Governatooorr Autonolas Spec

Project outline describing the Governatooorr - Oaksprout the Tan 's and David Minarsch 's hack project for Eth Denver

Technical Specification **2**

Background 2

High level design 2

Autonolas Service Specification 3

Backend Server **3**

Rounds 3

Single FSM diagram 4

Chained FSM diagram 4

Roles and responsibilities 4

Other service details 5

Web App Spec **5**

Potential additional features 6

Appendix **6**

Operating costs **6**

Threat model **7**

Private key management **8**

Scope **8**

Policy **8**

Generation of new keys **8**

Storage of keys locally **9**

Technical Specification

Background

The Governatooorr autonomous service is an autonomous, AI-powered delegate that votes on on-chain governance proposals on the Ethereum mainnet (and later off-chain governance proposals on Snapshot). The Governatooorr is a solution to governance apathy and an experiment with AI-enabled governance where the AI in question is co-owned.

A user holding governance tokens can delegate them to the Governatooorr as the delegatee.

We explicitly want to make this a product that edges on and creates discussion around the potential and dangers of such a product. Hence, the design and language used is cartoonish.

High level design

The design of the Governatooorr breaks down into a few core functionalities:

1. A lightweight web app that allows token holders to delegate their tokens to the Governatooorr delegatee and express their general voting preference. The web app also lets people donate to the Governatooorr some ETH in order to pay for its operation.
2. An autonomous service that implements all the business logic of the Governatooorr including:
    a. Watches out for new delegations and adds them to a list. Only tokens for which a threshold of delegated tokens are received are actively voted in. This establishes the active set.
    b. Watches for proposals for tokens in the active set. Collects all votable proposals.
    c. Uses the expressed preferences by delegators to establish a prompt that is sent to ChatGPT together with the governance proposal. We use Langchain

    here. Each agent in the autonomous service does this independently. We get back a voting intention for the proposal.
    d. Autonomous service votes based on the voting intention.

3. The autonomous service multisig is tied to an ENS name. The address of the multisig is used as the delegatee.

The list of general voting preferences the user can select from:

- Good – if the aggregate voting preference is good, vote for the ideal governance option
- Evil – if the aggregate voting preference is evil, vote for any governance option that is not the ideal

Technical constraints:

- We restrict attention to DAOs which are available via the Tally API. This simplifies the design for the purpose of the Hackathon.
- We use ComposeDB to save data

Autonolas Service Specification

Autonomous services that use Autonolas technology are implemented as Finite State Machines where the logic is split across independent components: rounds define the rules to transition across different states and behavior, who implement the actual business logic. At the end of each round, agents making up the service agree on the round's data output. For more information about how this works, have a look at the key concepts and FAQ sections in our docs.

Backend Server

We have a standard http server that provides the following endpoints:

- POST [/delegate]: Accepts JSON containing the address of the delegator, the token address for which delegate was made to the delegatee, and the general voting preference. This is triggered by the frontend after a user has signed the actual delegate transaction. Returns status codes.
- GET [/delegations/{address}]: Returns JSON with a list of delegations for the address.

The backend server is implemented as a separate skill that writes to the agent's shared state.

Rounds

List of proposed rounds to achieve the functionality.

FSM 1:

1. **SynchronizeDelgationsRound**: This round is used to synchronize all new delegations received via POST requests on the server.
2. **VerifyDelegationsRound**: This round is used to verify the latest on-chain delegations. Specifically, we check that the data collected via POST requests is matching on-chain delegation status. We use this info to establish the “current_delegations” dictionary that maps tokens to users and their delegations amounts.
3. **CollectActiveProposalsRound**: This round collects active proposals for each token in current_delegations and creates the “active_proposals” dictionary mappen tokens to proposal ids.
4. **SelectProposalRound**: This round selects the proposal to vote on. We use the time to expiry and the amount of delegations we have for a proposal for prioritisation.

FSM 2:

5. **EstablishVoteRound**: This round contains the core voting logic. Each agent in the service will run a LangChain model to go from a prompt to the voting intention. The prompt will be based on the voting preferences and a template.
6. **PrepareVoteTransactionRound**: This round uses the vote intention to prepare a voting transaction.

Single FSM diagram

Keep in mind that the FSM diagram just accounts for the happy path and that error handling will be added once the complete technical requirements are known. Other already available functionality, like agent registration and reset and pause rounds will be chained to this simplified FSM (Figure 1) to form a complete FSM (Figure 3). The FinishedLoop round in the following diagram represents a chaining point and will be removed during chaining.

Figure 1: diagram of the simple FSM

Chained FSM diagram

Figure 2: diagram of the chained FSM

Roles and responsibilities

- **Service developer:** implements the rounds and its business logic, API integrations, FSM chainings, chain integrations and any other capability needed for the service to run.

- **Agent:** single instance of the service logic that runs the rounds.

- **Agent operator:** runs one or more agent instances as specified by the service owner. Updates to new versions when required by the service.

- **Service owner:** ultimate responsible for the service. Decides the service needs, the number of agents and operators and registers the service in the Autonolas protocol.

Other service details

- **Number of operators:** depending on the level of decentralization we could increase or decrease this number. We recommend starting the service with a set of one operator running four agents and scale it up over time.

- **Number of API integrations:** ChatGPT

- **Who runs the autonomous service:** independent operators would run this service in a best-case scenario. Of course, at the beginning, it will run on Propel.

- **Who pays for gas:** gas fees, as well as infrastructure costs, are both part of the operational costs and service owners are responsible for covering them. In this case, the service ownership would be entirely Autonolas'.

