



Integrating Your AI Agent with Pearl

Date: 2/9/2025

Approved for external use by: David Minarsch

1. Introduction

Pearl is the platform that enables users to run autonomous AI agents integrated with the OLAS network. Agent developers can now integrate their own AI agents into the Pearl app, and enable users to run their AI agents. This document outlines the necessary guidelines to prepare your AI agent for seamless integration with Pearl.

2. Prerequisites

We require that the agent is developed using one of two approaches:

1. **A regular Open Autonomy agent.** The developer designs the required packages and FSM and integrates any other required package from the Open Autonomy/Open AEA frameworks. See [the Open Autonomy guides](#) on the Olas Developer Documentation for more information on how to develop your agent.
2. **An Olas SDK agent.** An agent implemented with any other framework and integrated inside a minimal Open Autonomy agent. See the [Olas SDK](#) section on the Olas Developer Documentation to learn how to integrate an external agent into a minimal Open Autonomy agent.

Once your agent is completed and tested locally, you must [push](#) your agent dev packages to IPFS through the `autonomy push-all` command.

You must also [mint components and your Agent Blueprint](#) (not service) on the Olas Protocol. To do so follow [these instructions](#) that leverage upon the [autonomy mint CLI](#) command: It automatically hashes them and mints them using the provided package path and private key.

3. Agent architecture requirements

This section describes the high-level architecture and interface that your agent must satisfy to be integrated into Pearl.

Start and stop agent flows

Whenever Pearl starts an agent, it executes the following tasks:

- For Open Autonomy agents:
 - Start a Tendermint subprocess (<http://127.0.0.1:8080>). Pearl manages the lifecycle of the Tendermint process.
 - Start the agent instance through the `aea run` command as a subprocess.
- For Olas SDK agents:
 - Run the binary.

To stop an agent, Pearl executes the following tasks:

- For Open Autonomy agents:
 - Tries to gracefully stop the Tendermint process through the exit endpoint (<http://localhost:8080/exit>).
 - If this call fails, it kills the Tendermint process by sending the OS SIGKILL signal.
- For Olas SDK agents:
 - Kills the agent process by sending the OS SIGKILL signal.

Therefore, the agent must periodically save the required data and be able to recover its state following a SIGKILL signal.

Folders

Pearl manages the configuration of each agent on separate folders. Namely, each agent and its configuration will be stored in a folder `.operate/services/sc-<uuid4>`, Where `<uuid4>` is a random UUID Version 4. Your agent does not have to interact with the contents of this folder, except for producing logs and accessing the persistent storage. See below.

Some paths to be aware of:

- `.operate/services/sc-<uuid4>/<service_name>/`: Folder containing the service package. The service is fetched from IPFS.
- `.operate/services/sc-<uuid4>/deployment/`: Folder containing the deployment artifacts for your agent.
- `.operate/services/sc-<uuid4>/deployment/ethereum_private_key.txt`: File containing the private key of the agent EOA.
- `.operate/services/sc-<uuid4>/deployment/agent`: **Working directory of the agent. This is the root folder where the agent gets executed.**
- `.operate/services/sc-<uuid4>/persistent_data/`: **Folder for persistent storage. Your agent should not store or access contents outside this folder. The path to this folder is set as an environment variable `STORE_PATH`.**

Keys

The agent will receive a collection (typically one) set up Safe{Wallet} whose owner is the Agent EOA:

- The Agent EOA private key is passed as a text file `ethereum_private_key.txt` in the agent working directory, containing the hex-encoded Agent private key.
- Environment variable `SAFE_CONTRACT_ADDRESSES` contains the Safe addresses in the following format:

```
Python
{
  "ethereum": "<address1>",
  "gnosis": "<address2>",
  ...
}
```

Agent logs

Agents must produce a file with logging information named `log.txt` on their working directory. We recommend the following format for your logs:

```
[YYYY-MM-DD HH:MM:SS,mmm] [LOG_LEVEL] [agent] Your message
```

