ChatGPT

ChefsPlan 5.0 Scaffold Overview

ChefsPlan is a decentralized gig-work platform connecting chefs and restaurants. This scaffold outlines a modular architecture based on the ChefsPlan whitepaper's design principles. It includes zkSync Era-compatible smart contracts (for shift management, escrow, and reputation), off-chain agents for automation, decentralized storage via IPFS, and a Next.js front-end dApp. The focus is on an MVP implementation with a trustless escrow flow, basic matching, and reputation system. Below is the project structure and a breakdown of each component.

Project Structure

```
chefsplan/
├── contracts/                  # Solidity smart contracts (zkSync Era
compatible)
│   ├── ShiftManager.sol        # Shift posting and application logic
(upgradeable)
│   ├── Escrow.sol              # Escrow for payments (upgradeable)
│   ├── Reputation.sol          # Reputation tracking (upgradeable)
│   └── Proxies.sol             # Proxy contracts (e.g.,
TransparentUpgradeableProxy)
├── scripts/                   # Deployment and automation scripts (Hardhat)
│   ├── deploy.js               # Script to deploy contracts (with proxy
setup)
│   └── agentStartup.js         # (Optional) Script to launch off-chain
agents locally
├── test/                      # Hardhat test files for contracts
│   ├── escrow.test.js          # Tests for escrow functionality
│   └── shiftManager.test.js    # Tests for shift lifecycle
├── agents/                    # Off-chain autonomous agents (Node.js or
Python)
│   ├── matchingAgent.js        # Suggests chefs for new shifts (listens to
events)
│   ├── complianceAgent.js      # Verifies credentials (simulated, updates
contract)
│   ├── payoutAgent.js          # Monitors completed gigs and releases
payment
│   └── copilotAgent.js         # AI Copilot (monitoring, optional MVP
placeholder)
├── frontend/                  # React/Next.js dApp for chefs and
restaurants
│   ├── pages/
│   │   ├── index.js            # Landing page / shift listings
│   │   ├── chef/dashboard.js   # Chef view: available shifts, applications
│   │   └── restaurant/portal.js # Restaurant view: post shift, manage
applicants
```

```
│   ├── components/             # Reusable UI components (ShiftCard,
ProfileCard, etc.)
│   ├── context/                # Web3 context providers (wallet integration)
│   └── utils/                  # Helper libraries (e.g., IPFS client, zkSync
wallet utils)
├── hardhat.config.js           # Hardhat configuration (with zkSync plugins)
[1]
├── package.json                # Project dependencies (Hardhat, zkSync,
IPFS, etc.)
└── README.md                   # Documentation for setup and usage
```

Folder Explanations: The `contracts` directory contains all Solidity contracts, each following upgradeable patterns. The `agents` folder holds Node.js (or Python) scripts that run off-chain logic (e.g., event listeners and automation). The `frontend` contains a Next.js app divided into pages for chefs and restaurants and utilities for blockchain interactions. Hardhat configuration is tailored for zkSync Era, and scripts are provided for deployment and testing.

Smart Contracts

The platform's core logic is implemented in Solidity smart contracts, deployable on zkSync Era. We use **Hardhat** with zkSync plugins for compilation and deployment, ensuring compatibility with zk-rollup specifics. All contracts follow an **upgradeable proxy** pattern (using OpenZeppelin upgradeable libraries) so that logic can be updated while data is preserved [2]. Key contracts include:

- **ShiftManager.sol:** Manages gig postings and applications. Restaurants can post shifts, and chefs can apply. It emits events like `ShiftPosted`, `ShiftApplied`, `ShiftAccepted`, `ShiftCompleted`, and `ShiftDisputed` to signal state changes. It holds minimal data on-chain (e.g. references to IPFS for shift details) and references an escrow instance for payment status. Functions might include `postShift(detailsCID, payment)` for restaurants to create a shift, `applyToShift(shiftId)` for chefs to apply, `acceptApplication(shiftId, chefAddress)` for restaurants to select a chef (which triggers funding the escrow), `markComplete(shiftId)` for chefs to mark the job done, and `confirmComplete(shiftId)` for restaurants to release funds. A simplified snippet of the ShiftManager interface:

```
contract ShiftManager {
    event ShiftPosted(uint indexed shiftId, address indexed restaurant,
string detailsCID, uint payment);
    event ShiftApplied(uint indexed shiftId, address indexed chef);
    event ShiftAccepted(uint indexed shiftId, address indexed chef);
    event ShiftCompleted(uint indexed shiftId, address indexed chef);
    event ShiftDisputed(uint indexed shiftId, address indexed by);

    struct Shift { address restaurant; address chef; uint payment; string
detailsCID; bool open; bool completed; }
    mapping(uint => Shift) public shifts;
    uint public nextShiftId;

    function postShift(string memory detailsCID, uint payment) external
```

```
returns (uint) { /* ... */ }
    function applyToShift(uint shiftId) external { /* ... */ }
    function acceptApplication(uint shiftId, address chef) external { /* ...
requires msg.sender is shift.restaurant ... */ }
    function markComplete(uint shiftId) external { /* called by chef */ }
    function confirmComplete(uint shiftId) external
{ /* called by restaurant to release escrow */ }
    function disputeShift(uint shiftId) external { /* ... */ }
}
```

