Olas Automate

Summary: A spec for a cross-chain autonomous service deployment that can automate smart-contract execution and offers transaction relaying.

Authors: Valory AG

Olas enables a community-driven approach to developing off-chain infrastructure, inviting broad participation rather than relying on limited centralized teams. One key piece of infrastructure in the crypto space is decentralized automation. This document specifies a crypto-native automation solution built with the Olas stack.

Its features include:

- Unified API: Each agent in the service hosts it. This can be at their domain of choice.
- Single entrypoint via SDK (used to abstract above APIs via the open-autonomy client, to be rebranded Olas SDK).
- One autonomous service per chain on which Olas is deployed. Each service is identical in terms of feature set and offers:
    - Transaction Relay
    - Smart Contract Automation
    - Gas tank

Each agent service will be deployed against one chain for which it does the three tasks. If an agent receives a request not for its chain then it simply sends it via ACN to the right service.

Deliverable 0: Automation Architecture Core Components[1]

Event Listener
The Event Listener is responsible for continuously querying the chain and monitoring emitted events.

Checker/Resolver
The Checker defines and verifies the conditions required for a task to become executable.

Executor
An Executor is responsible for reliably working jobs and submitting transactions on chain. By design in autonomous services a role like the executor is split in different skills.

Fees Oracle
Returns an overall estimated fee (gas costs + relayer fee) for relay requests.

Paying for your fees
Initially the service would accept payment in the native network token or its wrapped version for each network that Automation supports.

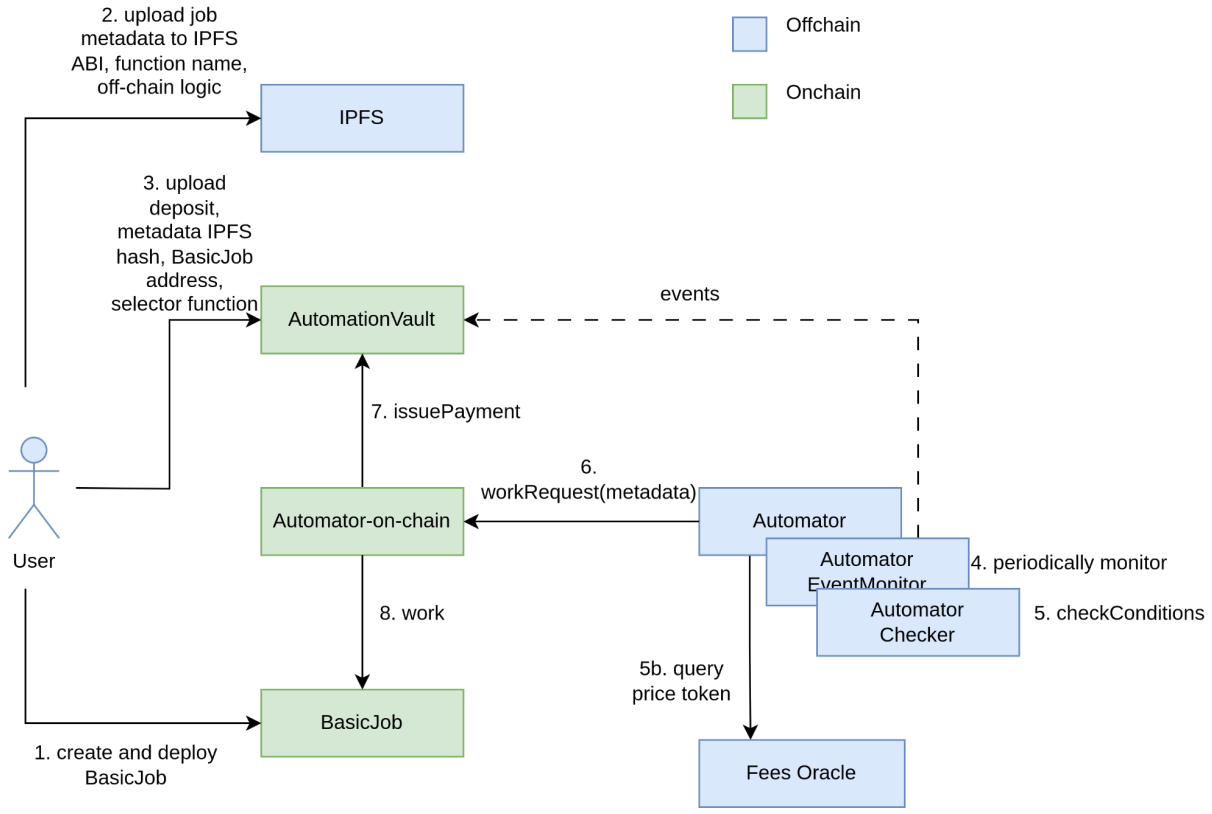
AutomationVault Balance
Simply deposit some tokens into the Vault for each of the network(s) on which you want to execute transactions. Each time an execution occurs, Vault will deduct the costs from your Vault Balance to cover the gas costs.

