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 a 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, and staking. 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 agent 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 reference to the IPFS hash of the metadata which references the underlying code via a further 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 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 minting) and the service management functionalities. The remaining functionalities, in particular transfer functionalities, are not pausable by 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.

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

Service Registry
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.

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.

ServiceManager
The contract via which service owners can create and manage their services.

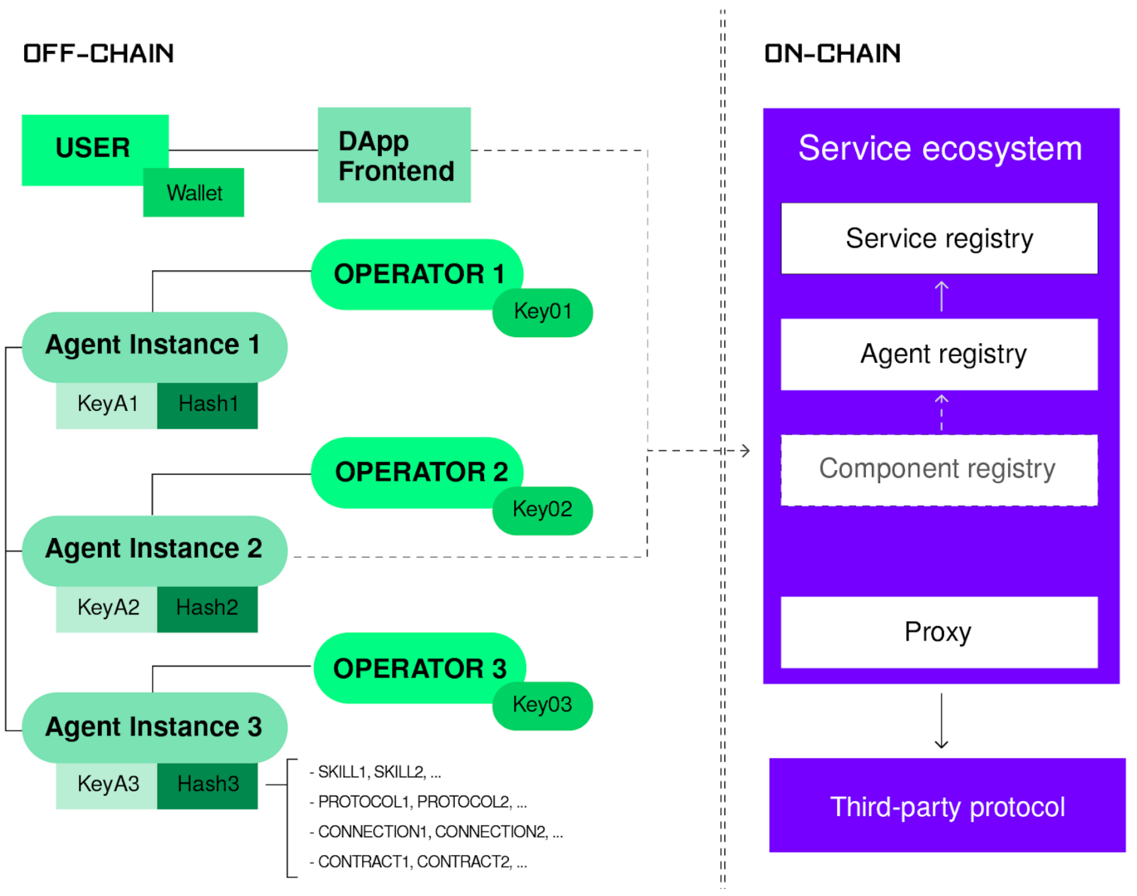

Fig. 5: Basic Autonolas Services Architecture

On-Chain Protocol Workflow

Next 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 an 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.

Once the Agent Operators have registered 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.

Finally, since Operators run agent instances, Developers contribute components and canonical agents code, and Service Owners build services, they should be rewarded proportionally for their efforts and to support the growth of the Autonolas ecosystem. A smart contract, the Payment Plan, allows the distribution of the corresponding rewards to Operators, Developers, and Service Owners. The funds for operating a service are provided by the Service Owner from the revenue of operating their service.

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).