Explanation: The `ShiftManager` keeps track of shifts and their state. Posting a shift records its details (with an IPFS CID for full description) and expected payment. Applying and accepting update the state and assign a chef. Completion and dispute actions emit events for off-chain agents or future on-chain arbitration. The actual funds are handled by the Escrow contract, referenced by shift ID.

- **Escrow.sol:** Holds and releases payments for gigs. When a shift is accepted, the restaurant deposits the payment into Escrow (e.g., by calling `deposit(shiftId)` with value). The Escrow contract locks funds until release conditions are met. It may implement functions like `deposit(shiftId)` (payable by restaurant), `confirmWork(shiftId)` to release payment to chef (callable by restaurant or an authorized agent when work is completed), and `refund(shiftId)` if a refund is needed (e.g., if no chef was selected or dispute resolved in favor of restaurant). If a dispute occurs, Escrow can hold funds until an off-chain resolution is reached. The Escrow contract ensures **trustless payouts** – funds can only go to the intended chef or back to the restaurant based on the agreed outcome. For simplicity in MVP, the restaurant's confirmation triggers immediate transfer to the chef. Example Escrow data structure and functions:

```
contract Escrow {
    struct Deposit { address restaurant; address chef; uint amount; bool
released; }
    mapping(uint => Deposit) public deposits;  // key: shiftId
    address public shiftManager;  // authorized contract

    modifier onlyShiftManager() { require(msg.sender == shiftManager, "Not
authorized"); _; }

    function deposit(uint shiftId) external payable onlyShiftManager {
        deposits[shiftId] = Deposit({ restaurant: tx.origin, chef:
address(0), amount: msg.value, released: false });
    }
    function assignChef(uint shiftId, address chef) external
onlyShiftManager {
        deposits[shiftId].chef = chef;
    }
    function release(uint shiftId) external onlyShiftManager {
        Deposit storage dep = deposits[shiftId];
        require(!dep.released && dep.chef != address(0), "Invalid release");
        dep.released = true;
        payable(dep.chef).transfer(dep.amount);
    }
```

```
    // (Potential dispute handling and refund functions can be added)
}
```

Explanation: The Escrow is kept simple: only the `ShiftManager` (as the controlling contract) can call its functions, ensuring logic goes through the intended flow. On shift acceptance, `ShiftManager` calls `deposit` (forwarding the restaurant's payment) and `assignChef`. On completion, `ShiftManager` calls `release` to pay the chef. This design isolates fund custody and allows extension (e.g., adding a dispute resolver that could call a refund). The contract is configured for zkSync, meaning using zkSync's ETH (or token) as currency; no special modifications are needed beyond using the zkSync Hardhat compiler and deployment tooling.

- **Reputation.sol:** Manages reputation scores for participants. After each completed gig, chefs and restaurants can rate each other (e.g., 1-5 stars). This contract stores accumulated ratings and computes an aggregate (e.g., average or feedback count). Functions might include `submitRating(address subject, uint8 score)` which can be restricted to be called only by participants of a recently completed shift (to prevent arbitrary ratings). The contract might maintain a mapping from user address to a struct with total score and count of ratings, and expose `getReputation(address)` to return a computed reputation (e.g. average score or a composite). For example:

```
contract Reputation {
    struct RatingStats { uint totalScore; uint count; }
    mapping(address => RatingStats) public ratings;

    function submitRating(address user, uint score) external {
        require(score <= 5 && score >= 1, "Score out of range");
        // In MVP, assume any caller can rate to simplify, but integrate with
ShiftManager for auth in real version
        ratings[user].totalScore += score;
        ratings[user].count += 1;
    }
    function getReputation(address user) external view returns (uint
averageScore) {
        RatingStats memory stats = ratings[user];
        if (stats.count == 0) return 0;
        return stats.totalScore / stats.count;
    }
}
```

Explanation: The Reputation contract is straightforward: it collects scores and calculates an average. In a full implementation, only an authorized entity (like the ShiftManager contract or the involved parties of a shift) could call `submitRating` to prevent abuse. This contract can be upgraded to more complex logic (like weighting by transaction volume or using ZK proofs for verifiable off-chain feedback) as needed.

