**Chainlink Verdikta Design**

DRAFT

10/4/2024



Shown in the figure above is a system using Chainlink to ensure execution (ultimately trustless) of AI models with results provided as an oracle service to smart contracts. The developed system will also include an example smart contract and user interface providing automated contract creation and resolution.

**Interface A**

As illustrated in the figure, some models may be run locally with a direct interface, while others, such as ChatGPT, will run remotely. Interface A enables interaction with both local and remote AI models by using a unified interface and a factory method to manage the different providers.

This design allows for seamless integration with local models (such as those supported by the Ollama framework) and remote models (such as closed models by OpenAI and Anthropic) through the same API. The interface calls the appropriate provider based on the request and routes it to the corresponding API, whether it’s local (Ollama) or remote (OpenAI, Anthropic).

When using one AI model, this will be effectively a relay from interface B. When using more than one AI model, the data returned from the AI model may be modified by combining it with results returned from the other models.

**Input Format:**

* Inputs are sent as HTTP POST requests with the body formatted in JSON. The request typically includes:
  + prompt (string): The input or question to be processed by the model.
  + model (string): The specific model identifier.
  + provider (string): Specifies the AI provider, such as “Ollama” for local models or “OpenAI” or “Anthropic” for remote ones.
  + Optional image (base64 encoded): For image support, though current open-source (local) models support text-based inputs only.

Example Input:

json

{

"prompt": "What is quantum mechanics?",

"model": "gpt-4o",

"provider": "OpenAI"

}

**Output Format:**

* Outputs are returned in JSON format, whether the model is local or remote. The response typically includes:
  + result (string): The AI-generated output or response.
  + error (optional): Error messages if the request fails.

Example Output:

json

{

"result": "Quantum mechanics is a fundamental theory in physics..."

}

The factory method ensures that the appropriate provider (local or remote) is called without requiring additional input modifications.

Example factory method

*usage:*

*const provider = LLMFactory.getProvider('OpenAI');*

*const response = await provider.generateResponse('What is AI?', 'gpt-4');*

**Messaging Protocol and Endpoints:**

* GET /api/tags: This endpoint fetches available models from all providers, both local and remote.
* POST /api/generate: This endpoint processes the input prompt and sends it to the selected provider. The response is returned in JSON format.

**Error Handling:**

Interface A catches and logs errors if a model is unavailable or fails to process the input, returning informative JSON error responses to the caller.

This unified structure enables Interface A to handle multiple providers (local and remote) through a consistent API, enhancing flexibility and scalability by allowing the addition of new providers with minimal changes.

**Interface B**

Interface B from the External Adapter to the AI Interface will typically be direct communications on a single computer. Data obtain by the External Adapter as a zip file containing a manifest and supporting documents will be uncompressed and formatted into JSON input data expected by the AI Interface. The return information will be in JSON form and includes a vector of rankings and textual justification for those rankings. This interface is open and free for an arbitrary new definition as part of Verdikta.

**Input Format:**

* The input is JSON-formatted data sent from the Chainlink External Adapter to Interface B. The input includes:
  + prompt (string): The query or task to be processed by the AI model.
  + models (vector): a vector of models to submit the quary to. In includes the following
  + Optional image (base64 encoded): For image-based AI queries. The model selected must support images.
  + Optional iterations (integer): number of times to call the AI model. Results will be averaged before returning rankings.
  + fields:
    - provider (strings): Specifies the AI provider (e.g., “Ollama”, “OpenAI”, or “Anthropic”).
    - model (string): model name (i.e. gpt-3.5-turbo)
    - weight (real): weight for combining results (0 to 1)

Example input:

{

"query": {

"prompt": "The woman in the image is wearing a red dress.",

"image": "base64\_encoded\_image", // Optional: a single base64 encoded image

"iterations": 3, // Optional: number of times to run the query through the LLMs

"models": [

{

"provider": "OpenAI", // Name of the provider

"model": "gpt-3.5-turbo", // Model name

"weight": 0.5 // Weight for combining results (0 to 1)

Count // number of times to run this model

},

{

"provider": "Anthropic",

"model": "claude-2",

"weight": 0.3

},

{

"provider": "Ollama",

"model": "llama3.2",

"weight": 0.2

}

]

}

}

Example Output:

{

"aggregatedScore": [999900, 100], // The final aggregated score vector

"justification": "The dress is most likely red based on the RGB values of the image."

