ROXONN Development Prompt (Decentralized GitHub Rewards Platform)

Project Overview

ROXONN is a decentralized bounty platform on Coinbase's Base chain (an Ethereum Layer-2) that links GitHub repositories to on-chain rewards. Contributors sign in with GitHub, browse funded issues, claim them, and earn crypto tokens when pull requests are merged. Repository owners (Pool Managers) fund issue-specific reward pools and approve contributions. The system uses Solidity smart contracts on Base to manage token pools and payouts, a GitHub App to monitor repository events, and web interfaces for users. All code should follow secure, modular design principlesopenzeppelin.comulam.io.

User Roles

- **Pool Manager (Maintainer):** Owner or maintainer of a GitHub repository. Creates and funds reward pools for issues, sets bounty amounts, reviews and approves/denies contributor submissions. Connects an external crypto wallet (e.g. MetaMask) to deposit tokens into the platform.
- **Contributor (Developer):** GitHub user who finds and solves issues. Signs in via GitHub OAuth, connects a crypto wallet, browses issues with bounties, claims an issue by linking a pull request, and receives tokens upon approval. Tracks earnings in a dashboard.
- **Platform Admin:** Operators of ROXONN who monitor system health, manage users or disputes, and moderate platform-wide settings (optional). They have an admin panel to oversee activity across all repos and users.

User Stories and Workflows

- *Contributor Story:* "As a Contributor, I log in with GitHub OAuth and link my crypto wallet. I browse issues across repos filtered by reward amount. I claim an open issue by submitting a pull request and linking it to ROXONN. Once my PR is merged, I receive the bounty tokens in my wallet. I can view my total earnings and claim history on my dashboard.”

- *Pool Manager Story:* “As a Pool Manager, I connect my wallet and select one of my GitHub repos. I create a funding pool by depositing tokens (up to a daily cap) and set bounties on specific issues. I review contributor pull requests and either approve or reject the reward. I receive notifications about claims and payouts. I can top up the pool or close it at any time.”
- *Admin Workflow:* The Platform Admin sets up system parameters, views aggregated metrics (total issued tokens, active users), and can intervene in disputes (e.g., refund an issue if needed). They manage email/notification templates and system-level settings.

Functional Requirements

- **Authentication:** Implement GitHub OAuth 2.0 for user loginstateful.com. After login, users link a blockchain wallet via WalletConnect/MetaMask.
- **Repository Selection:** Pool Managers can import or select their GitHub repositories (the platform reads available repos via GitHub API).
- **Funding Pools:** Pool Managers deposit ERC-20 tokens (or Base’s native token) from their external wallet into a smart-contract-managed pool for a repository. Enforce a 1000-token/day funding limit per repository to control spending.
- **Issue Bounties:** Managers set reward values on GitHub issues. The frontend shows issue lists with on-chain bounty info. Contributors filter and sort issues by reward size.
- **Claiming Issues:** Contributors claim an issue on the platform (mapping their GitHub username to their wallet). They indicate intent via the UI, and the system records the claim.
- **PR Linking:** Contributors submit a pull request on GitHub. The ROXONN GitHub App detects the merged PR (see GitHub App section) and notifies the platform. The platform verifies the merge and automatically triggers reward if the issue was claimed. If merging is ambiguous, the Pool Manager reviews it.

- **Reward Distribution:** On a merged PR (or closed issue), the backend calls the smart contract to transfer tokens to the contributor’s wallet (or refund to the pool if rejected). Smart contracts handle token disbursement and update balances. All transactions occur on Base Chain.
- **Pool Manager Approval Flow:** If a PR merge doesn’t clearly satisfy the issue (e.g. partial fix), the Pool Manager can manually approve or deny reward through the dashboard. The system then executes or cancels the token transfer accordingly.
- **Earnings Dashboard:** Contributors see a dashboard of their total rewards earned, pending rewards (claims in progress), and transaction history. Pool Managers see pool balances, daily spend, and pending claims.
- **Notifications:** The platform sends email and in-app notifications for key events: claim submitted, PR merged, reward paid, low pool balance, etc.
- **Admin Panel:** A secure admin interface for platform operators to view system metrics, manage user issues or disputes, and monitor contract fund levels.
- **Wallet Connectivity:** Use WalletConnect or MetaMask integration in the frontend. Contributors and Managers must connect wallets to perform blockchain transactions (token deposit, reward claim).

