XAUT/USDT Flash Loan Arbitrage Bot – System Blueprint

Amir Lehmam | 04/05/2025

Table of Content

Overview

This blueprint outlines a **high-performance flash loan arbitrage bot** that targets the XAUT/USDT trading pair across multiple decentralized exchanges (DEXs) on Ethereum and several EVM-compatible chains (Arbitrum, Polygon, BSC, etc.). The system uses a multi-language architecture to optimize for speed, safety, and flexibility: **Rust** for low-latency on-chain interactions, **OCaml** for analytical tooling and configuration, **Solidity** for atomic trade execution via flash loans, and **Python** for orchestration, machine learning (ML) integration, and monitoring. All arbitrage transactions are executed **atomically** within a single blockchain transaction using Aave flash loans, so if profit conditions aren't met the trades revert with no loss. The design prioritizes **deterministic execution** and security from day one, employing best practices like reentrancy guards and TWAP (Time-Weighted Average Price) oracle checks to mitigate flash-loan manipulation risks.

Key Goals and Features:

- **Multi-DEX, Multi-Chain Monitoring:** Continuously track XAUT/USDT prices on all available DEX pools (Uniswap, Sushiswap, PancakeSwap, etc.) across Ethereum mainnet and popular Layer-2/sidechains. This breadth increases chances of finding price disparities, especially since XAUT (Tether Gold) liquidity may vary by network.

- **Atomic Flash-Loan Arbitrage:** Execute two-leg arbitrage trades (buy low, sell high) in one atomic transaction using flash loans (e.g. borrowing USDT from Aave). The entire sequence (loan, swap, counter-swap, repay) happens within one block, guaranteeing the loan is repaid or the transaction reverts. This approach means no upfront capital is at risk aside from gas fees.

- **High Performance Hot Path:** Use Rust in the critical path to achieve near-real-time price sensing and trade execution with minimal latency. Rust's speed and memory safety allow the bot to handle rapid price updates and craft transactions faster than a Python script could. This performance is crucial since arbitrage opportunities are short-lived and competitive.

- **Advanced Analytics Off-Path:** Use OCaml for non-time-critical logic like configuration management and opportunity analysis. OCaml's strong typing and functional paradigm help create reliable tooling for, e.g., computing optimal trade sizes or scoring potential opportunities without impacting the reaction speed of the main bot.

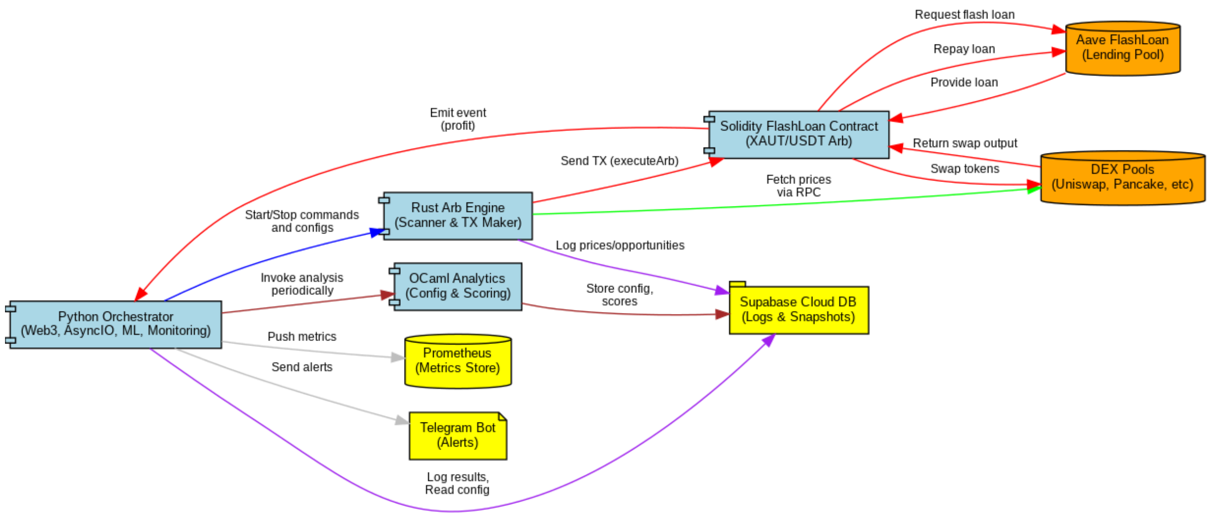
- **Orchestration & Intelligence:** Employ Python (with asyncio) as the "glue" – coordinating components, handling asynchronous events, and integrating ML models. Python's rich ecosystem makes it ideal for implementing monitoring (via web3.py for blockchain events), data logging, and applying any machine learning heuristics (e.g. predicting slippage or latency risk) to refine decision-making.

- **Integrated Logging & Alerts:** All components will feed into a unified logging and monitoring setup. We leverage **Supabase** (hosted Postgres) for quick prototyping of a database to store bot logs, price snapshots, and trade records. We also integrate **Prometheus** for metrics (e.g. latency, profit, success rate) and a Telegram bot for real-time alerts (e.g. when a big arbitrage profit is captured or if an error occurs).

- **Minimal-cost Infrastructure:** The deployment stack emphasizes free or low-cost resources suitable for a prototype: using public RPC endpoints or a local light node, Supabase's free tier for the database, and Docker containers to deploy services on a single machine. This ensures a developer can get the system running end-to-end without immediate investment, while keeping the design scalable for later growth.

Architecture Overview

At a high level, the system is composed of four main components (Solidity contract, Rust engine, OCaml analytics, Python orchestrator), each focused on a specific concern. These components interact through well-defined interfaces – primarily via on-chain calls and off-chain data stores – to collectively achieve fast and safe arbitrage. The **diagram below** illustrates the architecture and data flow between components and external services:


Request flash loan
Aave FlashLoan (Lending Pool)
Repay loan
Provide loan
Emit event (profit)
Solidity FlashLoan Contract (XAUT/USDT Arb)
Return swap output
Swap tokens
DEX Pools (Uniswap, Pancake, etc)
Send TX (executeArb)
Fetch prices via RPC
Start/Stop commands and configs
Rust Arb Engine (Scanner & TX Maker)
Log prices/opportunities
Invoke analysis periodically
OCaml Analytics (Config & Scoring)
Store config, scores
Supabase Cloud DB (Logs & Snapshots)
Python Orchestrator (Web3, AsyncIO, ML, Monitoring)
Push metrics
Prometheus (Metrics Store)
Send alerts
Telegram Bot (Alerts)
Log results, Read config

Description: High-level architecture of the XAUT/USDT arbitrage bot. Blue components are custom modules (Python orchestrator, Rust engine, OCaml analytics, Solidity contract). Orange cylinders are external on-chain systems (Aave lending pool, DEX pools). Yellow elements are off-chain support (Supabase DB, Prometheus, Telegram). Arrows indicate data/control flow: green = price data queries, red = on-chain arbitrage execution, purple = logging/config data, gray = monitoring & alerts.

As shown above, the **Rust Arbitrage Engine** continuously monitors DEX pools (green arrows) and, upon spotting a price imbalance, triggers the **Solidity flash-loan contract** to execute the arbitrage (red arrows). The **Solidity contract** interacts with Aave for the flash loan and with the DEXs to swap tokens, then returns profits. Meanwhile, the **Python Orchestrator** coordinates the system: it supplies configuration to Rust, listens for events (e.g. a profit event from the contract), and logs data to the **Supabase** database (purple arrows). The orchestrator also exposes metrics to **Prometheus** and sends alerts via **Telegram** (gray arrows). Periodically, an **OCaml Analytics** module may be invoked (brown arrows) to update DEX configuration or compute opportunity scores, which are stored for the Rust engine to use. This separation of concerns allows each part to be **optimized for its role** – for example, Rust handles time-sensitive tasks in parallel across chains, while Python/OCaml handle complex logic and integration in a more flexible environment.

