ChatGPT

ChefsPlan 5.0 – Development Roadmap & Checklist

Below is a structured development roadmap for **ChefsPlan 5.0**, a decentralized gig-work platform connecting chefs and restaurants. The roadmap is organized by key milestones/phases, and each section includes checklists of development tasks, team roles involved, and prerequisites or dependencies for that stage. This plan is based on the system architecture and phased approach described in the project's whitepaper and scaffold overview.

MVP Delivery – Core Platform Foundation (Phase 1)

In **Phase 1 (MVP)**, the focus is on implementing the core platform functionality and proving the concept in a real setting. This includes deploying essential smart contracts on the zkSync Layer-2 network for shift postings, applications, acceptances, and escrowed payments, along with a basic user-facing dApp and minimal off-chain automation [1]. Off-chain agents (e.g. Matching and Payout) run in a rudimentary form to automate notifications and payouts [2]. To expedite delivery, some features remain simplified at this stage – for example, compliance/KYC checks might be handled manually by an admin, and the reputation system can start as a simple off-chain rating stored in a database and periodically anchored on-chain [3]. By the end of Phase 1, ChefsPlan should be a working decentralized marketplace on zkSync with real escrowed funds, albeit with certain trust assumptions and basic features for a pilot user group.

Key Development Tasks:
- [ ] **Implement core smart contracts** – Develop and unit-test the **ShiftManager**, **Escrow**, and **Reputation** contracts (using Solidity) to handle shift lifecycle, payment escrow, and reputation tracking [4]. Follow upgradeable proxy patterns for future flexibility.
- [ ] **Deploy contracts on zkSync testnet** – Use Hardhat (with zkSync plugins) to compile and deploy the contracts to a zkSync Era test network for initial testing. Verify basic operations like posting a shift, applying, and releasing escrow.
- [ ] **Build the basic dApp (web frontend)** – Create a simple **Next.js** web application for chefs and restaurants. Implement pages for browsing available shifts, posting a new shift, viewing applications, and managing agreements. Integrate a Web3 wallet (e.g. MetaMask) for user authentication and transaction signing on zkSync.
- [ ] **Integrate wallet & accounts** – Ensure the dApp connects to zkSync (L2) via web3 providers. Enable wallet connection and switching to zkSync network in the UI (use zkSync's Web3Provider to wrap MetaMask for L2 transactions [5] [6]).
- [ ] **Implement off-chain agents** – Develop basic versions of the **Matching Agent** and **Payout Agent** as off-chain processes (Node.js or Python) that listen to contract events and perform automation [7]. For MVP, the Matching agent can simply log or notify when a new shift is posted (no complex AI matching yet), and the Payout agent can automatically release funds or send reminders when a shift is completed.
- [ ] **Basic testing & pilot run** – Conduct end-to-end testing of the MVP: simulate a restaurant posting a shift, a chef applying, acceptance of the application, and escrow payout upon completion. Fix any critical bugs. Prepare a small pilot with real users (e.g. a few restaurants/chefs) to gather feedback on the core workflow.

Team Roles Involved:
- *Blockchain Smart Contract Developer* – Implements and tests the Solidity contracts (ShiftManager, Escrow, Reputation) and ensures they follow security best practices.
- *Front-End (Web3) Developer* – Builds the Next.js dApp, integrates wallet connectivity, and ensures a smooth UX for posting shifts and managing applications.
- *Off-Chain Agent Developer* – Programs the initial automation agents (Matching, Payout) and sets up event listeners using ethers/zkSync provider. May be the same person as backend/blockchain dev in a small team.
- *Project Lead/Architect* – Coordinates the integration of smart contracts, backend, and frontend components; defines the overall module interfaces according to the whitepaper architecture.
- *QA/Test Engineer* – Creates test scenarios for the end-to-end flow (including edge cases like disputes) and verifies that escrow and application processes work on zkSync as expected.

Prerequisites & Dependencies:
- **Development Tooling:** Hardhat configured with zkSync plugins (e.g. `@matterlabs/hardhat-zksync-solc` and deploy plugins) for zkEVM-compatible compilation and deployment [8]. Node.js environment with project dependencies (Solidity compiler, OpenZeppelin upgradeable libraries, etc.) installed.
- **zkSync Network Access:** Access to a zkSync Era testnet endpoint (e.g. an RPC URL) with the appropriate network config (including an L1 link for bridging, and `zksync: true` flag in Hardhat config) [9]. Testnet ETH (on zkSync) in developer wallets for paying transaction fees.
- **Web3 Wallets:** MetaMask or similar wallet set up with the zkSync testnet chain (chain ID, RPC URL) for local testing of the dApp. Ensure team members have test accounts (chef and restaurant roles) with keys configured in the dApp for simulation.
- **IPFS/Storage Setup:** (For MVP, if including off-chain storage) Access to an IPFS node or pinning service to store shift descriptions and any uploaded files. This can be an Infura or Pinata account for convenience during development.
- **Contracts and Libraries:** OpenZeppelin upgradeable contracts library installed for proxy patterns; ensure contract addresses (once deployed) are recorded for the frontend and agents.
- **Environment Variables:** Securely store sensitive keys (private key for deployments, Infura API keys, etc.) in a .env file or the CI/CD pipeline for deployment. Configure the frontend with the correct contract addresses and network IDs.

zkSync Compatibility & Deployment Pipeline (Phase 1)

ChefsPlan 5.0 is built on **zkSync Era**, so establishing compatibility and a robust deployment pipeline is critical from the start. This involves tailoring the development and DevOps process to zkSync's environment. The smart contracts must be compiled with zkSync's VM in mind, and deployment scripts should target the zkSync testnet (and later mainnet) network configurations [10]. The pipeline will also handle migrating from testnet to mainnet, including any bridging of ETH for gas and verifying contract deployments. By setting up continuous integration early, the team can deploy updates quickly and ensure the MVP is consistently synced with zkSync's latest tooling.

Key Development Tasks:
- [ ] **Configure zkSync toolchain** – Update Hardhat configuration for zkSync Era compatibility. This includes using Matter Labs' zkSync compiler plugin and enabling the `zksync` flag in network configs [10]. Install the zkSync upgradable plugin to deploy proxy contracts on zkSync (supports Transparent and UUPS proxies) [11].
- [ ] **Write deployment scripts** – Create automated scripts (e.g. `scripts/deploy.js`) to deploy the ShiftManager, Escrow, and Reputation contracts to the zkSync testnet [12]. The script should handle

deploying logic contracts and initializing proxy contracts correctly. Include logic to output or save the deployed addresses for use in the frontend and agents.
- [ ] **CI/CD pipeline setup** – Set up a continuous integration pipeline (using GitHub Actions or similar) to run tests and deploy contracts to testnet on new releases. This should pull secrets (private keys, RPC URLs) from a secure store and execute the Hardhat deployment. Include a review step before promoting deployments to the zkSync mainnet.
- [ ] **zkSync testnet testing** – Deploy the MVP contracts to a public zkSync **testnet** environment and run integration tests. Ensure that contract interactions (transactions) are finalizing and that the L2 behaves as expected. Test edge cases like re-orgs or bridging if relevant (zkSync being an L2 requires considering L1 finality for certain operations).
- [ ] **Deployment on mainnet** – Once the MVP is stable on testnet, prepare for zkSync **mainnet** deployment. This involves bridging the required ETH to zkSync for gas fees, updating configurations to the mainnet RPC, and double-checking all addresses and initial parameters (like proxy admin address). Plan a mainnet deployment window and a rollback strategy if issues occur.
- [ ] **Contract verification & monitoring** – Verify the deployed contracts on the zkSync block explorer (if available) or publish the ABI/source for transparency. Set up monitoring for the contracts (e.g. listen to critical events or use a service to track contract errors) so the team is alerted to any on-chain issues post-deployment.