Agent healthcheck interface

The agent must expose a healthcheck endpoint (GET <http://127.0.0.1:8716/healthcheck>) producing a JSON with the following format:

```
Python
{
  "is_healthy": "<boolean>", # If not, the agent will be restarted by Pearl.
  "seconds_since_last_transition": "<float>",
  "is_tm_healthy": "<boolean>", # Is Tendermint healthy (not required for Olas
  SDK agents)
  "period": "<integer>",
  "reset_pause_duration": "<integer>",
  "rounds": [ # Latest N rounds in chronological order
    "<round_name_i>",
    "<round_name_i+1>",
    "<round_name_i+2>",
  ]
}
```

```

    ...,
],
"is_transitioning_fast": "<boolean>", # Used to determine agent reset
"agent_health": {
    "is_making_on_chain_transactions": "<boolean>",
    "is_staking_kpi_met": "<boolean>",
    "has_required_funds": "<boolean>",
    "staking_status": "<string>"
},
"rounds_info": { # Information about ALL possible rounds of the agent
    "<round_name_1>": {
        "name": "<string>", # Human-readable name
        "description": "<string>",
        "transitions": {
            "<event_1>": "<next_round_1>",
            "<event_2>": "<next_round_2>",
            ...
        }
    },
    "<round_name_2>": {
        "name": "<string>",
        "description": "<string>",
        "transitions": {
            "<event_1>": "<next_round_1>",
            "<event_2>": "<next_round_2>",
            ...
        }
    },
    ...
}
"env_var_status": {
    "needs_update": <boolean>, # Update required for env vars
    "env_vars": { # List of env vars required to be updated
        "<env_var_1>": "<message_1>",
        "<env_var_2>": "<message_2>",
        ...
    }
}
}
}

```

Health checker

When the agent starts, Pearl runs a background Health checker process that monitors it through the healthcheck endpoint. The key field is `is_healthy`, which indicates whether the agent is healthy or not.

- Pearl periodically polls the healthcheck endpoint (currently every 30 seconds) and will restart the agent if `is_healthy` is reported as `false` for a specified number of consecutive failures (currently set to 5).
- It is up to the agent developer to define the condition under which `is_healthy` is set to `false`.
- Currently, services on Pearl trigger this condition when the agent remains in a particular state for longer than `2 * RESET_AND_PAUSE_TIMEOUT`.

Note: the healthcheck polling condition may be changed in the future. Be aware that you might need to update this output accordingly (i.e., providing additional status on the healthcheck endpoint).

Monitoring and activity tracking

Pearl displays information about your agent by using the `rounds` and `rounds_info` parameters from the healthcheck endpoint. Also, Pearl will display a warning to the user if your agent provides `env_var_status.needs_update=true` in the healthcheck endpoint.

The UI to show agent activity is Pearl-native and nothing is required from the agent developers, apart from the rounds and rounds info.

Agent user interface (optional)

The agent can optionally serve a user interface webpage on the endpoint <http://127.0.0.1:8716/> and Pearl will display it through an embedded web browser. It is the responsibility of the agent to handle and request going through this endpoint. In particular, the agent can process post requests so that the user can actively communicate data in real time with the agent.

Standard environment variables

The following standard environment variables are set before running the agent:

- `ETHEREUM_LEDGER_RPC`
- `GNOSIS_LEDGER_RPC`
- `BASE_LEDGER_RPC`
- `CELO_LEDGER_RPC`
- `MODE_LEDGER_RPC`
- `STAKING_TOKEN_CONTRACT_ADDRESS`

- `ACTIVITY_CHECKER_CONTRACT_ADDRESS`: Activity checker contract for the associated staking contract.
- `SAFE_CONTRACT_ADDRESSES`: Safe multisig addresses for the agent.
- `SAFE_CONTRACT_ADDRESS` (legacy environment variable for single-chain agents).
- `STORE_PATH`: Path within the service directory that the agent can use to store persistent data. Usually located in `.operate/services/sc-<uuid4>/persistent_data/`

Note that these variables are not automatically set for your agent unless you define them in the service template as `computed` variables. See section [Service template](#) below.