Non-Functional Requirements

- **Scalability:** Support many repositories and concurrent users. Use pagination or lazy loading for issue lists. Ensure the smart contracts can manage multiple pools efficiently.
- **Performance:** Frontend should be responsive; backend APIs should cache GitHub data where possible. Smart contract calls should be efficient (optimize gas). Use Base Chain’s fast block times for quick transaction confirmation.
- **Reliability:** Design contracts to be fault-tolerant (e.g., using OpenZeppelin’s SafeERC20 and pull-payment patternsopenzeppelin.com). Backend services should retry on transient errors (e.g., blockchain or API timeouts).
- **Usability:** Provide clear UI for fund flows. Use consistent wallet connect flows. Offer tooltips or guides for first-time users (e.g., how to link GitHub with wallet).

- **Maintainability:** Code should be modular and well-documented. Follow conventions (e.g., consistent naming, coding style).
- **Compliance (Base Specific):** Leverage Base's EVM compatibility. If using a stablecoin (like USDC) on Base, ensure proper bridging. Use Base's testnet for development and base mainnet for production.
- **Daily Funding Cap:** Enforce a hard limit of 1000 reward-tokens per repo per day in the smart contract logic.

Blockchain Architecture (Base Chain)

- **Base Chain Overview:** Base is an Ethereum Layer-2 using Optimistic Rollupsulam.io. It supports Solidity contracts and is fully EVM-compatible. Smart contracts should be written in Solidity ^0.8.x (for overflow safety). Use Hardhat or Truffle with Ethers.js for deployment/testing.
- **Smart Contracts Design:** Implement at least two contracts:
  (1) **RewardPoolManager**: maintains a mapping of GitHub repos to funding pools, tracks manager addresses, daily caps, and total deposits;
  (2) **BountyContract**: handles individual issue bounties, claims, and payouts. Alternatively, a single contract with structs for each pool/issue could suffice. Use OpenZeppelin libraries for Ownable (or AccessControl) and SafeMathopenzeppelin.com.
- **Data Models:** On-chain track: pool balance, per-issue reward, claim status, and contributor addresses. Off-chain (backend DB): link GitHub issue IDs and PR IDs to on-chain issue bounties for quick lookup.
- **Token Handling:** Rewards use an ERC-20 token on Base (could be Base's native wrapped ETH or a stablecoin). Pool Managers deposit tokens into the contract; when a bounty is paid, tokens transfer from the contract to the contributor.
- **Event Triggers:** Contracts emit events for deposits, claims, approvals, and payouts. The backend should listen for these events (via web3 or by reading transaction receipts) to update UI or send notifications.
- **Security Measures:** Contracts must prevent re-entrancy (use nonReentrant modifiers) and ensure only the designated Pool Manager can withdraw or

approve funds for their repo. Use a pull-over-push payment model where possible. Include a circuit breaker or pause function for emergencies.

Security Requirements

- **Code Security:** Follow OpenZeppelin best practices for secure Solidity developmentopenzeppelin.com. Use audited libraries (Ownable, ReentrancyGuard, SafeERC20). Fix the compiler pragma to a specific version (e.g., 0.8.17) to prevent inconsistencies.
- **Smart Contract Auditing:** Structure code for audit readiness: write clear NatSpec comments, keep functions short, and write comprehensive unit tests. Minimize complex logic.
- **Authorization:** Validate that GitHub App payloads come from GitHub by verifying webhook signatures. Only allow authenticated Pool Managers to modify their own pools (check `msg.sender` or use access control).
- **Data Protection:** Do not store private keys on the server. Use environment variables or secret management for OAuth secrets and node private keys.
- **Network Security:** Use HTTPS/TLS for all API endpoints. Escape and validate all inputs to backend (e.g., GitHub usernames, issue IDs). Avoid common web vulnerabilities (CSRF, XSS) in the frontend.
- **Error Handling:** Gracefully handle failed transactions (e.g., out-of-gas, reverted transfers). Notify users if an action fails and allow retry if safe.
- **Auditing and Monitoring:** Log critical events. Consider integrating a monitoring service for smart contracts (e.g., Tenderly or OpenZeppelin Defender) to detect abnormal behavior.
- **Dependencies:** Keep all dependencies (npm packages, contract libraries) up to date and minimal.

