ChatGPT

Python-Based Architecture for an Expanded Airdrop Farming Bot

Overview and New Airdrop Opportunities

The airdrop farming bot needs to expand beyond its current support (Hyperliquid, LayerZero, Scroll, EigenLayer, zkSync) to target new high-potential opportunities. Below we identify each target platform and outline how to **interact programmatically via Python** – whether on testnet or mainnet – including any known APIs, SDKs, contract interfaces, or RPC endpoints. We also incorporate strategies for **randomized, Sybil-resistant behavior**, and provide architectural advice for a modular bot design, task scheduling, and wallet management.

High-Potential Airdrop Platforms and Interaction Steps

Monad – EVM-Compatible L1 Blockchain

About Monad: Monad is a **fully EVM-compatible Layer-1 blockchain** focused on high performance and scalability [1]. It launched a public testnet (Feb 2025) and, while no token is confirmed yet, early testnet interactions are believed to count toward a potential airdrop [2].

Programmatic Interaction (Testnet):

- **Network Connection:** Treat Monad like an Ethereum network. Add the Monad testnet RPC to Web3 (e.g. chain ID and RPC URL from community endpoints [3]). For example, using `web3.py`:

  ```
  from web3 import Web3
  MONAD_RPC = "https://monad-testnet.rpc.hypersync.xyz"  # example public RPC
  w3_monad = Web3(Web3.HTTPProvider(MONAD_RPC))
  ```

  Ensure to set the correct chain ID when signing transactions (check Chainlist or official docs for Monad testnet chain ID).

- **Faucet and Funding:** Use Monad's faucet to obtain test tokens. The official testnet site provides a faucet to get test MON tokens [4]. This can be automated if an API exists (e.g., by sending an HTTP request with your address if supported) or by scripting a browser interaction. At minimum, the bot can pause and alert you to manually fund the test wallets via the faucet.

- **On-Chain Actions:** Perform a variety of on-chain transactions to simulate real usage. Examples:

- **Deploy a Contract:** Since Monad is EVM, you can deploy a simple Solidity contract or interact with an existing one to generate activity. For instance, compile a basic ERC-20 or storage contract and use `web3.eth.account.signTransaction()` to deploy it. This shows developer-like activity.
- **Interact with DApps:** Monad's testnet focuses on NFTs and games. For example, an NFT marketplace integration with Magic Eden was highlighted [5] [6]. You can script interactions such as *purchasing a testnet NFT* by calling the marketplace contract. In practice, find the marketplace's contract address and ABI (if available) and use `contract.functions.buy(tokenId, ...).buildTransaction()`. Signing such transactions via the bot (with the wallet's private key) demonstrates active participation.
- **Gameplay Transactions:** If Monad's testnet game (e.g. "Fantasy Top") requires on-chain moves, automate those moves. This might involve calling a game contract's method (identified via block explorer or docs) to simulate gameplay. Adding random delays between moves (see randomness section) will mimic human play.

APIs/Endpoints: Monad is EVM-based, so standard Ethereum JSON-RPC calls work (for both transaction submission and data retrieval). Use **Web3.py** for simplicity. No specialized SDK is required beyond web3; any Ethereum library (web3.js, ethers via Python binding) could also be used. For contract ABIs, use official testnet explorers to fetch ABIs or use standard ERC-20/ERC-721 ABIs for token and NFT interactions.

Abstract – Cross-Chain ETH Staking Platform (ZK Rollup)

About Abstract: Abstract is a blockchain platform built on Matter Labs' **ZK Stack**, operating as an Ethereum rollup (Layer2) with EVM compatibility [7]. It focuses on features like cross-chain ETH staking (via LayerZero) and account abstraction [8] [9]. Its public testnet was on Sepolia (Ethereum testnet) and user actions (bridging and quests) likely count toward a future token airdrop [10] [11].

Programmatic Interaction (Testnet):

- **Network Connection:** Connect to Abstract's chain via its RPC. For example, Abstract testnet uses chain ID **11124** with RPC `https://api.testnet.abs.xyz` [12]. Using `web3.py`:

```
w3_abs = Web3(Web3.HTTPProvider("https://api.testnet.abs.xyz"))
w3_abs.middleware_onion.inject(Web3.middleware.geth_poa_middleware,
layer=0)  # if PoA
```

  (If the chain is ZK-rollup, it may not need PoA middleware; include if RPC requires it.)

- **Bridging ETH via LayerZero:** Abstract provides a **native bridge** (L1 Sepolia ↔ L2 Abstract) for ETH and tokens [13]. To automate:

- **On Sepolia (L1):** Call the bridge's deposit function. The bridge likely has an Ethereum-side contract (check docs or `Deployed Contracts` for addresses). For example, if the bridge contract ABI indicates a function `depositETH`, use:

```
bridge_contract = w3_eth.eth.contract(address=BRIDGE_ADDR, abi=BRIDGE_ABI)
tx =
bridge_contract.functions.depositETH(wallet_address).buildTransaction({...})
signed = w3_eth.eth.account.sign_transaction(tx, private_key=ETH_PRIVKEY)
w3_eth.eth.sendRawTransaction(signed.rawTransaction)
```

Include a small random ETH amount (within faucet limits) to deposit. **Randomize** deposit timing and amounts per wallet to avoid uniform patterns.

- **On Abstract (L2):** The deposit will mint ETH (or WETH) to the same address after ~15 minutes [14]. The bot should poll the Abstract RPC for balance changes. After receiving funds, it can also **withdraw** back to L1:

  - Call the Abstract-side bridge contract's withdraw method (e.g., `withdraw(uint256 amount, address to)`). This triggers a message that unlocks funds on L1 after the delay [14]. Automate a **withdrawal a day or two later**, at a random time, to simulate real usage (and test the full bridge cycle).

- **On-Chain Tasks on Abstract:** Beyond bridging, Abstract had incentivized tasks:

- **Quests and NFT:** The project offered quests (possibly via Intract or Galxe) and a commemorative NFT [15]. If these quests have on-chain components (e.g., claiming an NFT), script them. For instance, if an NFT claim contract is provided, call its `claim()` function. Use the wallet's private key to sign, just as with Ethereum.
- **ETH Staking:** Abstract's core feature is staking ETH on L2 with high yield [8]. If an **staking contract** exists (e.g., a liquid staking or validator delegation contract), you can interact via web3:
  - Approve any required token (if staking uses WETH/ETH, maybe just an `stake()` call with value).
  - Call stake functions with a random small amount of ETH. For example:

  ```
  staking_contract = w3_abs.eth.contract(address=STAKING_ADDR,
  abi=STAKING_ABI)
  tx = staking_contract.functions.stake().buildTransaction({"value":
  amount_wei, ...})
  ```

  This shows active use of the platform's main feature.

APIs/Interfaces: Use **JSON-RPC** for Abstract (same as Ethereum). The **LayerZero** integration is abstracted behind the bridge contracts, so you don't need to call LayerZero APIs directly – interacting with the bridge contracts via web3 is sufficient. If needed, Abstract's SDK or CLI (they provide a CLI for ZKSync-like interactions [16]) could be used for advanced scenarios, but for the bot, raw web3 calls are adequate.

Caution: Abstract's rollup uses **account abstraction**; if transactions require special handling (e.g., paying fees in ETH vs. needing a paymaster), consult docs. The provided RPC likely supports standard transactions with ETH as gas, simplifying bot development.

Eclipse – Solana VM Layer-2 on Ethereum

About Eclipse: Eclipse is an innovative L2 that runs the **Solana Virtual Machine (SVM)** on an Ethereum-aligned rollup [17]. In essence, it combines Solana's high-throughput runtime with Ethereum's security and liquidity [17]. It raised substantial funding (seed and Series A) and is expected to launch a token; early adopters on testnet/mainnet are prime airdrop candidates [18] [19].

Programmatic Interaction (Testnet & Mainnet):

- **Network Connection:** Unlike EVM chains, Eclipse uses Solana's developer tooling. Connect via Solana JSON RPC endpoints:
- **Testnet RPC:** `https://testnet.dev2.eclipsenetwork.xyz` (per docs) [20].
- **Devnet2 RPC:** (alternate dev environment) [21].

- **Mainnet:** Likely will have its own RPC (not yet public at testnet stage). Using the **Solana Python SDK** (`solana-py`), you can set up a client:

```python
from solana.rpc.api import Client
eclipse_client = Client("https://testnet.dev2.eclipsenetwork.xyz")
```

  With Solana, wallet keys are Ed25519 keypairs. You might load each bot wallet from a mnemonic or `.json` private key file:

```python
from solana.keypair import Keypair
kp = Keypair.from_secret_key(<64-byte_secret>)
pubkey = kp.public_key
```

- **Faucet & Airdrop:** To pay for transactions on Eclipse, you need **tSOL or tETH** (depending on how Eclipse structures fees). Eclipse testnet uses a token called tETH for gas (a testnet ETH equivalent) [22]. They provide a **faucet** in their developer portal [23] [24]. If available publicly, the bot can call the faucet programmatically:

- Check if there's an RPC method like `request_airdrop(pubkey, lamports)` as in Solana devnet. Eclipse may have a similar endpoint or a dedicated faucet UI. If an RPC airdrop is enabled, call it via `eclipse_client.request_airdrop(pubkey, int(1e9))` (for 1 SOL worth of lamports, adjust as needed).