}

**Endpoint for Processing Ranked Results:**

An endpoint will be created to process the AI-generated content in a way that supports output rankings. It will also support calling and aggregating responses from multiple AIs. This endpoint will handle both the retrieval of AI responses from Interface A and the subsequent formatting of the output into rankings and justification. The endpoint is designed to allow the existing interface and functionality to remain available for other applications that don’t require rankings.

Supported Endpoints

• GET /api/generate: Endpoint for retrieving available models from the supported AI providers (both local and remote).

• POST /api/generate: Endpoint for sending prompts and receiving generated text results (for applications not requiring rankings).

• POST /api/rank-and-justify: This endpoint will process the AI model’s response(s), extract the necessary rankings, and generate a textual justification. It will use the same AI model interaction methods but format the output differently from the standard generation endpoint.

**Messaging Protocol:**

JSON will be used as the messaging format for both input and output, providing consistency across all interfaces and endpoints. The ranking and justification responses will adhere to the same JSON structure as the standard outputs but will include new fields (rankings and justification).

**Error Handling:**

If the AI model fails to generate the rankings or an error occurs during processing, the interface will return a structured error message. This error is captured in the error field of the JSON response, allowing smart contracts to handle exceptions gracefully.

Example error output:

{

"error": "Failed to generate rankings due to invalid model input."

}

**Interface C**

Interface C between the Chainlink Node and the External Adapter will typically be direct communications on a single computer. It will be defined in a form that is required for Chainlink, as the Chainlink Node software will not be modified from the standard release for this use. The Chainlink Node will provide a JSON file that contains 1) a CID for an IPFS file, 2) functional information, including a job run ID (a value, also called a request ID, unique to each instance of calling the External Adapter) for callback, as required by Chainlink. The return will be an immediate 200 response to the HTTP post to convey receipt and a later asynchronous callback to the Chainlink Node to pass a JSON file with a vector of integers and a CID for a text document holding descriptive text.

This interface will be set using the Chainlink node GUI under the “bridges” tab. The External Adapter will be a RESTful interface that communicates through HTTP-type commands and is stateless.

*Detailed Description*

Input (from Chainlink Node to External Adapter):

1. cid: A Content Identifier (CID) for a zip file on IPFS, which contains data for the AI to evaluate.
2. id: A unique identifier for this call generated by the Chainlink Node. It will be used to identify the information sent later through a callback.

Output (from External Adapter to Chainlink Node):

1. vector: A list/vector of integers representing the results of the AI evaluation.
2. description\_cid: A CID for a text file on IPFS containing the reasoning or explanation behind the returned numbers.

Example input will be of the following form (posted to the specified URL):

POST /evaluate

{

"id": "jobRunId-123456789", # Unique jobRunId for callback

"data": {

"cid": "QmdfTbBqBPQ7VNxZEYEj14VmRuZBkqFbiwReogJgS1zR2n"

}

}

The immediate response from the External Adapter to this POST will be of the following form:

HTTP/1.1 200 OK

Content-Type: application/JSON

{

"jobRunId": "jobRunId-123456789",

"status": "in\_progress",

"message": "Job received and processing"

}

Then the results of the AI execution will be provided later through a callback URL. Example output will be as follows (posted to the specified URL):

POST /v2/runs/jobRunId-123456789

{

"vector": [12, 35, 46, 57],

"description\_cid": "The action was justified…"

}

Additional Data

Here is an example Chainlink job specification (including a “bridge” task description) for use with this External Adapter:

type = "directrequest"

schemaVersion = 1

name = "AI Evaluation Task"

maxTaskDuration = "0s" # No hard timeout

observationSource = """

cid\_input [type = "ethuint256"]

async\_task [type = "bridge" name = "ai\_evaluation\_bridge" requestData="{ \\"cid\\": \\"$(cid\_input)\\" }" allowUnrestrictedNetworkAccess=true]

# The result will be returned when the external adapter posts back

"""

**Interface D**

Interface D between the Smart Contract using the new service on chain and the Chainlink Node will be defined by Ethereum and Chainlink requirements. This communication will happen over the Ethereum blockchain.

