Autonolas Agent Services On-chain Representation

Introduction

The on-chain protocol anchors the Autonolas agent services running off-chain on the target settlement layer and provides the primitives needed to create, operate and secure such services. Autonolas benefits from a modular design with a clear separation of concerns and opportunity for extensibility without compromising its security and permissionless nature.

This suggests that the contracts:

- follow the core-periphery architecture (such as in Uniswap), which allows for changing out periphery functionality without changing the data models at the core,
- allow for extension via modules (such as in MakerDAO).

The protocol is not directly upgradable and instead relies on various upgrade alternatives, prominently it features upgradable modules, strategies and parameters, to name a few. Direct protocol upgrades are achieved by launching a new version of the protocol with the old deployment staying intact, developers using Autonolas agent protocol will need to actively migrate to newer versions of the protocol. Examples of modules include governance. Governance is particularly important in a modular system, as governance is used to vote on the adoption or abandoning of modules. By ensuring an immutable core the Autonolas protocol provides guarantees to the ecosystem that their components, agents and services, once created, are not mutable by governance, an important guarantee of censorship resistance.

Whilst the Autonolas on-chain protocol is built with the open-aea framework in mind as the primary framework for realizing autonomous services, it does not prescribe usage of the open-aea and allows for services to be implemented on alternative frameworks.

Elements

Agent Component
A piece of code together with a corresponding configuration that is part of the component code. In the context of the open-aea framework, it is either a skill, connection, protocol or contract. Each component is an ERC721 NFT with a reference to the IPFS hash of the metadata which references the underlying code via the code specific IPFS hash.[1] We describe how code is referenced on-chain via hashes next.

[1] Although currently code is referenced using IPFS this nested design allows for forward-compatibility with other platforms like Arweave.

The hash in the minted NFT refers to the metadata as per the ERC721 metadata standard. It has the following form:

```
{
    "title": "Name of the component",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "Identifier of the component/agent"
        },
        "description": {
            "type": "string",
            "description": "Describes the component/agent"
        },
        “version”: {
            "type": "string",
            "description": "semantic version identifier"
        },
        "component/agent": {
            "type": "string",
            "description": "A URI pointing to a resource with mime type image/* representing the asset to which this NFT represents. Consider making any images at a width between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5 inclusive."
        }
    }
}
```

The metadata file itself then points to the underlying code via the “component” or “agent” field.

Similarly, the service has a config hash. This also points to a metadata field as above:

```
{
    "title": "Configuration of an agent service",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
```

```
        "description": "(Off-chain/human readable) Identifier of the service"
      },
      "description": {
        "type": "string",
        "description": "Describes the service"
      },
      “version”: {
        "type": "string",
        "description": "semantic version identifier"
      },
      "config": {
        "type": "string",
        "description": "A URI pointing to a resource with mime type image/* representing the asset to which this NFT represents. Consider making any images at a width between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5 inclusive."
      }
    }
}
```

The on-chain protocol has no means of validating anything off-chain. Hence, the protocol optimistically assumes that any code referenced in it is correct. This assumption is reinforced through the tokenomics: components, agents or service configurations which are incorrectly referenced are unusable and therefore not showing up in the reward mechanism. Furthermore, if a service owner misspecified the mapping on the contract and config level then this should be obvious to operators in the service who can then choose to flag it or ignore it (if benign).

Canonical Agent
A configuration and optionally a code that defines the agent. In the context of the open-aea framework, this is concretely the agent config file which references various agent components. The agent code and configuration are identified by its IPFS hash. Each agent is an ERC721 NFT with reference to the IPFS hash of the metadata which references the underlying code via a further IPFS hash.

Whether the author of a component or agent correctly makes this association is up to them (i.e. an optimistic design approach is followed). Autonolas tokenomics incentivises the correct mapping.

Agent Instance

An instance of a canonical agent, running off-chain on some machine. This runs the agent code with the specified configuration. Each agent must have, at a minimum, a single cryptographic key-pair, whose public address identifies the agent.

