Onchain Riddle Challenge - Final Report

Onchain Riddle Challenge - Final Report

1. Project Overview

This document explains the architecture, development process, and my thought process behind building the Onchain Riddle Game for the Zama challenge. The challenge required creating a smart contract-based riddle game where users could read a riddle and submit answers via a connected wallet. Additionally, it required implementing a bot that automatically publishes a new riddle as soon as the current one is solved.

I approached this problem as a full-stack Web3 engineer, leveraging tools I'm deeply familiar with: Solidity, Hardhat, React, and Node.js. My primary goal was to build a modular, maintainable system that was also easy to deploy using Docker-based infrastructure.

2. Architecture and Folder Structure

The project repository consists of three main components:

- **blockchain/**: This folder contains the smart contract built using Hardhat. It defines the on-chain riddle logic, events (`RiddleSet`, `AnswerAttempt`, `Winner`), and access control (only the bot can set new riddles).

- **onchain-riddle-frontend/**: A React app built using Wagmi v2 and RainbowKit to interface with the contract. This app allows users to view the current riddle and submit their answers through their wallets (e.g., MetaMask).

- **onchain-riddle-bot/**: A Node.js service that listens for the `Winner` event on-chain and sets the next riddle. It uses `ethers.js` for blockchain interaction and is designed to run continuously as a background service.

At the root, I've added a `docker-compose.yml` file that enables running the bot and frontend services together with a single command. Both the frontend and the bot have their own Dockerfiles for isolated builds.

3. Thought Process and Development Strategy

I started with the smart contract since it's the source of truth in this system. I ensured that the contract was minimal but secure, with proper event logging and access control. Once the base functionality was verified using Hardhat, I moved to the frontend.

For the frontend, I decided early to use Wagmi v2 instead of v1 because of its composability and simplified contract interactions using `useReadContract` and `useWriteContract`. This allowed me to avoid boilerplate web3 handling and focus on UX.

The bot was the last piece. I built it to react to on-chain events instead of polling, as that's more efficient and realistic. I used a simple array of riddles to simulate a riddle bank and designed it so new riddles can be fetched dynamically in future iterations.

Onchain Riddle Challenge - Final Report

During development, I encountered several challenges:
- Wagmi v2 required a completely new setup compared to v1, including breaking changes in hooks and providers.
- The WalletConnect modal opened automatically; I had to figure out how to properly configure `autoConnect: false` through custom connectors.
- Dockerizing the frontend was tricky at first because of how Create React App serves static files. I had to ensure Nginx was properly configured and the build output was correctly copied.

Throughout the process, I tried to keep the implementation simple, extensible, and testable. I often switched between modules (contract frontend bot) to ensure the full integration worked smoothly.

4. Dockerization and Deployment

To streamline development and enable simple multi-service deployment, I used Docker from the start. Each component (frontend and bot) has a Dockerfile, and the root `docker-compose.yml` orchestrates them.

This approach gave me the following advantages:
- Consistency across environments
- Isolated builds and logs for each service
- Simplified deployment to platforms like Railway or Render

It also helped me iterate faster by allowing me to test the entire stack locally before deploying anything.

5. Final Thoughts

This assignment allowed me to demonstrate my experience in end-to-end Web3 engineering. From writing secure smart contracts to building reactive frontend apps and automating workflows through bots, I was able to tie together various pieces into a cohesive solution.

My decisions were guided by modularity, deployment-readiness, and developer experience. The result is a project that can be maintained, scaled, and extended without significant rewrites. Future improvements could include answer deduplication, user leaderboards, and integration with a real riddle API or database.

Overall, this was a rewarding build that let me combine technical skill with creative architecture in a real-world Web3 scenario.