Critically, the design ensures that the **hot execution path (price check → trade tx)** is isolated in the Rust & Solidity components, which can operate very deterministically and fast, whereas **supporting tasks** (analytics, logging, user interaction) happen asynchronously so as not to bottleneck the arbitrage execution. This modular approach also makes the system easier to maintain and extend (for instance, adding a new DEX or chain mainly involves updating the OCaml config and perhaps a Rust adapter, without touching core logic).

Below, we detail each major component, including responsibilities, key modules, and how they work together:

Smart Contract – Flash Loan Executor (Solidity)

Role: The Solidity smart contract executes the arbitrage transaction on-chain, ensuring atomicity and security. It is essentially the "agent" that the off-chain bot calls to perform the flash loan and the two swaps. We will deploy one instance of this contract on each target chain (Ethereum, BSC, Polygon, Arbitrum, etc.), configured with the addresses of the XAUT and USDT tokens on that chain and integrated with that chain's flash-loan provider (e.g. Aave V3 on Ethereum, Aave or other protocols on Polygon, etc.).

Key Functions: The core function (e.g., `executeArbitrage`) will: (1) initiate a flash loan for USDT from a lending pool, (2) upon receiving the USDT, swap USDT→XAUT on the designated DEX where XAUT is cheaper, (3) then swap XAUT→USDT on the DEX where XAUT is pricier, (4) repay the flash loan (USDT + fee), and (5) send any remaining USDT profit to the bot's address. All these sub-steps occur in a single transaction and either **fully complete or abort** (revert) as a whole. The bot passes in parameters to control which DEXs to use and how much to trade (these come from the Rust engine's calculation). For example, parameters might include: the flash loan amount in USDT, the address of the "buy low" DEX (and any data needed for its swap function), and the address of the "sell high" DEX (with corresponding data).

Flash Loan Integration: The contract implements the interface required by Aave (or the respective protocol) for flash loans. On Ethereum, this means inheriting from Aave's `IFlashLoanReceiver` and implementing the `executeOperation` callback. When our contract calls Aave's lending pool contract's `flashLoan` function, Aave will transfer the requested USDT to our contract and then invoke our `executeOperation` function in the same transaction. In `executeOperation`, we embed the arbitrage logic – performing the token swaps and ensuring we have at least `amount+fee` USDT to return. After this function finishes, control returns to Aave which verifies the loan repayment. If we fail to repay exactly on time, Aave (and the whole transaction) will revert, which protects the lender by design. Our contract will be coded to guarantee the repay happens unless something unexpected occurs (in which case revert anyway). Essentially, **"Use the chosen protocol to initiate the flash loan, ensuring you have a clear plan for repayment within the same transaction."**

DEX Trades: The contract needs to interact with different DEXs (likely Uniswap/SushiSwap clones) to swap tokens. To keep it flexible, we can integrate with common AMM interfaces. For Uniswap V2-like AMMs, we can call their `swap` functions on pairs or use router contracts (e.g., `UniswapV2Router02.swapExactTokensForTokens` for USDT→XAUT and

XAUT→USDT). For Uniswap V3, the logic is more complex (requiring exact path and fee tier), so an alternative is to limit initial scope to v2-style pools or rely on an aggregator or custom logic for v3. In this design, we assume the presence of liquid v2 pools for XAUT/USDT on each chain, or we can deploy a simple adapter in the contract to handle both v2 and v3. The contract will take care of approving the DEX router to spend the flash-loaned tokens and acquired tokens as needed. We hardcode or store the token addresses (XAUT, USDT) in the contract for safety (so it doesn't trade wrong tokens). The specific DEX addresses or call data might be supplied by the bot per transaction to allow flexibility in choosing platforms.

Security Measures: Security is paramount since flash loan bots operate in adversarial environments. We implement the following in the contract:

- *Reentrancy Guard:* The contract uses OpenZeppelin's `ReentrancyGuard` (nonReentrant modifier) on the entry function (`executeArbitrage`) to prevent any reentrant calls. This ensures that even though the contract will call out to external protocols (DEXs), those external calls cannot re-enter our contract's sensitive functions unexpectedly. (DEXs like Uniswap shouldn't call back, but as a general best-practice we include the guard.)

- *Access Control:* Only an authorized address (the bot's EOA address) can call the arbitrage execution function. We enforce this with an `onlyOwner` modifier (using OpenZeppelin's Ownable, for example). This prevents arbitrary users from triggering our contract with possibly malicious parameters.

- *Slippage and Oracle Checks:* We include parameters and checks to avoid bad trades. The bot will compute expected outcomes (expected amount of USDT after selling XAUT). The contract can require that the amount of USDT obtained at the end is >= `loan + fee + minProfit` or else it reverts (ensuring a minimum profit or break-even). Additionally, to guard against *price manipulation attacks*, we use a TWAP oracle or reference price feed. For example, we can consult a Chainlink price oracle for XAU (gold) vs USD, or a TWAP from Uniswap itself, to verify that the price difference we are arbitraging is genuine and not just due to a momentary manipulation of a single DEX's liquidity. If the on-chain instantaneous price deviates too far from a longer-term average price (or external price), the contract can abort. This prevents scenarios where an attacker could use a flash loan to push the price on one DEX, trick our bot into arbitraging a fake "profit" that disappears once they remove their manipulation (a known vector in some oracle attacks).

- *Failure Handling:* If any step fails (e.g., a swap doesn't execute due to insufficient liquidity or slippage tolerance), the contract will immediately revert, which automatically unwinds the flash loan. The bot off-chain will detect the failure (no profit event emitted) and can log the attempt. Reverts are expected occasionally – the system should tolerate them (besides the lost gas).

Optional Design Elements: We might design the contract to handle *multi-step arbitrage* (triangular trades) in the future or support other token pairs, but for now it's focused on a single round-trip between XAUT and USDT. The contract is fairly self-contained; once deployed, the off-chain bot only needs to call it with the right parameters. By deploying this contract to each chain and supplying the correct addresses (Aave lending pool, DEX factories/routers, token addresses), we create a unified interface for the Rust engine to perform arbitrage on any chain. This **separation of execution logic into a smart contract** aligns with common MEV bot designs that use on-chain execution to bundle multiple swaps (some even use Flashbots or private relay, which could be an enhancement for Ethereum mainnet to avoid public mempool competition).

Real-Time Arbitrage Engine (Rust)

Role: The Rust engine is the brains of real-time operation – it continuously monitors price data and decides when to strike an arbitrage. It runs as a high-performance, async service that interfaces directly with blockchain nodes. The choice of Rust ensures we can maximize throughput (scanning many exchanges and chains in parallel) and minimize latency between detecting an opportunity and launching the transaction. In a domain where **"Speed is crucial in arbitrage trading. Use automated trading systems or an arbitrage bot to execute trades as soon as an opportunity is identified"**, Rust gives us an edge by being compiled and thread-safe, avoiding the overhead of garbage collection or interpreter locks.

Price Monitoring: Upon startup, the Rust service loads the configuration of all XAUT/USDT pools to watch – likely provided as a list of pool addresses or router addresses per DEX and chain (from the OCaml config output or a config file). It then establishes connections to each blockchain network. For each chain, we use an RPC or WebSocket endpoint (e.g., Infura/Alchemy for Ethereum, public RPC for BSC, etc.) and spawn an asynchronous task or thread to handle that chain's data. The engine employs a combination of techniques to track prices in real time:

- **Direct On-Chain Queries:** It can periodically call `getReserves()` on AMM pair contracts (for Uniswap V2 style) or `slot0` on Uniswap V3 pools to get the latest price. This could be done every new block or every few seconds.

- **Event Subscription:** To reduce redundant queries, the engine subscribes to relevant events. For Uniswap V2, listening to the `Sync` events (which update reserves) or `Swap` events of the pair contract can inform us of price changes. For Uniswap V3, `Swap` events carry price info too. Using WebSocket subscriptions, the Rust engine can get pushed events as they happen, rather than polling. In practice, an approach is to subscribe to new block headers and then fetch all target pools' states in that block, or subscribe directly to pool events. The Whack-A-Mole project (a Python arbitrage bot) uses a similar multi-stream approach: *"loop through all chains and start an asynchronous websocket stream to retrieve data for new headers, Uniswap V2 Sync events, Uniswap V3 Swap events."* We will mirror this strategy in Rust: e.g., spawn tasks to listen to each DEX's events on each chain concurrently. Rust's async runtime (Tokio) can manage hundreds of such streaming tasks efficiently.