- Alternatively, if testnet tETH is bridged from Ethereum Sepolia, use the **canonical bridge**: deposit small Sepolia ETH on L1 to get tETH on Eclipse. The one-click guide suggests an "Instant bridge" available via UI [25] – possibly using Hyperlane or a similar service. To automate, you might interact with a **Hyperlane API/contract** or directly call the **Eclipse bridge** program:

  - If a **Wormhole** or Hyperlane contract is deployed on Sepolia for Eclipse, use web3 (similar to Abstract's bridging) to lock Sepolia ETH, and then listen for a corresponding credit on Eclipse (Solana side) via the event or by querying Eclipse RPC for balance changes.

- **On-Chain Actions on Eclipse:**

- **Mint a Domain:** Eclipse integrates with AllDomains (a Solana domain service) on testnet [26]. Programmatically, minting a domain likely involves calling the **AllDomains program** on Eclipse. If you have the program ID and instruction format (e.g., via Anchor IDL), use `solana-py` or **AnchorPy** to send the transaction:
    - Authenticate via Twitter (this step cannot be fully on-chain – likely a centralized check). If required, you might need to pre-link Twitter manually, as automating OAuth is non-trivial.
    - After linking, call the domain mint instruction with your wallet's public key and chosen name. The bot can generate a random name if needed. *Note:* Because this requires an off-chain component (Twitter), consider handling this part semi-automatically or skipping if automation is not feasible.
- **Complete On-Chain Quests:** Eclipse's "tap" page lists on-chain tasks/quests [27]. These could be simple transactions (e.g., transferring tokens, interacting with a demo dApp). For example:
    - Transfer a small amount of tUSD or other test token to another address (perhaps another one of your wallets) to simulate usage. Use Solana token program via the SDK or by constructing a transfer instruction.
    - If an on-chain game or app is available (they mention possible small quests), identify the program and simulate interactions. For instance, if there's an OpenBook DEX deployed (they have an OpenBook quickstart in docs) [28], you could place a test order or provide liquidity: *Use AnchorPy:* find the program ID for OpenBook on Eclipse testnet, and use AnchorPy to construct an instruction to swap or add liquidity (this requires familiarity with OpenBook's interface; alternatively, use a simpler system program call).
    - **Mint tETH (Liquid Staking):** The guide suggests minting tETH (Eclipse's liquid staking token) [22]. This might involve calling a **staking program** on Eclipse that stakes Celestia's TIA or Ethereum's Sepolia ETH to yield tETH:
    - If it's an Ethereum staking derivative, perhaps deposit Sepolia ETH into their contract (similar to Lido on testnet). If it's Celestia-related (stake TIA externally [29]), that part might be out-of-scope for this bot. On Eclipse itself, focus on what can be done directly:
    - Call the **tETH mint program** with your Eclipse wallet. Likely, you just provide some ETH (or even *maybe they allow testnet ETH conversion to tETH via a program*). If an API or command exists (the UI might simply call a program to mint tETH one-for-one with your deposited ETH), replicate that with the SDK.
- **Deploy a Smart Contract (Advanced):** Eclipse supports deploying **Solana programs** (in Rust) on its testnet [29] [30]. For the truly advanced usage (and *if* your bot operators are comfortable), you could automate contract deployment:
    - Compile a simple Solana program (e.g., a no-op or counter program) outside the bot. Then use the Solana CLI or RPC to deploy it. The bot could call a script or utilize anchor's Python bindings to deploy. This demonstrates a power-user profile (developers might get special retroactive rewards).
    - This is optional and requires significant setup (Rust toolchain, etc.), so consider it only if you aim to mimic developer activity for Sybil resistance.

Developer Notes: Interacting with Eclipse means working in the **Solana developer paradigm**. Key libraries: - `solana-py`: Low-level transactions (compose instructions manually or via imported `PublicKey` and `SystemProgram` classes). - `anchorpy`: If Eclipse programs are written with Anchor (common in Solana ecosystem), AnchorPy can use the IDL (Interface Definition) to easily call instructions. For example, using an IDL for a known program (like token or domain service) to build and send transactions. - **RPC usage:** Use

Solana JSON RPC calls like `get_balance`, `get_program_accounts` for monitoring. For instance, after bridging, use `eclipse_client.get_balance(pubkey)` to confirm tETH arrival. - **Concurrency:** Solana transactions finalize fast (seconds). The bot can submit multiple transactions quickly, but **throttle them** to avoid all wallets acting at once. Insert random `time.sleep()` delays (with jitter) between actions.

Axiom – Ethereum ZK Coprocessor and On-Chain Proofs

About Axiom: Axiom is a **ZK-coprocessor for Ethereum** that lets smart contracts trustlessly access historical blockchain data [31]. It uses zero-knowledge proofs (Halo2) to prove past chain state to contracts [32]. Axiom ran a testnet (often called the "Test Circuit") where users who generated proofs and queries could earn points or future airdrop allocations [33] [34].

Programmatic Interaction (Testnet/Mainnet):

- **Environment Setup:** Axiom's interactions involve generating cryptographic proofs off-chain and submitting them on-chain. The bot's architecture should integrate with Axiom's tooling:
- **SDK/Proving API:** Axiom provides an SDK (mostly in Rust and JavaScript) and a cloud Proving API [35]. Check if a Python binding or REST API exists for proof generation. If not, you might call a CLI tool from Python (subprocess) or use their GitHub libraries [36]. For example, the **Axiom V2 core SDK** on GitHub provides guidance for generating proofs of historical account states [37].

- **Contracts:** Identify Axiom's on-chain contract on Ethereum (likely a verifier contract). The **Axiom Ethereum library** [36] or docs should have an address/ABI for a `Verifier` or `AxiomProxy` contract.

- **Airdrop-Related Actions:**

- **Generate ZK Proofs:** In the testnet tasks, users had to generate and submit proofs (e.g., proving some historical fact). Programmatically, this means:
  1. Call Axiom's proving API or run a local proof generator for a given query (for instance, "prove ownership of an address at block X" or similar). This yields a proof blob.
  2. Construct the Ethereum transaction to send the proof to Axiom's contract. Likely the contract has a method like `submitProof(bytes proofData)`.
  3. Use `web3.py` to send this transaction. **Include randomness:** generate proofs for varied queries per wallet (not all wallets proving the same data) to avoid Sybil patterns.
  4. *Note:* Proof generation can be resource-intensive. If a direct Python route is not available, consider using Axiom's **Hosted Proving API** (if they offer one) where you send a request and get back a proof. Automate this via Python's `requests` library.

- **Query and Claim Mechanism:** Axiom's example *"autonomous airdrop"* contract allowed users to claim tokens by proving eligibility via Axiom [38]. If the airdrop requires a proof-based claim, your bot can do:

  - Generate the required proof (e.g., a Merkle proof of being in an off-chain snapshot, converted to a ZK proof through Axiom).
  - Call the airdrop contract's claim function, providing the proof. This might be a single call if the airdrop contract integrates Axiom verification. For example:

```
airdrop_contract = w3.eth.contract(address=AIRDROP_ADDR,
abi=AIRDROP_ABI)
tx =
airdrop_contract.functions.claimWithProof(proof_data).buildTransaction({...})
```

Sign and send as usual.
    - This process is advanced; ensure the bot can reliably get the proof data. Use logging and error handling – if a proof fails verification, catch it and perhaps try again or mark that wallet as needing manual review.

- **Staking/Points:** Axiom may reward running a **node or participating** in their network. If they allow testnet **validators or data providers**, and you have the resources, consider running an Axiom node and interacting via Python (monitoring node status, etc.). However, this might be beyond the typical bot scope, so focus on the on-chain proof submission which is more straightforward to automate.

APIs and Tools: The key here is Axiom's unique SDK: - No native Python SDK for ZK proof gen, but Python can interface with compiled binaries or web APIs. Keep an eye on Axiom's official channels for any developer tools announcements. - Use standard Ethereum RPC for on-chain submission. (Mainnet or Sepolia if they used testnet). - **Security:** Handling private keys to sign these proof transactions is as usual. But also manage any API keys or secrets if using a hosted proving service.

Mitosis – Cross-Chain Liquidity L1 (Programmable Liquidity)

About Mitosis: Mitosis is an **ecosystem-owned liquidity Layer-1** blockchain aiming to redefine the DeFi liquidity provider (LP) experience [39] . It's essentially a cross-chain liquidity protocol for modular blockchains [40] . Mitosis ran **Game of MITO** – an incentivized testnet campaign – where users earned points (XP) and **Morse tokens** for completing tasks [41] . A large portion of testnet-earned $MITO will carry over to mainnet [42] , meaning testnet actions directly influence airdrop rewards.

Programmatic Interaction (Testnet):

- **Network Connection:** Mitosis provides an EVM-compatible testnet:
- **RPC URL:** e.g., `https://124832.rpc.thirdweb.com` (Thirdweb's endpoint) [43] or `https://testnet.mitosis.org` if available.
- **Chain ID:** 124832 [44] .
- Connect with `web3.py` similar to other EVM chains:

```
w3_mitosis = Web3(Web3.HTTPProvider("https://124832.rpc.thirdweb.com"))
```

(Thirdweb's RPC is free to use; alternatively, use chainlist-provided RPCs.)

