concurrency allows many address checks to happen simultaneously, maximizing throughput when IO-bound by external API calls. - **Address Queue:** Initially, the queue is seeded with addresses derived from the primary xpub. The scanner will derive addresses sequentially (index 0, 1, 2, ...) and push them to the queue for workers to process. It does this in batches for efficiency – e.g., fetch 100 addresses at a time from the HD context to an internal list (`_addr_cache`) to avoid computing keys one-by-one [50]. - **Gap Limit:** The scanner implements a *gap limit* (common in HD wallet scanning). It keeps track of how many consecutive addresses have been seen with no activity. If `max_gap` (e.g. 20) empty addresses in a row are reached, it stops deriving further addresses [51] [52]. This prevents scanning an infinite range when it's unlikely any further addresses have funds (a reasonable assumption from BIP44 standard gap limit). The gap counter resets whenever an address with activity is found [52]. - **Address Handling:** Each worker takes an address from the queue and checks it: - If the target chain is **Bitcoin (or UTXO-based)**: It calls the provider's API to fetch the full transaction history for that address [53] [54]. For example, using Tatum's `GET /bitcoin/transaction/address/{address}` endpoint which returns a list of transactions (paged). The scanner will retrieve all pages so it gets the complete history [54] [55]. Alternatively, if configured, it might use Blockstream's API which provides UTXO and transaction lists [56] [57]. The result is a list of transactions (`txs`). The scanner then computes: - `tx_count = len(txs)` (number of transactions involving this address) [58] [59]. - `balance` by summing the output values to this address (and subtracting any spent outputs if we had that data). In this implementation, a simpler approach is used: whenever the address appears as an output in a transaction, add the output value if it's indeed to this address [60]. (This gives total received; tracking spends would require more data or using the Blockchair summary API). - `next_addrs`: in Bitcoin, if the scanner is in recursive mode (follow_depth > 0), it will also collect any new addresses found in the outputs of this address's transactions [61]. These could be change addresses or addresses this one sent funds to. All such addresses not seen before are added to the queue for further scanning [62]. This is a **recursive scan** feature that can trace through the transaction graph up to a certain depth. - If the chain is **Ethereum (account-based)**: It uses Infura's JSON-RPC to get basic info – specifically `eth_getBalance` and `eth_getTransactionCount` for the address [63] [64]. This gives the current balance and the number of transactions sent from that address (nonce). If either value is > 0, the address has activity. Full transaction history isn't fetched here (as that's more involved), but a non-zero tx count or balance indicates the address is active. The scanner sets `tx_count` and `balance` accordingly [65] [66]. (If needed, an Ethereum scan could be extended via a block explorer API to list transactions, but Infura

provides a quick lightweight check). - **Other Chains:** The architecture is extensible – additional provider APIs can be integrated similarly (e.g., for Litecoin, BSC, etc., using Tatum or Blockchair for multi-chain support). The scanner design abstracts the data fetch with a `fetch_transactions` function and could route to different APIs based on chain. - **Updating Stats:** As each address is processed, the scanner updates its stats: - It increments the `addresses_scanned` counter [17]. - If an address had any transactions (`tx_count > 0`), it increments `active_addresses` and also increments counters for thresholds: e.g., if >1 tx, >2 tx, etc., these are tallied in `tx_gt_1`, `tx_gt_2`, etc. [18] [67]. If the address had a balance >0, it increments a `with_balance` counter [68]. - These stats are available via the `/stats` or `/metrics` endpoints for a real-time or summary view. - **Batching DB Writes:** The scanner accumulates results and periodically writes to the MongoDB. Each address result (address, tx_count, balance, etc., plus maybe a timestamp) is added to an in-memory batch [69]. After a batch reaches a certain size (e.g., 50), it performs a bulk insert into the `addresses` collection [70] [71]. This reduces the overhead of writing to the database for each address, which is important when scanning thousands of addresses quickly. - **Follow Depth:** If configured to follow outputs (for UTXO chains), the scanner keeps track of depth (distance from the original xpub's addresses). It only enqueues new addresses while `depth < follow_depth` [62]. This prevents an infinite crawl of the transaction graph. For example, with `follow_depth=2`, it will scan addresses that the original addresses sent money to (depth 1) and the ones those addresses sent to (depth 2), but no further. - **Stopping Conditions:** The scan will stop when: - The gap limit condition is met (for sequential HD derivation). - The user manually stops the scan (which sets a flag that breaks out of loops on the next check) [72] [73]. - The range (if doing a finite range scan via SSE) is exhausted (e.g., reached the `end` value). - All queued addresses (including followed addresses) are processed and queue is empty. - **Cleanup:** After stopping, the scanner cancels any still-running worker tasks and flushes any remaining batch to the DB [74]. This ensures all data up to the stop point is saved. The backend `scan_task` completes at this point.