Success Metrics

- **Adoption:** Number of repositories onboarded and active Pool Managers.
- **Engagement:** Count of issues claimed and pull requests rewarded.

- **Token Throughput:** Total tokens distributed through the platform, and daily transaction volume on Base.
- **User Retention:** Number of repeat Contributors and frequency of Pool Manager funding.
- **Performance:** Average time from PR merge to reward payout. System uptime and response times.
- **Security:** Zero critical vulnerabilities (tracked by audits). No unauthorized fund transfers.

Core Features to Implement

- **GitHub OAuth Login:** Secure third-party login for Contributors (OAuth 2.0 flow with GitHub).
- **Wallet Connection:** MetaMask/WalletConnect integration for both Pool Managers and Contributors.
- **Pool Funding:** Pool Manager can deposit tokens into a new or existing pool. Enforce per-repo funding cap (1000 tokens/day).
- **Issue Bounty Setting:** Pool Manager can tag GitHub issues with reward amounts via the dashboard.
- **Issue Browser:** Contributor dashboard lists all funded issues (from all repos they have access to), with filters (by reward size, date, repo).
- **Issue Claim & PR Link:** Contributors click “Claim” on an issue, then later link a GitHub pull request to that claim. The system verifies ownership.
- **Automatic Payout:** On PR merge (or issue close), the platform auto-distributes the bounty tokens to the contributor’s wallet. Use the GitHub App to detect merges and trigger a contract call.
- **Manager Approval Flow:** If an issue requires manual review, Pool Manager can accept or reject the contribution. Only upon approval does the payout occur.
- **Contributor Earnings Dashboard:** Shows individual rewards history, pending claims, and wallet balance.

- **Smart Contract Audit Readiness:** Contracts should be clean, documented, and fully tested. Include unit tests covering edge cases.
- **Admin Panel:** System operators have a UI to monitor all pools, users, and platform health. Admins can issue refunds or override disputes.
- **Notifications System:** Email and in-app alerts for events like "Your PR was approved and reward sent", "Issue claimed by contributor", "Pool running low on funds", etc.

Components & Deliverables

- **Smart Contracts (Solidity):** Contracts for managing pools, bounties, and token payments. Use OpenZeppelin for ERC-20 and access control. Write unit tests (Hardhat/Truffle). Emit events for all key actions. Example structures:
    - `ROXRewardPool.sol` – handles deposits, issue mapping, and daily cap logic.
    - `ROXBounty.sol` – linked to an issue, handles claiming and payout by Pool Manager approval.
- **Frontend (React):** Two main views (Pool Manager and Contributor).
    - **Manager UI:** Repo selection, fund pool (via contract call), set/edit bounties on issues, view claims, approve contributions.
    - **Contributor UI:** Sign in, wallet connect, browse bounties, claim issues, link PRs, view earnings.
    - Use a UI framework (e.g., Material-UI or Tailwind) and web3 libraries (Ethers.js or Web3Modal).
    - Handle Ethereum provider detection (MetaMask) and WalletConnect sessions.
    - Interact with backend APIs for GitHub data and with smart contracts for blockchain actions.
- **Backend Server (Node.js):**
    - **GitHub App Webhooks:** Listen to repository events (PR merges, issue closures, comments if needed) and trigger reward logic. Use a library like Probot or Octokit. Verify webhook signatures.