The outcome is that the engine maintains an updated view of the **current price of XAUT in USDT on each DEX** across chains. For example, if Uniswap V3 on Ethereum has XAUT at $1850 and SushiSwap on Ethereum has it at $1840, the engine will detect a ~$10 difference. Similarly across chains (though cross-chain arb isn't atomic, the engine might still note price differences).

Opportunity Detection: On each price update (or each block), the engine evaluates all possible arbitrage pairs of markets *on the same chain*. (We restrict to intra-chain arbitrage for atomicity; a cross-chain price difference might trigger a different strategy, but that's outside our flash-loan one-transaction scope.) Essentially, for each chain and for each pair of DEXs on that chain that both have XAUT/USDT markets, check if there is a profitable spread after accounting for fees. If DEX_A's price of XAUT (in USDT) is lower than DEX_B's price by more than the sum of trading fees + flash loan fee + gas costs, then buying on A and selling on B yields profit. The engine will calculate the potential profit. Importantly, it also calculates the **optimal trade size** – taking too large a position will incur slippage and could erase profit. There is known methodology to determine the optimal amount to trade in a two-DEX arbitrage given constant-product AMMs: essentially where the marginal gain equals marginal cost in terms of price impact. We can solve this analytically or via iteration. This is one area where an offline OCaml analysis or embedded logic can help. We may use formulas or even incorporate an ML model's output to predict slippage. For now, the engine can do a quick binary search on trade size using pool reserve data to find the size that maximizes profit (or use a simplified approach of trading until the price gap closes). The difference in price might be small, so precision is key.

To decide to execute, the engine applies thresholds (to avoid chasing very tiny opportunities). These thresholds can be dynamic – e.g., require expected profit > X USDT, or profit margin > Y% – possibly adjusted by the ML "score" (discussed later). When an opportunity meets criteria, the engine immediately triggers an arbitrage execution.

Transaction Construction & Execution: Using the parameters of the chosen opportunity, the Rust engine constructs the transaction data to call the Solidity flash-loan contract on that chain. We use the ABI of our contract (the function signature of `executeArbitrage`) and fill in: amount of USDT to borrow, identifiers for DEX A and DEX B (this could be encoded as an enum or as addresses + method selectors for the swaps), and any other necessary details (like swap path if using a router). We rely on a Rust Ethereum library (such as `ethers-rs` or `web3` crate) to encode the call and sign the transaction with the bot's private key. The engine will have the bot's EOA private key loaded (from a secure source or environment variable) to sign transactions.

To maximize success, the engine attaches an appropriate gas price. On low-cost chains (Polygon, BSC), just a reasonably high gas price is fine. On Ethereum, if competition is fierce, we might need to use a priority fee (EIP-1559) and possibly even consider sending via Flashbots (private relay) to avoid being seen in mempool. For now, we can simply set a high gas max fee and priority to try to get mined quickly. (In a later iteration, integrating with MEV relay or using time-backrun strategies could be considered, but our 30-day prototype will focus on functionality first.)

Once signed, the engine **submits the transaction** to the network (via the RPC sendTransaction or via a direct JSON-RPC call). It then monitors the transaction's status. Because everything is atomic, we either see the transaction succeed (with profit) or fail (reverted). The engine or orchestrator will detect the outcome: if succeeded, the contract's event (Profit event) will be emitted; if failed, we may get a revert reason or just a drop. The Rust engine can log the result and then continue scanning for the next opportunity.

Concurrency & Low Latency: The scanning and execution loop is highly optimized: the engine is effectively doing a lot of things in parallel – listening to events on multiple chains, updating prices in memory, and performing arbitrage checks. Rust's strength is that it can handle this with minimal overhead and without memory leaks. We'll use data structures like lock-free maps or atomic references to store the latest price info per DEX. The moment a new price update comes in, we can recompute potential spreads. If using async event streams, our design will be event-driven: e.g., upon receiving a price update for XAUT/USDT on DEX_A, immediately check against the last known price on DEX_B, etc. This way we react in sub-second timescale. **(For perspective, many arbitrage bots in Python might operate on a few seconds loop; our goal with Rust is to react within milliseconds.)**

Additionally, by handling multiple chains, the engine can opportunistically earn on smaller networks where competition is lower. *A community MEV bot noted that it was "profitable on lesser known EVM compatible chains where there isn't much competition," while on big networks like Ethereum or BSC competition can zero out profits.* Our bot is built to perform on major chains by being fast, but also can capitalize on less crowded networks thanks to the multi-chain support.

Module Structure (Rust): We organize the Rust project as follows:

- `src/main.rs` – entry point, loads config and launches async tasks.
- `src/scanner.rs` – contains the logic for subscribing to events or polling pools on each chain, and detecting arbitrage opportunities. This might be further broken into submodules by chain or DEX. For example, a trait `DexPool` defining how to get price from a pool, with implementations for UniswapV2-type and UniswapV3-type, etc.
- `src/tx_crafter.rs` – functions to build the calldata for the Solidity contract call, given an arbitrage plan (which DEXs, how much to borrow, expected minimum output, etc.). It might use the ABI JSON or hardcoded function selector. Could integrate `ethers::contract` abis for easier encoding.
- `src/executor.rs` – handles sending the transaction and awaiting confirmation. Possibly also handles retry logic (if a transaction is stuck or gas needs bumping, though that might be advanced).
- `src/config.rs` – structures to parse config data (like JSON from OCaml output or a TOML file) containing chain RPC URLs, contract addresses, DEX addresses, etc.

We use dependencies such as `tokio` (for async runtime), `ethers-rs` (for convenient Ethereum interactions, ABI encoding, and potentially wallet management), and possibly `serde` for config parsing. The engine will also include logging (to console or file) using Rust's `log` or `tracing` crate, so that important events (opportunity found, tx sent, success/failure) are recorded.

Example Flow: Suppose on Polygon the engine sees XAUT/USDT at 1850 on QuickSwap and 1865 on SushiSwap, and estimates that borrowing 10,000 USDT could yield ~5 USDT profit after fees. This exceeds our profit threshold, so: the engine crafts a transaction calling our Polygon contract with parameters to borrow 10k USDT, buy XAUT on QuickSwap, then sell on SushiSwap. It signs and broadcasts the TX. A few seconds later, the TX is mined. The contract emitted an event with profit = 5.x USDT. The engine (or orchestrator) logs "Trade executed on Polygon: +5.2 USDT profit." The engine continues monitoring for more opportunities (perhaps the act of our trade closed the price gap, so none immediate on that chain). The entire detection-to-execution cycle could be under 1 second in ideal conditions, illustrating the need for a robust, fast loop.

(Notably, a production arbitrage system might integrate mempool monitoring to see pending transactions that could create an arb or to avoid clashing, and might use strategies like bundling transactions via Flashbots for priority. Those are beyond our initial scope but the architecture leaves room to incorporate such optimizations later.)

Analytics & Tooling Module (OCaml)

Role: The OCaml component handles **off-critical-path analytics** and configuration tasks that support the arbitrage bot's operation. These are tasks that don't need to run every second, but are important for optimizing the bot's performance and maintaining correctness. By using OCaml, we gain the benefits of a powerful type system and functional paradigm for writing reliable, mathematically-oriented code (a philosophy used by certain trading firms in traditional finance). In our system, OCaml is used to manage DEX configuration data and to perform advanced computations like opportunity scoring and parameter optimization that would be cumbersome to do in the Rust hot loop or the Python orchestrator.