- **Test Tokens & Faucet:** Mitosis testnet uses **MITO** as native token (for gas) [44] , and likely has test versions of assets (wETH, stablecoins, etc.). The official faucet (on their testnet site) distributes test tokens daily [45] . Automation steps:

- The faucet required social login ("Activate Faucet" with Discord/Twitter) [45], which is hard to automate fully. However, you can integrate a **headless browser** (e.g., Selenium) to simulate the OAuth flow if needed. This is complex; alternatively, do the social connection once manually per wallet and then script the **token request** if the faucet has an open endpoint.

- If direct faucet automation is not feasible, pre-fund your wallets via the faucet and store their balances. The bot can then focus on using those tokens for tasks.

- **On-Chain Task Automation:** Game of MITO had a series of tasks:

- **Play the "MITO" Collecting Game:** On the homepage, users collaborate to collect letters forming "MITO" [46]. This likely logs on-chain participation or is simply a front-end game awarding XP. If the game triggers on-chain events (e.g., a contract call when a letter is collected), capture that by inspecting network calls. If it's off-chain (server-coordinated), automation may not help much aside from simulating user activity with browser automation. In the latter case, consider using a **web automation tool** to periodically perform the game action (e.g., click a button at random intervals). Ensure each wallet (account) only does it at random times, not all simultaneously.
- **Voting:** There's a voting tab for governance-like participation [47]. Likely on-chain via a governance contract or snapshot. If it's on-chain (perhaps a simple contract storing votes or a dummy DAO proposal):
    - Use web3 to call the `vote(proposalId, choice)` function. The bot can fetch current proposals via a public call (if ABI given) and then cast votes. Randomize vote choices if not consequential, or follow a pattern that looks human (some wallets abstain, etc.).
- **EOL Opt-in & Deposits:** Users needed to **opt-in to EOL (Ecosystem-Owned Liquidity)** and deposit assets [48]. This likely involves:
    - Calling an **opt-in contract** (maybe setting a flag for your address). Possibly `optIn()` or signing a message. Automate this by calling the function once per wallet.
    - **Depositing Assets:** Mitosis accepts deposits of assets from multiple chains (Ethereum, Arbitrum, Optimism, etc.) which are then represented as "miAssets" on Mitosis [49] [50]. To simulate:
    - Simplest route: use the faucet tokens (they likely provided testnet versions of ETH, USDC, etc. on Mitosis itself [45]) and call the deposit function **on Mitosis chain**. For example, deposit testnet USDC:
        1. Ensure your wallet has test USDC on Mitosis (the faucet likely provided some).
        2. Use web3 to approve the Mitosis vault/bridge contract to spend your USDC.
        3. Call `deposit(token, amount)` on the Mitosis deposit contract (the "Deposit" tab contract) [51].
    - Advanced: If time permits, actually deposit from other chains:
        - On Goerli/Arbitrum-Goerli/etc., call a bridge contract to send assets to Mitosis (the Binance guide suggests bridging from Sepolia or testnets of Arbitrum/Optimism [50]). This is complex, as it might involve interacting with multiple testnets and waiting for finality. The bot could coordinate such cross-chain actions using **Celery tasks** (one task to send from L2, another to finalize on Mitosis after some delay).
        - Given complexity, focusing on on-chain deposit on Mitosis itself is usually enough to register as active.
- **Track Progress & Pet Upgrade:** The testnet had a "pet" representing progress [52]. This likely updates automatically as you complete tasks. Ensure the bot checks *"My Page"* to confirm tasks are registered (if there's an API or by scraping the page HTML for confirmation strings). This can be a

simple HTTP GET to the profile page (if cookies/session are handled after login) – possibly out-of-scope for pure on-chain, but useful for logging.

APIs/Resources: Mitosis being EVM means standard Web3 calls for on-chain parts. Off-chain components (faucet social auth, game coordination) may require creative solutions: - **Selenium or Playwright:** for automating web tasks (login, clicking daily faucet, playing the game). - **Thirdweb SDK:** They provide a Connect SDK [53] which can simplify wallet connections and contract calls if one prefers a higher-level approach. In Python, it's easier to stick to web3, but thirdweb's contracts or **Thirdweb Python (if exists)** could manage some tasks (not widely used, so web3 is safer). - **dApp UIs/GraphQL:** Check if the testnet site offers a GraphQL or REST API (some campaigns have an API for submitting proofs of task completion). Inspect network calls when using the site manually – the bot might replicate those HTTP calls with appropriate auth tokens.

Sybil Consideration: Mitosis explicitly used social logins to reduce Sybil farming. If each wallet is tied to a unique Discord/Twitter, you may need distinct accounts. Avoid reusing one social account across wallets. The **bot architecture** should allow storing per-wallet credentials (perhaps a JSON config mapping wallet -> Twitter/Discord token). Incorporate random delays and different behavior patterns per wallet (e.g., some wallets deposit only one type of asset, others deposit multiple; vary deposit amounts and timings).

Pump.fun – Solana Memecoin Launchpad

About Pump.fun: Pump.fun is a **Solana-based memecoin launchpad** that lets anyone create and trade memecoins easily [54]. It automates token creation and fair launch mechanics (no prior technical skill needed). A token ($PUMP) and airdrop are rumored, with possibly 10% of supply for early users [55]. Qualifying actions likely include creating tokens, adding liquidity, and general platform usage [56].

Programmatic Interaction (Mainnet Solana):

- **Wallet Setup:** Use **Solana SDK** as with Eclipse (solana-py). Load each wallet's keypair (if you used the same wallets as for Ethereum, ensure they have corresponding Solana keys or generate new ones specifically for Solana usage).

- **Creating a Token (Memecoin):** The primary action on Pump.fun is token creation. Automate this via Solana program calls:

- Pump.fun likely has a **program** (smart contract) that, when invoked, mints a new SPL token and sets up initial liquidity. They mention tokens can be launched for as low as ~$2, which in Sol terms might be done by depositing a small amount of SOL or USDC as initial liquidity. The program likely:
    1. Creates a new token mint (using Solana's SPL Token program under the hood).
    2. Creates a liquidity pool (maybe an AMM on Serum or a simple constant product AMM).
    3. Lists the token on their interface.
- If Pump.fun's program is open-source or if an API exists, obtain the program ID and instruction schema. It may be an Anchor program, in which case get the IDL (Interface Definition). The bot can then use AnchorPy:

```
from anchorpy import Program
# Assume we have IDL and program ID for PumpProgram
```

```
pump_program = Program(pump_idl, PUMP_PROGRAM_ID, eclipse_client)  # or
solana mainnet client
```

Then call the instruction (for example, `pump_program.rpc["create_token"](...parameters...)` if defined). Parameters likely include the token name, symbol, initial supply, and maybe a fee account.

- If direct program interaction is complex, an alternative is to script the front-end via **Selenium**:

  - Navigate to pump.fun, fill the form (name, symbol, upload a random image if required – perhaps skip image or use a placeholder), and click “Create”. This will prompt the Phantom wallet – which you can automate using a wallet adapter or by intercepting the transaction request if possible. (There are headless Phantom plugins but that’s very advanced).
  - A simpler approach: Use **Solana CLI** via Python (`subprocess.run(["solana", "program", "send-tx", ...])`) if Pump.fun provides a CLI hook or if their creation can be mimicked with standard CLI commands (e.g., using `spl-token` CLI to create a token, then adding it to a known DEX). However, Pump.fun’s value is in the automated fair launch, which might not be trivial to replicate manually.

- **Trading and Liquidity:** To show usage:

- **Trade tokens:** After creating some tokens, your bot can simulate trading. If Pump.fun has an API or if trades go through an on-chain DEX (like a Serum market), you can interact with that:
  - Use the **Serum DEX program** via Python to place a buy or sell for a token your other bot wallet created. Or use Jupiter aggregator’s API (if available via HTTP) to swap a small amount of SOL to some newly created token. This demonstrates engagement with the platform’s coins.

- **Airdrop distribution feature:** Pump.fun might also allow **airdropping tokens** to other users (the search result mentioned a feature for mass token drops [57]). As a power-user move, your bot could create a token and then use the “mass airdrop” feature to send small amounts to a set of addresses. Programmatically, if this is an on-chain instruction, call it (likely iterating transfers). Or if off-chain, skip unless API available.

- **Monitoring:** Use Solana RPC calls to track the number of tokens created and maybe volume:

- The bot can fetch created token account stats (though Pump.fun likely has its own indexer). For safety, after initiating token creation, poll the token mint address to ensure it exists and has your supply. This confirms the action succeeded.

Libraries & APIs: - **Solana Web3 (solana-py)**: use for any direct calls. For example, to create a token manually you could use solana-py’s `create_token` and `create_associated_token_account` instructions from the SPL token library. - **Bitquery or Solana Indexer API:** The search shows a *Bitquery Pump Fun API* [58] . This could provide read access (e.g., to get token data, creation info). Not necessary for sending transactions, but useful for verifying state or ensuring your actions registered (for example, query how many tokens your wallet created). - **Rate limiting:** Pump.fun on Solana mainnet means transactions incur fees (tiny SOL amounts) and possibly slippage costs for liquidity. Spread out actions over time to avoid drawing attention (e.g., don’t create 100 tokens in one minute; maybe a few per day). - **Sybil safety:** Each wallet should create a different number of tokens (some 0, some 1-2, some maybe 5 over time). Vary token names (the bot can generate random meme-like names). This avoids a pattern where all tokens look auto-generated by the same script. Also, be mindful that **Solana addresses are public** – if all your bot wallets immediately send tokens to each other or interact exclusively with each other's creations, it might raise flags. Instead, consider having them primarily interact with the platform (creating and maybe trading on public pools) and not just with each other.

