

**Flappy Bird TON Game – Onboarding & Handoff**

**Guide**

**Project Overview**

This document provides a comprehensive handoff guide for the **Flappy Bird-style webapp game on TON** (The Open Network). The project is a classic Flappy Bird clone enhanced with Web3 features on the TON blockchain. It includes a React-based game front-end, a TON smart contract for handling in-game payments/rewards, and a Telegram bot integration for deploying the game as a Telegram Mini App. The new administrator (agent) will receive full access to all systems and will assume responsibility for development, deployment, and operations.

**Note:** Placeholder values (in <ANGLE\_BRACKETS> ) indicate sensitive credentials or URLs to be provided by the project owner. The agent should replace these placeholders with actual values during setup.

**Repository Access & Development Tools**

**GitHub Repository Access:** The game’s source code is managed in a GitHub repository. Ensure the new agent has **full access** (e.g. added as an Admin collaborator or ownership transferred) to the repository at **<REPO\_URL>**. The repository contains all code for the front-end (and possibly backend services), so unrestricted access is required to push updates and manage issues.

**Development Environment:** The project ecosystem uses modern web and blockchain development tools. Key tools and technologies include:

• **Node.js & npm/yarn:** The webapp is built with Node.js. Install the recommended Node.js LTS version (e.g. v16 or v18) and package manager (npm or Yarn) to run and build the project. All JavaScript/TypeScript dependencies are listed in package.json – e.g., React for the front-end UI, and TON SDK libraries for blockchain integration.

• **React Framework:** The front-end is a React application (likely bootstrapped with Create React App or a similar toolkit). Component code (such as the Flappy game logic in FlappyGame.jsx ) is written

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| in JSX/JavaScript | [1](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L8-L16) | |  | | --- | | [2](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L14-L19) | | . Familiarity with React hooks and state management is necessary for |

making UI or game logic changes.

• **TON Development Tools:** Set up the TON blockchain development toolchain. This may include:• **TON SDK or Tonweb:** Libraries to interact with TON smart contracts from JavaScript/TypeScript.

• **Smart Contract Compiler:** If the smart contracts are written in FunC (TON’s smart contract language) or solidity (via an EVM for TON), install the appropriate compiler or tool (for example, TON Compiler or any build scripts provided).

• **Deployment Scripts:** The repository might contain scripts or configurations (e.g. Hardhat, toncli, or custom scripts) to compile and deploy contracts. Ensure these are installed and configured (check README or scripts in the repo).

1

• **Other Utilities:** If the project uses Docker (for containerized deployment) or other services, install those. For example, the official TON Flappy Bird example uses Docker and Ngrok for development

|  |  |  |  |
| --- | --- | --- | --- |
| [3](https://github.com/ton-community/flappy-bird#:~:text=Ngrok) | |  | | --- | | [4](https://github.com/ton-community/flappy-bird#:~:text=Running) | | – check if this project does similarly. If Docker is used, have Docker and Docker Compose |

installed.

Make sure to verify the repository’s README or documentation for any project-specific setup instructions (such as environment variables, build commands, etc.). Do a fresh clone of the repo and run the development server (commonly npm install then npm start ) to ensure the environment is correctly configured.

**Initial Credential Setup**

Several accounts and credentials must be prepared to enable the agent to manage the system. Below is a checklist of required credentials, with guidance on creating or obtaining each:

• **GitHub Account:** The agent should have a GitHub account ( <AGENT\_GITHUB\_USERNAME> ). The project owner will add this account to the GitHub repository with appropriate permissions. No new login needs creation if the agent already has an account; just ensure access is granted. (Placeholders: <REPO\_URL> for repository link, <AGENT\_GITHUB\_USERNAME> for the agent’s GitHub.)