Service
A service is made up of a set of canonical agents, a set of service extension contracts (defined below), a number of instance agents per canonical agent and a number of operator slots. A service defines the blocktime at which the service needs to fill slots and when agent instances need to go live. Each service is an ERC721 NFT.

Service Multisig or Threshold-Sig
A multisig associated with a given service. It contains the assets the service owns and validates the multi-signed transactions arriving from the agents. The protocol supports multiple multisig implementations from which the service owner can choose. Most prominently, Autonolas builds on the popular and well-audited multisig Gnosis Safe, for which the protocol calls the Gnosis Proxy Factory to create a Proxy instance.[2]

Service Extension Contract
A set of custom on-chain contracts which extend the functionality of a service beyond what the agent protocol offers. These contracts are outside the scope of the Autonolas on-chain protocol. The service extension contracts are also responsible for defining the payment plan under which operators and the Autonolas protocol (and therefore developers) are rewarded.

Third-Party Contract
Any contract which might be called by the service extension contract or the protocol. These contracts are outside the scope of the Autonolas on-chain protocol.

Roles

Developer
Develops components and agents. Registers agents and components. Registering involves minting an NFT with reference to the IPFS hash of the metadata which references the underlying code.

Service Owner

[2] The Proxy does not contain any substantial logic. For signature validation and other utilities it calls into the Proxy Master, a singleton contract.

Individual or entity[3] that creates and controls a service. The service owner is responsible for all aspects of coordination around the service, including paying operators and ensuring the agent code making up the services is present and runnable.

Operator
An individual or entity operating one or multiple agent instances. The operator must have, at a minimum, a single cryptographic key-pair whose address identifies the operator. The operator may also use a smart contract wallet which provides a multi-sig functionality. In the case of a single key-pair, it must be different from any key-pair used by the agent. Operators register for a slot in a service with an agent instance address they control and their operator address.

Service User
Any individual or entity using a given service. These users are outside the scope of the Autonolas stack. It is the responsibility of a service owner to incentivise users to use their service.

Core Smart Contracts

Core smart contracts are permissionless. Autonolas governance controls the process of minting of new components and agents (i.e. it can change the minting rules and pause the minting) and the service management functionalities. The remaining functionalities, in particular transfer functionalities, are not pausable by the governance.

GenericRegistry
An abstract smart contract for the generic registry template which inherits the solmate ERC721 implementation.

UnitRegisty
An abstract smart contract for generic agents/components template which inherits the GenericRegistry.

Component Registry
A contract to represent agent components which inherits UnitRegistry.

Agent Registry
A contract to represent agents which inherits UnitRegistry.

Service Registry

[3] This includes institutions, companies, machines etc.

A contract to represent services and provide service management utility methods which inherits GenericRegistry.
Autonolas extends the ERC721 standard to support appending additional hashes to the NFT over time. This allows developers and service owners to record version changes in their code or configuration and signal it on-chain without breaking backwards compatibility.

Service Registry Utility Token
A core contract that allows secure autonomous services with a custom ERC20 token. This contract wraps the implementation of the original Service Registry contract in order to manage the extra storage variable containing all the custom ERC20 token information. This will moreover contain the slash and drain functionalities.

Periphery Smart Contracts

Periphery contracts are fully controlled by governance and can be replaced to enable new functionality, but also to restrict existing functionality.

GenericManager
An abstract smart contract for the generic registry manager template.

RegistriesManager
The contract via which developers can mint components and agent NFTs.

Service Manager Token
The contract via which service owners can create and manage their services.

Other Contracts

Contracts used to extend existing functionality of periphery or core smart contracts.

OperatorWhitelist
A contract that allows service owners to have a permissioned set of operators able to registrering agent instances and manage their operators whitelist.

OperatorSignedHashes
A contract that allows to get message hash that can be approved by operators in order to let service owners to unbond and/or register agent instances via the operator's

pre-signed message hash. Moreover, the contract allows verifying the provided message hash against its signature.