- **Upgradeability:** All contracts are deployed behind proxies for upgradeability. We utilize OpenZeppelin's proxy pattern (e.g., TransparentUpgradeableProxy or UUPS proxies) with a proxy admin. Using the zkSync Hardhat toolchain, we integrate the `hardhat-zksync-upgradable` plugin which supports Transparent, UUPS, and Beacon proxies on zkSync Era [2]. Each contract has an `_initialize` function instead of constructor (for proxy use), and deployment scripts

use `deployProxy` from the zkSync upgrades plugin to deploy. For example, using Hardhat's zkSync plugin, one might write:

```
// Hardhat deploy script snippet for upgradeable contract
const ShiftMgr = await hre.deployments.getArtifact("ShiftManager");
const shiftMgr = await hre.zkUpgrades.deployProxy(deployer, ShiftMgr, [], {
initializer: 'initialize' });
```

This ensures we can seamlessly upgrade logic (e.g., add features or fix bugs) without disrupting contract state or requiring migration, aligning with the platform's need for long-term maintainability.

- **zkSync Compatibility:** The Hardhat config is adjusted for zkSync Era. We use Matter Labs' plugins for zkSync, for example `@matterlabs/hardhat-zksync-solc` to compile with zkEVM-compatible solc, and `@matterlabs/hardhat-zksync-deploy` and `...-upgradable` for deployment. The network config includes the zkSync Era testnet endpoint and an Ethereum L1 for bridging context. For instance, the config might include:

```
networks: {
  zkSyncTestnet: {
    url: "https://testnet.era.zksync.dev",  // zkSync Era testnet RPC
    ethNetwork: "goerli",                   // L1 network (if bridging
needed)
    zksync: true,                           // enables zkSync plugin
    accounts: [ process.env.PRIVATE_KEY ]
  }
}
```

As shown in zkSync's docs, the `zksync` flag must be `true` for the network configuration [1]. We follow zkSync deployment practices such as using the custom ETH (which is basically bridged ETH on L2) and paying fees in ETH. All solidity code avoids unsupported EVM patterns to be compatible with zkRollup constraints.

Off-Chain Autonomous Agents

Off-chain agents are programs running off the blockchain to automate and support the platform's operations. They listen to blockchain events and perform actions that smart contracts alone cannot, such as complex computations, external integrations, or conditional triggers. We outline four key agents (Node.js for accessibility, but Python could also be used):

- **Matching Agent:** Monitors newly posted shifts and suggests potential chefs. It listens for the `ShiftPosted` event from the ShiftManager contract. When a new shift is posted, the agent could query an off-chain database of registered chefs (with their skills, location, availability) to find matches. In the MVP, this can simply log recommendations or automatically notify those chefs (e.g., via off-chain messaging or by calling an on-chain function to record a suggestion). For example, using Ethers.js in Node:

```
const provider = new ethers.providers.JsonRpcProvider(zksyncRpcUrl);
const shiftMgr = new ethers.Contract(shiftManagerAddress, ShiftManagerABI,
```

```
provider);
shiftMgr.on("ShiftPosted", (shiftId, restaurant, detailsCID, pay) => {
  console.log(`New shift posted: ID=${shiftId}, payment=${pay}`);
  // TODO: fetch suitable chefs from off-chain storage and notify them
});
```

Explanation: The agent uses a JSON-RPC provider connected to zkSync. It subscribes to `ShiftPosted` events. On each event, it triggers some logic (in MVP, perhaps a console log or a stubbed function call). In a full system, this agent might call an on-chain method like `recommendChef(shiftId, chefAddr)` (if such exists) or send notifications through a backend service. The Matching Agent helps automate the discovery of talent for a given gig, improving platform efficiency.

- **Compliance Agent:** Responsible for verifying credentials and other compliance checks for chefs and restaurants. For example, it could ensure that a chef who applied to a shift has the necessary food handling certification or work authorization. In a decentralized context, this might involve zero-knowledge proofs (ZKPs) – e.g., a chef provides a ZK proof of a certification without revealing personal data. In the MVP scaffold, we **simulate** this process. The agent listens for an event like `ShiftApplied(shiftId, chefAddress)`. When triggered, it checks the chef's status (perhaps by looking up an IPFS document or an off-chain database for a verification record). If the chef is unverified, the agent could update a state (maybe call a smart contract function like `markUnverified(chef)` or simply log a warning). If verified, it might log approval or even call a contract to store a verification flag for that chef's address. This agent could also interact with a zkSync-compatible **Verifier contract** if ZK proofs were used, but for MVP we just outline the process. Pseudocode:

```
shiftMgr.on("ShiftApplied", async (shiftId, chefAddr) => {
  const verified = await checkChefCredentials(chefAddr);  // off-chain check
or IPFS data
  if (!verified) {
    console.log(`Compliance warning: Chef ${chefAddr} is not verified for
shift ${shiftId}`);
    // Potential action: call ShiftManager.markChefUnverified(shiftId,
chefAddr) or similar
  } else {
    console.log(`Chef ${chefAddr} verified for shift ${shiftId}`);
  }
});
```

Explanation: This shows the agent catching applications. The `checkChefCredentials` could be a stub that reads from a JSON file or IPFS data that was previously uploaded (e.g., KYC/credentials proof). In future, this agent would handle actual ZK verification: for instance, verifying a proof that "chef has certification X" by calling a zkSync Verification contract or using an identity system.