A specialized smart contract affiliated with the Chainlink Node will be used for this purpose, called the Chainlink Node oracle contract. This smart contract is a standard contract available to support fielding chainlink nodes (Oracle.sol). This contract will have an address, called the oracle contract address, that any smart contract calling it will need to know.

The input from the Smart Contract to the Chainlink Node will include the following information: 1) a CID for the data, the oracle contract address, the calling contract address, the Chainlink job ID (a value specific to the AI request call on the Chainlink Node that persists over time), and a fee. The returned information from the Chainlink Node to the Smart Contract will use a callback function contained in the calling Smart Contract. The retuned information will include a vector of integers and a CID corresponding to the textual reasons for the results

Details

Here is a framework for a smart contract making a call to the Chainlink Node:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "@chainlink/contracts/src/v0.8/ChainlinkClient.sol";

contract AIChainlinkRequest is ChainlinkClient {

using Chainlink for Chainlink.Request;

// The Oracle and job spec

address private oracle;

bytes32 private jobId;

uint256 private fee;

// Result from the AI evaluation

uint256[] public evaluationResult;

string public descriptionCID;

// Constructor sets up Chainlink contract parameters

constructor(address \_oracle, bytes32 \_jobId, uint256 \_fee, address \_link) {

setChainlinkToken(\_link); // Chainlink token contract address

oracle = \_oracle; // Chainlink node oracle contract address

jobId = \_jobId; // Job ID from Chainlink node

fee = \_fee; // Fee for the request (in LINK tokens)

}

// Request AI evaluation via Chainlink node

function requestAIEvaluation(string memory cid) public returns (bytes32 requestId) {

Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this), this.fulfill.selector);

// Send the IPFS CID to the external adapter

request.add("cid", cid);

// Send the request to the Chainlink node

return sendChainlinkRequestTo(oracle, request, fee);

}

// Callback function called by Chainlink node with the AI evaluation results

function fulfill(bytes32 \_requestId, uint256[] memory \_evaluationResult, string memory \_descriptionCID) public recordChainlinkFulfillment(\_requestId) {

evaluationResult = \_evaluationResult;

descriptionCID = \_descriptionCID;

}

}

Future Development

In the figure above, a block labeled “ac” is shown as part of this interface. This represents an aggregator contract, which will be used to provide trustlessness by polling multiple Chainlink Nodes and eliminating outliers. This component will be added in the future.

**Interface E**

Interface E between the User Interface Backend and the Smart Contract will be defined by Ethereum and requirements. This communication will happen over the Ethereum blockchain. The User Interface Backend will communicate using tools that are part of the Web3.js JavaScript library. The User Interface Backend will use the third-party service Infura to access the Ethereum (or L2) blockchain.

The User Interface Backend will communicate with the Smart Contract by posting a transaction. The backend will sign the transaction using information provided by the user’s cryptocurrency wallet (e.g. Metamask or Ledger).

**Interface F**

There will be two parts to this interface, between the User Interface Backend and the External Adapter, which will primarily serve to limit information conveyed over the Ethereum blockchain. Firstly, this interface will convey the bulk of the information for the AI query in the form of a zip file holding manifest, textual, and multimedia information. This zip file, possibly encrypted, will nominally be placed on IPFS by the User Interface Backend and downloaded using IPFS by the External Adapter. Secondly, the textual explanations given by the AI Models for their numerical determinations will be placed on IPFS, again, possibly encrypted, by the External Adapter, with the User Interface Backend downloading this information for presentation to the user.

Though IPFS is the nominal form of communication for this data, it may be communicated by any means. The IPFS CID is a cryptographic hash, and the zip or text file it represents can absolutely be validated using the CID, as (to astronomical probability) only the originating file gives its specified CID. A separate, possibly encrypted communication channel could also be used to convey this data.

**Interface G**

The User Interface Frontend and Backend components will communicate through a RESTful interface on the backend.