Team Roles Involved:
- *DevOps Engineer* – Designs and maintains the CI/CD pipeline for deploying contracts and the application. Ensures secure handling of keys and smooth promotion from testnet to mainnet.
- *Blockchain Developer* – Configures the Hardhat environment for zkSync, writes the deployment and migration scripts, and handles network-specific issues (like gas token differences on zkSync). Also responsible for verifying that contracts behave identically on zkSync as they would on Ethereum.
- *QA Engineer* – Focuses on deployment testing, ensuring that after deployment on testnet, the system works end-to-end. They might create automated test suites that run on a local zkSync node or testnet to validate each deployment.
- *Project Manager* – Schedules and coordinates the go-live on mainnet, ensuring that prerequisites (like funding the deployer account with ETH on L2) are done and that all stakeholders sign off on the deployment.

Prerequisites & Dependencies:
- **zkSync SDKs and Plugins:** The development environment must include Matter Labs' Hardhat plugins for zkSync (for compilation and deployment) and any necessary zkSync libraries (e.g. `zksync-web3` for interacting with contracts). Confirm compatibility of Solidity version and features with zkSync's compiler.
- **Network Configuration:** Access to zkSync Era Testnet RPC (e.g. `https://testnet.era.zksync.dev`) and a corresponding Ethereum testnet (Goerli/Sepolia) RPC for L1 bridging context [9]. These should be configured in Hardhat and environment files. For mainnet, obtain the zkSync mainnet RPC and ensure the team has L1 mainnet access if needed for bridging.
- **Private Keys and Accounts:** A deployment Ethereum account (EOA) with its private key configured for Hardhat, funded on the zkSync testnet (and later mainnet) with enough ETH to cover gas. Typically, this means having some ETH on L1 and then depositing to zkSync. Acquire testnet ETH from faucets and ensure the deployer address is funded on L2.
- **Continuous Integration Tools:** Set up GitHub repository secrets for the private key and RPC URLs. Ensure the CI runner has Node.js and can install the zkSync-specific packages. Verify that the pipeline can fail safely (e.g., halts deployment if tests fail).
- **Proxy/Upgrade Infrastructure:** The OpenZeppelin Upgrades library (via the zkSync Hardhat plugin) must be configured. Determine proxy admin ownership (the address that will own the proxies and can upgrade logic contracts). This might initially be a dev multisig or an EOA; note that later in governance phase this might transfer to the DAO.

- **Documentation:** Document the deployment steps and configurations. This includes a runbook for deploying to new networks (for example, instructions to deploy to another L2 or how to initialize a new proxy admin) so that the process is repeatable and team members can assist or take over if needed.

Compliance Agent Integration (Simulated Verification, Phase 1)

As part of the MVP, ChefsPlan introduces a **Compliance Agent** to handle participant requirements (e.g. work eligibility, certifications) in a *simulated* manner. At this early stage, the compliance checks are not fully decentralized or automated via zero-knowledge proofs – instead, the system uses a placeholder/off-chain process to verify that chefs meet certain criteria. The Compliance Agent runs off-chain, listening for events (such as a chef applying to a shift) and then referencing an external record or admin input to decide if the chef is "verified" or not [13]. In practice, this could mean the agent checks a JSON file or database of pre-verified users, and logs a warning or updates a flag if an unverified chef tries to take a gig [14]. This simulation lays the groundwork for Phase 2, where real KYC/credential verification will be done via zero-knowledge proofs – for now, it ensures the compliance logic is in place without blocking the MVP launch.

Key Development Tasks:
- [ ] **Develop Compliance Agent script** – Create an off-chain service (Node.js script or Python daemon) called **Compliance Agent** that connects to the zkSync network (using a Web3 provider) and subscribes to relevant events (e.g. `ShiftManager.ShiftApplied` when a chef applies for a shift) [13]. Implement callback logic that triggers on these events.
- [ ] **Simulate credential checks** – Within the agent, implement a placeholder check for chef credentials. This could be a function like `checkChefCredentials(chefAddress)` that looks up the chef's status in an off-chain data source [15]. For MVP, this might simply reference a hardcoded list or a JSON file of "verified chefs" (simulating that those chefs have provided required documents offline).
- [ ] **Handle verification outcomes** – Define the agent's actions based on the check above. If a chef is not verified for a certain requirement, the agent might log a warning or call a specific smart contract function to record this. For example, it could call `ShiftManager.markChefUnverified(shiftId, chefAddr)` or a similar function (if implemented) to mark that the application needs attention [16]. If the chef is verified, the agent can log approval or take no action, allowing the process to continue normally.
- [ ] **Integrate with on-chain data (if applicable)** – If the smart contracts include any compliance-related hooks (e.g. a field in ShiftManager or a separate `ComplianceRegistry` contract that can store verification flags), integrate the agent to update those. In MVP, this might be minimal or even skipped (the agent could operate purely off-chain), but the scaffolding should allow later plugging in actual on-chain verification.
- [ ] **Testing the compliance flow** – Simulate scenarios to ensure the Compliance Agent behaves correctly. For example, attempt a shift application with a "verified" chef vs an "unverified" chef in the off-chain list and observe that the agent logs or flags the latter. Ensure that these conditions appropriately inform the team or the application (for instance, the restaurant might see an admin warning that a chef's credentials are pending verification).
- [ ] **Prepare for future integration** – Although the verification is simulated, design the agent's structure to be extensible. Cleanly separate the verification check (which is a stub now) so that in Phase 2 it can be replaced with calls to a real ZK verification system. Possibly include configuration for different credential types (ID, work permit, certifications) even if the logic just returns "true" for all in MVP. This foresight will ease the Phase 2 upgrade.

Team Roles Involved:
- *Backend/Agent Developer* – Implements the Compliance Agent service. Needs familiarity with web3

libraries and event listeners to interact with the zkSync network. Also simulates the business logic for verification criteria.
- *Smart Contract Developer* – (If needed) updates or writes any smart contract hooks for compliance (e.g., functions or events in ShiftManager related to verification status). In MVP this role is minor, but they ensure the contracts emit the events the agent listens to, and later will modify contracts to require proofs.
- *DevOps Engineer* – Sets up the runtime environment for the Compliance Agent. This could involve a server or cloud function to run the agent continuously, and ensuring it can securely connect to the blockchain (RPC endpoint, keys if needed). Also handles logging or alerting if the agent flags an issue (so the team can intervene in a pilot).
- *Compliance Analyst (advisor role)* – Though not heavily involved in coding at MVP stage, a domain expert can define what "being verified" means in context (e.g., which certifications are required for chefs). They provide the criteria that the simulated logic will check, ensuring the simulation aligns with real compliance needs.