- **Payout Agent:** Ensures that once work is completed, payments are released promptly. It monitors events like `ShiftCompleted` and `ShiftAccepted`. In a simple flow, when a restaurant calls `confirmComplete(shiftId)` (meaning the job is done), the Escrow contract's `release` function might be invoked directly by that same transaction. However, if we want automation or a timeout-based release, the Payout Agent could intervene. One approach: if a chef marks a shift as completed (`ShiftCompleted` event emitted by ShiftManager when chef

calls `markComplete`), and the restaurant does not confirm within a certain time, the Payout Agent could trigger a fallback release. For MVP, we assume the restaurant will call confirm, but we include the agent for monitoring. The agent might do the following:

```
shiftMgr.on("ShiftCompleted", (shiftId, chefAddr) => {
  console.log(`Shift ${shiftId} marked complete by chef. Waiting for
confirmation...`);
  // Start a timer for auto-payout if needed
});
shiftMgr.on("ShiftAccepted", (shiftId, chefAddr) => {
  console.log(`Shift ${shiftId} accepted; escrow funded. Monitoring for
completion.`);
});
escrow.on("PaymentReleased", (shiftId) => {
  console.log(`Payment for shift ${shiftId} released from escrow.`);
});
```

In an advanced scenario, if a certain time passes after `ShiftCompleted` without a `PaymentReleased`, the agent (which could hold a privileged role or use a predefined contract method) might call `Escrow.release(shiftId)` itself to pay the chef, assuming no disputes. The agent thus acts as a safety net to enforce the *trustless escrow* — ensuring the chef gets paid even if the restaurant becomes unresponsive after completion (the specific policy would be defined in smart contracts or off-chain agreements).

- **AI Copilot Agent (Optional):** This is a placeholder for future enhancement where an AI agent oversees the system's health and provides decision support. In the scaffold, we include `copilotAgent.js` as a stub that could, for example, aggregate system data or detect anomalies. It might subscribe to various events (posts, disputes, ratings) and log insights. For instance, it could flag if a particular user accounts for many disputes or suggest optimal pricing based on historical data. In an MVP, this agent might simply output periodic summaries (e.g., number of active shifts, average reputation scores, etc.) without direct on-chain interaction. This component underscores the extensibility of the platform to include AI-driven features later.

All agents can run independently, likely as Node.js scripts or services. They utilize **web3/ethers** libraries to connect to zkSync. Note that to listen to events in real-time, a WebSocket or polling provider is needed (e.g., using an RPC endpoint that supports `eth_subscribe`). The agents should handle reconnection and error cases, ensuring robustness in a decentralized environment where network hiccups can occur.

Decentralized Storage (IPFS Integration)

ChefsPlan uses IPFS for storing non-transactional data, ensuring decentralization and reducing on-chain storage. Examples of data stored on IPFS include detailed **shift descriptions**, chef profiles or resumes, and **KYC/credential proofs** (or references to them). Rather than storing large strings or documents in smart contracts, the contracts will store content identifiers (CIDs) that point to IPFS files. We integrate with IPFS via services like **Web3.Storage** or Infura's IPFS API for ease of use.

In the project, the `utils/` or `agents/` directories include an IPFS client setup. For example, using **web3.storage** in Node.js/Browser:

```
// utils/ipfs.js
import { Web3Storage, File } from 'web3.storage';
const client = new Web3Storage({ token: process.env.WEB3STORAGE_TOKEN });

export async function uploadJSON(data) {
  // Convert JSON data to File object
  const buffer = Buffer.from(JSON.stringify(data));
  const files = [ new File([buffer], 'data.json') ];
  const cid = await client.put(files);  // upload to IPFS via Web3.Storage
  console.log('Stored data on IPFS with CID:', cid);
  return cid;
}
```

As shown in Web3.Storage docs, the `client.put(files)` method returns a CID string after uploading the files [3]. The scaffold uses this in places like when a restaurant posts a shift: the frontend or an agent can upload the shift details (role, time, location, requirements, etc.) as a JSON to IPFS and get a CID. That CID (a content hash) is then passed to the `postShift` contract call on-chain. The smart contract stores the CID (e.g., in `Shift.detailsCID`). Later, anyone (chefs, restaurants, agents) can retrieve that data from IPFS by referencing the CID. For example, the frontend might use an HTTP gateway or IPFS API to fetch `ipfs://<CID>` which contains the shift description.