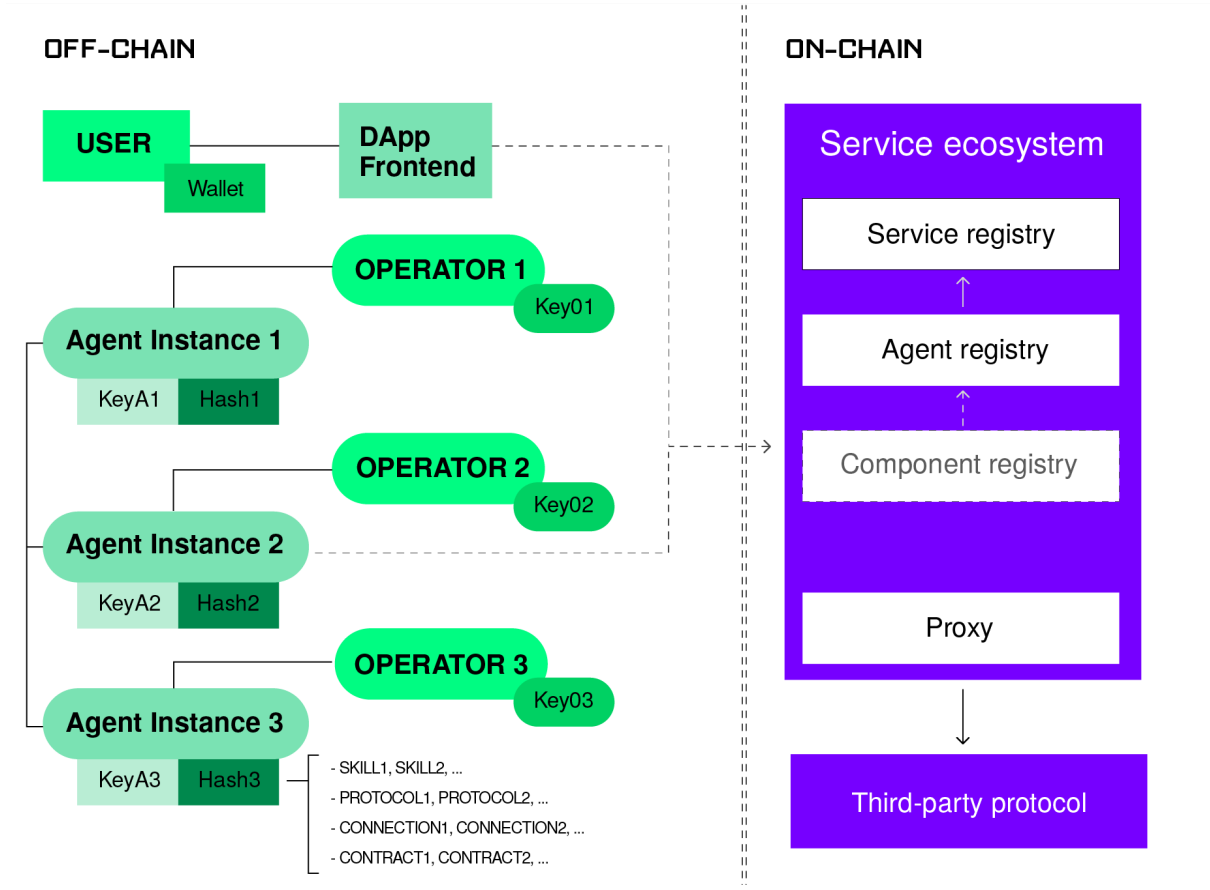

Fig. 1: Basic Autonolas Services Architecture

On-Chain Protocol Workflow

Here we give a succinct description of the basic workflow of the on-chain protocol.