Prerequisites & Dependencies:
- **Event Interface from Contracts:** The ShiftManager (or related) contract must emit events on key actions (e.g., a chef applying to a shift) that the Compliance Agent can subscribe to. According to the scaffold, events like `ShiftApplied` are available for this purpose [17]. Ensure the ABI and contract address are available to the agent.
- **Off-Chain Verification Data:** Prepare a data source to simulate verification. This could be as simple as a JSON file (mapping chef addresses to a boolean "verified" status) stored locally or on IPFS, or a small database of test users. The agent should know how to access this (for instance, load the JSON at startup, or fetch from an IPFS CID if simulating a decentralized storage of credentials).
- **Node Access & Credentials:** The agent needs access to a zkSync RPC endpoint (WebSocket or HTTP with polling) to listen for events. If using a WebSocket, ensure the provider supports subscriptions (e.g., an Alchemy/Infura ws endpoint for zkSync if available) [18]. If the agent will make transactions (e.g., calling a contract function to mark unverified), it needs a private key and account with enough zkSync ETH to send that transaction – set up an agent Ethereum account if needed for these calls.
- **Runtime Environment:** Decide where the Compliance Agent will run during development and MVP deployment. Options include running locally (manual launch when testing) or deploying it on a server/VM. For MVP/pilot, a simple approach is running it as a background process by the dev team. In production, it might be containerized and deployed. Ensure Node.js (if using Node) is installed in that environment and necessary packages (ethers, etc.) are available.
- **Logging & Monitoring:** Set up a way to observe the Compliance Agent's activity. During dev, console logs are sufficient. For a pilot, integrate a logger that writes to a file or dashboard. This will help in debugging if, for example, an application was blocked due to verification – the team can see the agent's logs to know why.

IPFS-Based Data Handling (Shift Details & Credentials, Phase 1)

To maintain decentralization and reduce on-chain storage, ChefsPlan uses **IPFS (InterPlanetary File System)** for data that is too large or sensitive to store on Ethereum. In the MVP, this means details like shift descriptions, chef profiles, or attached documents are stored off-chain in IPFS, and only their content hashes (CIDs) are referenced in smart contracts [19]. By doing so, the system keeps on-chain transactions lightweight while still ensuring data integrity (since the IPFS hash is tamper-evident). The ShiftManager contract, for instance, will record an IPFS CID for each shift's full description rather than the raw text [20]. Likewise, any credentials or agreements can be placed on IPFS and fetched by users or agents when needed. This phase involves setting up the IPFS integration and ensuring the dApp and agents can smoothly read/write to distributed storage.

Key Development Tasks:
- [ ] **Set up IPFS node or service** – Decide on the IPFS backend for the project. For development, this could be running a local IPFS node or using a third-party IPFS pinning service (like Pinata or Infura's IPFS API). Configure the project with the API endpoint and credentials if using a service. Make sure the IPFS node is accessible by the frontend (possibly via a gateway URL) for retrieving files.
- [ ] **Implement file upload & pinning** – In the dApp, implement logic for uploading data to IPFS. For example, when a restaurant posts a new shift, the form details (title, description, requirements, etc.) should be bundled into a JSON object or similar format. Upon submission, the frontend (or a cloud function) uploads this JSON to IPFS and obtains the content hash (CID). Ensure that the uploaded data is *pinned* so that it persists on IPFS (if using a pinning service, through their API).
- [ ] **Store content hashes on-chain** – Modify the ShiftManager contract's `postShift` function to accept an IPFS content hash (CID) for the shift details [20]. The contract will store this string (or a Keccak-256 digest of it if needed) as part of the Shift record. The escrow or matching processes then only deal with this reference. This task may involve choosing the data type for the CID (e.g., a `string` or `bytes` in Solidity) and ensuring it can handle the IPFS hash length.
- [ ] **Retrieve IPFS data in the dApp** – When displaying shift details in the frontend (e.g., chef browsing shifts), implement a fetch from IPFS. Using the stored CID from the contract, the frontend can query via an IPFS gateway or client library to get the JSON and then render the full description. Handle the case where data might not load (e.g. if IPFS is slow) by providing loading states or caching frequently accessed content.
- [ ] **Store user credentials/docs** – If the MVP involves any user-uploaded documents (like a chef's certification or an agreement PDF), integrate those with IPFS as well. For instance, a chef's profile might have a field for a "credential file CID" that points to an uploaded certificate image or PDF on IPFS. Although full ZK verification isn't live yet, storing these off-chain sets the stage for later proving knowledge of them. Update the Compliance Agent to know where to find a chef's credential file (e.g., perhaps the agent can read a CID from the chef's profile in the contract or a central list, then fetch that file from IPFS during verification checks).
- [ ] **Ensure data privacy where needed** – Consider if any data needs to be encrypted before going onto IPFS. Public shift details can likely be plain JSON, but personal documents might be sensitive. At MVP stage, it could be acceptable to store them unencrypted but access-controlled by only sharing CIDs with authorized parties. However, note in design that later phases might introduce encryption or ZK proofs, so structure the data storage with that future upgrade in mind (e.g., keep personal data separate so it could be replaced by a proof).

Team Roles Involved:
- *Full-Stack Developer* – Handles integration of IPFS on both frontend and backend. On the frontend, implements upload and retrieval logic (possibly using `ipfs-http-client` or an API); on the backend/DevOps side, sets up the IPFS node or configures the third-party service.
- *Smart Contract Developer* – Updates smart contracts to include IPFS hash references. Ensures that storing the hash is efficient (maybe using content identifiers directly) and that events or return values expose those hashes for the frontend/agents to consume. Also responsible for any hashing scheme if needed (for example, if storing a digest vs full CID).
- *DevOps Engineer* – If self-hosting IPFS, maintains the IPFS node. Ensures it's pinned to the required data and is reachable. If using a service, they manage API keys and monitor pinning status. Also sets up backup strategies (IPFS is decentralized but pinning ensures availability – the team might want multiple pins or a cluster for important data).
- *Security/Privacy Expert (consulting role)* – Advises on what data can safely be public on IPFS vs. what might need encryption or access control. At MVP, this might be minimal involvement, but decisions here affect user privacy, so having guidelines (like "okay to store shift descriptions publicly, but personal IDs should be encrypted or redacted") is useful.

Prerequisites & Dependencies:
- **IPFS Access:** An operational IPFS node or account with a pinning service. For local testing, running `go-ipfs` daemon or using the JS IPFS can work. For a persistent environment, get API credentials for a service like Pinata. Also, a chosen method for the frontend to fetch data (e.g., using a public gateway like `ipfs.io` or a dedicated gateway URL provided by the pinning service).
- **Data Format Definition:** Define how shift details and other data will be structured off-chain. For example, a JSON schema for a "Shift" object (with fields like title, description, location, etc.), and maybe another for "Profile" or credentials. Standardize this so that both producers (the form that uploads) and consumers (the UI that downloads and shows it) agree on format.
- **Content Identifier Handling:** Ensure the system can handle IPFS CIDs correctly. If using CIDv1 (base58 or base32 strings), confirm the smart contract storage and events are accommodating those (string length ~46 characters for base58). If needed, decide on storing a multihash vs the entire URI. Also include support in the web app to convert or properly display truncated versions if needed.
- **Bandwidth and Caching:** Accessing IPFS can introduce latency. For a smoother UX, set up caching for frequently accessed data. This could be done at the application layer (storing fetched JSON in memory or localStorage) or using a service that caches gateway responses. Ensure the development environment has sufficient bandwidth to fetch IPFS content, and consider using a closer gateway for performance.
- **Fallback Plan:** In case IPFS is temporarily unreachable, decide what happens – e.g., the dApp could show "Content temporarily unavailable, please retry." For MVP, having a single source is acceptable, but document these cases as things to improve (like multi-gateway queries or having the data also on a backup server for emergency).

