How it works

Goal

The goal of Galadriel is to enable on-chain autonomous AI apps, especially agents. Specifically, we want to enable developers to create on-chain programs which are capable of:

1. calling LLMs and multimodal models, including from closed-source API-providers,
2. permanently storing large amounts of data and retrieving relevant parts of it on-demand, including using embedding-based semantic search,
3. making requests to external services like search engines,
4. executing all the above in a loop running for several minutes.

Why these requirements? First, models are the core of any AI app. Second, storage allows extending and specializing the models' knowledge with relatively little effort. Third, external tools are crucial for extending the AI's capability to act in the real world, in real time. And fourth, long runtimes are a fact of life for state of the art large models, and looping a model (which further extends inference time) improves the AI's reasoning ability.

Design

The core of how Galadriel enables on-chain AI is theoracle .

The oracle enables contracts to make external API calls (including LLMs). It offers an interface for Solidity contracts on the Galadriel chain to call out LLMs or other models, use tools, and retrieve data.

The oracle is implemented as contract that is called asynchronously and has an off-chain component. This is similar in architecture to ChainLink. However, because tool calls (e.g. web searches) and LLM calls do not produce deterministic results nor cannot be averaged, consensus on those requests is not possible. To solve this, we execute the oracle in a trusted execution environment — see theTEE section .

To make your own on-chain AI, you need to build your Solidity contract in a specific way to interact with the oracle. Making a call to most contracts on EVM chains is synchronous: the call is made, and the result is returned immediately. However, the oracle is asynchronous: the call is made, and the result is returned later, when the oracle has finished processing the request off-chain. The async nature of the oracle is because on-chain programs cannot typically execute long-running tasks (block time is the ceiling on execution time).

Due to the above, to make a call to an LLM, generative image model, external tool, or anything else via the oracle you need to use a callback function. This function is called by the oracle when the result is ready.

Usage patterns

Based on this, here are the typical usage patterns for Galadriel.

Simple generative image model call

Assume you want to make one call to a generative image model during the execution of your contract. This is what you need to make e.g. agenerative AI NFT minting dApp .

To do so, you need to add two things to your contract:

- Make a call to the generative image model. This is done by calling a function on the oracle.
- Add a callback function to your contract. This function is called by the oracle when the result is ready.

And the execution flow is as follows (see the diagram below):

1. Your contract calls the oracle contract to make a call to the generative image model — say, DALL-E.
2. The oracle backend makes a call to DALL-E API, off-chain.
3. (About 10-20 seconds passes.)
4. DALL-E API returns a result to the oracle backend.
5. The oracle backend calls the oracle contract in a new transaction, containing the DALL-E response.

6. The oracle contract calls the callback function in your contract, containing the DALL-E response.

For details on implementation, please see the Solidity reference .

Simple LLM call, with history

Generative image models are stateless: you put a prompt in and get an image out. However on-chain ChatGPT and many other AI apps require storing state in the form of conversation history. For example, for OpenAI Chat Completions API you need to pass to the API a list of previous messages.

To enable calling LLMs with history, you need to implement two more functions in your contract for fetching history which is stored in your contract, on-chain. Overall, to call an LLM, you need to add the following things to your contract:

- Make a call to the LLM. This is done by calling a function on the oracle.
- Add a callback function to your contract. This function is called by the oracle when the result is ready.
- Add a history getter function to your contract. This function is called by the oracle to get the history. (Detail: this actually needs to be two seperate functions, one for fetching message contents and the other for fetching message roles.)

In this scenario, the execution flow is as follows:

1. Your contract calls the oracle contract to make a call to the LLM, say OpenAI GPT-4 API.
2. The oracle backend calls the oracle contract, which in turn calls your contract, to fetch the message history.
3. The oracle backend makes a call to the GPT-4 API, off-chain.
4. (About 5-20 seconds passes.)
5. The GPT-4 API returns a result to the oracle backend.
6. The oracle backend calls the oracle contract in a new transaction, containing the LLM response.
7. The oracle contract calls the callback function in your contract, containing the LLM response.

For details on implementation, please see the Solidity reference .

Complex usage: AI agents

AI agents are more complicated. The core of an AI agent is a loop. In each iteration, the output of an LLM call determines what tool to use (e.g. web search), or whether the agent should terminate. After the tool is executed, its output is fed back into the next iteration, where the LLM decides on the next action, etc.

To make an on-chain agent with Galadriel, your callback (that is called when the oracle is ready) needs to contain code for choosing what to do next. For agent, this will typically be one of:

1. Terminate (if the task is complete, or impossible)
2. Make a call to an LLM
3. Make a call to an external tool

This way, the callback function will be repeatedly called by the oracle, until the agent decides to terminate. This creates the loop, even though no explicit loop is present in the contract.

Security model

A lot of trust is placed in the oracle. This is the first step towards a working system while we are designing and building a minimal-trust approach to this oracle.

In the current architecture of Galadriel, the oracle runs inside a trusted execution environment (TEE) and can be verified by anyone to be executing the correct code; see more in the TEE section.

TEE

The oracle is executed in a trusted execution environment, specifically an AWS Nitro Enclave . Upon startup, the enclave generates a private key which can only be accessed inside the enclave, and is used for signing the transactions made by the oracle. Since the image running in the enclave is public and auditable (see links below) and for every execution of the oracle an attestation is provided, you can verify every execution of the oracle to be correct after-the-fact.

See more details about the TEE setup in the following links.

- Oracle contract
- Oracle back-end code
- Enclave setup and how to verify attestations: see galadriel-ai/teeML

See also

- Implementations of the above on in galadriel-ai/contracts
- :* Example contracts
-
    - Oracle back-end
-
    - Oracle contract
- FullSolidity reference

Quickstart Solidity reference twitter github discord Powered by Mintlify