Roughly speaking, the standard workflow starts by a developer building an Agent Component or a Canonical Agent. Alternatively, a given Service Owner starts the workflow by publishing a specification of the Agent Component or Canonical Agent for which they ask developers to provide an implementation. Next, the developer uses the RegistriesManager contract to interact with the Component Registry or the Agent Registry, with the aim to register a representation of the new code on-chain in the form of an NFT. The following information needs to be included during registration: the address of the owner of the code (usually owner and developer coincide), the address of the

developer of the component of the code, the component dependencies of the to be registered agent component/canonical agent and the IPFS hash that references the agent component/canonical agent. Through the minting the developer becomes the holder of the NFT representing the agent component/canonical agent. Off-chain, the code matching the IPFS hash of the registered agent component/canonical agent is pinned on an IPFS node for later retrieval. Together, the on-chain protocol and the off-chain code storage enable code sharing and reuse by providing a place in which canonical agents or agent components can be contributed.

Next, a Service Owner creates a Service from Canonical Agents and registers the corresponding Service to the Service Registry interacting with the ServiceManager. The information that needs to be added to the registry includes: the address of the owner of the Service, the NFTs that represent the Canonical Agents that make up the Service, the number of Agent Instances that need to be created per Canonical Agent, the threshold or minimum number of Agent Instances that can sign the multisig created for the Service, and an IPFS hash which points to metadata referencing a configuration hash of the service. Moreover, if service owners want to secure their service with a ERC20 token and not with Ether, they need to specify the token address. Finally, service owners can eventually opt-in for a permissioned set of Agent Operators.

Once the Agent Operators (or the service owners via the operators' pre-signed message hash) have registered the Agent Instances and the actual slots for running Agent Instances for a given Service are filled, the Service Owner instantiates a multisig with all Agent Instances addresses configured and the stipulated threshold.

Although currently the framework is implemented in Python, agents and their components can be developed in arbitrary programming languages and utilizing arbitrary frameworks as long as they satisfy a number of specific technical requirements enabling their interoperability (e.g., correct implementation of protocols and application logic).

DAO management of PoSes

Protocol-owned-Services are autonomous services owned by an on-chain DAO-governed protocol. Specifically, a DAO assumes the role of a service owner and the service is managed via the DAO governance process. Services are run by third-party operators and can be improved by the means of a new code delivered by third-party developers.

If necessary, service owners can update their services to incorporate an up-to-date code or increase their fault tolerance. However, upgrading and re-deploying a deployed service requires several steps including terminating the service, un-bonding registered agent instances, activating agent instance registrations, registering new agent instances, and re-deploying the service (see

https://github.com/valory-xyz/autonolas-registries/blob/main/docs/FSM.md for more details).

While service owner privileges are required for some steps, operator inputs are needed for others, bearing a necessity for a separate on-chain governance vote. Given the timing with which a software becomes obsolete, such a procedure would be impractical. To simplify the DAO management of PoSes, Agent Operators can pre-sign message hashes that allow service owners to un-bond (*unbondWithSignature()*) and/or register (*registerAgentWithSignature()*). In such a way, DAO members can propose and vote on a unique governance round to enable the governance executor contract to complete the upgrade procedures steps at once.

Secure autonomous service

Autonolas protocol enables secure autonomous services by allowing service owners to choose between securing their services with ETH or any ERC20 token. Service owners are required to ensure a proper service termination by providing a security deposit that is returned when the service is terminated, while agent operators deposit a security bond that can be slashed in case of misbehavior. Using a third-party token can be beneficial for the following reasons:

1. boost adoption and provide utility of a third-party (ERC20) token which in turn is beneficial for regulatory compliance and, ultimately, benefits the overall ecosystem of the project.
2. Incentivize DAOs to move their off-chain processes in the form of service and spend ‘less’ to secure by using DAO governance own native token
3. PoSes owned by a DAO-governed protocol having an ERC20 token as governance token allows more flexible and better agreements between the DAO and operators and ultimately lead to more successful and sustainable services.