Withdrawal

Pearl allows the user to withdraw from the wallets associated with an agent to a specified wallet address. Namely, upon user request, Pearl will withdraw the funds of each chain the agent is defined from the following wallets in order:

1. Agent Safe
2. Agent EOA
3. Master Safe
4. Master EOA

to the specified address. Note that Pearl does not (and can not) manage any funds that are particular for a given agent, for example, funds invested in a liquidity pool. It is the responsibility of the agent to withdraw these funds before Pearl initiates the withdrawal from Agent EOA and Agent Safe.

Note: We are currently working on defining a methodology to:

- **Let Pearl know that the agent has finished all withdrawal operations before proceeding with withdrawal.**

Security practices

In addition to the QA recommendations above, the source code of your agent must adhere to robust security standards to mitigate vulnerabilities. We strongly recommend following best practices guidelines such as the [OWASP Developer Guide](#) and the [CWE Top 25 Most Dangerous Software Weaknesses](#) to prevent common security issues.

Architecture requirements specific to Open Autonomy agents

This section only applies to Open Autonomy agents, and it does not apply to Olas SDK agents.

Framework and Python version

We require that the agent be developed in the current Open Autonomy version being used by [Pearl repository](#) (typically, it will be the latest release of Open Autonomy). The Python version must be in accordance with the framework version. See the `pyproject.toml` files for more information about versions.

Python dependencies

Currently, Pearl executes Open Autonomy agents through a binary compilation of the Open AEA library using Pyinstaller. If an agent requires specific Python libraries to work, agent developers must open a Pull Request on the [Pearl repository](#) to request that these libraries be included in the binary.

Special characters

To ensure compatibility across various systems and avoid encoding issues, the agent source code must only include characters within the ASCII printable range (32-126). While Python supports UTF-8 encoding, which can represent a wide range of characters from various languages and symbols, using characters outside the ASCII range, such as accents, icons, and emojis, can cause unexpected behavior and potential errors, particularly when the code is executed across different systems.

Note on AEA binaries

As stated above, Pearl uses the Open AEA framework to run your agent through the `aea run` command. This command is encapsulated into a binary compiled with Pyinstaller for the following configurations:

- Windows x64 (`aea_win.exe`)
- Mac ARM x64 (`aea_bin`)
- Mac Intel x64 (`aea_bin`)

We recommend that you test your agent with the latest version of these compiled binaries, that come shipped with the Pearl release.

Code QA

The Open Autonomy framework encourages the use of a number of linter and security tools (Isort, Black, Mypy, Bandit,...) to ensure that your agent code meets good coding practices. We recommend that you read the section [Analyse and Test](#) on the Olas Developer Documentation.

Security practices

In addition to the QA recommendations above, the source code of your agent must adhere to robust security standards to mitigate vulnerabilities. We strongly recommend following

best practices guidelines such as the [OWASP Developer Guide](#) and the [CWE Top 25 Most Dangerous Software Weaknesses](#) to prevent common security issues.

4. Integration with Pearl

Service template

You must open a PR on the Pearl repository providing a service template [here](#), with the same format as other agents.

Funding requirements

Funds are transferred from the Master Safe to the agent EOA/agent Safe as required. A Pearl user only has to transfer funds to the Master Safe, and Pearl will distribute the funds accordingly to the agent EOA and agent Safe.

The template defines funding requirements per chain, token and entity (agent EOA/agent Safe). Typically, agent EOA fund requirements are only for paying chain transactions, and any funds used for the operation of the business logic itself (e.g., investing in a liquidity pool) have to be specified on the agent Safe.

Pearl periodically checks whether the specified entity funds for any token fall below half of their specified value, and displays an alert for the user prompting a refill of the Master Safe.