PlushieAI – AI Companion & Reward Platform

About PlushieAI: PlushieAI is an interactive AI-driven platform offering virtual companions and creative AI products (like transforming plush toy images with AI) [59]. Its ongoing airdrop distributes **$PLSH** tokens to users for participating in daily activities and community engagement [60] [61]. Key participation involves connecting a wallet, and using Telegram/Discord bots for quizzes, lucky draws, etc., rather than heavy on-chain transactions [62] [63].

Programmatic Interaction (Hybrid Off-chain & On-chain):

- **Wallet Integration:** The first step is to connect a wallet on the Plushie airdrop page [63]. This likely involves a Web3 **signature login** (signing a message to prove ownership). The bot can automate this by:
- Using web3.py's `Account.sign_message()` with the private key. If the site gives a nonce or message to sign (perhaps via an API call or during a WebSocket session), the bot should fetch it and produce a valid signature.
- Submit the signature via an HTTP POST if the site's backend expects it (monitor network calls when a normal user connects their wallet to replicate this).

- Once connected, the wallet address is recorded for receiving airdrop tokens later. Ensure the bot stores any session token if returned (like a JWT or cookie) to use for subsequent interactions.

- **Daily Engagement Automation:** PlushieAI's airdrop is gamified:

- **Telegram Bot Tasks:** Users earn random PLSH prizes by interacting with a Telegram bot (and possibly Discord) [63]. Automating Telegram requires using the **Telegram Bot API** or a library like `python-telegram` to simulate a user:
    - If Plushie's Telegram bot has commands (like `/dailyspin` or answering quiz questions), you could have a Telegram account for each bot wallet (this means phone numbers – not trivial at scale). This may not be worth the effort for full automation due to Telegram's anti-bot measures.
    - Alternatively, if the bot rewards link back to on-chain (e.g., dispensing tokens on Solana or Ethereum), you might skip the Telegram step and focus on any on-chain claim.
- **Lucky Wheel & Quiz:** The lucky wheel spin and quiz are likely on the website or Telegram. If on the website (maybe a React app), there might be an API call when you spin:
    - Inspect the network calls when spinning. Possibly a call like `POST /spin` returning the random prize. If so, the bot can hit that endpoint daily. It might require the session token from the wallet connection step as authentication.

    - For the quiz, if there's a set pool of questions or a specific code (the guide mentions a quiz code) [64], the bot could attempt to answer automatically. This is more static content; perhaps not needed if points are small.

- **Frequency & Randomness:** Schedule these interactions as periodic tasks (e.g., spin the wheel every 20-26 hours instead of exactly 24, to seem human). Use APScheduler/Celery (discussed later) to set up daily jobs with jitter.

- **On-Chain Aspect:** Eventually, $PLSH tokens will likely be distributed on a blockchain (possibly Ethereum or an L2). For the bot, the main on-chain action might simply be **receiving the airdrop**. Ensure each wallet is prepared:

- Have the wallets on the appropriate network (if announced, say Arbitrum or Solana; not confirmed in docs).
- If on Ethereum, keep a bit of ETH for gas to claim if needed. If they airdrop via a claim contract, use web3 to call `claim()` when it's live.
- If on Solana, ensure the wallet is active (create an associated token account for PLSH token mint if known, using `spl-token` create account via code or solana-py).
- *Note:* The airdrop page itself might allow a **direct claim** when available, in which case automating clicking "Claim" (with wallet signature) will be needed. Monitor announcements for when distribution happens.

Tools & Libraries: This platform blurs on-chain and off-chain automation: - **Requests + Web3:** for web APIs and signing messages. - **Selenium/Playwright:** If certain interactions (like the wheel spin) are easier to handle in a headless browser (in case of CSRF tokens or complex front-end logic), you can use Selenium to load the page, inject the web3 signature (perhaps via a custom JS snippet if needed), and trigger the spin. This is heavy but can mimic a real user completely. - **Telegram API:** Only if you attempt to automate Telegram tasks. Each bot instance could log into Telegram (using Telethon or python-telegram-bot library) as a user and interface with Plushie's bot. **Be cautious:** Telegram might ban multiple accounts if they appear automated.

Sybil Resistance: PlushieAI actively uses social tasks to prevent Sybil bots. If you automate many accounts: - Use unique Telegram/Discord for each (which may be impractical beyond a small number of identities). - At minimum, ensure the on-chain wallet tasks are done – those are your control point. The bot may skip some social tasks to avoid detection. A few fully-compliant wallets might be better than dozens of incomplete ones here. - If running multiple instances, use proxies for web requests so all hits don't come from the same IP.

STAU Platform – Gold Trading on Blockchain

About STAU: STAU (often referred to as **STAU Gold**) is a platform bridging physical gold and blockchain, enabling online trading of gold items and jewelry with a cryptocurrency (the **$STAU** token) [65] [66]. The $STAU token fuels transactions on the platform (purchases, platform fees, and rewards) [65]. An airdrop or incentive program likely rewards early users of the marketplace.

Programmatic Interaction (Mainnet, likely EVM-based):

- **Network and Token Setup:** The STAU platform is built on blockchain (the description suggests Ethereum or BSC). Check where $STAU is deployed (e.g., if it's listed on MEXC, it might be BSC or ETH ERC-20). Assuming Ethereum:
- Load Ethereum Web3 (or BSC web3 if needed). Use the token contract address for STAU (from CoinGecko or official sources) to instantiate an ERC-20 `contract = w3.eth.contract(STAU_ADDR, abi=ERC20_ABI)`.
- Ensure the bot wallet has some of the chain's native token for gas.

- **Marketplace Actions:** The core interaction is **buying and selling gold-backed items** on the STAU platform:

- Likely, each gold item is represented on-chain, possibly as an NFT or a unique ID in a smart contract. When you buy an item, you probably transfer STAU tokens in exchange for ownership (which could be an NFT transfer).
- Identify the **marketplace contract**. It might be an escrow contract where the method `purchase(itemId)` transfers the item NFT to buyer and escrows tokens to seller. If the platform provides an SDK or if the contract ABI is published, use it:
    - Example:

    ```
    marketplace = w3.eth.contract(address=MARKET_ADDR, abi=MARKET_ABI)
    tx = marketplace.functions.purchase(item_id).buildTransaction({
        "from": wallet_address,
        "value": 0,
        "gas": 200_000,
        "nonce": w3.eth.get_transaction_count(wallet_address)
    })
    # Note: value=0 if payment is in STAU token, not ETH
    ```

    Before calling, the bot must approve the marketplace to spend the required STAU amount:

    ```
    price = marketplace.functions.getPrice(item_id).call()
    stau_token.functions.approve(MARKET_ADDR, price).transact({"from":
    wallet_address})
    ```

    Then proceed to call `purchase`.
- If items are NFTs, another approach is interacting with an NFT contract:
    - Use `ERC721.safeTransferFrom(seller, buyer, tokenId)` if the platform relies on a standard transfer after off-chain order matching. However, given STAU token's role, the custom marketplace contract approach is more likely.
- The bot can simulate a **buy** and **sell** cycle with minimal financial impact:
    - Buy a cheap item (the platform might have lower value items in STAU).

    - Later, list it for sale again (if selling is user-driven). If there's a `listForSale(itemId, price)` function in the contract or via an API, call it after purchase. This completes a round trip.
    - These actions show you are an active trader on the platform.

- If the platform requires using their web interface (for off-chain matching), consider using their **API** if provided:

    - Some marketplaces have REST endpoints (e.g., for searching items or placing orders). Check STAU docs or community for any developer integration guides. If found, the bot could POST an order to sell or similar.

- **Staking or Rewards:** The description mentions *"receiving token-based rewards"* [65]. If STAU has a staking or loyalty program:

- Automate staking of STAU into whatever contract yields rewards. For instance, if there's a `StakePool` contract with `stake(amount)`. Use web3 to approve and stake a small portion of tokens.
- If rewards accrue, the bot can periodically call `claimRewards()` and either compound (restake) or just log the event.
- Stagger staking times and amounts per wallet to avoid uniform behavior (one wallet stakes 100 STAU, another 50, at different times, etc.).