To enable the use of custom ERC20 tokens, an additional contract called ServiceRegistryTokenUtility is used to manage the extra storage variable containing all ERC20 token related information. This contract also provides slash and drain functionalities for the ERC20 token bonds. Specifically, all the methods but *deploy()* and *update()*, which allow a *state* transition and require a token-related information, are handled on the serviceRegistryTokenUtility contract, while the original ServiceRegistry contact handles all the service management procedure and performs checks that are not related to the the custom ERC20 token used.

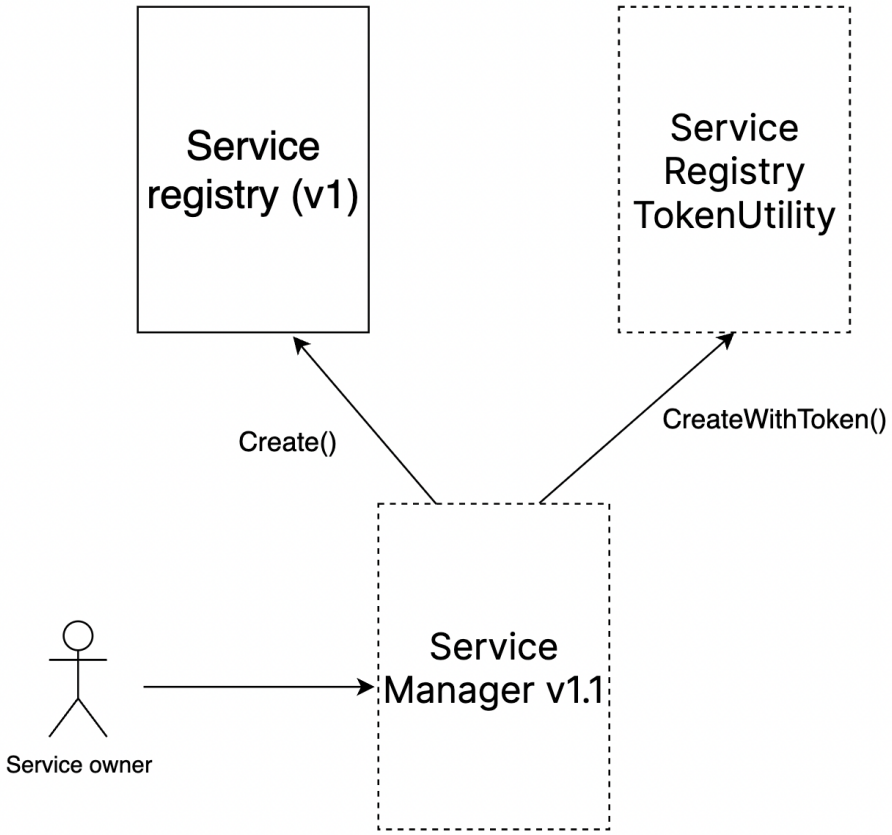

Fig. 2 Secure autonomous services with custom tokens.

When a service is registered, the service owner has to specify the address of the token address which they aim to secure their service with. Depending on the token address used (e.g. ETH address or a custom ERC-20 token), the ServiceManager contract determines whether a service is primarily secured with ETH or a custom ERC20 token. The ServiceManager contract combines calls to the original ServiceRegistry contract and to the ServiceRegistryTokenUtility. Specifically, when the service is secured with ETH, the ServiceManager contract just forwards all the calls to the ServiceRegistry contract. While, if a service is secured with a custom ERC20 token, both ServiceRegistry and serviceRegistryTokenUtility contracts are involved in the calls from the ServiceManager contract.

Autonomous service operation

As already mentioned in the document, Agent Operators can run one or multiple agent instances in an autonomous service. Service owners can choose to enable any agent

operator to opt-in to run their services in a permissionless manner or to whitelist a specific set of operators address to opt-in to running their services.
The OperatorWhilest contract allows service owners to establish a permissioned operation and to maintain a whitelist of operator addresses.

Introducing the ability to opt for a permissioned operation set can have several benefits. It enables a more effective mechanism to incentivize good behavior by considering the behavior of the service operator over time and it can help build trust between service owners and operators. Whitelisting operators can also prevent unwelcomed operators from registering agent instances and thereby registration frontrunning issues.