DEX Configuration Generator: One responsibility of the OCaml tool is to keep an updated list of all relevant XAUT/USDT markets. For example, it can query each blockchain for DEX pools containing XAUT and USDT. This could be done via reading factory contracts or using subgraph data. Rather than coding those queries in Rust or Python, we use OCaml to write a script that does it perhaps once a day. Why OCaml? We can utilize its robust pattern matching and possibly existing libraries for JSON-RPC or even GraphQL (for TheGraph) in a safe manner. The output would be a structured data file (JSON or YAML or even an OCaml-serialized data structure) listing, for each chain: the addresses of XAUT and USDT tokens, the addresses of pools or routers on each DEX, fee parameters, and any quirks (e.g., if a DEX uses a different swap function signature). This acts as the master config that the Rust engine consumes at runtime. Keeping this generation logic in OCaml means we can easily extend and verify it (perhaps even write unit tests in OCaml to ensure, for example, that all addresses retrieved are indeed XAUT/USDT pairs and not some scam token). Once generated, the config can be fed to the bot (Rust reads it on start) or stored in Supabase for the orchestrator to pull.

Opportunity Scoring & ML Prep: The OCaml module can also serve as an **analytics engine** to evaluate arbitrage opportunities more deeply than the real-time engine can afford to. For instance, using historical data from the logs, OCaml code can analyze how often certain DEX pairs present arbitrage and with what characteristics (price gap, liquidity, etc.). It can derive a scoring model to help prioritize which opportunities are likely to be most profitable or likely to succeed. This scoring can incorporate various factors: liquidity depth (a deep pool can handle larger trades with less slippage), volatility of the pair, past success rate, etc. We might implement a formula or a small heuristic model here.

Additionally, OCaml could be used to **compute optimal trading strategies** offline. For example, solving the optimal flash loan amount to maximize profit for a given pair of AMMs is essentially solving an equation derived from the AMM constant product formulas. OCaml, being good at math-heavy logic, can solve or at least simulate this. We could produce a table of optimal trade sizes for various pool reserve ratios, which the Rust code can then use as a lookup or a quick calculation rather than doing iterative computation at runtime. If we have an ML model (trained in Python) that, say, predicts slippage or success probability, we could even port a simplified version of that model into OCaml (or directly use the coefficients in Rust). However, an easier approach is OCaml can be used to **verify or validate ML outputs**. For instance, if the ML model suggests trading X amount, OCaml could simulate the trade on a recorded state to double-check profit.

In summary, OCaml acts as our **"strategic planning"** component: it is run periodically (perhaps via CLI by the orchestrator) to refine the bot's strategy: update config, compute any new parameters or thresholds, and output them for the Rust and Python components.

Integration: The output from OCaml could be written into the Supabase database or a file that the Python orchestrator reads. For example, OCaml might update a table `dex_config` in Supabase with columns (chain, dex, XAUT_address, USDT_address, pool_address, fee, etc.). The Rust engine on startup (via orchestrator) can query this table. Similarly, a table `arb_scores` could list (chain, dexA, dexB, score, recommended_flashloan_amount). This might be overkill for the prototype; as an alternative, the OCaml tool could simply print out a config which we manually feed into the Rust binary. In a 30-day build, the essential part is to have at least a basic OCaml script that produces the list of DEX addresses to confirm correctness and possibly calculates a simple risk score for each DEX (e.g., based on liquidity). We prioritize having this separation (even if one could hardcode the config) because it enforces a clean model where adding a new DEX or chain is a matter of updating the OCaml logic, not touching the core bot code.

Example Use: Suppose a new DEX launches on BSC with XAUT/USDT pair. We run our OCaml config updater, it queries the BSC factory and finds the new pair address and adds it to the list. It might also flag that this DEX has very low liquidity, so it gives it a low "score" (meaning the bot should only attempt very small trades there or perhaps ignore unless price gap is huge). This info is stored in the DB. The orchestrator sees an updated config (or we restart the bot with the new config) and now the Rust engine will include this new DEX in its monitoring. This way, the expansion to new markets is seamless and less error-prone.

Development: We use OCaml's ecosystem (dune build system) to create a small command-line program (e.g., `arb_analysis.exe`). Key modules could include:

- `DexConfig.ml` – functions to query known DEX factory addresses on each chain (we'll maintain a list of factory/router addresses for Uniswap, Sushi, Pancake, etc. in the code or a JSON file) and find if XAUT/USDT exists, then output the addresses. Possibly using an OCaml web3 library or calling out to an API. If direct on-chain queries in OCaml are complex, we could call Python for this part; however, assume we can do it with OCaml for consistency.

- `Scoring.ml` – functions to compute arbitrage scores. Could load recent price+trade data (perhaps from a CSV dump of Supabase logs) and apply some heuristics. For example, using a simple formula: score = (avg % price gap * liquidity) – (volatility * factor) – (latency risk * factor). The actual formula can be tuned with experimentation.

- Possibly use OCaml's strength in formal verification: if time permits, we could even formally verify some aspects of the arbitrage math (e.g., prove that for a given constant product formula, our computed optimum is correct). This is advanced and optional, but showcases why a language like OCaml (with tools like Coq or Isabelle available) could be chosen for a safety-critical arbitrage strategy.

Why OCaml here? While one might use Python for these tasks, using a statically-typed FP language helps ensure our complex logic has fewer bugs (helpful when calculating financial formulas). It's similar to how high-assurance trading systems use functional languages for the core logic. And since these computations are not time-sensitive, we don't mind the extra step of an external tool – we value correctness. For instance, one could model slippage mathematically and solve for optimal order size: *"we can analytically model our expected slippage given the recent price history of the asset and then size our orders to minimize the impact of slippage."* Such calculations fit well in an OCaml script that can be run offline and its results fed into the online system.

Orchestrator & AI Integration (Python)

Role: The Python orchestrator is the operational **command center** for the entire bot. It supervises the running of the Rust engine, triggers OCaml tools as needed, interfaces with the blockchain for any tasks not handled in Rust (like contract deployment or event listening), and provides integration points for monitoring and AI. Python was chosen because of its ease of use for scripting diverse tasks and rich libraries (Web3.py for blockchain, SciPy/Scikit for ML, etc.), making it ideal as the glue layer in a complex system. While performance-critical loops run in Rust, Python can handle everything from startup/shutdown routines to reacting to on-chain events in a human-friendly way.

Startup & Deployment: The orchestrator can have a CLI with subcommands. For example, `arbbot deploy` would use Web3.py to deploy the Solidity contract to each chain (if not already deployed) and record the addresses. It will load the compiled contract bytecode (likely compiled via Hardhat or Foundry beforehand) and send the deployment transactions (requires having deployer keys/funds on those chains). In the 30-day plan, we might deploy contracts manually and just configure their addresses. Either way, the orchestrator holds the configuration (it could be a single config file or environment variables for RPC endpoints, contract addresses, etc.).

To start the bot, `arbbot start` will do the following:

- Launch the Rust engine process (if we run it as a separate process). This could be done via Python's `subprocess` call to run the compiled Rust binary. Alternatively, we could compile the Rust engine as a library and call it via FFI or use a Rust-Python binding (like PyO3). But for simplicity, treat it as an external service. The orchestrator ensures it's running, possibly reading its stdout/stderr for logging.

- Ensure connections to all services: e.g., test that Supabase is reachable (maybe by a test insertion), ensure Prometheus is up (or will be up), and Telegram bot token is valid.

- Load any ML models or data needed at runtime.

Event Monitoring: Once running, the orchestrator sets up asynchronous tasks (using Python's `asyncio`) to listen for specific blockchain events. A crucial one is the **profit event** from our Solidity contract(s). Each time our contract executes an arb, we have it emit an event like `ArbExecuted(address indexed user, uint profit)`. The orchestrator subscribes to these events on each chain (Web3.py can use WebsocketProvider to subscribe to logs). When a new event comes in, the orchestrator logs it and triggers post-trade actions (like recording profit, updating ML training data with this outcome). Similarly, the orchestrator could listen for certain error events or transaction reverts, though detecting a revert might require either parsing the transaction receipt of the submitted TX (which we can also do: Rust could communicate the TX hash, and Python can watch for its receipt). If a transaction failed, we log that too. Essentially, the orchestrator is aware of each arbitrage attempt and its result, which is important for learning and monitoring.

The orchestrator also monitors the **health of the Rust engine**. If the Rust process crashes or exits, the orchestrator can restart it (perhaps after a delay and sending an alert). It might also periodically ping the engine (for instance, via a simple HTTP or reading a heartbeat file) to ensure it's not hung. This ensures high availability – if something goes wrong in the hot path, we attempt to recover quickly.