All of the above scanning logic runs asynchronously without blocking the main server thread. FastAPI (with Uvicorn/Starlette) handles other requests or websocket communication in the meantime. Thanks to `asyncio`, the scanner can await network calls, allowing other operations to interleave.

Real-Time Updates via WebSockets

To provide live feedback, the backend uses WebSockets (or SSE as an alternative) to push scanning results to the frontend in real-time. This avoids requiring the client to continuously poll for updates. Here's how it can be structured: - **WebSocket Endpoint:** The FastAPI app defines a `/ws` endpoint for websocket connections. When the Next.js frontend opens a websocket (e.g., using `WebSocket` API or Socket.IO client), the server accepts the connection and keeps it open [75] [76]. We maintain a set of connected clients (or a dict if we want to address them individually) [77] [78]. - **Streaming Results:** As the scanner finds addresses with activity or updates its progress, it sends messages to the client. There are a few ways to implement this: - **Direct Push on Events:** The scanner could be passed a callback or have access to a websocket connection to send data whenever an interesting event occurs. For example, right after an address is found to have `tx_count > 0`, we could send a JSON message to the client with that address and its details. Similarly, after every X addresses scanned, send an update with the new count. - **Broadcast Loop:** Another approach is to have the scanner continuously report stats (say every second) over the socket. The FastAPI app could spawn a background task that periodically checks `scanner.stats` and sends a JSON message to all clients with the current `addresses_scanned` and `active_addresses` counts. - **Using MongoDB Change Streams:** (Advanced) If all results are logged to MongoDB, we could tap into MongoDB Change Streams to get notified when a new address document with activity is inserted [79] [80].

This was the approach in a similar real-time dashboard example, where changes in the database trigger websocket messages [81] [82] . In our case, this might be overkill, but it's an option if we wanted the DB to be the single source of truth. - **Message Format:** We use JSON messages for the websocket. A message could look like:

```
{
  "event": "address_found",
  "address": "<address>",
  "chain": "<chain>",
  "tx_count": 5,
  "balance": 0,
  "index": 42
}
```

for a found address, or

```
{ "event": "progress", "addresses_scanned": 10000, "active_found": 6 }
```

for a periodic progress update. The frontend can update the UI accordingly (e.g., append a new row in the results table for an address_found, or update a counter display for progress). - **Client-Side Handling:** On the Next.js side, we can use the WebSocket API inside a React effect hook. For example, open the socket in a useEffect when the component mounts or when a scan starts. Handle incoming messages by updating React state (like an array of found addresses, and progress numbers). If using Socket.IO, the mechanism is similar but with its event-based API (subscribe to events). - **Closing Connection:** When the scan finishes or is stopped, the server can send a final message (e.g., `event: "done"`) or simply close the socket. The client should handle the socket closing by maybe showing a "Scan complete" status. If the user navigates away or cancels, the client should close the socket from its side to free resources. - **Alternate SSE:** If WebSockets are not desired, Server-Sent Events can be used (as demonstrated in the implementation). SSE is one-way (server->client) but for our use-case that's sufficient. The `/range_stream` and `/random_stream` endpoints already use SSE to push JSON text events continuously [24] [25] . The frontend can attach an `EventSource` listener and update the DOM in the `onmessage` callback. The provided code shows parsing the JSON and appending it to an output log in real-time [83] [84] . The choice between WebSocket vs SSE might depend on need for bi-directional communication or preference; WebSockets offer more flexibility (client can send commands like pause/resume, which in the code is handled by closing/re-opening the SSE). - **Live UI Updates:** Regardless of transport, the user sees a live feed of results. Each result entry could be a small card or table row showing: - *Address* (with maybe a copy button), - *Chain/Type* (e.g., "BTC SegWit" or "ETH"), - *Tx Count* and *Balance* (in native units or converted to a human-readable format), - *Status* (for instance "active" if tx_count > 0, or the scanner might only send active ones to begin with). Possibly, the UI could highlight addresses that have funds or many transactions. In the code, they even triggered a sound (`ding()`) if an address with transactions was found [84] [85] – indicating a "hit" to the user. - **Progress Display:** The UI can also show an updating counter or progress bar. If using websockets, the progress messages can drive this. If not, a fallback could be a setInterval that polls `/stats` every second to update the numbers (the code's `refresh()` function does exactly that for health/stats/metrics endpoints) [86] . The combination of push for important events and occasional polling for aggregate stats can make the UI both reactive and accurate.