Tech Notes: Since STAU is likely EVM, rely on: - **Web3.py & ERC-20 interactions:** Standard procedure as with Ethereum. We have the advantage of reusing much of the approach from Hyperliquid or others if they involved ERC-20 transfers. - If any **SDK** from STAU exists (perhaps a JS SDK for their marketplace), it's not likely in Python. So manual contract calls are the way. - **Transaction Randomness:** For each wallet, randomize which item to buy or how many items (some wallets might skip buying to emulate different user profiles – e.g., some just browse or stake). If the bot has to actually spend STAU, you might obtain some STAU tokens for test purposes or use minimal amounts. If it's not feasible to buy real items due to cost, consider focusing on staking and platform interactions that don't require large spend (perhaps they have test mode or promo codes – not evident though).

dFusion AI Protocol – Decentralized Knowledge Base

About dFusion AI: dFusion AI is building a **community-driven, decentralized knowledge base** with an incentive system for data curation [67]. It uses AI filtering plus human validation to ensure quality data, and contributors earn **Knowledge Ingestion points** convertible to future tokens [68] [69]. dFusion's token ($VFSN) and early participation via staking $VANA on their "Social Truth Data License Protocol (DLP)" are noted – e.g., staking 100k $VANA yields a 1:1 airdrop of $VFSN [70].

Programmatic Interaction (Testnet/Mainnet hybrid):

- **Contribution Tracking:** Currently, contributions (adding articles, videos, APIs to the knowledge graph) are likely handled off-chain in the beta phase (users submit data on a website, earn points off-chain [71]). This is hard to automate meaningfully with Python since it's not on-chain yet:

- If there's an API for submitting knowledge (not publicly known), the bot could post entries. But content quality matters; spam submissions could be filtered out. Automation here might violate terms, so it's safer to **not auto-spam contributions**.

- Instead, focus on **on-chain signals** of participation that *are* planned:

  - dFusion might airdrop to users who linked an address and earned points. So ensure each bot wallet's address is linked to a dFusion account. This likely requires signing up on their site and connecting the wallet (sign message or simple login).
  - The bot can automate the wallet connection on the dFusion site (similar to PlushieAI – sign a message to log in).
  - Optionally, script a minimal level of activity: e.g., if there's a daily check-in or upvote/downvote mechanism on the platform via API, use the bot to perform those actions.

- **Staking $VANA:** A known on-chain activity is staking $VANA tokens to receive $VFSN:

- Determine the chain $VANA is on. It could be Ethereum mainnet if $VANA exists (the tweet doesn't specify, but likely Ethereum). If $VANA isn't publicly trading, this step may be meant for future mainnet.
- Should $VANA be obtainable (maybe a placeholder token or points system token):
  - Acquire or mint $VANA on the designated network. For test purposes, maybe $VANA was distributed or can be requested if testnet.
  - Identify the **DLP staking contract** (Social Truth DLP). This might be a smart contract where users lock $VANA and it records their stake.
  - Use `web3.py`: approve the contract to spend 100,000 VANA, then call `stake(100000 * 1e18)` (assuming 18 decimals) [70]. This locks the tokens. The contract might emit an event or give you a receipt token.
  - The bot should record that it staked and possibly monitor if it needs to keep the stake until a snapshot (to avoid early withdrawal).

- If 100k $VANA per wallet is unrealistic (depending on token availability), consider staking a smaller amount if the goal is simply to show activity. However, the condition specifically says first 100k staked get 1:1 airdrop [70], implying a threshold. You might choose one or two "primary" wallets to stake if possible (instead of all), to guarantee at least those get the reward, rather than many wallets failing to meet the threshold.

- **On-Chain Governance or Usage:** dFusion's future might involve a governance token ($VFSN) with on-chain proposals. Keep an eye out:

- If any testnet governance votes or token airdrops via Galxe/OATs occur (the search result suggests whitelists via Galxe in some campaigns [72]), script participation by interacting with those platforms' APIs (Galxe has an API for querying tasks, but manual completion may be required for tasks like Twitter follow).
- For now, the main actionable step is the staking.

Tools & Integration: - **Web Scraping/Requests:** To integrate with dFusion's site for off-chain contributions or linking accounts. Possibly use `requests` to hit their endpoints (if you capture a request like `POST / api/contribute` with JSON payload when adding knowledge, you could replicate it with your own content). - **Error handling:** Off-chain tasks are prone to failures (e.g., content rejected by AI filters). The bot should log responses and perhaps back off if contributions don't succeed. It might be wise to **not automate content submission** and focus on the guaranteed on-chain parts (staking and wallet linking). - **EVM Web3:** likely needed for staking if on Ethereum. If $VANA and $VFSN are ERC-20, treat them like any token: get contract addresses, ABIs (ERC-20 standard ABI for approve/transfer), and the staking contract ABI (might be a simple `deposit(uint256)` function).

Sybil Consideration: dFusion's design encourages unique contributions, which bots could struggle with: - If you do attempt contributions, have each bot wallet submit *distinct data*. You could fetch random Wikipedia paragraphs or YouTube links as dummy contributions. But ensure they are contextually relevant (or at least not obvious spam) to avoid bans. - Staking $VANA is straightforward but on-chain visible. If many wallets stake exactly 100,000 VANA at the same time, that's conspicuous. Stagger staking times and even amounts (some stake 100k, some 110k, some 80k if the airdrop might also reward smaller stakes proportionally – unclear, but variation looks more organic).

ZenithX – Ethereum-based Digital Asset Exchange

About ZenithX: ZenithX is an Ethereum-based platform for trading digital assets (possibly an NFT marketplace or a social token exchange). It acts as a "large digital ledger" for selling digital goods and aims to be user-friendly with minimal capital required [73]. Its token, **ZENX**, is used for trading, supporting creators, and accessing content [74]. Prior early users received a feature ("ETH Mainnet Module") as a reward, hinting that using the platform early could yield airdropped tools or tokens [75].

Programmatic Interaction (Mainnet or Testnet):

- **Platform Access:** First, identify if ZenithX is live and if it has an API:
- If it's an NFT or content platform, check for developer docs or an open API (some platforms offer GraphQL endpoints for marketplace data).

- If not public, the bot can still use web3 if ZenithX's functionality is on-chain. For example, if it's an NFT marketplace on Ethereum, each listing/purchase might be a transaction through a known contract (like a modified version of OpenSea's Seaport or a custom contract).

- **Trading Digital Assets:** To mimic usage:

- **Listing an Item:** If you have a content creation aspect (e.g., creating a "unique online content" as mentioned [76]), you may need to mint an NFT or create a listing. This could be on-chain (minting an NFT representing your content):
    - Use an ERC-721 contract (if ZenithX has its own, use that; or deploy a dummy one if needed).
    - The bot can call `mint()` on ZenithX's content token contract to create a new asset (if allowed).
    - Then call the marketplace contract to list it for sale (likely a function like `createOrder(tokenAddress, tokenId, price)`).
- **Buying/Backing Artists:** ZenithX allows backing creators and trading their content [77]. This suggests a social token or personal token concept. Possibly each creator has a token (ERC-20 or ERC-1155) you can buy:
    - If an API exists, use it to get a list of creators or tokens.

    - Then use web3 to buy a small amount of one creator's token or content. For instance, if creators' tokens are traded through a Uniswap-like contract, the bot can swap some ETH->CreatorToken via that contract.
    - Or if backing is done via a specific contract call (`back(creatorId, amount)`), call that with a small `amount`.

- **Using Features:** The mention of an "ETH Mainnet Module" given free to users suggests that interacting with new features could be key. Stay updated via ZenithX's announcements. If they introduce a feature (like a module or new service), have the bot try it:

    - For example, if they allow connecting an Ethereum wallet to use some tooling, do that connection via web3 or API.
    - If they launch a testnet (less likely since described as mainnet-based), then do testnet tasks similarly.

- **Token Interaction (ZENX):** If ZENX is already live or on testnet:

- Treat it as ERC-20. The bot can acquire small amounts (maybe via a faucet if testnet, or via swap if mainnet and listed).
- Use ZENX in a transaction on the platform. E.g., pay for a content subscription or tip a creator with ZENX if such actions exist.
- Alternatively, just hold some ZENX to be eligible for any holder airdrop (some projects reward early holders). The bot could swap a bit of ETH for ZENX using Uniswap (if trading live) – done via Python by calling Uniswap's router contract:

```
uniswap = w3.eth.contract(UNISWAP_ROUTER, abi=UNI_ROUTER_ABI)
# Swap ETH for ZENX (assuming ZENX has a pool with WETH):
path = [WETH_ADDRESS, ZENX_ADDRESS]
tx = uniswap.functions.swapExactETHForTokens(
    0, path, wallet_address, int(time.time())+60
).buildTransaction({"from": wallet_address, "value": w3.toWei(0.01,
'ether')})
```

Sign and send to execute the swap. Now the wallet holds ZENX.

Note: Without official docs, some of this is speculative. The strategy is to **demonstrate typical user actions**: creating content, buying/selling/trading assets, and using the platform's native token.

Modularity: The bot should encapsulate ZenithX interactions in a module (e.g., `zenithx_module.py`) that handles all these steps. If ZenithX later publishes an API, swap out direct web3 calls with API calls accordingly.

With the above platform-specific strategies, the bot will cover a broad range of interactions. Next, we discuss how to incorporate **randomized scheduling and Sybil avoidance**, and how to design the bot architecture for maintainability and scalability.