Web App Spec

There are 4 main user stories associated with the web app:

- "As a user I want to be able to delegate tokens to the Governatooorr delegate address"
  - Nice to haves:
    - "As a user I want to be able to see what I have already delegated."
    - "As a user I want to be able to undelegate tokens."

- “As a user I want to be able to tell Governatooorr how I would generally like it to vote.”
- “As a user I want to see what voting preference I set.”
    - Nice to have:
        - “As a user I want to be able to update my general voting preference.”
- “As a user I want to see what the current voting preference for the Governatooorr is”
- “As a user I would like to be able to donate ETH to Governatooorr to fund its gas operations.”
- “As a user I want to understand what Governatooorr is and how it works”

The web app will use

- Next.js for the base application framework
- Antd as main UI kit
- `Autonolas-frontend-library` for specific components, e.g. wallet connection

The user stories above suggest the following required components:

- Delegate
    - Allows the user to specify a token and amount. Allows them to build and submit a transaction on Ethereum mainnet in order to delegate those tokens to the Governatooorr address.
- SetVotePreference
    - User can input a preference and submit it to the autonomous service.
- Donate
    - A user can input an amount of ETH before submitting a transaction to send that ETH to the Governatooorr address.

Delegate & SetVotePreference

- Process
    - Complete delegation -> on success, trigger availability of SetVotePreference form
- Delegate
    - Needs governance address
    - Needs access to relevant token contract addresses
    - Token address input
    - Amount input
- SetVotePreference
    - Radio input – 1 of 2 options:
        - Good
        - Evil
    - Form submission → post request to autonomous service endpoint

Donate

- Input amount of ETH
- Submit transaction

Potential additional features

- This is intentionally just an MVP. Over time, the project can be extended to make it more safe and useful. E.g. data currently held in memory could be persisted to Ceramic.

Appendix

Operating costs

The following table outlines an approximation of the costs operators incur when running agents as part of an autonomous service on cloud solutions like AWS. This example uses m5.large instances (8 GB, 2vCPU) as reference, and assumes one agent per instance. Lower requirements might be enough for simple services. No transaction gas cost or any other cost apart from the hosting has been taken into account.

Total Agent instances	Daily costs (USD)	Monthly costs (USD)
1	2.304	69.12
2	4.608	138.24
3	6.912	207.36
4	9.216	276.48
5	11.52	345.6
6	13.824	414.72
7	16.128	483.84
8	18.432	552.96

9	20.736	622.08
10	23.04	691.2

Table 1: Autonomous service running costs as a function of agent instances.

Threat model

It is important to note that autonomous services (and hence agent services) do not need to store the history of the common service state permanently. This implies that agent services may be internally using blockchain-like technology, but they never implement a standard L1/L2 layer.

Autonomous services work under the following threat model:

- A service is managed by a service owner, who is in charge of managing the service lifecycle (e.g. sets it up and can shut it down)
- A service is run by a set of operators, each running at least one agent instance, for a total of *n* agent instances
- Every pair of agent instances in the service can securely and independently communicate
- A majority of the *n* agent instances run the agent code defined by the service (typically at most ⅓ of the instances are allowed to be malicious for the service to be guaranteed to run)
- A malicious agent instance can deviate arbitrarily from the code that is supposed to run
- A service is registered in a L1/L2 blockchain from which the economic security of the service is bootstrapped
- Every operator must lock a deposit for each agent instance they own in the L1/L2 blockchain where the service was registered
- Agents can punish each other's misbehavior by submitting fraud proofs to the underlying chain, causing slashing of the deposit of the malicious instance
- The service owner locks a deposit equal to the total deposits requested from the agent instances. This is used to incentivise the service owner to release the agents deposits at the end of the lifetime of the service

An autonomous (or agent) service is decentralized by virtue of minimizing the trust placed on individual agent instances. Although a service owner could potentially be penalized for misbehavior, they are largely assumed to act honestly, as it has no control over the service when it is live (similarly to the entity that deploys a smart contract in a programmable blockchain).

Private key management

The purpose of this policy is to establish standards for the secure generation and storage of private keys within the context of Valory.xyz deployment of Multi Agent Systems within the web3 space. Effective implementation of this policy will reduce the risk of irrecoverable loss of assets and promote best practices amongst both internal and internal stakeholders.

Scope

This policy applies to private keys generated in the course of business operation, and entails best practices when producing keys for both Testnets and Mainnets. The document covers processes for:

- Generation of new keys
- Storage of keys locally
- Access of keys within deployment environments
- Transfer of Assets between internal and external Addresses

Policy

Keys should **never** be transferred across common communication channels such as Discord, Google Chat or email. Any keys which have been transferred as so should instantly be considered as compromised and efforts should be made to recover funds from these keys wherever possible.

Multi-signature wallets should be used wherever possible, where this is unavoidable, access to the newly generated key is limited to the deployer/application owner.

Generation of new keys

All keys must be generated using the **`autonomy generate-key`** tool. This encourages password protection and ensures that the risk of internal users storing unencrypted keys is minimized.

Keys must be generated as and when required, on the basis of one key per application/usage.

Storage of keys locally

Key Generation flow

- Create a new folder locally
- Create a new virtual environment
- Install open-autonomy
- Bulk generate keys

Upon generation of keys locally, it is the responsibility of the developer to ensure that;

- The local device is password protected and encrypted.
- The keys are never located in a git repository directory

Deployment keys

Keys used for production contract deployment should be considered throw away and must be destroyed after;

- The deployment is completed
- The newly deployed contract has been transferred to a multi-sig

Keys used for on-going operations in deployment i.e. individual agent keys must also be backed up and encrypted on an air-gapped offline device.