Zero-Knowledge Credential Verification & Selective Disclosure (Phase 2)

In **Phase 2**, ChefsPlan enhances privacy and trust by integrating **zero-knowledge proofs (ZKPs)** for user credentials and introducing selective disclosure techniques. The goal is to allow chefs and restaurants to prove they meet certain requirements (identity, certifications, etc.) *without revealing sensitive personal data*. This phase likely involves partnering with a decentralized identity solution or framework (e.g. integrating with Polygon ID, Sismo, or similar ZK identity providers) to issue verifiable credentials [21]. Users will obtain ZK-based credentials (for example, a proof of having a work permit or food safety certification) and present proofs to the platform. The smart contracts and/or the Compliance Agent are updated to verify these proofs on-chain, or to accept attestations from trusted ZK attesters, instead of relying on manual or off-chain checks [22]. Additionally, the platform can implement *selective disclosure*: users could prove statements like "I am certified to level X" or "I have a rating above 4 stars" without exposing all underlying data [23]. By the end of Phase 2, all participants would be cryptographically verified, eliminating the need for ChefsPlan (or any central admin) to store personal info, thus greatly improving privacy and compliance adherence.

Key Development Tasks:
- [ ] **Integrate a ZK identity solution** – Research and choose a ZK credential system (e.g., Polygon ID, Sismo, or custom circom circuits). Implement the integration such that chefs and restaurants can obtain verifiable credentials. For instance, a chef might use a mobile app or web interface to generate a proof of identity/qualification issued by a trusted issuer. ChefsPlan's platform needs to accept these proofs. This could mean deploying a verifier smart contract (if using custom proofs) or using an existing protocol's verifier. Update the Compliance Agent and smart contracts to require a valid proof for certain actions (e.g., applying to a shift might now require a ZK proof of certification).
- [ ] **Design ZK circuits or use templates** – If a custom approach is required, design zk-SNARK circuits that encode the verification logic (e.g., "user's government ID is valid and indicates they are over 18").

Alternatively, use template circuits from an identity platform. Ensure circuits are efficient (proving time is reasonable) and secure. If using something like Polygon ID, configure the appropriate credential schema (e.g., a credential that says "Chef is registered with Dutch Chamber of Commerce" or "Chef has completed food safety training Level 2") and how it will be verified.
- [ ] **Update smart contracts for verification** – Modify or add smart contract methods to handle on-chain proof verification. For example, a new `verifyCredential(bytes proof, bytes32 merkleRoot)` function that can be called to validate a proof against a known verifier contract or a state root. The Compliance Agent could invoke this on-chain verification when a new user registers or applies to a shift. If using an attestation approach, the contract might accept a signed attestation from a trusted issuer instead (which could be simpler but semi-centralized). In either case, the contract system must record that a given user is verified without exposing *why* (no personal data on-chain, just a flag or credential hash).
- [ ] **Implement selective disclosure for reputation** – Extend the reputation system to utilize ZK proofs for privacy. For instance, allow a chef to prove **qualitatively** that their average rating is above a certain threshold without revealing the exact number. This could involve issuing a ZK credential for reputation (e.g., the platform or a oracle-like agent could issue a credential "Reputation > 4.0" to a chef, which the chef can then prove). Alternatively, design the Reputation contract to support zero-knowledge queries (this is complex and might use cryptographic accumulators or range proofs). As a first step, focus on one or two practical selective disclosures (like rating threshold or number of completed shifts) and implement a prototype proof for those. [23]
- [ ] **Migrate data to decentralized storage** – If any personal data was temporarily kept in a centralized database during MVP (for example, copies of IDs or certificates), Phase 2 is the time to remove that. Replace those with either IPFS-stored encrypted files plus ZK proofs of their contents, or with purely on-chain verifiable credentials. Essentially, ensure the platform no longer requires storing raw personal info on its servers [24]. This task includes writing migration scripts or procedures for existing user data: e.g., reach out to pilot users to provide new ZK credentials or re-verify through the new system.
- [ ] **Security audits and testing** – Introduce comprehensive testing for all new ZK features. This means unit tests for the proof verification logic (if custom circuits, include tests with valid/invalid proofs), integration tests where a user goes through the whole flow (obtaining credential, submitting a proof to smart contract, doing an action like applying to a shift successfully). Engage an external auditor to review the smart contracts, especially the new verification logic and any new contracts deployed for ZK credentials [25]. Also audit the zero-knowledge circuits themselves if custom – ensure there are no vulnerabilities (zero-knowledge code should be checked for soundness, no trivial bypass, etc.). Deploy the updated system to a testnet environment (could be zkSync testnet or even a local zk-EVM if easier) and conduct a testnet beta with a larger user group to get feedback on the ZK onboarding flow.

Team Roles Involved:
- *Zero-Knowledge (ZK) Engineer* – Specializes in zk-SNARK/STARK technologies. This person designs or integrates the credential verification circuits and ensures the proofs are correctly verified on-chain. They might write circuits in Circom, ZoKrates, or use off-the-shelf solutions from an identity protocol.
- *Smart Contract Developer* – Modifies the Ethereum contracts to include verification logic, possibly deploys new contracts (like a Verifier contract or an Identity registry). Works closely with the ZK engineer to make sure the on-chain and off-chain parts connect (public inputs match, etc.). Also upgrades the Reputation contract if needed for on-chain reputation handling with privacy.
- *Frontend/UX Developer* – Updates the user interface to incorporate the new verification steps. For example, adding a **verification flow** where a user connects an identity wallet or scans a QR code to generate a proof. Ensures the process is user-friendly (ZK can be complex, so abstract it with clear prompts like "Click here to verify your credentials"). Possibly integrates with wallet plugins or SDKs provided by identity solutions (e.g., an SDK to request a Polygon ID proof from the user).
- *DevOps/Backend* – If the ZK solution requires backend components (like an issuance server or a database for credentials), the DevOps or backend engineer sets those up. For instance, some systems

require running a service that issues credentials to verified users – the team might run an issuer node that grants a credential after manually checking someone's ID (in early phases). Also responsible for ensuring the proving key and verification key (if applicable) are managed securely and deployed correctly.
- *Compliance Officer / Legal Advisor* – Although much is technical, having a compliance expert is important. They ensure that whatever KYC/credential is being proven actually satisfies legal requirements (e.g., if Dutch law requires checking a chef's ID, the advisor confirms that using a ZK proof of ID is acceptable). They might also help select which identity platform has credibility (or help establish ChefsPlan as an issuer of work credentials).
- *Security Auditor (external)* – Not a daily team role, but at this phase the project engages security auditors to evaluate the new cryptographic implementations. They will look at smart contracts and possibly the ZK circuits or the integration with identity providers.