  - **API Services:** Expose endpoints for the frontend to fetch data (e.g., list of issues with active bounties, user profile, claim status).
  - **GitHub Integration:** Use GitHub OAuth token to fetch user's repos/issues. For Pool Manager, fetch own repos; for Contributors, fetch claim history by their GitHub ID.
  - **Blockchain Interaction:** Use Ethers.js to call smart contract functions (e.g., deposit tokens to pool, payout bounty). Maintain a secure signer wallet for automated tasks if needed (or trigger user wallet).
  - **Database (optional):** Store mappings between GitHub IDs and wallet addresses, issue claims, and events history. Use a database like PostgreSQL or MongoDB for persistence.
  - **Notifications:** Hook into an email service (SendGrid/Mailgun) and frontend push notifications.
- **GitHub App (ROXONN Contribution Rewards):**
  - Must be installable on any GitHub repo.
  - **Webhooks to subscribe:** `pull_request` (to detect merges/closures) and `issues` (optional, to detect issue closings).
  - **Logic:** On `pull_request.closed` with `merged: true`, verify if the PR was linked to a claimed issue bounty. If so, signal the backend to execute the payout contract call. If an issue is closed without a merged PR, optionally refund or close the bounty.
  - **Permissions:** The app needs `read & write` access to Pull Requests and Issuesdocs.github.com.
- **Wallet & Payments:**
  - Integrate MetaMask/WalletConnect so users sign transactions (e.g., `approve` and `deposit` for Pool Managers, `claim` for Contributors).
  - Use a standard ERC-20 token for bounties; if none is chosen, create a `ROX` token. Pool Managers must first `approve` the contract to transfer tokens on their behalf, then `deposit` into the pool.

  - All monetary flows should be trackable in the smart contract state (no off-chain IOUs).

PRD Sections to Include

- **Introduction:** Describe ROXONN's purpose and goals (decentralized GitHub bounty platform on Base Chain).
- **User Roles:** Define Contributor, Pool Manager, Admin (as above).
- **User Stories & Workflows:** Outline scenarios for signing in, funding a pool, claiming an issue, merging PR, and payout. Include edge cases (e.g., dispute workflow).
- **Functional Requirements:** Summarize the above bullet points of what features the system must have.
- **Non-Functional Requirements:** Summarize performance, scalability, security, usability considerations.
- **Blockchain Architecture:** Detail the Base chain specifics (Optimistic Rollups, EVM) and how smart contracts are structured to handle pools and bountiesulam.io.
- **Security Requirements:** Highlight contract security (OpenZeppelin patternsopenzeppelin.com), OAuth security, webhook validation, secret management.
- **Success Metrics:** List how to measure success (active users, issues resolved, tokens distributed, etc.).

Development Guidelines

- **Modularity:** Structure code into reusable modules. E.g., separate contract files for each component, React components for each view, and small backend services. Use MVC or similar patterns for the backend.
- **Best Practices:** Follow industry standards (e.g., OpenZeppelin Contracts for Solidity, ESLint/Prettier for JS). Lock down Solidity pragma, use SafeMath or Solidity 0.8 overflow checksopenzeppelin.com.

- **Testing:** Write comprehensive tests. For contracts, use Hardhat or Truffle to test all functions (normal and edge cases). For backend, use Jest or Mocha. For frontend, use React Testing Library or Cypress for key flows.
- **Documentation:** Document all code. Include a README with setup instructions. Provide API documentation (Swagger or similar) for backend endpoints.
- **Continuous Integration:** Set up CI to run linting, tests, and contract verification on each push (e.g., GitHub Actions).
- **Code Security:** Review code with linters and tools (MythX, Slither, or OpenZeppelin's Defender). Ensure contracts are "audit-ready" by minimizing complexity.
- **Deployment:** Use environment variables for config (API keys, secrets). Deploy contracts first to Base testnet, verify functionality, then to mainnet. Keep deployment scripts idempotent.

Coding and Output Requirements

- Instruct the AI coder to produce **concise, clean, and well-commented code** for each component. Encourage using TypeScript for type safety in frontend/backend.
- **Modular Code:** Every contract and service should be in its own file/module. Reuse components where possible.
- **Testable:** Include unit tests and specify how to run them. For example, "Provide tests for each Solidity contract using Hardhat/Chai."
- **Secure:** Follow OpenZeppelin security patternsopenzeppelin.com and validate inputs. Use HTTPS, secure cookie/session management on the backend. Do not hard-code secrets.
- **Documentation:** Each deliverable (contracts, API, front-end) should have usage instructions in comments or separate docs.
- **Ready for Deployment:** The AI should output code that can be directly deployed/run after filling in configuration (e.g., OAuth client secrets, Base RPC endpoint).

Note: This prompt is intended for an AI coding assistant. The assistant should interpret these specifications and output development code accordingly. All generated code must align with these requirements, be structured logically by component, and adhere to high-quality software practices. The assistant's output should reflect the full-stack nature of the project and include comments linking to these design points (e.g., indicating security checks, event handling, etc.).