Data stored: - **Shift Descriptions:** Instead of a large text field on-chain, `detailsCID` points to an IPFS JSON or PDF containing job details, pay rate, location, etc. - **User Credentials:** A chef's certifications or background check can be uploaded to IPFS (perhaps encrypted if sensitive) and the hash stored either on-chain or off-chain in a registry. The Compliance Agent can fetch and verify these. In a future ZK-based approach, the actual credential could be verified by a proof, but the existence of a file on IPFS with a given CID might be used as a reference. - **Dispute Evidence:** If a dispute arises, evidence like photos or documents could be stored on IPFS. The dispute resolution process (though not fully on-chain in MVP) can reference these CIDs.

We ensure that **no private data is stored unencrypted on IPFS**, in line with best practices (anyone with the CID can fetch the content) [4] [5]. For MVP, most data can be considered non-sensitive (job postings are public, and basic profile info can be public). If sensitive data (like personal IDs) were needed, it should be encrypted before uploading to IPFS, with decryption keys shared as appropriate off-chain.

Integration-wise, the project's `package.json` includes `web3.storage` or `ipfs-http-client`. Web3.Storage offers a straightforward API and pinning service to keep data available on IPFS and Filecoin. Alternatively, using Infura IPFS API or a self-hosted IPFS node could be configured in the `utils/ipfs.js` file.

Frontend dApp (React + Next.js)

The user interface is a decentralized application (dApp) built with **React and Next.js** for simplicity and performance. It interacts with the smart contracts via a Web3 library (Ethers.js or web3.js) and connects to users' Ethereum/zkSync wallets.

Key features and pages:

- **Wallet Integration:** Users (chefs or restaurants) connect their Ethereum wallet (e.g., MetaMask or WalletConnect) configured for zkSync Era. We utilize zkSync's ethers provider extension – `Web3Provider` from `zksync-ethers` – to wrap the injected provider [6]. This allows signing transactions on zkSync seamlessly with familiar wallets. For example, in the Next.js app, we might have a context hook that does:

```
import { Web3Provider } from "zksync-web3";
// On user clicking "Connect Wallet":
const ethProvider = window.ethereum;  // MetaMask's provider
const zkProvider = new Web3Provider(ethProvider);  // zkSync-compatible
provider
const signer = zkProvider.getSigner();
```

This signer is then used with contract ABIs to send transactions (e.g., `shiftManagerContract.connect(signer).postShift(...)`). The UI should display the connected network (to ensure it's on zkSync testnet/mainnet) and handle chain switching if necessary.

- **Restaurant Portal:** A Next.js page (e.g., `restaurant/portal.js`) where restaurant users can post new shifts and manage ongoing gigs. It includes a form to create a shift (job details, date/time, payment amount). When submitted, the front-end calls `uploadJSON` (to IPFS) for the details, then calls the `postShift` function on the ShiftManager contract via the connected wallet. The UI will list shifts the restaurant has posted, showing their status (open, accepted, completed, etc.). For an open shift, the portal shows incoming applications (by listening to events or polling the contract for applicants). The restaurant can then select an applicant (calls `acceptApplication(shiftId, chefAddress)` which triggers funding the escrow – likely prompting the wallet to send the payment). Once a shift is in progress or completed, the restaurant sees a button to confirm completion (calls `confirmComplete` to release funds). They will also have a way to rate the chef after completion (calls `Reputation.submitRating(chef, score)`).

- **Chef Dashboard:** A page like `chef/dashboard.js` where a chef can find available shifts and manage their applications. This page queries the ShiftManager for all open shifts (or listens to `ShiftPosted` events). It displays a list of gigs (with details fetched from IPFS via the CID). The chef can click on a shift to view details and then choose to apply (`applyToShift(shiftId)` transaction). The dashboard also lists shifts the chef has been accepted for or completed, prompting them to mark completion when the time comes (`markComplete(shiftId)` transaction when they finish the job). After a job, the chef can rate the restaurant (`Reputation.submitRating(restaurant, score)`). The chef profile section can show their current reputation score (queried from `Reputation.getReputation(chef)`).

- **Common Components:** The `components/` folder includes reusable UI pieces. For example, `ShiftCard` component to display a shift summary (title, date, payment, etc.) on both the listing page and dashboards; `ApplicationList` to show which chefs applied to a shift (for a restaurant user); `WalletConnectButton` for connecting the wallet; and so on. These ensure a consistent look and feel.

- **Web3 Utilities:** In `frontend/utils/` or a dedicated `frontend/context/`, we define helpers for interacting with the blockchain. For instance, loading contract ABIs and addresses into Ethers.js, setting up the `Web3Provider` as shown above, and perhaps a convenience function to fetch data from IPFS (using an IPFS gateway or `axios`). We also include the zkSync Era chain configuration (chain ID, RPC URLs) so the dApp can prompt network switching if the user is on a different network.