Database Logging (Supabase): The Python layer is responsible for pushing comprehensive logs to the database. While the Rust engine might do some logging to console, the orchestrator can take those logs (if captured) or use the events it sees to construct database entries. For example, when an `ArbExecuted` event is received, Python inserts a row into a `trades` table with columns: timestamp, chain, dex_buy, dex_sell, amount_borrowed, profit, gas_used, etc. It can fetch additional info like gas used from the transaction receipt and include that. If an attempt failed, it logs a row in maybe a `failures` table with the reason (if we captured the revert reason or at least the fact it reverted). Python can also log periodic snapshots of prices or spreads (though that might be overkill to store all the time; instead, we might store any time a spread above X% was seen, even if not taken due to threshold). Supabase (Postgres) will hold this data, which we can later analyze or feed into ML training. The **Supabase Python client or a simple HTTP POST** to the Supabase REST endpoint will be used for these inserts. This decouples the Rust component from direct DB connections (keeping the hot path lean). The database can also be used to store **config data** that OCaml updates – the orchestrator would read that (with a SELECT query via Supabase's API) and update the Rust process if needed (perhaps by restarting it with new config or sending a signal—though dynamic reloading in Rust might not be implemented in 30 days, a restart is acceptable for config changes).

Machine Learning Integration: The orchestrator is where we integrate any meaningful ML/AI logic to enhance the bot's decisions. We will keep the ML component **focused** and driven by data – no frivolous AI use. Two concrete examples:

1. *Slippage & Optimal Execution Prediction:* We can train a simple model that predicts the *effective profit* of an arbitrage trade given the observed price difference, trade size, and pool liquidity. The model could be a linear regression or decision tree that was trained on historical simulations or past trades. It would take features like: price_gap%, pool1_liquidity, pool2_liquidity, trade_amount, perhaps network congestion, and output a predicted realized profit (accounting for slippage and price movement during transaction). This can help decide the optimal trade amount or whether a given opportunity is worth it. Indeed, **"Linear Regression models or Decision Trees can predict the expected slippage of a trade based on past execution performance."**. We might gather data by simulating various trade sizes on historical pool states (which we can do offline) to train such a model. During runtime, when Rust finds an opportunity, it could either use a precomputed formula from this model (e.g., the model might tell us that for a 2% price gap and pool liquidity X, the ideal trade is Y tokens to maximize profit). To avoid slowing Rust with Python calls, we might embed a simplified model as parameters. Alternatively, Rust can output the raw opportunity to a queue, Python fetches it (quickly), runs the model to refine the trade amount or decide go/no-go, and instructs Rust accordingly. In the interest of time and simplicity, the first version might not have a feedback loop; instead, the Python ML might periodically adjust global parameters (like "for DEX A vs B, don't trade more than $Z or you'll get too much slippage" based on model). Over 30 days, a feasible integration is: Python reads the last day's trades and outcomes from Supabase, retrains or updates a simple model (even a few rules), and updates a config for the Rust engine (via file or env var) with new parameters (like adjust profit threshold up if many fails observed, etc.).

2. *Latency-aware Scoring:* The bot can use AI to assess *the risk that an opportunity will vanish or be beaten by a competitor* given current network conditions. For example, if Ethereum is congested and the arbitrage gap is small, by the time our transaction mines, the gap might close or another bot might have taken it. We could train a classifier that, given features like current block's gas price, pending TX count, and the size of the arbitrage gap, outputs a probability of success. If probability is too low, the bot might skip or wait. Similarly, a model could help decide how much priority fee to include in the transaction – essentially treating it as a multi-armed bandit problem:

how much to pay to ensure inclusion before others. This is advanced, but even a heuristic learned from data (like "if gap <1%, 80% of time someone else gets it on Ethereum") could be encoded. In the prototype, we might implement a simpler form: e.g., measure the bot's typical transaction mining latency and require that expected profit > X * (latency in blocks) to proceed, where X could be tuned or learned.

The orchestrator can run these models using libraries like **scikit-learn** (for quick models) or even load a small TensorFlow/PyTorch model if we had one. Given the time, a rule-based or linear model approach is realistic. The integration points are either *pre-trade filtering* or *post-trade analysis*. Pre-trade, as discussed, is tricky to sync with Rust in real-time, but we can set conservative parameters that the model updates occasionally. Post-trade, Python can use the outcomes to refine the model (reinforcement loop).

Alerts and Notifications: The orchestrator will incorporate a submodule for sending alerts. Using the **Telegram Bot API**, we set up a bot and get a token/chat_id for our notifications channel. The orchestrator, upon certain events, will send messages. For instance:

- If an arbitrage trade succeeded, send a Telegram message: "✅ Arb trade on [Chain]: +$X profit (tx hash: …)".
- If a trade failed (reverted or lost money due to fees), perhaps "⚠️ Trade on [Chain] failed or broke even."
- If the Rust engine or any component crashes, an alert to check the system.
- Periodic summary alerts, e.g., daily P/L or important stats.

These alerts keep the developer informed in real-time, which is crucial for a production system running 24/7. (Prometheus will handle automated alerts too, but a Telegram message is a quick human-friendly notice.)

Metrics & Monitoring: The orchestrator uses the **Prometheus Python client** to define metrics for things like number of opportunities detected, number of transactions sent, success count, failure count, latency of execution (time between detection and mined tx), etc. It can also track balances (if the bot's profit accumulates in its address, check that), and any other health indicators (e.g., memory usage of Rust process if accessible). The Python process can run an HTTP server exposing a `/metrics` endpoint that Prometheus will scrape. We'll containerize Prometheus and configure it to scrape this endpoint at e.g. 5s intervals. In a prototype, we might skip a full Prom setup and instead just log metrics, but we aim to include it to mirror a production-ready environment. The orchestrator might also push custom alerts through Prometheus Alertmanager if set up, but since we have Telegram, that covers critical alerts in a simpler way.

Finally, the orchestrator can provide a simple **CLI or web interface** for the developer. For example, one could implement a small command like `arbbot status` to print current status (using data it has, like "X chains connected, last trade 10 min ago, total profit $Y").

This isn't required, but helps demonstrate a holistic system. Even a read of the database or Prom metrics can serve as status.

Module Structure (Python):

- `orchestrator.py`: main script using `asyncio.run` to start the event loop and tasks. It loads config, starts Rust, sets up event subscriptions.
- `events.py`: using Web3.py to subscribe to contract events and handle them.
- `database.py`: functions to write to Supabase (could use the `supabase` PyPI library or direct HTTP).
- `alerts.py`: functions for Telegram notifications (using `python-telegram-bot` or simple `requests.post` to Telegram API).
- `ml.py`: functions to load/train/apply ML models for slippage/latency. Possibly using `joblib` to load a pre-trained model from disk.
- `metrics.py`: setup of Prometheus metrics (from `prometheus_client`).

We will use libraries: **web3.py** (to interact with Ethereum and listen to events), **asyncio** (for concurrency), **psycopg2 or postgrest** (to write to Supabase if not using their SDK), **sklearn/ numpy** (for any simple model), and standard logging.

Summing up Python's role: It does not participate in the tight arbitrage decision loop (thus doesn't slow it down) but it *oversees* everything and adds intelligence and reliability. This approach is akin to having a "manager" process supervising a high-frequency trading engine – it can stop/start it, feed it improved parameters (from analytics), and report upwards (to humans or dashboards) what's going on. The combination of these layers (Rust for speed, Python for brains/monitoring) is a proven pattern in many systems – e.g., a prototype arbitrage bot had "InfluxDB monitoring" and "Telegram alerts" as non-essential but useful components, which we have included here to ensure **production-grade operability from day one**.

Data Storage and Infrastructure Stack

Off-Chain Data (Supabase): We utilize **Supabase** as a quick, hosted solution for a Postgres database and RESTful interface. This is mainly for convenience during prototyping – it avoids the need to set up a separate database server and provides an easy API and dashboard. All sensitive or time-critical data remains on-chain or in-memory; Supabase is used for **logging, analytics, and persistence of configuration**. For instance, tables we create might include:

- `trade_logs(timestamp, chain, dex_buy, dex_sell, vol_usdt, profit_usdt, tx_hash)` – each successful trade entry.