Prerequisites & Dependencies:
- **ZK Infrastructure:** Acquire or set up the necessary infrastructure for zero-knowledge proofs. If using a known solution like Polygon ID, get API keys or set up an issuer organization within that system. If building custom, generate proving and verifying keys for the circuits (which can be time-consuming if circuits are large – plan for the ceremony or key generation step). Ensure the team has access to libraries (e.g., snarkJS, circom runtime, or identity SDKs) needed to generate and verify proofs.
- **Trusted Issuers:** Identify who or what will issue the initial credentials. For example, if verifying identities, perhaps integrate with an existing KYC provider that can issue a ZK attestation that "passport verified" for a user. If it's going to be done in-house initially (for pilot, the team might act as the issuer of "verified chef" credentials), set up a secure process for that. This might involve a simple web form where an admin checks a document and then triggers issuing a credential to the user's wallet.
- **User Wallet Compatibility:** Ensure that user wallets can handle the ZK credentials. Some solutions have their own apps (e.g., a separate mobile app for holding credentials). Plan how chefs and restaurants will store and present credentials – for instance, they might need to install a specific wallet or app. Provide instructions or integrate support into the ChefsPlan dApp (maybe via WalletConnect if a mobile app is used).
- **Performance Considerations:** Be aware of the performance impacts – generating a proof on a mobile device can be intensive. Choose ZK algorithms that are mobile-friendly (or allow server-side proving if that doesn't compromise trust too much). Also ensure the on-chain verification doesn't use excessive gas; zkSync might have cheaper costs, but verifying a proof still has a cost. Optimize circuits to keep proof verification within reasonable gas limits or use recursion if needed to combine multiple proofs efficiently.
- **Regulatory Check:** Before fully rolling out, do a check that using ZK proofs meets compliance requirements. For example, if audited, can the platform demonstrate compliance? (Perhaps by showing the list of attestation issuers and the fact that all users have valid attestations, without revealing user identities.) Having a strategy to reassure regulators (maybe maintaining an off-chain log of verifications done, signed by an issuer) could be a prerequisite in highly regulated contexts.
- **Migration Plan:** A plan to transition existing users to the new system. If Phase 1 had 100 users who were manually KYCed, Phase 2 might require them to re-verify via the new method. Prepare communications and possibly incentives (like "verify your profile with our new privacy-preserving system and earn a badge or token reward") to encourage smooth adoption.

Reputation System Enhancements (Phase 2+)

The **Reputation system** introduced in the MVP (basic star ratings) will be significantly enhanced in Phase 2 and beyond. The aim is to make reputation more robust, fair, and privacy-preserving. In earlier stages, the reputation may have been as simple as an average star rating stored off-chain or in a basic contract. Now, we plan to move it fully on-chain and improve its fidelity. Only authorized actions should affect reputation (e.g., only participants of a completed shift can rate each other, to prevent arbitrary ratings). We may introduce weighted scoring (for example, a rating from a high-stakes gig might count more than a small gig) or other advanced logic [26] . Additionally, to align with the privacy goals, we can integrate zero-knowledge elements: users might prove aspects of their reputation (like being above a threshold) without revealing exact details, as mentioned in Phase 2 with selective disclosure [27] . By upgrading the reputation system, ChefsPlan can better incentivize good behavior and provide reliable signals to users, all while protecting individual data.

Key Development Tasks:
- [ ] **Upgrade on-chain reputation contract** – Refactor the **Reputation.sol** contract to support more complex functionality. If it was initially basic (just storing totals and counts [28]), consider expanding it to maintain additional metrics: e.g., number of gigs completed by the user, cumulative earnings (to weight feedback by experience), or separate ratings for different categories (professionalism, skill, etc.). Implement restrictions so that `submitRating` can only be called by parties of a completed shift (perhaps the ShiftManager contract now calls Reputation.sol when a shift is finalized to record mutual ratings). Introduce an **authorization check** in the contract (or rely on ShiftManager) to prevent any invalid inputs [29] . Ensure the contract remains upgradeable for future tweaks.
- [ ] **Integrate ZK-proof for reputation (if applicable)** – Building on Phase 2's ZK infrastructure, allow certain reputation info to be proved in zero-knowledge. For example, after each gig, instead of directly storing a numeric rating on-chain, the platform could issue a credential to the chef like "Chef X has a 5-star review from gig Y" which could later be aggregated off-chain and proved. However, this approach can be complex; a simpler approach is to keep storing ratings on-chain but use cryptographic techniques to hide or abstract them when needed. A concrete task could be: Implement a function or off-chain service for *selective disclosure of reputation*: e.g., generate a proof that user's average > 4 without revealing all underlying ratings. This likely involves heavy cryptography (like a ZK circuit that takes all ratings as inputs); it might be more realistic to plan this for beyond Phase 2, but Phase 2 can lay the groundwork (like storing data in a way that is ZK-friendly later).
- [ ] **Reputation as SBT (Soulbound Token)** – Consider minting a **Soulbound Token** that encapsulates a user's reputation. An SBT (a non-transferable NFT) could represent a chef's reputation score or level. For instance, levels like Bronze/Silver/Gold chef based on number of successful gigs and average rating. These tokens can be updated (or reissued) as the reputation changes. The advantage is that it's easily shareable across platforms (and could be used in multi-chain scenarios later) and it's inherently tied to the user's identity. Task: design the SBT contract (or use an existing standard), and have the Reputation system mint/update SBTs when thresholds are reached. This must be done carefully to avoid spam or frequent updates – maybe only significant changes trigger an update.
- [ ] **Front-end and UX improvements** – Enhance how reputation is presented to users and how they can use it. For example, display more detailed profiles: show total gigs completed, badges for top ratings, etc. Also implement any features for selective disclosure in the UI – e.g., a chef can click "Prove my rating" to get a sharable proof or to satisfy a requirement for certain high-end shifts that require a minimum rating. Additionally, add mechanisms to prevent abuse: perhaps integrate a **dispute resolution** for unfair ratings (this might tie into the DAO phase later, but start considering it).
- [ ] **Backfill and migration** – If the reputation system logic changes, plan to migrate or recompute existing reputation data. For instance, if we move from off-chain to on-chain, import the existing ratings from the pilot database into the new contract (maybe via a one-time script that the contract owner can call to set initial values). If introducing SBTs, mint tokens for users based on their current scores. Ensure no data is lost in transition. Also, communicate to users if there are changes (e.g., "We've updated how ratings are calculated to better reflect experience").
- [ ] **Testing and validation** – Test the enhanced reputation flows thoroughly. This includes unit tests for the new Reputation.sol functions (ensuring only valid callers can rate, etc.), simulation tests for SBT issuance (if using), and usability testing on the front-end to make sure reputation info is clear and

trustworthy. If adding complexity like weighted ratings or category ratings, validate that these actually improve the system (perhaps by simulating different scenarios or getting user feedback).