- **Next.js Config:** Since we are dealing with an Ethereum-like environment (zkSync), we might need to polyfill certain Node.js modules for web3 libraries to work in the browser (if using web3.js). With ethers, this is usually not an issue. We ensure environment variables (like Infura API keys or Web3.Storage token) are handled properly (perhaps in `.env.local` for the Next app, and not exposed unless needed). The front-end is mostly client-side for web3 interactions, but Next.js can pre-render static content (like a home page listing recent shifts, which could be fetched server-side from an indexer or subgraph if one existed; for MVP, we can fetch directly from chain on the client side).

- **Security & UX:** The dApp should clearly indicate transaction costs and await confirmations. On zkSync, transactions finalize quickly, but the UI will handle waiting for receipts. We integrate WalletConnect to allow mobile users or those with other wallets to connect – this is facilitated by libraries such as `@walletconnect/web3-provider` or the newer WalletConnect v2. In the scaffold, we include it as an option in the WalletConnectButton component.

By structuring the front-end with separate views for chefs and restaurants, we ensure each user type has a tailored experience. Under the hood, both use the same contracts but with different UI paths. This separation is done via Next.js dynamic routes (e.g., `/chef/[page]` and `/restaurant/[page]`) or simple conditional rendering based on user role (which could be determined by whether an address has any chef activity vs. restaurant activity).

Deployment and Testing Setup

To make development straightforward, we provide configuration for both local and testnet deployments:

- **Local Deployment (zkSync Local Node):** For a purely local setup, developers can use zkSync's local node (via `hardhat-zksync-node` plugin or the zkSync CLI that runs a local environment). However, zkSync local setup can be complex; an alternative is deploying to an official testnet like zkSync Era Testnet (e.g., on Sepolia). The Hardhat config is prepared for testnet by default, but it also includes a configuration for a local node if needed. For example, using the `hardhat.config.js`:

```
networks: {
  ZKsyncEraTestnet: {
    url: "https://testnet.era.zksync.dev",
    ethNetwork: "sepolia",     // L1 network for cross-chain ops, if any
    zksync: true,              // enable zkSync in Hardhat 7
    accounts: [process.env.PRIVATE_KEY]
  },
  // Optional local network
  ZKsyncLocal: {
    url: "http://localhost:3050",  // example local zkSync RPC
```

```
    ethNetwork: "http://localhost:8545", // local L1 for bridging if needed
    zksync: true,
    accounts: [process.env.PRIVATE_KEY]
  }
}
```

Running `npx hardhat run scripts/deploy.js --network ZKsyncEraTestnet` will deploy the contracts to zkSync testnet. The deploy script uses Hardhat's zkSync deploy plugin, which handles contract deployment and verification. We also include deployment of proxies via the upgradable plugin. After deployment, the script can output contract addresses and save them to a file or the frontend config.

- **Deployment Scripts:** The `scripts/deploy.js` automates deploying all contracts and linking them. For instance, it will deploy ShiftManager (upgradeable proxy), Escrow (upgradeable), and Reputation (upgradeable). After deploying Escrow, it likely needs its address set in ShiftManager (so ShiftManager knows where to call for deposits/releases). This could be done via an `initialize` function or a separate call after both are deployed. The script ensures the right initialization order (e.g., deploy Escrow, then deploy ShiftManager passing Escrow's address to its initializer, etc.). It also might set the authorized contract in Escrow (e.g., calling `escrow.setShiftManager(shiftManager.address)` if not done in constructor). All this can be done in the Hardhat script and verified with console logs. Finally, it could store addresses in a JSON for the frontend or output them for copy-pasting.

- **Running Off-Chain Agents:** We provide an `agentStartup.js` or documentation in the README on how to run each agent. For example, each agent script can be run with Node (or `npm run agent:match`, `npm run agent:compliance`, etc., using package.json scripts). During local development, one might run a local Ethereum (if needed for zkSync, though zkSync L2 can run without a local L1 if using testnet) and then run the agents to interact with the deployed contracts. The scaffold includes basic logic for agents (as discussed earlier) – developers can expand these as needed.

- **Testing:** The `test/` directory contains unit tests for the smart contracts using Hardhat's test framework (Mocha/Chai). For MVP, we write tests for crucial flows:

- Escrow: ensure deposit and release functions work and money goes to the chef.
- ShiftManager: ensure posting, applying, accepting, completing flows change state and emit events as expected. We simulate the full lifecycle in a test: a restaurant posts a shift, a chef applies, the restaurant accepts (and escrow is funded), the chef marks complete, the restaurant confirms (releasing escrow), and ratings are submitted. These tests use Hardhat's zkSync network integration to run on an in-memory zkEVM or actual testnet.
- Reputation: test that ratings accumulate correctly.

Running `npx hardhat test` executes these tests. We ensure that tests are written in a way that is compatible with zkSync's testing tools (the Matter Labs Hardhat plugins support running tests on a local zkEVM environment or against testnet with a proper configuration).