[1] Takes inspiration from https://docs.gelato.network/developer-services/automate & https://docs.keep3r.network/

Deliverable 1: Basic Automation

Minimal architecture (on-chain and off-chain part) and diagram for an Automation (autonomous) service with an AutomationVault for native payment tokens or wrapped payment tokens that live in the same chain as the Automation Vault.

Workflow:



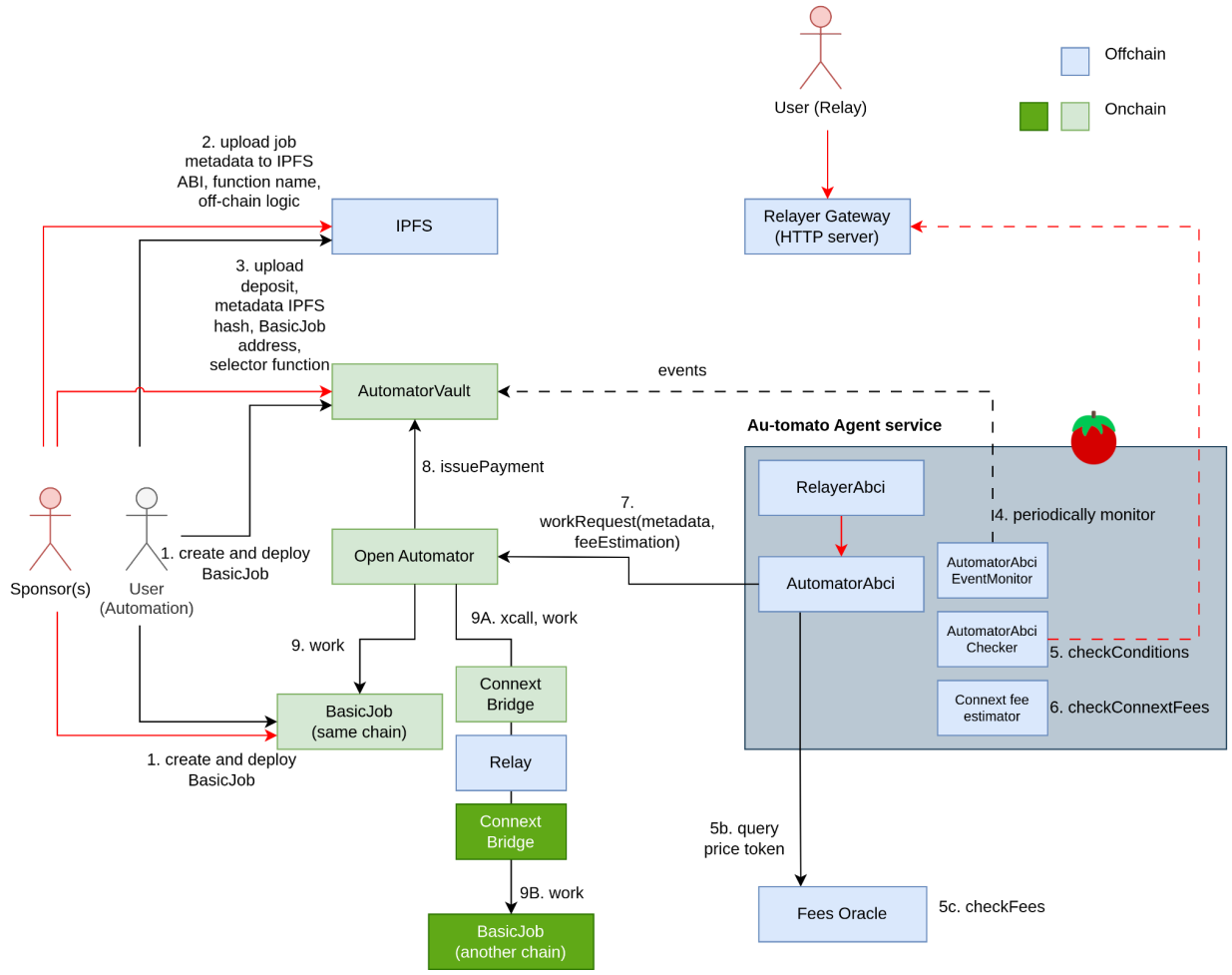
1. User uploads **BasicJob** to the blockchain which contains a custom work() method.
2. User deposits on the **AutomationVault**:
    a. Tokens
    b. Metadata containing pointing to the the ABI information for the contract
    c. Logic to call this function: code that will execute this function. (*optional)
3. **Automator_EventMonitor** “monitors” for new AutomationTasks posted.
4. **Automator_EventMonitor** updates **Automator** state including new jobs to automate.
5. **Automator** periodically checks with the **Automator_Checker** for executable jobs