Team Roles Involved:
- *Blockchain Developer* – In charge of modifying the Reputation contract (and possibly deploying a new version via proxy upgrade). Should have a good sense of designing incentive systems to ensure the contract changes align with desired behaviors (for example, thinking through how users might try to game the rating system and mitigating that in code).
- *Data Analyst* – Helps determine the improved reputation formula or weights. Using data from the pilot, they might analyze how ratings were used and propose enhancements (like "if we weight by gig value, the top performers change in this way…"). This role ensures the reputation metric is meaningful and not easily exploited.
- *Frontend Developer* – Updates the profile and search UI to incorporate the richer reputation info (graphs, badges, etc.). Also handles any user interaction with the reputation system (such as viewing proofs or understanding their rating). They need to make complex concepts (like an SBT or ZK-proof of rating) understandable to non-technical users.
- *ZK/Blockchain Engineer* – If implementing ZK aspects or soulbound tokens, this engineer handles those specialized tasks. They might design how the SBT metadata is structured or how the proof generation for "reputation > X" would work. Ensures that the integration between reputation data and any cryptographic proof is solid.
- *Community Manager (indirectly)* – As reputation is a user-facing feature, the community/outreach person might gather user feedback on the fairness and utility of the reputation system. They relay if chefs feel the ratings accurately reflect their work or if restaurants trust the scores, which can inform further tweaks.

Prerequisites & Dependencies:
- **Existing Reputation Data:** Have access to the current repository of ratings (from MVP phase). This might be on-chain in a simple form or stored off-chain. It's needed for analysis and for migration into the enhanced system. Make sure this data is backed up and consistent.
- **OpenZeppelin or Libraries:** If using an SBT approach, leverage standards (ERC-721 with non-transferable extension, or the emerging ERC-5192 for soulbound). Import any libraries or base contracts needed. If sticking to a pure contract upgrade, ensure the storage layout can accommodate new fields (for proxy upgrade, new variables for more stats, etc.).
- **ZK Proof Tools (optional):** If pursuing selective disclosure via ZK, ensure the ZK stack from Phase 2 is available. This includes any proving key, circuits that might need to incorporate reputation. For example, if building a circuit "prove average rating ≥ 4", that circuit needs the user's ratings as inputs, which means we need a way to input those privately – possibly requiring an off-chain data pull and user interaction. The tools to do this (like user running a prover in their browser) should be prototyped.
- **Community Governance Input:** (Forward-looking dependency) – If planning to eventually allow the community to shape the rep system (in DAO phase, e.g., adjust how reputation is calculated), consider documenting the parameters that could be tweaked. This is not an immediate dependency, but preparing the system with configurable parameters (like weights or thresholds that could be voted on later) could be wise. For now, decide these parameters internally but keep in mind they might become governance-controlled.
- **Testing Framework:** Up-to-date tests from Phase 1 should be extended. Utilize a testing framework (like Hardhat/Foundry tests) to verify that after an upgrade, existing data remains correct (write tests for upgrade by initializing a proxy with old data, then upgrading and seeing that data is still accessible and new functions work). Also, possibly integrate a static analyzer or formal verification tool for the new reputation logic if it's critical (ensuring no overflow in calculations, etc., especially if using averages or division).

Scalability & Multi-Agent Optimization (Phase 3)

With core functionality and privacy features in place, **Phase 3** shifts focus to scaling up the platform and optimizing the network of autonomous agents. This involves both technical scalability (handling more users and transactions) and operational scalability (agents and processes that can manage a high volume of gigs efficiently). On the blockchain side, ChefsPlan can explore deploying on multiple networks or an **app-specific rollup** to reach a broader audience and distribute load [30]. The architecture's modular design means ChefsPlan could run instances on zkSync, Polygon, etc., while keeping a unified cross-chain reputation via bridged soulbound tokens or oracles [31]. Off-chain agents must also evolve – the Matching Agent, for example, should use more sophisticated algorithms (potentially AI/ML-driven) to match thousands of chefs to gigs in real time [32]. The AI Copilot agent could take a larger role in system oversight, optimizing postings, or detecting anomalies. Additionally, performance enhancements like batched transactions, **meta-transactions** (gas abstractions), and contract optimizations will be implemented to ensure the platform remains fast and cost-effective at scale. Overall, Phase 3 is about ensuring ChefsPlan can grow from serving a pilot group to serving many cities, with robust infrastructure.

Key Development Tasks:
- [ ] **Optimize smart contracts for gas and throughput** – Review and refine the Solidity contracts to handle higher load. This might include minimizing gas costs of frequent operations (e.g., using more efficient data structures or caching values) – following known gas optimization techniques [33]. Implement batch operations where possible (for instance, if a restaurant wants to post 10 shifts at once, allow a batch posting function to save gas). Consider leveraging zkSync-specific features like native account abstraction or meta-transactions to improve user experience (e.g., a gas relay contract so that restaurants can pay gas fees on behalf of chefs) [33].
- [ ] **Horizontal scalability & multi-chain** – Deploy ChefsPlan to additional networks if needed. For example, launch the smart contracts on another zk-rollup or sidechain to serve users who prefer that network (while zkSync remains primary). If doing so, create a strategy for **cross-chain coordination**: ensure that reputation and identity are recognized across chains. One approach is deploying a unified reputation oracle that aggregates data from all deployments and can issue a combined SBT or update a global record [31]. Another approach: limit certain networks to certain regions to reduce cross-chain interaction. Task: set up deployment scripts for new chain(s), adjust agent configurations to monitor multiple networks, and implement a bridging or sync mechanism for reputational data (could be as simple as a periodic export/import or a trust-minimized oracle feed of top-level stats).
- [ ] **Scale out off-chain agents** – Upgrade the architecture of the off-chain agents for reliability and speed. This could involve running multiple instances of each agent type in a load-balanced way. For example, multiple Matching Agents could each handle a subset of shifts (maybe partitioned by region or by chef categories) and publish suggestions, with the best suggestion taken or combined. Introduce more **AI/ML**: use machine learning to rank or recommend matches (e.g., train a model on past successful gigs to predict good chef-restaurant matches). The AI Copilot agent can be enhanced to manage agent coordination – e.g., monitoring if one Matching Agent is overwhelmed and spinning up another, or analyzing system metrics to anticipate demand spikes [32].
- [ ] **Agent decentralization prep** – While still running centrally, prepare to allow community participation in running agents. This could include open-sourcing the agent code and designing a protocol for how multiple agents might compete or collaborate. For instance, define a system where multiple Matching Agents can submit recommendations for a shift, and perhaps the one that consistently performs best could be rewarded. This is forward-looking to the DAO phase, but Phase 3 can pilot it in-house (run two different matching algorithms concurrently and compare outcomes, etc.). Document how external parties might plug in an agent in the future, to smooth the transition.
- [ ] **Performance and load testing** – Conduct thorough scalability testing. Use scripts or simulation to create a large number of transactions (e.g., simulate thousands of shifts and applications) on a testnet

or a zkSync local node [34] [35]. Monitor the system's response: identify bottlenecks (maybe the Matching Agent CPU usage, or a spike in gas costs for a certain contract function). Tune parameters accordingly – for example, if the Matching Agent is too slow, maybe reduce the frequency of its checks or simplify its algorithm for large loads. Also test the user interface under load (can the frontend handle showing hundreds of shifts without lag? Does the notifications system scale?).
- [ ] **UX improvements for scale** – As the platform grows, small UX issues can become big problems. Revisit the UI design to incorporate better filtering, sorting, and management of a high volume of content (e.g., a restaurant with 50 active shifts needs dashboard tools to manage them). Possibly integrate real-time features (like websockets for instant updates when applications come in) to avoid constant polling. Implement gas abstractions in UX: e.g., if using meta-transactions, allow a one-click action for chefs that doesn't prompt for gas (the gas could be taken care of by the system or sponsor). Also consider multilingual support and other enhancements as user base diversifies.