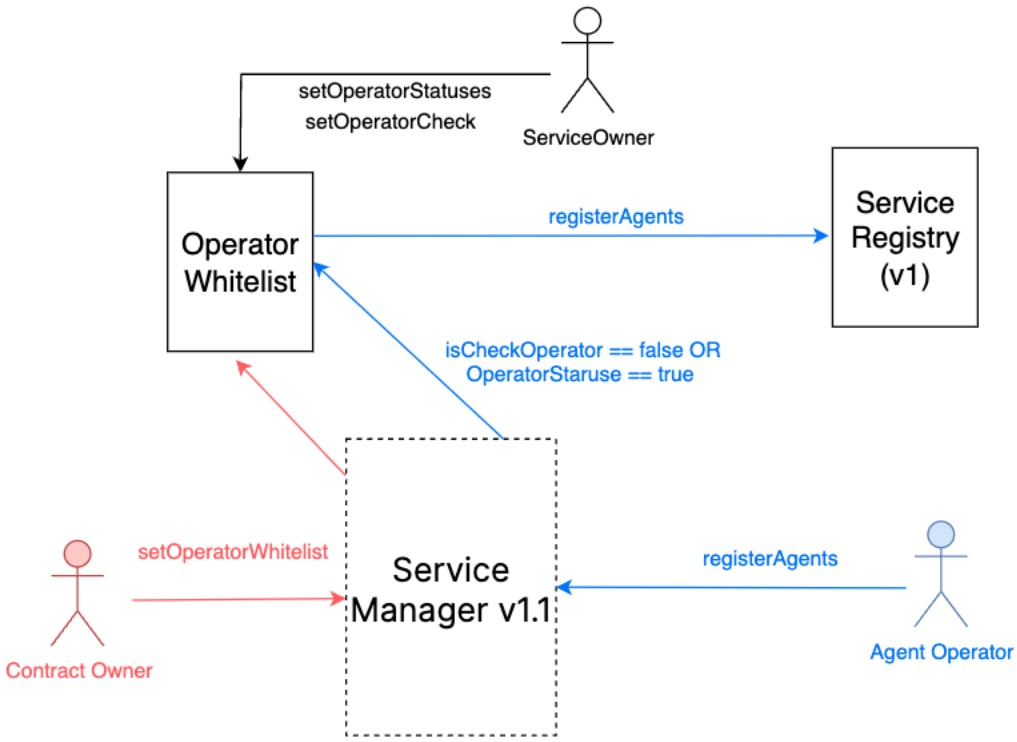

Fig. 3 Once the operatorWhitelist contract is settled, either service owners set a check for permissioned operations of their services and only whitelisted operators are able to register agent instances, or in case of permissionless operations, any operators are able to register agent instances for the service.

Workflow for service owners.
Service owners can maintain a whitelist of operators and enable a permissioned operation using the *setOperatorsStatuses()* function. By setting the operator addresses status to *true* or *false* and setting the operator check to *true,* service owners can ensure that only whitelisted operators can register agent instances for that service.

Once the operator check is set to *true,* it is the responsibility of the service owner to ensure that at least some operator addresses are whitelisted, otherwise only the service owner can register agent instances. If a set of operator addresses is whitelisted and it is aimed for a permissioned operator set, the service owner must ensure that the check on the operator set is set to *true.* Such a check can be eventually set to *true* also with the method *setOperatorCheck().* If the check is set to *false,* the operator can register agent instances in a permissionless manner.

Service owners can de-whitelist operator addresses by setting the operator statuses to *false* in *setOperatorsStatuses()*. If all the previously whitelisted operators are de-whitelisted but *setOperatorCheck()* for a specific service ID is still set to true only the service owner can register agent instances.

Lastly, service owners can set the check in *setOperatorCheck()* to *false*, even if there are still whitelisted operators. This will result in agent instance registration being operated in a permissionless manner.