By using WebSockets, we ensure the system is **fully asynchronous end-to-end** – the backend pushes data as soon as it's available, and the frontend updates the view without user refresh. This real-time flow is crucial for a good user experience in a scanner that may check thousands of keys (the user shouldn't have to guess if something is happening). As noted in similar real-time systems, WebSockets are ideal for streaming live updates from backend to frontend [87].

Logging and Data Storage (MongoDB)

All scans and their outcomes are stored in a MongoDB database for persistence and analysis: - **Scan Jobs (Keys Collection):** Each initiated scan can be logged as a document in a collection (call it `keys` or `scans`). The document would contain: - Timestamp of start (and end if completed). - Input type and values: e.g. `mode: "random"` or `"range"`, the hex seed or range that was used, or the specific xpub provided. - Derived master keys: the xprv/xpub (or tpub, etc.) that was actually scanned. In the current implementation, they store all variants of extended keys for reference [2]. This can be useful for later reproducing the scan or extending it. - Perhaps an array of addresses found (or count summary). However, since addresses are being logged in another collection, this might be kept minimal in the scan document. - Chain info: which chain(s) were scanned. - Any performance stats (total addresses scanned, etc.) if we want to log that at end. - **Addresses Collection:** Each address scanned (especially those with activity) is stored as a document in an `addresses` collection [2]. Fields might include: - The address string, - Chain (e.g., `"BTC"` or `"ETH"`), - `tx_count` (number of transactions found for that address), - `balance` (in satoshis or wei, or null if not applicable), - `last_seen` timestamp (when we found activity or scanned it), - Possibly a reference to which scan/job it came from (could store the `_id` of the scan document to tie them together, though the current code does not do this explicitly). - `provider_data` or `raw_txs`: The raw response from the provider for that address, which might include transaction hashes or other details [88] [89]. This is useful for later inspection. In the implementation, they saved the entire list of transactions (or Infura info) under a field so that no data is lost [90]. Storing raw data can increase DB size, so one might choose to store only summary info and keep raw data out or in a separate collection. - **Batch Inserts:** As mentioned, addresses are inserted in batches for efficiency [70] [69]. MongoDB can handle bulk inserts well, and this also reduces the impact on the event loop. - **Indexing:** We would likely index the addresses collection on fields like `address` and `chain` (to ensure uniqueness or quick lookup), and maybe on `tx_count` or `last_seen` if doing queries like "find all active addresses" or recent activity. - **Real-time Logging:** Because we insert as we scan, the database is getting updated in real-time. This means we could have another service or just use Mongo's data for live dashboards. In a real deployment, one could use MongoDB Change Streams as hinted, to watch the `addresses` collection and broadcast changes [79] [80]. But since we already push via WS, that's redundant for the user-facing side. - **Example Document:** A sample address document might look like:

```
{
  "address": "1BoatSLRHtKNngkdXEeobR76b53LETtpyT",
  "chain": "BTC",
  "tx_count": 12,
  "balance": 0.0054321,
  "last_seen": 1693580340,
  "provider_data": [ { "txid": "...", "value": ..., ... }, ... ]
}
```