Incorporating Randomness and Avoiding Sybil Behavior

To make the bot-operated wallets appear as independent legitimate users (and not a coordinated Sybil attack), **programmed randomness and variability** must be woven into both scheduling and execution:

- **Randomized Task Scheduling:** Using APScheduler (or Celery Beat), schedule tasks with randomness:
- Stagger the execution times for each wallet. For instance, rather than every wallet bridging at 3:00 PM, assign each a random minute/second within a window (or even a random hour in the day). APScheduler allows adding a jitter or scheduling jobs at random intervals. For example, schedule daily tasks with a variance:

```
import random
from apscheduler.schedulers.background import BackgroundScheduler
sched = BackgroundScheduler()
# Schedule daily job between 8-10 AM
exec_time = datetime.time(hour=8 + random.randint(0,2),
minute=random.randint(0,59))
sched.add_job(run_daily_plushie_tasks, 'cron', hour=exec_time.hour,
minute=exec_time.minute)
```

  Each wallet's `run_daily_plushie_tasks` job could be set up with different random times by generating `exec_time` per wallet.

- For event-driven tasks (like reacting to a new proposal or testnet event), introduce a random delay before responding. Example: if a new governance vote is detected, each bot waits a random 0–120 minutes before voting. This prevents all votes clustering in the same block or minute.

- **Randomized Transaction Data:** Vary the parameters of actions:

- **Amounts:** When possible, randomize amounts within realistic ranges. If bridging ETH, one wallet might bridge 0.1 ETH, another 0.2 ETH, etc., instead of all bridging an identical amount. Ensure amounts remain small if real value is involved (so as not to risk much capital) but still varied.
- **Order of Actions:** Do not execute tasks in the same sequence for every wallet. For example, on Mitosis, one wallet might do faucet -> opt-in -> deposit, while another does faucet -> deposit (skipping opt-in until later, or performing the game before deposit). Shuffle the order of sub-tasks:

```
actions = [do_opt_in, do_deposit, do_vote, do_game]
random.shuffle(actions)
for action in actions:
    action(wallet)
```

  This makes the pattern of interactions unique per wallet.
- **Skipping and Failing Gracefully:** Real users sometimes miss tasks. Intentionally have a subset of wallets skip a day or two of activity. For instance, not every wallet needs to perform **every** daily PlushieAI spin or every vote on Mitosis. Randomly decide that 10% of the time a wallet "takes a

break". This absence of perfect participation can make them seem more human (some users are lazy or forgetful).

- **Transaction Metadata:** If possible, randomize gas prices (within a safe range) and include subtle differences in transaction nonce spacing. For example, wait a random extra few seconds after receiving a transaction receipt before sending the next, to avoid an almost deterministic cadence of transactions.

- **Network Diversity:** If all transactions originate from the same IP or RPC node, it could be a giveaway:

- Use multiple RPC endpoints (Alchemy, Infura, public nodes, etc.) and assign different nodes to different wallets. For example, 5 wallets use one RPC URL, 5 use another. This distributes the source of transactions.
- Similarly, if using proxies for web interactions (PlushieAI, etc.), route different wallets through different proxy IPs or VPN endpoints. This prevents the platform from seeing one IP doing everything.

- Vary **User-Agent strings** if using web requests or automation. When simulating browsers, use a variety of agent profiles (some Chrome, some Firefox on different OS).

- **Human-like Behavior Simulation:**

- Insert delays between multi-step processes to simulate "think time". E.g., when the bot uses a dApp: connect wallet -> wait 5-30 seconds -> perform action -> wait 10-60s -> perform next action. Real users click around, read, etc., so not doing everything within milliseconds is important.
- Incorporate conditional logic that sometimes cancels an action. For instance, maybe a wallet connects to a site but then doesn't actually execute a trade because "they changed their mind". This could be simulated by occasionally skipping an interaction after a login.
- Use **logging and monitoring** to ensure the randomness stays effective and doesn't accidentally sync up. Over time, check that tasks remain desynchronized.

By implementing the above, the bot's pattern of life per wallet will be distinct. The goal is that even if someone analyzes on-chain data or platform usage data, the activities look like a bunch of individual enthusiasts rather than a single scripted entity. This dramatically lowers the risk of being flagged as a Sybil farm and thus **improves the chances of qualifying for airdrops** (many projects disqualify detected Sybils).

Architectural Design for Modular Bot Construction

To manage the complexity of multiple platforms and tasks, the bot's architecture should be **modular, extensible, and maintainable**. Key design considerations:

- **Module Per Platform:** Organize the codebase such that each target platform has its own module or class encapsulating its logic. For example:
- `monad_module.py` handles Monad-related functions (bridge, nft_purchase, etc.).
- `eclipse_module.py`, `mitosis_module.py`, etc., similarly.

- Each module exposes a common interface, e.g., a `run_all_tasks(wallet)` method that executes that platform's sequence for a given wallet.
- This modular approach makes it easy to add or remove platforms. If a new airdrop opportunity arises, you add a new module without touching the core scheduler logic.

- **Core Scheduler/Controller:** A central scheduler (APScheduler or Celery Beat) triggers tasks. It can maintain a registry of platform modules and iterate through wallets:

```
PLATFORMS = [monad, abstract, eclipse, mitosis, pumpfun, plushie, stau,
dfusion, zenithx]
for wallet in wallets:
    for platform in PLATFORMS:
        sched.add_job(platform.run_all_tasks, args=[wallet],
trigger='cron', ...)
```

  However, running all platforms in one job per wallet might be too sequential; you might schedule each platform task separately with its own frequency. For instance, PlushieAI tasks daily, but bridging tasks maybe weekly. So schedule per platform per wallet with appropriate intervals.

- **Task Queue vs Direct Calls:** Consider using **Celery** as a task queue for asynchronous execution:

- Celery workers can run tasks in parallel (e.g., handling multiple wallets at once, if you want concurrency).
- Define Celery tasks for each action, e.g., `bridge_eth_to_abstract.delay(wallet)` or `mint_domain_eclipse.delay(wallet)`.
- Use **Celery Canvas / Chains** to string together sequences for a wallet. For example, a Celery chain could ensure a sequence: faucet -> deposit -> withdraw, while Celery itself parallelizes across wallets.

- Celery's retry mechanism is helpful. If a transaction fails (maybe due to nonce issues or temporary RPC failure), you can have the task auto-retry after a delay. This is harder to do with APScheduler alone.

- **State and Configuration Management:** Use a configuration file or database to store:

- Wallet details: private key (encrypted, if possible), associated social accounts/tokens (for PlushieAI, Mitosis faucet, etc.), and any state (e.g., last time tasks completed, whether certain one-time tasks like "opt-in" have been done).
- Platform config: contract addresses, RPC URLs, etc., for each platform. This should be easily updatable (if a platform moves from testnet to mainnet, just update RPC and contract addresses in config).

- You could use YAML/JSON for simple config, or a small SQLite/JSON file for per-wallet state. For example, mark in the DB that wallet X has already minted an Eclipse domain, so you skip that one-time task next run.

- **Wallet Management & Key Handling:** Since the bot uses private key based wallets:

- Keep private keys **securely**. Avoid hardcoding them. Instead, read from an encrypted file or an environment variable. For instance, use `python-dotenv` to load keys from a `.env` file (not tracked in git).
- If many wallets are used, consider implementing a wallet loader that can read a list of keys (or mnemonic phrases) from a secure store. Example:

```
import os
wallets = []
for i in range(NUM_WALLETS):
    key_hex = os.getenv(f"WALLET{i}_KEY")
    wallets.append(Account.from_key(key_hex))
```

- *Modularity in wallet usage:* The wallet object could be passed into platform modules. Define a Wallet class that abstracts details like chain accounts:

```
class Wallet:
    def __init__(self, eth_key, sol_key=None):
        self.eth_account = Account.from_key(eth_key)
        self.sol_keypair = Keypair.from_secret_key(sol_key) if sol_key else
None
        self.address = self.eth_account.address  # For EVM tasks
        self.public_key = self.sol_keypair.public_key if sol_key else None
```

  This way, platform modules can use `wallet.address` or `wallet.public_key` as needed, without each module managing raw keys separately.

- If using many wallets concurrently, be mindful of **nonce management** on each chain. For Ethereum, if the bot sends multiple transactions quickly from one wallet, track the nonce and manage it manually:

  - Use `w3.eth.get_transaction_count(address)` to get current nonce at start.
  - Increment nonce locally as you build multiple transactions in one run.
  - Celery tasks might conflict if not coordinated; consider a per-wallet lock or have one worker handle all tasks for a given wallet sequentially to prevent nonce race conditions.

- **Abstraction of Interaction Layer:** Create utility functions for repeated patterns:

- EVM utilities: e.g., `send_erc20(token_address, to, amount, wallet)` or `call_contract_function(contract, func_name, args, wallet)`. These wrap the building, signing, and sending of a transaction, returning the tx hash or receipt. Having these will reduce code duplication across modules.
- Solana utilities: e.g., `send_solana_tx(instructions, signers)` that constructs a Transaction, signs with the provided signers (the wallet's Keypair), and sends it via RPC, optionally confirming it.
- Web/HTTP utilities: e.g., a generic function `perform_web_login(url, wallet)` that navigates a headless browser or sends a signature to login, which modules like PlushieAI and dFusion can use.

- Randomness helpers: e.g., `wait_random(min_secs, max_secs)` that does a time.sleep for a random duration (to easily insert delays).