Team Roles Involved:
- *Blockchain Architect* – Leads the multi-chain expansion strategy. Evaluates which networks to expand to (based on user demand or cost benefits), designs the cross-chain reputation solution, and ensures contract optimizations are safe and effective. Coordinates any additional deployments on new chains.
- *DevOps/Infrastructure Engineer* – Focuses on the deployment and monitoring infrastructure. In Phase 3, this includes scaling the backend: setting up additional servers or cloud resources for multiple agent instances, using container orchestration if needed (Docker/Kubernetes) to manage services, and implementing robust monitoring (dashboard for agent health, alerts if an agent crashes or lags). Also handles any nodes or oracles that facilitate cross-chain communication.
- *AI/ML Specialist* – If AI is introduced for matching or recommendations, an ML engineer will develop and train models. They will need historical data (collected from Phase 1/2 usage) to train algorithms that can, for example, predict no-shows or optimize match quality. They work on integrating these models into the Matching Agent or a new "Recommendation Service."
- *Front-End Lead* – Improves the front-end to handle scale. This involves optimizing code, maybe implementing virtual scrolling for lists, ensuring that new data (like many notifications or messages) doesn't overwhelm the browser. Also leads UX adjustments to manage complexity (like better dashboards for power users). Coordinates with the blockchain team to incorporate new features like gas-less transactions (which might require UI changes to hide gas or explain sponsored transactions).
- *Quality Assurance Engineer* – In a scaled environment, QA must test under various scenarios. They might use automated load testing tools or coordinate large group testing sessions. They also test on multiple network configurations if multi-chain: e.g., scenario where a user has profiles on two chains – does the UI aggregate that correctly? They ensure the system remains stable and user-friendly as features are added.

Prerequisites & Dependencies:
- **Historical Data & Analytics:** Have sufficient usage data from earlier phases to inform scaling decisions. For example, metrics like average transactions per user per day, peak usage times, or which features are most used. This data can identify where optimizations are needed most and train AI models. Set up analytics collection (if not already done) in earlier phases so that by Phase 3 there's a repository of info to work with.
- **Multi-Chain Tooling:** Research and select tools for cross-chain operations. This could include oracles (Chainlink or custom solution) to relay information (like aggregated reputation scores) between chains, and bridges if assets or data need to move. If launching an app-specific rollup via the zkSync Stack, engage with Matter Labs early to plan that and ensure the team has the expertise to operate a rollup (which is more complex than deploying to an existing chain).
- **Computing Resources:** Scaling agents and adding AI might require more computing power. Ensure the budget for servers or cloud services is available. For instance, running an AI service might need a machine with a GPU or at least substantial CPU if doing real-time inference. Plan for horizontal scaling:

maybe have the infrastructure as code (Terraform or similar) ready to spin up multiple instances behind load balancers.
- **Testing Environment:** Maintain a robust staging environment that mirrors the scaled architecture. This could mean having a mini-network of agents and maybe a dedicated test chain to simulate multi-chain. For example, deploy a copy of contracts to a Polygon testnet in addition to zkSync testnet, and test the bridging of rep between them. The environment should allow the team to practice upgrades and catch issues before they hit real users.
- **Community Communication:** As scaling often comes with changes (like potentially moving to additional networks or introducing new agent behaviors), have a communication plan for the community. This is not a strict technical dependency, but important for smooth roll-out: e.g., if enabling meta-transactions, educate users that they can now do X without gas; if launching on a new chain, guide interested users on how to access it. Possibly recruit power users to beta test new improvements (like an improved matching algorithm) before full deployment.

Decentralized Governance & Token Model (Phase 4)

In the final phase, ChefsPlan will transition to a **community-governed platform** by introducing a governance token and setting up a Decentralized Autonomous Organization (DAO). The aim is to empower users (chefs, restaurants, and other stakeholders) to have a say in the platform's evolution, aligning ChefsPlan with decentralized ethos not just in infrastructure but also in decision-making [36]. The governance token (let's call it $CHEF for now) would be distributed to participants, for example as rewards for completing gigs, contributing to the ecosystem, or through a retroactive airdrop to early adopters. Token holders can then use their $CHEF to vote on proposals – these could include changes to platform fees, dispute resolution processes, new feature rollout, or electing a council of representatives. The DAO could control key smart contract parameters (like escrow rules or treasury funds), and over time even assume ownership of the upgradeable proxy admin roles, fully decentralizing control. An initial treasury will be established (possibly funded by a portion of platform fees or an allocation of tokens) to fund community-driven initiatives. Furthermore, previously off-chain services can be opened up – for instance, community members could run their own Matching Agents and get rewarded by the DAO for good performance [37]. Phase 4 thus involves not just technical steps (deploying token and governance contracts) but also careful economic and community planning to ensure a successful handover to the community.

Key Development Tasks:
- [ ] **Design tokenomics** – Define the total supply of the governance token, distribution mechanisms, and utility. For example, decide what portion goes to the team/investors vs. community, how much is allocated for user rewards, and if there's a vesting schedule. Outline how users earn tokens: e.g., a certain number of tokens per completed shift (perhaps weighted by transaction size or given equally to chef and restaurant), or a reputation-linked reward. Ensure the design incentivizes positive actions (good work, honest reviews) and doesn't encourage abuse (for instance, guard against someone gaming rewards by creating fake gigs). Consult industry models for reference (like other DAO-driven marketplaces).
- [ ] **Implement the token contract** – Develop a smart contract for the governance token $CHEF. This will likely be an ERC-20 token with additional features such as snapshots (useful for voting, so voting power can be calculated at proposal block) or inflationary mechanisms if ongoing rewards are needed. Consider using OpenZeppelin templates for ERC-20 and possibly extensions for governance. If planning to use the token for fee payments or staking, ensure the contract supports those interactions. Conduct a security audit on token contract given its importance.
- [ ] **Deploy governance framework** – Set up the DAO governance contracts. This could be a custom setup or using known frameworks like OpenZeppelin **Governor** contracts, DAOstack, Compound's governance model, or Snapshot (off-chain voting with on-chain execution). A likely approach is