If your agent uses staking, Pearl will read the OLAS staking and security deposit requirements from the specified staking contract and display this information before the user can run the agent for the first time.

Additionally, Pearl requires that the Master EOA has an operating balance of the chain's native currency to pay for transactions and urges the user to refill the Master EOA once this balance drops from a given threshold.

All the UI elements and components of the funding mechanism are Pearl-native and no UI is required to be built by the agent team for the funding mechanism.

Environment variables

- Fixed environment variables always take the value specified in the template.
- User-provided environment variables are requested from the Pearl user interface to be filled by the user and passed to the agent. This is useful to set API keys or other secret parameters.
 - The UI elements to update the environment variables by the user is Pearl-native and no UI is required by the agent developer.

- Backend-computed environment variables are filled by Pearl at runtime. There are a number of [standard environment variables](#) that are computed by Pearl (mainly related to staking contracts). **However, they must be explicitly set in the service template so that your agent can read them.**

If your agent requires additional backend-computed environment variables, you must open a PR to the [Olas Operate Middleware repository](#).

Updating the agent

To update your agent, you must provide the updated `hash`, `service_version`, and the `agent_release` version in the configuration template above. Open a Pull Request on the [Pearl repository](#) with the updated values.

Frontend requirements

The requirements can/must be fulfilled via ``/frontend`` source code.

Agent enums (identifier)

New agents will require new enum values to ensure consistency through the application.

[AgentType](#) values should only be added when relevant to a new *type* of agent, i.e. trader, optimus – regardless of the chain, or surface level branding used.

Agent service API classes

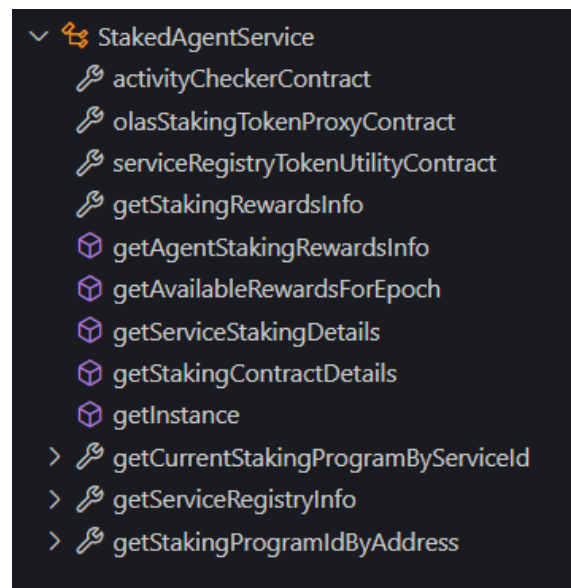
Agents require an agent service class.

Each class exposes an API for interacting with various contracts required and used by an agent.

It is responsible for defining functions for Pearl to access on-chain values related to staking, reward epochs, Olas registry state, and any other abstracted-away on-chain requests.

Active service classes can be found at ``\frontend\service\agents``.

Staked agents can extend **``StakedAgentService``** – an abstract class with commonly required function definitions. This class assumes your agent has been built using the Olas stack, and has staking contracts created via Olas' factory contracts.



Changing implemented service-class-exposed methods will require custom UI implementations to integrate with Pearl.

Importantly, the agent service APIs – as they stand – do not implement state. They are `abstract`, and used as singletons with static methods; they are not instantiated.

Agent service classes are later referenced in the `frontend/config/agents.ts` config – via the `serviceApi` key, though, also accessible by importing them.