- `opportunity_logs(timestamp, chain, dex_buy, dex_sell, price_buy, price_sell, spread_pct, decided_amount, executed BOOLEAN)` – records when a notable opportunity was detected. This could help training ML (executed = false means we skipped, and later we can see if it would have been profitable or not).

- `config_dex` and `config_params` – storing config as mentioned (addresses, and any tunable parameters like current profit threshold, risk levels etc. that can be adjusted without redeploying code).

- `system_events(timestamp, component, event, details)` – logs of things like "Rust restarted" or "Error on chainX RPC" for debugging.

Supabase's free tier can handle our volume (likely a few inserts per block at most) and we can query it on their web UI or retrieve data for analysis. It also allows perhaps building a quick front-end if desired, but that's out of scope. The reason to use a cloud DB at all is to have a historical record and to enable data-driven improvements. If running entirely locally, we could skip this and use local files or a local Postgres; however, including Supabase from the start means our design cleanly separates runtime memory (Rust) from persistent data (Postgres), which is useful if we scale to multiple bot instances or need to analyze data remotely.

Logging and Monitoring: Beyond database records, the system generates logs and metrics:

- Each component (Rust, Python, even OCaml when run) will output logs to stdout or log files. We aim to structure these logs (JSON format or clearly labeled text) so they can be aggregated. In a production deployment, one might use a logging service or at least `docker logs` to capture them. For now, we'll focus on crucial logs (transactions, errors) being captured by the orchestrator for inclusion in Supabase.

- **Prometheus** will be used to scrape metrics. We plan to run a Prometheus instance in Docker on the same host (since free hosted Prometheus usually requires some payment or pushing metrics externally, which is unnecessary here). The orchestrator opens a metrics endpoint (e.g., on port 8000). Prometheus is configured (via prometheus.yml) to scrape that endpoint for metrics like `arb_profit_total{chain="Ethereum"}`, `arb_trades_count{success="true"}` etc. We also expose Rust engine metrics if possible. If the Rust engine is separate, we could instrument it with a metrics library and either push to a **Pushgateway** (Prom component for short-lived jobs) or have it open its own endpoint. A simpler route: Rust can emit metrics as log lines and the orchestrator can update Prometheus metrics accordingly (basically using Python as the metric aggregator). This avoids running an HTTP server in Rust.

We'll implement the latter for simplicity: e.g., when orchestrator receives an event "trade executed", it calls `prometheus_counter.inc()` for trades and adds to a profit gauge, etc.

- **Grafana** (optional): If we want to visualize metrics nicely, we could spin up Grafana (free) with Prometheus as data source. This would give dashboards of performance. Given time constraints, we may skip Grafana setup in the initial 30 days, but mention it as a future convenience. The key is the metrics collection is in place to allow adding such visualization easily.

Infrastructure for Development & Deployment:
For development, we will use **local blockchain nodes** and testnets:

- We can simulate scenarios using a Mainnet fork (e.g., using Hardhat or Foundry's Anvil) to have real XAUT/USDT pools and Aave available in a sandbox. This is extremely useful to test our Solidity contract logic with real liquidity pools. We'll write some tests (perhaps in Hardhat) to ensure the flash loan and swaps execute correctly and profit is as expected in a controlled price scenario.

- We will also test on a public testnet if possible: the challenge is XAUT (a real token) might not exist on testnets. Instead, we might deploy dummy ERC20s on a local testnet and simulate an arbitrage between two dummy DEXs we set up. However, to fully test flash loans, we need a flash loan provider on testnet – Aave operates on some testnets (like Aave v3 on Goerli or Mumbai with test tokens). We could adjust to use, say, DAI or USDC on testnet as an analog and not focus on XAUT for testing. In any case, thorough testing of the mechanics will be done in a sandbox environment.

For deployment (even for prototype), **Docker** is our friend. We'll create a Docker Compose setup that includes:

- A service for the **Python orchestrator** (based on a Python 3.10 image, installing web3.py, etc., and running our orchestrator.py).

- A service for the **Rust engine**. We can multi-stage build: compile the Rust binary in a cargo builder image, then use a slim runtime image to run it. This container will be very lightweight.

- (Optional) a service for **OCaml tasks**. Since OCaml tool is run on demand (not a daemon), we might not need a persistent container. We could just run the OCaml binary manually when needed. Alternatively, include it in the orchestrator container if we compile it to a binary and copy it in, so that orchestrator can invoke `./dex_config` or so.

- A **Prometheus** service (using the official Prometheus image) with a configuration that scrapes the orchestrator (and Rust if applicable).
- A **Pushgateway** if we decide Rust will push metrics rather than be scraped.
- Possibly a **Grafana** service (with a pre-provisioned dashboard for our metrics).
- Supabase is not run in Docker since we use the cloud service; we will provide the URL and API key as environment variables to the orchestrator.

Everything can be configured via a `.env` file (containing RPC URLs, private keys, supabase keys, telegram keys, etc.). The deployment guide will emphasize minimizing cost: for instance, instead of running an Ethereum full node, use Infura's free tier (with rate limits in mind) or use a service like Ankr's public RPC. Many chains have public RPC endpoints that are sufficient for light loads. For example, BSC and Polygon offer free RPC URLs that we can use initially. If those are slow, we might run a light client or a mid-tier provider later. But since our focus is one pair and not too many requests (we only watch that pair's pools), we can likely stay within free tier limits.

Security & Key Management: On a prototype, we might simply store private keys in env files or a local keystore. For production, one would secure them (perhaps using an HSM or at least an encrypted vault). We'll note that for now, keys are kept locally and never shared. The flash loan arbitrage by nature doesn't hold funds for long on-chain (profits accumulate in the bot's wallet), but over time the bot's address will store profits in USDT (or XAUT if we choose to keep profit in XAUT somewhere). The orchestrator should monitor the balance and perhaps periodically transfer profits out to a cold wallet or at least alert if a large balance is on the hot wallet.

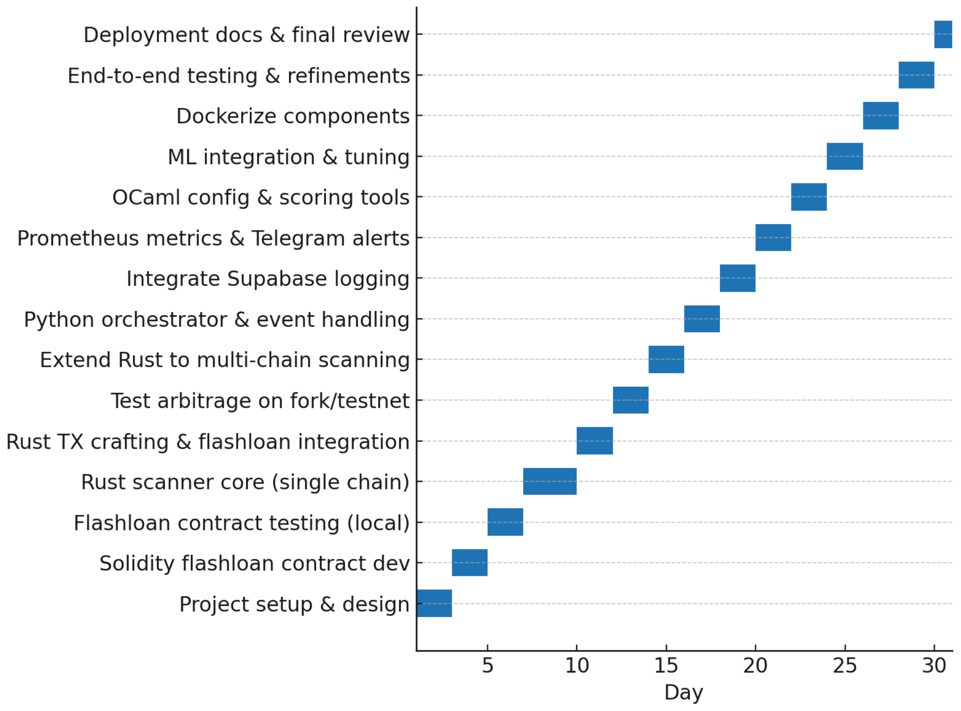
Scalability Considerations: While the initial deployment might be on one machine, the architecture allows scaling out. For example, one could run multiple Rust engine instances, each focusing on a subset of chains or DEXs, possibly on servers located near the respective blockchain nodes for lower latency. The orchestrator can also be scaled or distributed (though careful to avoid double-triggering the same arb). We can also scale vertically by moving to more powerful hardware if needed. The use of Docker means we can deploy on cloud VM, Kubernetes, etc. easily.