- **Error Handling and Alerts:** Because the bot operates autonomously, implement robust error handling:

- Surround external calls with try/except. If a transaction fails (throws an error), catch it and log which wallet, which platform, and the error (e.g., nonce too low, RPC unavailable, out-of-gas).
- Use logging with timestamps and maybe push critical issues to an alert (could integrate with a Discord webhook or email if a wallet consistently fails tasks).

- Continue with other tasks even if one fails (isolate failures so one platform's error doesn't halt the whole bot). This can be achieved naturally with Celery (tasks are independent), or with APScheduler by configuring `misfire_grace_time` and error callbacks.

- **Scalability:** If you eventually manage dozens of wallets, you may want to **parallelize** the workload:

- Running tasks sequentially might become too slow. Celery allows scaling horizontally (multiple worker processes or even distributed across servers). APScheduler in a single process might become a bottleneck if each job is waiting on network calls.
- A hybrid approach: Use APScheduler to kick off high-level jobs (like "run daily tasks for all wallets on Platform X"), and within that job, dispatch Celery subtasks for each wallet. This way scheduling is centralized but execution is concurrent.
- Example:

```
def run_mitosis_all_wallets():
    for wallet in wallets:
        mitosis_tasks.delay(wallet.id)
sched.add_job(run_mitosis_all_wallets, 'interval', days=1)
```

  Here, `mitosis_tasks` is a Celery task that internally calls the sequence for that wallet.
- Ensure your system (and RPC endpoints) can handle concurrency. If you spam 50 transactions at once, some might drop. Rate-limit within tasks if needed or use Celery's concurrency settings to only allow, say, 5 transactions per second overall.

By following this design, the bot will be easier to extend. For instance, if next month a new project "OmegaNetwork" appears with an airdrop, you can just add `omega_module.py`, implement its tasks (using the existing utility functions where possible), add it to the platform list, and schedule its jobs.

APScheduler vs Celery for Task Scheduling

The current bot uses **APScheduler** for randomized scheduling. As we integrate more tasks and possibly need more robustness, it's worth comparing APScheduler to **Celery** (with Celery Beat for scheduling) and possibly using them in tandem:

- **APScheduler (Advanced Python Scheduler):**

- *Pros:* Simple to use in a single-process scenario. It runs inside your Python application and can schedule jobs with cron-like syntax or intervals. Great for triggering functions at certain times or after delays. It supports adding jobs at runtime, and you can easily incorporate jitter manually (by computing random times or using `misfire_grace_time` to allow slippage).
- *Cons:* It's single-process (unless using the experimental distributed options) – meaning all tasks run in the same process. If a job runs long or hangs, it could delay others. In our context, interacting with networks can sometimes hang or slow down, which might throw off the scheduler timing. Also, if the process crashes, all scheduled jobs stop (no built-in persistence or fault tolerance, unless you use the job stores to DB and custom restart logic).

- APScheduler is excellent for smaller scale or where precise distribution isn't needed. It can handle dozens of tasks, but hundreds might become unwieldy.

- **Celery (Distributed Task Queue):**

- *Pros:* Designed for scalability and reliability. Tasks are queued in a broker (Redis, RabbitMQ, etc.) and executed by worker processes. You can run many workers for parallelism. Celery has **retries, timeouts, and error handling** features built-in. If a worker crashes, tasks can be picked up by another. Celery Beat (or a separate scheduler like cron feeding tasks into Celery) can schedule tasks similarly to APScheduler. It's a more robust solution for production and large-scale task execution.
- *Cons:* More complex to set up. Requires running a broker service and one or more worker processes alongside your main app. Overkill for a small, local bot with a handful of tasks. Debugging can be a bit more involved (need to look at worker logs, etc.). Also, Celery tasks are typically stateless (passing arguments), which means sharing large context (like complex objects) isn't straightforward – though in our use case, passing a wallet address or ID is fine.
- Celery's scheduling (via Beat) isn't as straightforward for random times, but you can create periodic tasks and then add randomness in the task execution (like tasks that sleep random durations or use countdowns).

Recommendation: For this airdrop bot, a **hybrid approach** can yield the best results: - Continue to use **APScheduler** in the main bot controller for high-level scheduling because it's easy to set up one-off times like "every day at ~9am" or "every 3 days" with some randomness. - Offload the actual execution of heavy tasks to **Celery workers**: - APScheduler job would simply enqueue Celery tasks. E.g., schedule a job for each morning that does: `celery_app.send_task("mitosis.run_all", args=[wallet_id])` for each wallet, or a group of such tasks. - This way, the timing/jitter logic is handled by APScheduler (in one process), and the parallel execution and robustness is handled by Celery. - If maintaining two systems is too much, you could also try using just Celery Beat for scheduling. Celery Beat can schedule periodic tasks, but adding random offsets might require custom crontab entries or logic inside tasks to sleep. - Another improvement: use Celery's result backend to track task success/failure. You can have a monitoring job (maybe APScheduler-driven) that checks for any failed tasks in the last day and logs or alerts them.

Celery's advantage will become evident if you scale up the number of wallets or tasks. It will allow the bot to handle many simultaneous operations (e.g., bridging on multiple chains concurrently) without one action blocking another. It also gives the opportunity to distribute tasks across multiple machines if needed (for example, if interacting with rate-limited APIs, you could run workers from different IPs).

APScheduler's advantage is simplicity – if the bot remains at, say, <50 tasks a day, one process can likely manage it with sleep/wait time in between. But if the bot grows (both in tasks and complexity), transitioning to Celery is wise. In summary, **start with APScheduler for scheduling and integrate Celery for executing tasks concurrently and reliably**, achieving a balance of simplicity and power.

Code Snippets and Libraries

Below are some code-level illustrations of key operations, using Python libraries:

1. Web3.py Transaction Example (EVM): Sending a transaction (e.g., approving and bridging tokens):

```python
from web3 import Web3
from eth_account import Account

w3 = Web3(Web3.HTTPProvider("https://sepolia.infura.io/v3/YOUR_PROJECT_ID"))
acct = Account.from_key("0x...<PRIVATE_KEY>")  # wallet private key
bridge_addr = Web3.to_checksum_address("0x1234...")  # Abstract bridge contract
bridge_abi = [...]  # ABI for the bridge contract

# Prepare a deposit transaction on Sepolia -> Abstract bridge
bridge = w3.eth.contract(address=bridge_addr, abi=bridge_abi)
tx = bridge.functions.depositETH().build_transaction({
    "from": acct.address,
    "value": w3.toWei(0.1, 'ether'),
    "nonce": w3.eth.get_transaction_count(acct.address),
    "gas": 1000000,
    "gasPrice": w3.toWei(5, 'gwei')
})
signed_tx = acct.sign_transaction(tx)
tx_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
print("Deposit submitted, hash:", tx_hash.hex())
# The bot would later listen for or poll the Abstract L2 for credit.
```

This snippet uses **web3.py** to build and send a deposit transaction. Note we manually set gas/gasPrice; in practice, use `w3.eth.estimate_gas(tx)` and perhaps a gas price oracle for mainnet.

2. Solana-py Transaction Example: Creating and sending a basic Solana transfer (could be adapted for Eclipse or Pump.fun actions):

```python
from solana.rpc.api import Client
from solana.keypair import Keypair
from solana.system_program import TransferParams, transfer
from solana.transaction import Transaction

client = Client("https://api.devnet.solana.com")  # or Eclipse testnet RPC
sender = Keypair.from_secret_key(bytes([...]))    # sender private key bytes
recipient_pubkey = "Fg6PaFpoGXkYsidMpWxTW...destination_address..."
```

```
# Build a transfer of 0.01 SOL
txn = Transaction()
txn.add(transfer(TransferParams(
    from_pubkey=sender.public_key,
    to_pubkey=Web3.to_public_key(recipient_pubkey),
    lamports=10000000  # 0.01 SOL in lamports
)))
# Send transaction
response = client.send_transaction(txn, sender)
print("Solana transfer result:", response)
```

This uses the Solana JSON RPC. For Eclipse, you'd change the RPC URL and ensure the keypair is valid for the Eclipse network (which in testnet likely uses the same key scheme as Solana). The code sends a small SOL amount to another address.

3. APScheduler with Random Jitter: Scheduling a repetitive job with randomized interval:

```
import random, datetime
from apscheduler.schedulers.background import BackgroundScheduler

scheduler = BackgroundScheduler()

def run_plushie_spin(wallet):
    # Code to perform the wheel spin for given wallet
    ...

# Schedule each wallet's daily spin at a random hour/minute
for wallet in wallets:
    hour = random.randint(0, 23)
    minute = random.randint(0, 59)
    scheduler.add_job(run_plushie_spin, 'cron', args=[wallet], hour=hour, 
minute=minute,
                      id=f"plushie_{wallet.address}")
```

This sets each wallet's spin at a random time of day. We could also add jitter on a smaller scale by computing next run time dynamically each day.