And a corresponding scan document:

```
{
  "started_at": 1693580300,
  "mode": "range",
  "start_hex": "000000...0000",
  "end_hex": "000000...0FFF",
  "chain": "BTC",
  "xpub": "xpub6CUGRUonZSQ4TWtTMmzXdrXDtyPWKi...W",
  "zpub": "zpub6n.....",
  "eth_address": "0x1234...abcd",
  "status": "completed",
  "addresses_scanned": 4096,
  "active_addresses": 3
}
```

This schema can be adjusted, but the idea is to retain enough info to resume or analyze scans later. In the provided code, the `keys` collection holds the extended keys and even the first few addresses' history (under "Tatum Data", "Infura Data", etc., which contain what was found for index 0 of each type) [91] [92]. This is more of a diagnostic snapshot per scan.

By logging to MongoDB, we gain persistence (the scanning results don't vanish if the server restarts) and a way to aggregate or query historical data. For instance, one could easily query "how many addresses with balance were found in the last week" by looking at the `addresses` collection with a date filter. The **metadata** such as source (which API was used), input type, etc., can also be stored; for example, one could add a field `input_type: "random"` or `"range"` in the scan document, and maybe `source_api: "tatum"` vs `"blockstream"` if that varies (the code uses an env var to switch BTC provider) [93].

Importantly, writing to the database is done **in real-time as scanning progresses**, not just at the end. Each address result is inserted right after it's obtained (batched) [88] [89]. This means if a scan is long-running, one can monitor the database for partial results. It also means that if the scan stops early, we still have all results up to that point saved.

Throttling, Batching, and Rate Limiting

Given the potentially high volume of external API calls (scanning many addresses quickly), it's crucial to implement throttling and batching strategies: - **Concurrency Limits:** The `concurrency` parameter (16 by default) limits how many addresses are being fetched at the exact same time [48]. We should tune this based on external API rate limits. For example, if Infura allows 100 requests per second, concurrency 16 is safe; if using a public node with lower capacity, we might dial it down. The system can also dynamically adjust – e.g., if many timeouts or rate-limit errors occur, reduce the concurrency. - **Rate Limiting:** In addition to concurrency, a simple rate limiter (requests per second cap) can be implemented. For instance, using a token bucket or even just `asyncio.sleep()` between batches of requests. The current design uses the gap limit and natural asyncio scheduling as basic control, but we could add: - A counter of requests made to each provider in the last second and delay new ones if a threshold is exceeded. - Use external libraries like `aiolimiter` to enforce a QPS (queries per second) limit per domain. - Since the scanner uses

a queue, one approach is to only put a new address into the queue after a tiny delay or to have workers sleep for a short duration after each API call. For example, after processing an address, `await asyncio.sleep(0)` is used just to yield control (as in the SSE stream generator) [23]; this could be increased to `sleep(0.1)` to throttle overall speed slightly if needed. - **Batching API Calls:** Some providers allow batch queries. For example, Infura's JSON-RPC API accepts an array of requests in one HTTP call. We could batch the balance and transaction count calls for multiple Ethereum addresses in one request (though the code currently does one address at a time for simplicity). Blockchair API can return data for multiple addresses in one call (by comma-separating addresses). Utilizing these can drastically cut down the number of HTTP requests. A possible optimization: collect, say, 10 addresses and query them in one batch, then distribute the results. This adds complexity but is worthwhile if rate limits are tight. - **Exponential Backoff:** If a provider responds with a **rate limit error** or any transient failure, the system should retry after a delay. Wrap the API calls in try/except; on exception or HTTP 429, wait e.g. 5 seconds, then try again (perhaps reducing concurrency temporarily). This ensures that a spike in usage doesn't completely fail the scan but rather slows it down gracefully. - **Limiting Range Size:** If using range mode with explicit start and end, the frontend should ideally prevent extremely large ranges (since scanning millions of keys would overload the system). The backend can also enforce a max range length – e.g., if `end - start > 100000`, reject or require confirmation. In practice, users will likely test small ranges. - **One Scan at a Time:** In the current design, only one scanning job runs at a time on the backend (a global `scanner` object). This is a form of self-throttling to avoid excessive load. If multiple users or multi-scan support is needed, we'd implement a job queue with limits (e.g., max 1 scan per user, or a queue where extra requests wait). We could spin up separate scanner instances per job, but then careful resource management is needed. - **Caching Results:** An optimization: if the same address gets derived multiple times or across multiple scans (which can happen especially in random scanning – albeit low probability to hit the exact same address – or if ranges overlap), we could cache addresses already seen recently to avoid redundant API calls. For example, keep an in-memory LRU cache of address -> result. The scanner's `_seen` set partly serves this purpose for a single scan (to not double-scan addresses found via multiple paths in the graph) [94] [62]. - **Batch DB Writes:** As discussed, using `insert_many` for every 50 addresses is a form of batching to reduce load on MongoDB [69] [70]. This threshold can be tuned as well (larger batch for faster insert throughput vs. smaller for more real-time data). - **Monitoring and Backoff:** We should monitor the latency of external calls. If calls start slowing down (perhaps the API is throttling us), we can implement a backoff: e.g., if a request takes > X seconds or we get timeouts, pause new requests for a bit. The scanner's design with asyncio makes it easy to insert `await asyncio.sleep(n)` in the loop if needed to backoff.