• **Telegram Bot & Web App:** Set up a Telegram bot which is used to host the game as a Telegram Mini App.

|  |
| --- |
| • **Create Bot:** Using Telegram’s[@BotFather](https://t.me/BotFather), create a new bot via the /newbot command. Choose a |

name and username for the bot (e.g. “FlappyTONBot”). BotFather will provide an **API token** – record

|  |  |  |
| --- | --- | --- |
| this as <TELEGRAM\_BOT\_TOKEN> | [5](https://github.com/ton-community/flappy-bird#:~:text=Telegram%20bot%20%26%20Telegram%20Web,App) | . |

• **Link Web App:** Still in BotFather, link a Web App to the bot using /newapp . Select the bot just created, then provide the required details: app name, description, an image, and the domain where the game will be hosted (the URL of the webapp with SSL, e.g. https://<YOUR\_DOMAIN> ). Also set a unique short name for the game. BotFather will confirm and give a URL (something like https://

|  |  |  |  |
| --- | --- | --- | --- |
| t.me/<YourBotName>/<GameShortName> ) which launches the game inside Telegram | |  | | --- | | [6](https://github.com/ton-community/flappy-bird#:~:text=Telegram%20bot%20%26%20Telegram%20Web,App) | | . Save |

this URL and ensure the domain matches where you’ll deploy the webapp.

• These steps generate credentials and links: the <TELEGRAM\_BOT\_TOKEN> (keep it secret) and the <TELEGRAM\_WEBAPP\_URL> . The token will be used in the backend server or bot script to interact with Telegram (if the bot will send messages or verify users), and the URL is used to access the game via Telegram.

• **TON Wallet (Testnet & Mainnet):** Create a TON cryptocurrency wallet for the project, or obtain the keys to the existing one:

• For **Testnet**: Use a wallet app (e.g. Tonkeeper or official TON Wallet) and switch it to test network mode. Generate a new wallet (you will get a seed phrase). Label this as   
 <TON\_TESTNET\_WALLET\_SEED> and keep it secure. Also note the testnet wallet address   
 <TON\_TESTNET\_WALLET\_ADDRESS> . Fund this wallet with test Toncoin from a faucet. TON provides a Telegram bot “Test Giver” that dispenses testnet tokens – you can request free TON test coins

2

|  |  |  |
| --- | --- | --- |
| (around 2 TON each hour) for development | [7](https://www.datawallet.com/crypto/get-ton-testnet-tokens#:~:text=Summary%3A%C2%A0You%20can%20get%20free%20TON,2%20tokens%20every%2060%20minutes) | . This ensures you have funds to deploy and test |

contracts on testnet.

• For **Mainnet**: If the game is to handle real value, set up a mainnet TON wallet. This could be the same wallet after switching network or a separate one. Securely store the <TON\_MAINNET\_WALLET\_SEED> and address <TON\_MAINNET\_WALLET\_ADDRESS> . The mainnet wallet will be used to deploy the smart contract and receive real revenue, so it should be controlled by the project owner or transferred to the agent as part of the handoff (with multisig or other security if needed). **Note:** Do not share actual seed phrases in this document; use placeholders and exchange them securely off-document.

• **Smart Contract Deployment Credentials:** The smart contract (e.g. the royalty payment splitter) may require its own key pair or use the wallet’s key for deployment. In TON, a contract’s address can be deterministic from a deployer key and initial data. Typically:

• If using a deployment tool (like a script or toncli), you might need to configure a deployer wallet or provide the contract’s future address and the wallet’s secret key. Ensure the agent has access to any **private keys** or mnemonic phrases required for deploying/upgrading the contracts.

• Identify the **contract address** on testnet and mainnet. For example,   
 <PAYMENT\_SPLITTER\_TESTNET\_ADDRESS> and <PAYMENT\_SPLITTER\_MAINNET\_ADDRESS> . If the contract is already deployed on mainnet, provide that address to the agent, along with any admin keys if the contract has an admin role.

• If the contract requires an admin or owner key to adjust settings (such as changing payee addresses or withdrawing funds), that key must be handed over. It could be the same as the mainnet wallet or a separate key stored in code or config (check the contract code to verify how ownership is handled).

• **Other Services:** If any other third-party services are used, prepare credentials for those. For example:

• **Hosting**: credentials or access to the server or cloud platform where the webapp is hosted (could be AWS, Vercel, Heroku, etc.). Provide <HOSTING\_SERVICE\_URL> and an account login or invite the agent’s account.

• **Domain**: if a custom domain is used for the webapp (especially since Telegram requires an HTTPS domain), ensure the agent can manage DNS or SSL certificates as needed (or provide an admin contact at the domain registrar).