- **zkSync Specific Considerations:** Deploying to zkSync Era requires an Ethereum L1 wallet for the payer. Since zkSync is Layer 2, deployment and transactions still use an Ethereum-like account. We follow best practices such as funding our test account with ETH on zkSync testnet (e.g., using the faucet) before deploying. The Hardhat deployment prints the **L1 → L2 deposit** info if

necessary (the plugin often handles bridging for deployment behind the scenes). For verifying contracts, zkSync's explorer has a verification endpoint configured in the Hardhat config (as seen in the config snippet above, `verifyURL`). The scaffold can include a `package.json` script like `"verify": "hardhat verify --network ZKsyncEraTestnet <contractAddress> <constructorArgs>"` for manual verification if needed.

- **Bootstrapping Data:** To test the system end-to-end, we might include a script or instructions to create some sample data. For instance, a script to post a few sample shifts (calling the contract via Hardhat or ethers), and maybe simulate a chef application. However, because we have a frontend, one can also use the UI to do this after deployment. The README guides the developer to:

- Deploy contracts on testnet.
- Run `npm run dev` in the `frontend` to start the Next.js app.
- Connect two different wallets in two browser windows (one as a restaurant, one as a chef) on zkSync testnet.
- Use the UI to post a shift (restaurant) and apply (chef), etc., verifying that the whole loop works.

The deployment and testing setup is geared towards making it easy to develop on this stack even if one is not deeply familiar with zkSync. By using widely adopted tools (Hardhat, Next.js, Ethers.js), developers can leverage their Ethereum knowledge, while the scaffold's configuration handles the zkSync details.

MVP Workflow Example

Finally, to illustrate how all the components come together, here's a step-by-step walkthrough of the **MVP escrow flow** in ChefsPlan:

1. **Restaurant Posts a Shift:** A restaurant user (with a zkSync-compatible wallet) navigates to the portal and fills out a form for a new gig (e.g., "Dinner service on Sep 30, need a sushi chef, 4 hours, $200"). When they submit, the dApp uploads the shift details to IPFS and obtains a CID. Next, the dApp calls the `postShift(detailsCID, payment)` function on the `ShiftManager` smart contract. The transaction (on zkSync) creates a new shift entry on-chain, emits a `ShiftPosted` event, and expects a payment amount (though actual payment is not yet transferred at this stage in our design, it could also be designed to allow optional pre-funding). The new shift is marked as open for applications. The IPFS CID is stored with the shift so others can read the full description.

2. **Matching Agent Suggests Chefs:** Upon the `ShiftPosted` event, the off-chain **Matching Agent** wakes up (triggered by the event). It logs the new shift and could query its database for matching chefs (e.g., those with "sushi" in their skill set and available that date). Suppose it finds a few; in an advanced setup, it might automatically notify those chefs off-chain (via email or push notification), or even on-chain if there was a mechanism. In the MVP, this agent's action is minimal (just logging or printing suggestions), but it lays the groundwork for automated matching.

3. **Chef Applies to Shift:** One of the chefs sees the new shift (either via the front-end browsing or via a notification). The chef goes to the dApp's shift listings (the front page or their dashboard) and reads the details (fetched from IPFS). If interested, they click "Apply". The dApp prompts a transaction to call `applyToShift(shiftId)` on the `ShiftManager` contract. On-chain, this records the chef's application (could emit `ShiftApplied` event and possibly store the

applicant's address in a list of applications for that shift). The shift remains open but now has at least one candidate.

4. **Compliance Check:** The **Compliance Agent** catches the `ShiftApplied` event. It checks the applying chef's address against known verification records. In our MVP, maybe it simply checks a hardcoded list or a JSON file of "verified chefs". If the chef is not verified, the agent could flag this (e.g., print a message or even call a contract function to mark the application as pending verification). Let's assume the chef is verified (has priorly uploaded credentials to IPFS and the agent had marked them verified in some registry). The compliance agent logs that the chef is verified for the shift. (If not, the restaurant might later see a warning in the UI that the chef's credentials are unverified, which could influence their decision).

5. **Restaurant Accepts an Application:** The restaurant, seeing the list of applicants (perhaps just one in this case), decides to hire this chef. In the portal UI, they click "Accept" next to the chef's application. This triggers a transaction calling `acceptApplication(shiftId, chefAddress)` on `ShiftManager`. The contract logic likely does two things: (a) mark the shift as no longer open (assign the chef to it, emit `ShiftAccepted` event), and (b) require the **escrow deposit**. The `acceptApplication` function can be designed as payable or it might internally call the Escrow contract. In our design, the `ShiftManager` might have required the payment amount when posting, but to keep funds secure, it's better to collect payment at acceptance. So the `acceptApplication` function could be `payable` and immediately forward the `msg.value` to `Escrow.deposit(shiftId)`. Alternatively, it could enforce that the restaurant has approved a certain amount of tokens to Escrow if using ERC-20. For simplicity, assume it's payable in ETH. The restaurant's wallet will prompt to pay the shift amount (e.g., 0.1 ETH for $200 equivalent on testnet) plus gas. Once this transaction is confirmed, the Escrow contract now holds the funds for the shift under that `shiftId`, and the shift state is updated. The `ShiftAccepted` event carries the chosen chef's address.