4. Celery Task Definition Example: A Celery task for bridging on Abstract:

```
from celery import Celery

app = Celery('airdrop_bot', broker='redis://localhost:6379/0')

@app.task(bind=True, max_retries=3)
```

```
def abstract_bridge_task(self, wallet_addr, private_key, amount_eth):
    try:
        w3 = Web3(Web3.HTTPProvider("https://sepolia.infura.io/v3/..."))
        acct = Account.from_key(private_key)
        # ... build tx as above ...
        tx = bridge.functions.depositETH().build_transaction({...})
        signed = acct.sign_transaction(tx)
        tx_hash = w3.eth.send_raw_transaction(signed.rawTransaction)
        return tx_hash.hex()
    except Exception as e:
        # retry after a delay if something goes wrong
        raise self.retry(exc=e, countdown=60 + random.randint(0,30))
```

This Celery task will attempt to perform the bridge and, if it fails (say due to a nonce issue or RPC failure), retry up to 3 times with a 1-1.5 minute delay. The `bind=True` allows using `self.retry` and tracking state. We pass in minimal data (address and key) to avoid storing large state in the task.

5. Using Web Requests for Off-chain actions: (Pseudo-code for brevity)

```
import requests

def plushie_spin(wallet):
    # Assume we have a session token from prior login
    token = wallet.plushie_session
    headers = {"Authorization": f"Bearer {token}"}
    resp = requests.post("https://airdrop.plushie.ai/api/spin", headers=headers)
    if resp.status_code == 200:
        result = resp.json()
        print(f"Wheel spun for {wallet.address}, won {result['prize']} PLSH")
    else:
        print(f"Spin failed for {wallet.address}: {resp.text}")
```

This illustrates how the bot might call a hypothetical PlushieAI endpoint for spinning the wheel, using a stored auth token. In practice, you'd need to obtain `wallet.plushie_session` by performing the login steps (which could also be coded with `requests` if the login flow is known, or via an automated browser login captured once).

Libraries Summary: Throughout the development, leverage the following libraries:

- **Web3.py** for Ethereum-compatible chain interactions [1] .
- **eth-account (included in web3)** for signing transactions with private keys.
- **Solana-py** and **AnchorPy** for Solana/SVM chains (Eclipse, Pump.fun).
- **APScheduler** for simple scheduling of tasks with flexibility in Python.
- **Celery** (with a broker like Redis) for scaling and robust task execution.

- **Requests** (or **httpx**) for REST API calls to platforms (PlushieAI, etc.).
- **Selenium** for any web automation that cannot be done via direct requests (optional, used with care due to overhead).
- **Logging** via Python's `logging` module to keep structured logs of all actions (set up different log files per platform maybe, or at least log prefixes to distinguish).

By combining these tools in a cohesive architecture, the bot will be capable of handling complex multi-chain operations, remain resilient via retries and scheduling, and be easily extensible as new airdrop opportunities emerge.

Conclusion

This technical implementation plan provides a comprehensive roadmap for expanding the airdrop farming bot. By modularizing platform interactions, integrating both APScheduler and Celery for intelligent task scheduling, incorporating randomness to evade Sybil detection, and using appropriate Python libraries for blockchain and web interactions, the bot can safely and efficiently farm potential airdrops on **Monad, Abstract, Eclipse, Axiom, Mitosis, Pump.fun, PlushieAI, STAU, dFusion,** and **ZenithX**. Each platform's nuances – from Solana programs to ZK proofs – are handled in a targeted manner. Implementing these recommendations will result in a robust, scalable bot architecture positioned to capture upcoming airdrop rewards while minimizing risks.

Sources:

- Monad is a new high-performance EVM Layer-1 blockchain (testnet launched in 2025) [1].
- Abstract enables cross-chain ETH staking on a ZK-rollup testnet (Sepolia) [8] [9].
- Eclipse is an Ethereum SVM-based Layer-2 combining Solana speed with Ethereum security [17].
- Axiom provides on-chain access to historical data via ZK proofs (Halo2-based ZK coprocessor) [31].
- Mitosis incentivized testnet involved liquidity deposits and cross-chain interactions for $MITO rewards [42] [48].
- Pump.fun is a Solana memecoin launchpad to easily create and trade tokens (potential $PUMP airdrop) [54].
- PlushieAI distributes $PLSH tokens to users engaging with its AI companion via social channels [63] [61].
- STAU Platform uses a crypto token to facilitate trading of gold/jewelry with blockchain security [65] [66].
- dFusion AI rewards contributors to a decentralized knowledge base and involves staking $VANA for future $VFSN tokens [68] [70].
- ZenithX is built on Ethereum to trade digital assets, with a native token ZENX for transactions and supporting creators [73] [74].

---

[1] [4] [5] [6] Guide to Interacting with the Monad Testnet for A Potential Airdrop | CoinGecko
https://www.coingecko.com/learn/monad-potential-crypto-airdrop

[2] Potential Monad Airdrop » How to be eligible?
https://airdrops.io/monad/

3 2 Public / Free Monad RPC & API Endpoints - CompareNodes.com
https://www.comparenodes.com/library/public-endpoints/monad/

7 Abstract RPC Node Endpoints, APIs & Tools | QuickNode
https://www.quicknode.com/chains/abstract

8 9 10 11 15 Potential Abstract Airdrop » How to be eligible?
https://airdrops.io/abstract/

12 Connect to Abstract - Abstract
https://docs.abs.xyz/connect-to-abstract

13 14 Bridges - Abstract
https://docs.abs.xyz/tooling/bridges

16 Introduction - Abstract
https://docs.abs.xyz/overview

17 Eclipse
https://www.eclipse.xyz/

18 Eclipse Testnet Airdrop Farming Tutorial | Airdrop To Farm For 2024
https://www.youtube.com/watch?v=1IDUTXYIO6s

19 What is Eclipse? How to Claim the Latest Airdrop Rewards - Bitrue
https://www.bitrue.com/blog/what-is-eclipse-how-to-claim-the-latest-airdrop-rewards

20 21 RPC & Block Explorers | Eclipse Documentation
https://docs.eclipse.xyz/developers/rpc-and-block-explorers

22 25 26 27 29 30 Eclipse Token Airdrop: Complete Guide
https://www.oneclick.fi/blog/eclipse-airdrop-guide

23 24 28 Getting Started | Eclipse Documentation
https://docs.eclipse.xyz/

31 Open-sourcing the Axiom ZK Circuits
https://blog.axiom.xyz/open-sourcing-the-axiom-zk-circuits/

32 A deep dive into Axiom's Halo2 circuits - The Trail of Bits Blog
https://blog.trailofbits.com/2025/05/30/a-deep-dive-into-axioms-halo2-circuits/

33 Axiom Testnet Guide - Step-by-step. : r/Whitelist_Airdrop - Reddit
https://www.reddit.com/r/Whitelist_Airdrop/comments/1aprrxb/axiom_testnet_guide_stepbystep/

34 Potential Axiom Airdrop » How to be eligible?
https://airdrops.io/axiom/

35 Axiom
https://www.axiom.xyz/

36 axiom-crypto/axiom-eth - GitHub
https://github.com/axiom-crypto/axiom-eth

37 Axiom - Zero Knowledge Tools - Alchemy
https://www.alchemy.com/dapps/axiom

[38] autonomous-airdrop-example/README.md at main · axiom-crypto ...
https://github.com/axiom-crypto/autonomous-airdrop-example/blob/main/README.md

[39] Mitosis Testnet Airdrop [Limited Time] - Binance
https://www.binance.com/ar/square/post/17435845736402

[40] Mitosis Expedition Guide for Beginners: How to Earn MITO Points
https://university.mitosis.org/mitosis-expedition-guide-for-beginners-how-to-earn-mito-points/

[41] Potential Mitosis Airdrop » How to be eligible?
https://airdrops.io/mitosis/

[42] [45] [46] [47] [48] [51] [52] [72] Mitosis Airdrop guide: Steps to potential Reward | CryptoRank.io
https://cryptorank.io/drophunting/mitosis-activity177

[43] [44] [53] Mitosis Testnet: RPC and Chain Settings
https://thirdweb.com/mitosis-testnet

[49] Arbitrum x Mitosis: Scaling Cross-Chain Liquidity with Modular Power
https://university.mitosis.org/arbitrum-x-mitosis-scaling-cross-chain-liquidity-with-modular-power/

[50] Modular LP's famous testnet airdrop tutorial: Mitosis - Binance
https://www.binance.com/en/square/post/5304042438842

[54] [56] How to Create Memecoins on Pump.fun for Their Upcoming Airdrop | CoinGecko
https://www.coingecko.com/learn/pump-fun-guide-how-to-create-your-own-memecoins

[55] Pump.fun token rumors mount as protocol revenue drops 66%
https://cointelegraph.com/news/pump-fun-token-rumors-1-billion-sale-airdrop

[57] pump-fun-official/pump-fun-bot - GitHub
https://github.com/pump-fun-official/pump-fun-bot

[58] Pump Fun API - Bitquery V2 API Docs | Blockchain Data API (V2)
https://docs.bitquery.io/docs/examples/Solana/Pump-Fun-API/

[59] [60] [62] [63] [64] Plushie AI Airdrop
https://www.bitrue.com/blog/what-is-plushie-ai-airdrop-a-complete-guide-to-earning-free-tokens

[61] [65] [66] [68] [69] [73] [74] [75] [76] [77] Crypto Airdrops in 2025: All You Need to Know - CoinSwitch
https://coinswitch.co/switch/crypto/crypto-airdrops/

[67] dFusion AI
https://www.dfusion.ai/

[70] dFusion AI Protocol - X
https://x.com/dFusionAI/status/1871662134490218924

[71] dFusion AI: Genesis Knowledge
https://genesis.dfusion.ai/