In summary, the system is designed to **scan efficiently but responsibly**, using concurrency to speed up scanning, while employing gap limits and one-job-at-a-time to avoid unbounded resource usage. The use of asynchronous IO means that if one provider call is slow, other tasks aren't blocked – they can proceed to query other addresses or other providers.

Recommended Technologies and Libraries

Building this system involves several components, each with suitable libraries:

- **FastAPI** (Python) – for the backend API. FastAPI's built-in support for async endpoints and WebSockets makes it a great fit [75] [76]. Uvicorn is recommended as the ASGI server to run FastAPI. Pydantic (part of FastAPI) can be used to model request and response data (though here we mostly pass raw JSON).

- **Next.js (React)** – for the frontend. Next.js can handle the UI, and also possibly proxy API calls if the FastAPI is a separate service. React's state management will be used for live updates. The `EventSource` API or WebSocket API in the browser will be utilized for real-time communication. If using socket.io, one can include the Socket.IO client library in Next.js and use a Socket.IO server (there's `python-socketio` which can integrate with FastAPI or run alongside).
- **Cryptography/HD Wallet libraries:** The key derivation can use existing libraries like bip-utils or bit which support BIP32/BIP44, instead of writing our own. In the implementation, a custom lightweight module was written using *ecdsa* and *Crypto.Hash* (PyCryptodome) for hashing [95] [96]. For reliability, one could use the well-tested Bitcoin libraries to derive xpub, addresses, etc., especially if extending to many coins. For Ethereum, libraries like web3.py can derive addresses from a mnemonic or xpub as well.
- **aiohttp or httpx** – for making asynchronous HTTP requests to external APIs. The code uses `aiohttp.ClientSession` for reuse of connections [97] [98]. `httpx` is another modern async HTTP client that could be used. Both support timeouts and retries which we should employ.
- **External APIs:**
- *Infura* – no library needed, just HTTP POST to the JSON-RPC endpoint. But web3.py could also be used with an Infura provider for higher-level abstraction (e.g., `Web3.eth.get_balance(address)`).
- *Tatum API* – a RESTful API, can be used via simple HTTP GET calls (with API key header) [99]. No specific SDK is required, but Tatum provides a Python SDK if preferred.
- *Blockchair API* – simple REST calls; again, could use direct HTTP or any wrapper.
- These keys (Infura Project ID, Tatum API Key) should be loaded from environment or config (as seen in the `.env` usage) [100].
- **Motor or PyMongo** – for MongoDB access. If we want fully async interaction, Motor (async Mongo driver) is ideal [101]. In the code, they used a synchronous PyMongo via `asyncio.to_thread` for inserts [70], which is also workable. Motor would let us `await db.insert_one` directly. Define the Mongo URI and database in config.
- **WebSockets/Socket.IO:** FastAPI provides `WebSocket` class for handling connections natively [75]. This is fine for most uses. If we needed rooms or broadcast to specific users, we might incorporate Socket.IO which has those features built-in at the cost of an extra layer. The decision depends on scale and complexity – for a single-user or low-user-count tool, FastAPI's WS is sufficient. In either case, ensure to run the app with an ASGI server that supports WebSocket (Uvicorn does).
- **Task Scheduling/Queue:** The native approach uses `asyncio.create_task` to start the scan. This is acceptable for moderate load. If we anticipated very heavy usage or wanted to distribute work, a task queue like **Celery or RQ** could be introduced. For example, the `/start_scan` could enqueue a job in Redis and a worker process (separate from FastAPI) does the scanning, sending updates via a message broker or websocket. However, this adds complexity and is likely unnecessary unless scaling to many concurrent scans.
- **Performance considerations:** Use of C-optimized libraries for crypto (the ecdsa library is Python but has some optimization; PyCryptodome for hashing is in C). If scanning extremely large ranges, the key derivation becomes CPU heavy – one could offload that to a separate process or use vectorized libraries. But generally network calls will be the bottleneck, not key generation.
- **Security:** Ensure that any keys generated are not exposed beyond the necessary scope. The private key (hex) is shown in the UI for transparency, but in a real app you might not show it or you might allow exporting it. Since this is a scanner (often used for finding funded keys), we assume these are not user's own keys but random ones – security is less of a concern, but still, treat the data carefully.