6. **Payout Agent Starts Monitoring:** The **Payout Agent** sees the `ShiftAccepted` event and logs that shift X is funded and in progress. It might start a timer or simply note that it should watch for completion or any unusual inactivity.

7. **Chef Completes the Shift:** On the day of the gig, after finishing the work, the chef uses the app to mark the shift as completed. This triggers `markComplete(shiftId)` on the contract. The contract likely just records a flag (or emits `ShiftCompleted`) but does not release funds yet (to allow the restaurant to verify the work). The `ShiftCompleted` event is emitted.

8. **Restaurant Confirms Completion:** The restaurant, seeing that the chef marked the job done (perhaps they get a notification or they check the app), reviews and then clicks "Confirm Complete" if all is well. This calls `confirmComplete(shiftId)` on `ShiftManager`. The contract will then call `Escrow.release(shiftId)` internally (since the ShiftManager is authorized to do so), which transfers the escrowed funds to the chef's address. A `PaymentReleased` event might be emitted by Escrow. At this point, the chef has been paid out from the escrow in a trustless manner. The Payout Agent also sees the `PaymentReleased` and logs that payment went through successfully. (If the restaurant did nothing, the Payout Agent could, per design, step in after a timeout to call `release` itself, but let's assume cooperation in MVP.)

9. **Reputation Update:** After completion, both parties are prompted to rate each other. The chef, in their app, can give the restaurant (employer) a rating out of 5. The restaurant can rate the chef. These actions call `Reputation.submitRating(target, score)` on-chain. The Reputation contract updates the scores accordingly. Over time, these ratings accumulate to form each user's reputation score, visible in the app (for example, next to a chef's name it might show "★ 4.8 (5 jobs)" indicating average 4.8/5).

10. **(Optional) Dispute Scenario:** In the happy path above, no disputes occurred. If there was a problem (say the restaurant claims the chef left early, or the chef claims the restaurant's work conditions were unsafe), either party could initiate a dispute. In the scaffold, we have a `disputeShift(shiftId)` in the ShiftManager which emits `ShiftDisputed`. The contracts themselves do not resolve the dispute (that's complex and often involves arbitration). But the presence of a dispute could prevent `release` from happening automatically. The funds remain in Escrow. Off-chain, an arbitrator or a mutually trusted party would resolve the issue. The resolution could be enacted by a privileged contract method or by mutual agreement (e.g., both parties calling a function to split the funds or refund). The MVP might not implement full dispute resolution logic, but it leaves room for it. The **AI Copilot Agent** or a future "dispute agent" could flag disputes and even assist in resolution (not in MVP scope, but an extension).

11. **System Extensibility:** Throughout this flow, the design remains decentralized. The smart contracts enforce the core rules (escrow and basic workflow). Off-chain agents augment the system without compromising trustlessness (they mainly observe events and provide services; they don't hold custodial power except what the contracts allow, like an optional auto-release). IPFS ensures that even if ChefsPlan's web servers go down, the data about shifts and profiles is still available via the IPFS network (as long as it's pinned). The use of zkSync L2 provides low fees and fast finality, which is crucial for a gig-work platform where micro-transactions need to be efficient.

Each component in this scaffold can be developed and improved in isolation. For instance, one could upgrade the **Matching Agent** to integrate with a more sophisticated AI recommendation system, or implement a zk-SNARK-based credential verification for the **Compliance Agent** (so chefs can prove they have a certificate without revealing identity). The smart contracts can also be upgraded (thanks to proxies) to add features like partial payments, deposits, or slashing for no-shows. The modular architecture ensures ChefsPlan can evolve while adhering to decentralized principles laid out in the whitepaper.

Sources:

- zkSync Hardhat Upgradable Plugin – supports Transparent and UUPS proxies, integrating OpenZeppelin patterns [2].
- zkSync Era Hardhat config – example shows testnet setup with `zksync: true` and custom RPC [1].
- Web3.Storage IPFS upload – using `client.put(files)` returns a CID for stored content [3].
- zkSync Web3Provider – allows MetaMask/WalletConnect to interact with zkSync in the browser [6].

---

[1] [7] Getting started - ZKsync Docs
https://docs.zksync.io/zksync-network/tooling/hardhat/guides/getting-started

2 hardhat-zksync-upgradable - ZKsync Docs
https://docs.zksync.io/zksync-network/tooling/hardhat/plugins/hardhat-zksync-upgradable

3 4 5 Web3 Storage - The simple file storage service for IPFS & Filecoin.
https://old.web3.storage/docs/how-tos/store/

6 Web3Provider - ZKsync Docs
https://docs.zksync.io/zksync-network/sdk/js/ethers/api/v5/providers/web3provider