Extending config files

Pearl's frontend is mostly configured via. config files. These can be extended with additional values if required by your agent. Naming conventions are relatively straight forward.

agents.ts – agent configuration objects, indexed by AgentType enums

```
JavaScript
[AgentType.PredictTrader]: {
  isAgentEnabled: true,
  requiresSetup: false,
  name: 'Predict Trader',
  evmHomeChainId: EvmChainId.Gnosis,
  middlewareHomeChainId: MiddlewareChain.GNOSIS,
  requiresAgentSafesOn: [EvmChainId.Gnosis],
  requiresMasterSafesOn: [EvmChainId.Gnosis],
  serviceApi: PredictTraderService,
  displayName: 'Prediction agent',
  description: 'Participates in prediction markets.',
},
```

Values in the agent config identify generic metadata such as the chains the agent operates on, naming and description, and whether the agent requires a setup flow.

tokens.ts – chain-specific token configs, indexed by token symbol enums

```
JavaScript
const BASE_TOKEN_CONFIG: ChainTokenConfig = {
  [TokenSymbol.ETH]: {
    tokenType: TokenType.NativeGas,
    decimals: 18,
    symbol: TokenSymbol.ETH,
  },
},
```

```
[TokenSymbol.OLAS]: {
  address: '0x54330d28ca3357F294334BDC454a032e7f353416',
  decimals: 18,
  tokenType: TokenType.Erc20,
  symbol: TokenSymbol.OLAS,
},
} as const;
```

Pearl supports balances for common ERC20 tokens and native gas. Balances relevant to agent operations outside of deployment and staking should generally not be included in this config. RPC request budgets are limited at present, therefore, additional tokens i.e. LP tokens, or any temporary holdings outside of the required scope, should be monitored, displayed, and served to the user via. the custom agent UI interface.

chains.ts – blockchain details, metadata

Where your agent uses a new chain. You can update the chain configuration. Given this chain config may/will be used by other agents in the future it is **important to discuss this implementation with Valory to ensure an RPC is hosted**.

mechs.ts – defining mech contracts

Currently two mech types are supported, Agent Mechs (singular mechs) and Mech Marketplace (decentralized service for multiple mechs to serve requests).

olasContracts.ts – contracts required to interact with Olas

Not updated by third parties. This defines registry and registry token utility contracts.

Agent selection screen

The [AgentSelection](#) component will add your agent to the agent selection screen based on the values set in the agents config.

Agent introduction flow (first run)

```
JavaScript
export const AgentIntroduction = () => {
  const { goto } = useSetup();
  const { selectedAgentType, selectedAgentConfig } = useServices();

  const introductionSteps = useMemo(() => {
    if (selectedAgentType === 'trader') return PREDICTION_ONBOARDING_STEPS;
```

```
if (selectedAgentType === 'memeeoorr') return AGENTS_FUND_ONBOARDING_STEPS;  
if (selectedAgentType === 'modius') return MODIUS_ONBOARDING_STEPS;
```

The [AgentIntroduction](#) component should be extended with the relevant agent `introductionSteps`.

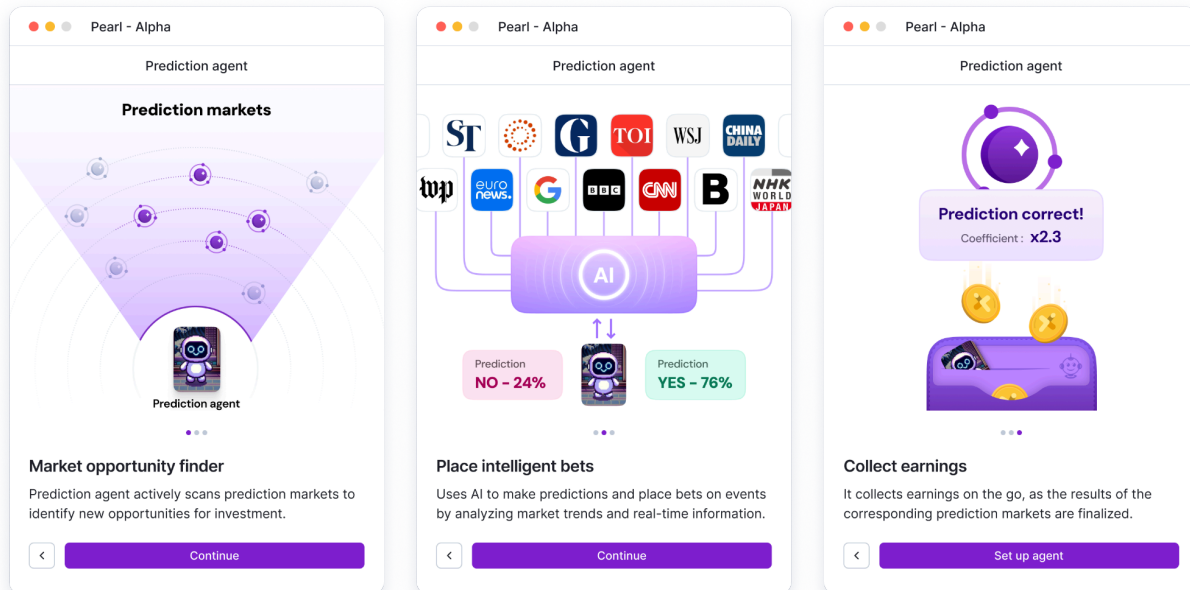
Additional onboarding steps should be defined for your agent, following the types used in existing OnboardingSteps[] constants. [See example](#):