Use HTTPS for all API calls, secure WebSocket (wss) in production, and perhaps an auth mechanism if this is not just a local tool.

- **Diagram Libraries (for documentation):** Not part of the product, but to create architecture diagrams one might use mermaid.js or draw.io. (This document itself uses textual description due to limitations, but a separate diagram could illustrate the flow from user input to result.)

Finally, some general best practices: - Use environment variables or config files for API keys (as shown in the example `.env` with `TATUM_API_KEY`, `INFURA_PROJECT_ID`) [100]. - Containerize the application (the repository uses Docker Compose to run the FastAPI and a MongoDB instance together) which is helpful for deployment [102]. - Write unit and integration tests for the key derivation and scanning logic, perhaps using a testnet or a dummy blockchain service to simulate API calls, to ensure the correctness of address generation and scanning.

In conclusion, this architecture consists of a responsive frontend that gives users control over scanning parameters and displays live results, and a robust asynchronous backend that performs the scanning logic – from key generation, address derivation across multiple chains, to querying blockchain data providers – all while streaming updates and logging to a database. By combining FastAPI's performance and concurrency with Next.js's interactivity, the system can efficiently scan large key spaces and immediately surface any "hits" (addresses with balance or transactions) to the user [47]. The use of asynchronous design and careful throttling ensures it remains both fast and respectful of rate limits. This setup provides a scalable foundation for a multichain wallet scanner that can be extended to additional blockchains or integrated with more advanced analysis in the future.

---

[1] [3] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [95] [96] bip32.py
https://github.com/BloodLuust/masterpig/blob/ca24c76f9c6bcf8dc6dae29982cdd143ca98580f/app/bip32.py

[2] [100] [102] README.md
https://github.com/BloodLuust/masterpig/blob/ca24c76f9c6bcf8dc6dae29982cdd143ca98580f/README.md

[4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [19] [20] [21] [22] [23] [91] [92] [97] main.py
https://github.com/BloodLuust/masterpig/blob/ca24c76f9c6bcf8dc6dae29982cdd143ca98580f/app/main.py

[17] [18] [47] [48] [49] [50] [51] [52] [58] [59] [60] [61] [62] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [88] [89] [90] [94] scanner.py
https://github.com/BloodLuust/masterpig/blob/ca24c76f9c6bcf8dc6dae29982cdd143ca98580f/app/scanner.py

[24] [25] [83] [84] [85] [86] index.html
https://github.com/BloodLuust/masterpig/blob/ca24c76f9c6bcf8dc6dae29982cdd143ca98580f/web/index.html

[53] [54] [55] [56] [57] [63] [64] [93] [98] [99] providers.py
https://github.com/BloodLuust/masterpig/blob/ca24c76f9c6bcf8dc6dae29982cdd143ca98580f/app/providers.py

[75] [76] [77] [78] [79] [80] [81] [82] [87] [101] Developing a Real-time Dashboard with FastAPI, MongoDB, and WebSockets | TestDriven.io
https://testdriven.io/blog/fastapi-mongo-websockets/