6. Whenever Automator determines that an executable job is in order, it executes the code that the user has provided and makes the transaction with the output of that code.
    a. The **Automator** contract will do 3 things,
        i. Call the work function of the **BasicJob**
        ii. Calculate how much gas was spent in the work function
        iii. Issue payment to the **Automator** for the amount of gas spent.

Missing for a next version:
- The user defines `BasicJob` and `metadata` and the off-chain/UI takes care of the rest

Deliverable 2: Adding Basic Relay functionality

Minimal architecture (on-chain and off-chain part) and diagram for a relay system on top of the basic automation architecture above.



On-chain

- AutomationVault
- OpenAutomator
- BasicJob

Third party:

- Bridge contract

Off-chain

The off-chain consists of two parts. The automation agent service, and an Http server (Relayer Gateway).

The Relayer Gateway has a user-facing API interface. There are 3 end points for relaying:

- One for which users pay with their own gas
- And one where users' gas fees are partly or entirely subsidized by the application
- Another endpoint for providing the meta-transactions to the agent service. In the future this can be extended to be a websocket, or use some popular queuing solutions.

The (automation) agent service, as described above, would have the capability for dynamically loading logic from IPFS. There could be a package for the Relayer, that would perform the following logic:

1. Get the meta-transactions from the Relayer Gateway.
2. Verify their validity.
3. Estimate fees
4. Prepare the transaction to be signed and sent via the Automator.

This essentially would make the Relayer a "plugin" to the Automation service, instead of having it embedded in the service.

Assuming the Automator has support for the relaying as explained above, then a possible flow could be:

1. User adds their meta-transactions to the **RelayerGateway**.
2. Automator makes a transaction

Integration of Connext[2]

Perhaps the simplest approach to manage cross-chain interoperability for the first version of the service is to use the provided API of Connext for the following reasons:

Pros:

- It provides an already available, tested and robust API for managing cross-chain payments in a service for multiple chains.
- Developing an infrastructure that provides the same level of service and functionality as Connext will require an excessive amount of resources without a clear gain. The Connext infrastructure comprises complex on-chain and off-chain architecture.

[2] For exemplary purposes Connext is chosen. It is to be decided which bridge provider is most suitable.

Cons:

- Connext fees would need to be taken into account for the transactions that are sent.

Compatibility with EIP 4337 standard

The suggested architecture for automation and relay is flexible and generic and thus compatible with the architecture and functionalities of EIP-4337.

Proposed next steps

1. Implement the required smart contracts and perform initial tests
2. Implement off-chain service/FSMs that supports automating a simple job
3. Add off-chain service/FSMs that implement basic transaction relaying for Safe multi-sig transactions
4. Full e2e tests
5. Autonolas Automate SDK

Estimated time effort :

- Task 1. 6-15 days
- Task 2. 5-12 days
- Task 3. 5-12 days
- Task 4. 5-12 days
- Task 5. 4-10 days