deploying a Governor contract where proposals are voted on by token holders and a **Timelock** contract that, upon proposal passage, executes the changes (such as calling functions to change fees or upgrade contracts). Also, consider a **Treasury** contract controlled by the DAO for managing funds. Task: configure parameters like voting delay, voting period, quorum, and proposal thresholds in the governance contract to suit the community size (initially, these might be low if community is small, and can be raised as it grows). [36]
- [ ] **Integrate on-chain governance control** – Gradually migrate control of key platform parameters to the DAO. This involves identifying what decisions the DAO should have power over. Examples: set the escrow fee percentage, choose to deploy on a new network, adjust reputation algorithm parameters, or allocate treasury funds to marketing or insurance pools. Technically, this means certain functions in contracts that were previously restricted to the admin or owner should now only be changeable via governance proposals. For instance, if Escrow.sol had an owner who could set a fee, modify it so that the owner is the DAO's Timelock contract. Similarly, the upgradeable proxy admin could be transferred to the DAO's multisig or timelock, so upgrades become subject to votes. Plan a safe transition – possibly start with less critical parameters and later hand off more control as the community proves capable.
- [ ] **Community launch and airdrop** – When the token and DAO are ready, execute a launch plan. This could include an **airdrop** of tokens to early users (chefs and restaurants who used the platform in Phases 1-3) to bootstrap governance with a diverse set of stakeholders. Also possibly allocate tokens to those who contributed (e.g., testers, partners). Develop a distribution script and verify it carefully (you may need to snapshot user activity up to a certain block and allocate tokens accordingly). Launch a web interface for the DAO (maybe integrate with the existing dApp or use a platform like Snapshot for proposal discussion). Provide educational materials so users know how to use their tokens to vote. Possibly run a few "test" proposals (like non-critical votes) to get everyone comfortable with the process.
- [ ] **Enable community-run agents and contributions** – Open source the code for the off-chain agents and provide guidelines for anyone to run them. The DAO can then decide to reward agent operators. For example, the DAO might set aside some tokens to pay out monthly to the best performing Matching Agent (as measured by, say, how quickly shifts get filled or user ratings of matches). Develop a mechanism for this: it could be as informal as the community voting each month on agent performance, or as automated as the agents themselves reporting metrics to a contract. Encourage community developers to propose improvements (through governance proposals) to agent algorithms, new features, or even protocol changes. Essentially, foster a **developer community** around ChefsPlan.
- [ ] **Legal and security final checks** – Before full decentralization, consult legal experts on the token distribution and DAO setup to ensure compliance with securities and regulations (especially since ChefsPlan deals with real work arrangements, there may be labor laws intersecting token incentives). Implement any necessary safeguards (for example, maybe the DAO cannot change certain core rules that keep the platform legally compliant, or if it does, there's a failsafe). Once clear, proceed to transfer any remaining admin keys to the DAO. At this point, announce that ChefsPlan is community-governed and the core team is just a part of the community, not in sole control.

Team Roles Involved:
- *Tokenomics Expert/Blockchain Dev* – Designs the token and writes the smart contracts for token and governance. Needs to understand both technical solidity implementation and economic implications. Will work on simulations or models to predict how tokens will circulate and be used.
- *Community Manager* – Crucial at this phase, acts as the liaison between the core dev team and the user community. Helps set up initial governance processes, moderates forums or Discord where proposals are discussed, and educates users on how to participate. They may also organize events (AMA, tutorials) around the token launch and DAO voting.
- *Legal Advisor* – Reviews the token model and DAO structure for compliance. Advises on whether the token might be considered a security, how to minimize legal risks, and ensures that the DAO's decisions (especially around real-world work arrangements) won't run afoul of laws. Possibly helps in drafting terms of service updates to reflect the new decentralized ownership.

- *Back-end/Full-Stack Developer* – Extends the platform's front-end to incorporate governance features. This could involve adding a "Governance" section to the dApp where users can see proposals, vote (if on-chain voting via web3) or be redirected to a Snapshot page. Also responsible for integrating any off-chain snapshot votes with on-chain execution if that route is chosen.
- *DevOps* – Helps with the token distribution mechanics (running the scripts to distribute tokens to possibly thousands of addresses, ensuring no hiccups). Also may set up any necessary off-chain infrastructure for governance (like a Discourse forum for proposal discussions, or ensuring snapshot.space is configured for the project).
- *Core Team (All)* – The existing core team should plan to gradually step back control but still be heavily involved in guiding the community initially. They might seed the DAO with some example proposals (like improvement proposals that were already planned) to show how it works. Each member might assume more of an advisory role to the community rather than unilateral decision-maker.

Prerequisites & Dependencies:
- **Legal Structure for DAO:** Decide if the DAO will have any legal wrapper (some DAOs set up a foundation or an LLC to represent them for contracts and legal interactions). If deemed necessary, set this up with the help of legal counsel. If not, at least establish clear terms in the smart contracts and documentation that outline the risk and responsibilities (DAO governance means the community decisions are final, etc.).
- **Treasury Funds:** Accumulate some funds for the DAO treasury prior to launch. For example, the platform might have collected fees during earlier phases (if any fees were enabled) – decide to transfer those to the DAO. Or allocate a percentage of token supply as the treasury. This treasury gives the DAO resources to fund proposals (like paying for further development, community initiatives, insurance pools, etc.).
- **Community Platforms:** Ensure there are channels for community discussion and proposal ideation. This may involve setting up a forum (like forum.chefsplan.com), a Discord/Telegram specifically for governance talk, and encouraging migration of conversations from private channels to public ones. Also, possibly deploy a voting interface: if using on-chain voting via Governor contract, something like Tally or Sybil can be used; if using off-chain Snapshot for signaling, configure that.
- **Snapshot of Participation:** Prepare the data needed for token distribution. For fairness, you'd likely consider metrics like number of shifts completed, total volume, reputation, etc., to distribute tokens. This data should be extracted from contract events or databases. Verify its accuracy and perhaps have it reviewed by an objective party or at least multiple team members to avoid mistakes in who gets what.
- **Security Audits:** The new governance contracts and token contracts should be audited. Also, consider auditing the whole system now from a perspective of how it all ties together (a holistic audit) since it's the final stage – especially because you're about to hand off control. Any vulnerabilities left could be catastrophic once control is decentralized (since no central kill-switch). So, get a thorough security review of the entire ChefsPlan 5.0 stack if possible.
- **Operational Transition:** Internally, the team should prepare for a different role. This includes possibly revoking deployer privileges, handing server control to community (if going fully decentralized on agents, else at least being transparent about agent operation). Draft a "Declaration of Decentralization" document that outlines which parts of the system are now community-controlled and which (if any) remain with the team. This transparency will help build trust that Phase 4 is complete and not superficial.

Throughout these phases, it's important to iterate and incorporate feedback. By following this roadmap, ChefsPlan 5.0 will gradually evolve from a team-driven MVP into a fully decentralized, community-governed ecosystem – delivering on the vision of a trustless gig-work platform for hospitality [38] [37]. Each phase builds on the last, adding functionality, security, and decentralization, while the provided checklists ensure that the development team covers all necessary tasks, roles, and preparations at every step. With this plan, ChefsPlan can systematically grow into a scalable, privacy-preserving, and self-sustaining network for chefs and restaurants worldwide.

---

[1] [2] [3] [19] [21] [22] [23] [24] [25] [27] [30] [31] [32] [33] [36] [37] [38] ChefsPlan_ A Decentralized Gig-Work Platform for Chefs and Restaurants.pdf
file://file-P78t1R65FUtcREdnM3QuBq

[4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [20] [26] [28] [29] [34] [35] ChefsPlan 5.0 Scaffold Overview.pdf
file://file-U5UDmRJQKoLKsW7pVKdXkc