JavaScript

```
export const PREDICTION_ONBOARDING_STEPS: OnboardingStep[] = [  
  {  
    title: 'Market opportunity finder',  
    desc: 'Prediction agents actively scan prediction markets to identify new  
opportunities for investment.',  
    imgSrc: 'setup-agent-prediction-1',  
  },  
  {  
    title: 'Place intelligent bets',  
    desc: 'Uses AI to make predictions and place bets on events by analyzing  
market trends and real-time information.',  
    imgSrc: 'setup-agent-prediction-2',  
  },  
  {  
    title: 'Collect earnings',  
    desc: 'It collects earnings on the go, as the results of the corresponding  
prediction markets are finalized.',  
    imgSrc: 'setup-agent-prediction-3',  
  },  
] as const;
```

The AgentIntroduction component is Pearl-native, so apart from the content required no more UI needs to be developed by the agent developer.

Agent Introduction flow - UI reference



Agent setup flow

Agent setup flows are optional, and depend on your agent. This flow creates a “dummy service” that is pre-populated with environment variables prior to its deployment.

Your agent may not require these steps. As with “Olas Predict”, there are no setup steps. Though, agents that require, for example, a user-submitted API key, or address, will.

Agent setup forms are conditionally displayed, based on the agent type. [See example](#):

```
JavaScript
{selectedAgentType === AgentType.Memeoorr && (
  <MemeoorrAgentForm serviceTemplate={serviceTemplate} />
)}

{selectedAgentType === AgentType.Modius && (
  <ModiusAgentForm serviceTemplate={serviceTemplate} />
)}
```

If required, a new setup form should be created. Ensuring [dummy service creation](#) (where the agent requires pre-deployment variables) is implemented.

The agent setup form UI is Pearl-native, so apart from the variables that are required and the type of each variable, no UI component needs to be developed by the agent developer.

The purpose of the setup flow is to overwrite and persist any default variables available in the service template file. ([See example](#)).

The overridden environment variables are set via [input fields in the created forms](#). Defaulted environment variables should be set in the service template, following the existing template. With three types associated with each value; COMPUTED (values computed by the middleware/agent), USER (values input and set by the user), and FIXED (values that do not change); and additional name and description metadata for each variable.

Staking contracts

Staking program identifiers

This [enum should be extended](#) with an identifier "slug" recognized by both frontend and middleware.

Implementing staking contract addresses, ABIs

Staking contract-related data is stored in `/config/stakingPrograms`. With a [core barrel file](#) that imports staking contract objects by chain.

Each chain has its own staking contract file. As per current implementation, in each file, there are generally two objects:

- An object for all [StakingProgramId → Address mappings](#) (contract addresses)
- An object for all [StakingProgramId → StakingProgramConfig mappings](#) (configs)
 - Each [StakingProgramConfig](#) type defines the required values, in each config, enabling you to infer values required in the used enums.