Cost and Resource Estimates: Since we use mostly free resources: the main costs would be transaction gas (which on test runs we minimize, and on real runs we hope is covered by profits) and maybe a few subscription services if needed (we aim not to, but e.g., Infura beyond free tier). The system in production on Ethereum mainnet could be costly in gas if it sends many failed transactions; hence the importance of the ML filters to avoid bad trades. On cheaper chains, the cost is negligible.

In summary, the infrastructure stack is designed to be **developer-friendly and low-cost** initially, but follows patterns (containerization, microservice separation, centralized logging) that set the stage for a robust production deployment.

Development Timeline (30-Day Plan)

To build this system in 30 days, we propose a phased approach with daily milestones, ensuring that we have a functional product as early as possible and refine it to production quality by the end. Below is a Gantt chart and breakdown of the tasks:



Description: 30-day development Gantt chart for the arbitrage bot. Each bar represents a task or milestone spanning the indicated days. We front-load core functionality (Solidity contract, Rust engine) and progressively integrate advanced features (multi-chain, orchestrator, ML, monitoring) toward the end.

- **Day 1-2: Project Setup & Design** – Set up version control (Git) and base project structure. Write a detailed design (this blueprint) and define the interfaces between components. Prepare development environment: Rust toolchain, Python env, Node (for compiling contracts), OCaml compiler, etc. Outcome: team is aligned on architecture and a skeleton repository with subfolders for `solidity/`, `rust/`, `ocaml/`, `python/` is created.

- **Day 3-4: Solidity Flash Loan Contract Development** – Implement the smart contract in Solidity. Start with a basic flash loan receiver that can borrow USDT from Aave and immediately repay (as a sanity test). Then add the dual-swap arbitrage logic between two DEXs. Include reentrancy guard, and owner access control. Use OpenZeppelin libraries. Write unit tests for contract functions (using Hardhat/Foundry with mocked calls or a mainnet fork to simulate Aave and DEX interactions). By end of Day 4, deploy the contract to a local Hardhat network and simulate a flash loan + swap sequence to verify it behaves as expected (e.g., hardcode a scenario with one pool cheaper than another).

- **Day 5-6: Flash Loan Contract Testing (Local)** – Rigorously test the Solidity contract on a fork of mainnet (so we have actual Uniswap pools and Aave). For example, fork Ethereum, give the contract some USDT, manually manipulate a Uniswap pool's reserves to create an arbitrage, and call the contract to see if it profits and reverts when appropriate. Fix any bugs (like mis-calculation of amounts or missing approvals). Also test failure paths (what if not enough profit, does it revert?). By Day 6, we should have high confidence in the contract's correctness and safety. Start deployment to a testnet (optional) or plan addresses for mainnet deployment.

- **Day 7-9: Rust Scanner Core (Single Chain)** – Begin coding the Rust arbitrage engine focusing on a single chain (Ethereum) and a simple scenario. Set up RPC connection (using Infura or local node), fetch XAUT/USDT pair addresses for Uniswap and Sushi on Ethereum (for instance), and implement logic to query their reserves and compute price differences. Initially, do this in a loop every block: get reserves, compute price on each, print if an arbitrage exists. No transactions yet, just detection. By Day 9, we have a Rust program that can detect a price imbalance on Ethereum between two exchanges (perhaps tested by manually tweaking one pool or using a fork where we can adjust state).

- **Day 10-11: Rust Transaction Crafting & Flashloan Integration** – Expand the Rust engine to integrate with the Solidity contract. Use the ABI of `executeArbitrage` and the contract address (from Day 5's deployment on testnet/fork) to craft a transaction. Implement signing with a private key and sending the transaction through the RPC. On a local test or Ethereum fork, simulate an actual arbitrage: trigger the Rust code when a known opportunity is present (maybe use the same scenario from contract testing) and ensure the transaction goes through and yields profit. Essentially, achieve an end-to-end arbitrage execution on one chain in a controlled environment. This involves coordinating the Rust logic (detect → call contract) and verifying the contract indeed executed (the Rust code can wait for receipt). By end of Day 11, we should have the **basic arbitrage loop working on Ethereum** (or a fork), end-to-end.

- **Day 12-13: Full Pipeline Test on Fork/Testnet** – These days are for thorough testing of the integrated system (Solidity + Rust) in increasingly realistic conditions. We might deploy the contract on a public testnet (if using test tokens for XAUT/USDT) or continue with mainnet forking. We test different cases: no arb available (bot should do nothing), small arb (bot executes and just covers fees), large arb (bot makes profit). Monitor how fast the detection to execution is and if any issues (like occasional mis-estimation of required gas or slipping price). If any bug arises (e.g., transaction not sending correctly, or rounding issues in calculations), fix them now. By Day 13, we want a reliable single-chain arbitrage bot that we can trigger on demand and it works. This is a good checkpoint to have a **Minimal Viable Product**: one chain, two exchanges, arbitraging XAUT/USDT with flash loan.

- **Day 14-15: Extend Rust Engine to Multi-Chain & Multi-DEX** – Now that the core logic is proven, generalize the Rust engine. Introduce configuration for multiple chains: e.g., arrays of RPC URLs, contract addresses per chain, DEX endpoints, etc. Modify the code to spawn separate async tasks for each chain's monitoring. Ensure thread-safety for any shared data structures (each chain can largely operate independently, so we might duplicate some logic per task). Also, extend to more DEXs: e.g., on Ethereum include Uniswap v3 in addition to v2 (this might require writing a small adapter for price calculation), on Polygon include QuickSwap, etc., as available. Essentially, broaden the list of pools being watched. This is also a point to optimize scanning – possibly implementing event-driven updates instead of polling where possible. By Day 15, the Rust engine should be capable of concurrently watching, say: Ethereum (Uniswap v2 & v3), Polygon (QuickSwap), BSC (PancakeSwap), Arbitrum (SushiSwap or Uniswap), etc., and detecting opportunities within each. We'll test by creating fictitious spreads on each to see if it logs detection. Transaction execution on each chain can be tested on testnets or forks similarly. We also ensure that multiple chain handling doesn't slow down one another (thanks to Rust async, it shouldn't).

- **Day 16-17: Python Orchestrator & Event Handling** – Begin implementing the Python orchestrator. First, focus on the basic orchestration: writing a script to launch the Rust engine (perhaps via subprocess), and establishing a Web3 connection to each chain for listening to our contract's events. Using Web3.py, subscribe to `ArbExecuted` events from each deployed contract. Also set up a basic asynchronous loop to handle these events as they come. On catching an event, print a message or log it. Additionally, implement a simple command interface: e.g., a CLI argument to `orchestrator.py` that can deploy contracts (for now, just gather addresses if already deployed) or start monitoring. By Day 17, we have Python code that can run alongside the Rust engine, get notified when an arbitrage trade happens (via contract event or polling receipts), and log those events. At this stage, orchestrator is not yet doing DB or ML, but it's coordinating multi-chain events.

- **Day 18-19: Integrate Supabase Logging** – Set up a Supabase project (if not already) and create the schema for logs. Use the Supabase Python client or HTTP API to connect. In the orchestrator, add routines to insert records into the database for events: each arbitrage event and any other relevant data. Also consider logging the bot's status periodically. We test this by triggering some events (maybe using the test environment or simulating calls) and verifying that entries appear in Supabase (using their dashboard or select queries). By Day 19, we should have persistent logs being recorded – effectively a basic **operational database** for our bot. This is also where we might log configuration data; the orchestrator could pull config from Supabase as well (for instance, store contract addresses per chain in a table and have the orchestrator read them on start instead of a local config file). Implementing that would allow easy reconfiguration without code changes.