Activity checkers

Contracts that track, check, verify activity when staking contracts are bounded by some set requirement being met each epoch. They can be referenced by adding them to the [activityChecker config](#) file.

Feature flags

There are standing feature flags which allow gating of features. The current flags can be extended, and [changed here](#).


Agent setup - UI reference

UI requirements

- Setup description text
- Tooltip description for each environment variable

Currently filled manually, dynamic solution to be developed. UI is Pearl-native, no requirement to be built by the agent developer.

Pearl - Alpha



Set up your agent

Set up your agent with access to a [Tenderly](#) project for simulating bridge and swap routes, and swap routes and provide a [CoinGecko API key](#) as a price source.

Tenderly access token ⓘ




Tenderly account slug ⓘ

Tenderly project slug ⓘ

CoinGecko API key ⓘ

Continue

Pearl - Alpha



<

Edit agent settings

Tenderly access token ⓘ

Tenderly account slug ⓘ

Tenderly project slug ⓘ

CoinGecko API key ⓘ

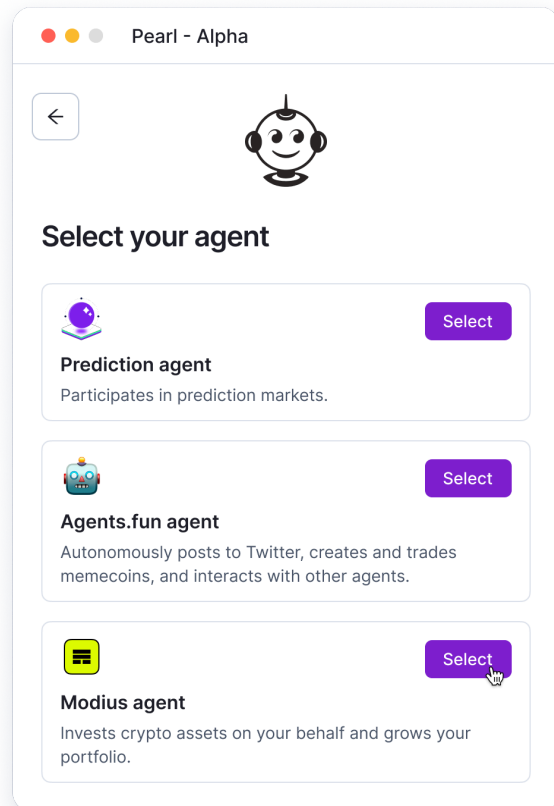
Save changes

Agent presence

The agent will be shown in the main agent selection screen. For this Pearl requires:

- Agent name
- Agent logo
- Agent description

Currently filled manually, dynamic solution to be developed. UI is Pearl-native, no UI required by agent developer.



Staking contract interaction

Streak - Rewards history

Staking rewards history depends on a few factors:

- the staking contract is product of an Olas staking contract factory
- the chain where the contract is deployed has a supported subgraph

Currently chains with supported subgraphs can be [found here](#). Assuming your agent uses/has an associated Olas staking contract, any epoch checkpoints and associated earning details will be available via the subgraphs.

Each subgraph is associated with a chain and managed by Valory. Do not update subgraphs that are “higher level” and used by multiple agents. Agent specific subgraphs should be restricted only to the agent you are integrating.

On Streaks, this value is calculated via an algorithm that determines whether the agent owner has run their agent consistently, once per day, inclusive of any contract switches – provided the staking contracts are deployed via a monitored Olas staking factory contract.

General contract reads

Contract reads should be abstracted away in each agent’s [service API class](#). As previously mentioned RPC request budgets are limited, please leverage multicalls to batch RPC requests and use reasonable intervals when polling RPCs.

5. Next Steps

For further assistance, reach out to the Pearl development team (PM: Iason Rovis - iason.rovis@valory.xyz)