- **Day 20-21: Prometheus Metrics & Telegram Alerts** – These two days focus on the monitoring aspect. First, integrate **Prometheus metrics**: incorporate the `prometheus_client` in Python, define counters and gauges for key metrics. Have the orchestrator spawn an HTTP server (or use `start_http_server` from prometheus_client) to expose metrics. Write a Prometheus config (prometheus.yml) to scrape this. Test locally that Prometheus can fetch the metrics (run Prometheus container, check the data). Define some example metrics: e.g., `arb_total_profit_usd`, `arb_trades_successful`, `arb_trades_failed`, `current_spread_eth` (maybe the last seen spread on Ethereum), etc. Next, set up the **Telegram bot**: create a bot via BotFather (if not done), get the token and chat ID. Use Python's `requests` or `python-telegram-bot` to send a simple message. Integrate into orchestrator: send a message on important events (success, fail, errors). Test by forcing those conditions (maybe simulate a success event). Also possibly send a startup message ("Bot started on X chains"). By the end of Day 21, we have real-time visibility: one can look at Prometheus (or logs) to see metrics, and get Telegram pings when trades happen. This makes the system much more **production-ready**, as issues or successes won't go unnoticed.

- **Day 22-23: OCaml Config & Scoring Tools** – Now implement the OCaml component. Day 22: focus on the DEX configuration script. Use web3 bindings or an HTTP approach to query each chain's factory for XAUT/USDT pair. If direct querying in OCaml is complex and time is short, an alternative is to use the data we already know (addresses found during development) and structure them properly. But assuming we proceed, get a list of known DEX factories (could hardcode addresses for Uniswap, Sushi, Pancake, etc.) in a data structure, then for each, call a function to get pair address for XAUT/USDT (by hashing the pair or using a subgraph). Store results in a JSON output. Day 23: implement a simple opportunity scoring function. Perhaps read the latest prices from Supabase logs (or we feed in some sample data) and compute, for each potential pair of exchanges, a score = (max historical spread)% * (min liquidity) to prefer high-liquidity and often profitable routes. It could also output a recommended trade size (maybe as a fraction of pool reserves). The OCaml program then outputs a config file that includes these scores and suggestions. We test the OCaml output by running it manually and checking the data. If time permits, integrate it: the orchestrator could call the OCaml binary periodically (e.g., via `subprocess.run`) and then update internal structures or DB with new parameters. If not, we at least document the process to update config via OCaml outputs. By Day 23, we have an OCaml tool that enhances configuration management and potentially the strategy, even if it's rudimentary (this establishes the multi-language pipeline fully).

- **Day 24-25: ML Integration & Tuning** – With data logs accumulating (from tests or any demo runs), attempt a basic ML integration in Python. On Day 24, formulate a simple predictive task: e.g., based on an opportunity's spread and pool sizes, predict the slippage or profit. Even without real training data, we can simulate some by using the AMM formulas. Train a model (could be as simple as a linear regression using scikit-learn) offline. Then integrate it into the orchestrator: load the model (pickle) at startup. In the detection logic, before Rust executes (if we have a hook or if Rust can accept an external stop), perhaps simpler: use the model to adjust a global `min_profit_threshold` dynamically. For instance, if model predicts high slippage for current pool conditions, increase the required profit threshold. This can be done every few seconds or whenever a big change in pool liquidity is logged. On Day 25, focus on testing this ML logic's effect. It might be subtle, but we could simulate a scenario where blindly the bot would try a trade with low margin, and see if with ML thresholding it skips it. Also tune parameters manually or via the model to avoid false positives (we don't want to skip good trades or take bad ones). The output of these days is a demonstration that the bot uses data-driven logic to improve decisions (even if the improvement is modest, we lay the groundwork for more sophisticated models later). We ensure that this ML component doesn't slow the system (run it in a background thread or infrequently). At this point, our bot is feature-complete: it has the core arbitrage logic and all supporting systems (config, monitoring, AI).

- **Day 26-27: Dockerize Components** – Write Dockerfiles for the Rust engine (from a Rust base image, compile in release mode) and Python orchestrator (from a Python slim image, install requirements). Also, prepare a docker-compose.yml including those plus Prometheus. Configure networking between them if needed (though orchestrator and Rust might just communicate via database and chain, so no direct link needed, but for simplicity, they can be on the same Docker network to share environment or volume if needed). Test the entire setup on a fresh machine using docker-compose: it should pull up all services and the bot should start running. We specifically check that environment variables (like private key, RPC URLs) can be passed in securely (maybe via a .env file that docker-compose reads). On Day 27, refine the Docker setup: ensure logs from Rust and Python can be seen via `docker-compose logs`. Also possibly include the OCaml binary: we could compile it on the host and just mount it, or create an OCaml container to run the analysis as a one-off job. For now, maybe skip containerizing OCaml due to time and run it as needed outside Docker. The deliverable is a one-command deployment: `docker-compose up` launches the entire bot stack. This is crucial for ease of use and for future scaling (could be deployed to cloud easily).

- **Day 28-29: End-to-End Testing & Refinement** – These days are reserved for final testing, bug fixes, and performance tuning. We will run the bot in a simulated environment continuously to observe behavior. For example, run it on a mainnet fork where we can control some price differences and ensure it captures them. Or run on a small scale on a real network (perhaps on Polygon or BSC with very low amount to see if it finds any arb, even if likely none, just to test stability). Monitor CPU and memory usage (Rust should be efficient; Python might use some CPU for event loop but should be fine). Check that no component crashes over a period of time, and if it does, improve error handling. This is also time to review security: ensure private keys are not logged anywhere, ensure contract addresses are correct (especially if deploying to mainnet, double-check addresses of XAUt and USDT on each chain). Add any missing features that are quick wins (for instance, if we noticed a particular failure mode, add a guard or alert for it). By end of Day 29, the system should be running smoothly in a continuous mode, ready for real deployment.

- **Day 30: Deployment Prep & Documentation** – On the final day, focus on documentation and deployment details. Write a **lean deployment guide** (in the README or a separate doc) explaining how to: set up environment variables (RPC endpoints, obtaining an Aave API key if needed, funding the bot's address with a bit of ETH/BNB for gas), how to deploy the contracts (the CLI method or using a script), and how to start the system (Docker instructions). Also document how to add a new DEX or chain in the config (e.g., update the config table and restart, etc.), so the system is maintainable. Additionally, list scaling recommendations: e.g., if moving to production, one should run their own Ethereum node or use a faster service, maybe isolate each chain's bot on a different machine if latency critical, and do security audits on the Solidity contract. We'll include a section on possible future improvements such as integrating with Flashbots for Ethereum to avoid front-running, more sophisticated ML (like reinforcement learning as hinted by research), and support for more arbitrage types (triangular, cross-chain via bridging arbitrage with some risk). Finally, ensure all code is pushed to the repository and do a final code review for cleanliness and readability. By the end of Day 30, we have a fully documented, production-grade prototype ready to be used or extended.

Each of these tasks builds on the previous, and by following this schedule, we progressively add complexity while keeping the system runnable at each stage (important for iterative testing). This ensures that by the time all pieces come together, most individual issues have been ironed out. The end result is an **advanced yet lean arbitrage bot** system – one that is immediately useful for XAUT/USDT trading and serves as a solid foundation for further expansion or deployment in a live trading context.

References

The design draws on known strategies and best practices in DeFi arbitrage and MEV bot development, including atomic flash-loan execution (rapidinnovation.io), multi-chain event streaming (medium.com), high-performance Rust integration (github.com), secure smart contract patterns (medium.com), and the use of monitoring/alerting tools that have become standard in trading systems (medium.com). The incorporation of AI for execution optimization is inspired by recent applications of ML in trade execution to predict slippage and adjust strategies (medium.com, stephendiehl.com). By combining these elements, the system is engineered to be **robust, fast, and intelligent** from the outset, even on a prototype-friendly infrastructure.