• **Pinata (IPFS)**: (if used for storing game assets or NFT metadata, as in some TON examples) – API key

|  |  |  |
| --- | --- | --- |
| and secret for the Pinata account <PINATA\_API\_KEY> / <PINATA\_API\_SECRET> | [8](https://github.com/ton-community/flappy-bird#:~:text=Pinata) | . |
|  |  |  |

• **Ngrok or Tunneling (for dev)**: If using Ngrok for local testing with Telegram, supply the Ngrok auth

|  |  |  |
| --- | --- | --- |
| token <NGROK\_AUTH\_TOKEN> or have the agent create their own Ngrok account | [9](https://github.com/ton-community/flappy-bird#:~:text=Ngrok) | . |
|  |  |  |

Each of these credentials should be created securely. Use strong passwords and enable two-factor authentication where applicable. Share secrets through a secure channel (not in plain text via email or documents). The new agent should confirm receipt and ability to use each credential (for example, log into GitHub, connect the Telegram bot, access the TON wallet, etc.) during the onboarding process.

3

**Smart Contract Overview & Deployment Pipeline**

**Smart Contract Description:** The project uses a **royalty-enforced payment splitter** smart contract on the TON blockchain. This contract is designed to automatically distribute incoming payments among multiple

|  |  |  |
| --- | --- | --- |
| parties according to predefined percentages, thereby enforcing a revenue share (royalty) arrangement | [10](https://www.cookbook.dev/contracts/royalty-splitter#:~:text=Royalty%20Splitter%20on%20Cookbook%20This,is%20specified%20is%20by) | . |

In practice, any Toncoin sent to this contract will be split and forwarded to the stakeholders’ addresses (e.g., a percentage to the game developer/treasury and a percentage to players or content creators as defined in the contract). This ensures that revenue (or prize payouts) from the game is trustlessly divided without manual intervention. If the game also involves NFTs or token sales, the contract can be used to enforce creator royalties by requiring secondary sale payments to go through this splitter contract.

**Contract Components:** The payment splitter contract likely includes: - A list of **payee addresses** and their corresponding **share percentages** (or fixed shares) defined in the code or set during deployment. The total must sum to 100% (or the full amount of any payment). - A function to distribute or withdraw funds. Depending on implementation, it might automatically split on each incoming payment, or hold funds and allow periodic distribution. - (Optional) Royalty enforcement logic, if dealing with NFTs – for example, the NFT’s transfer might be restricted unless a royalty fee is paid to this contract. If this is in scope, the contract could be linked with an NFT contract ensuring a portion of any sale price is sent to the splitter.

**Contract Language and Location:** On TON, smart contracts are often written in **FunC** (TON’s native smart contract language) or in Solidity (if using an EVM-compatible chain or a transpiler). Check the repository for a contracts directory or specific files (e.g., .fc files for FunC or .sol files). The agent should familiarize themselves with this contract code. Key files and their paths (with placeholders if applicable): -<CONTRACT\_FILE> – the main smart contract source code. - <CONTRACT\_TEST\_FILE> – any test scripts or documentation for the contract. - If the contract is not included in the main repo (sometimes kept separate), ensure the agent has access to that repository or file.

**Deployment Pipeline:** The process to deploy or update the smart contract is as follows: 1. **Development & Testing:** Use the testnet wallet and a TON dev environment to deploy the contract on **TON Testnet** first. Ensure you have test Toncoins (from the faucet) to pay for deployment fees. Compile the contract code using the appropriate compiler or SDK. For example, if using FunC, run the FunC compiler to get the .tvm binary and use the TON CLI to deploy. If using a JavaScript SDK (like tonweb or Ton SDK), there may be scripts to deploy via Node.js. 2. **Configure Payees:** Before deployment, set the payee addresses and splits. Typically, this is done in the constructor or an init data of the contract. In a royalty splitter, once deployed,

|  |  |  |
| --- | --- | --- |
| the payee list is immutable | [10](https://www.cookbook.dev/contracts/royalty-splitter#:~:text=Royalty%20Splitter%20on%20Cookbook%20This,is%20specified%20is%20by) | (unless the contract has an owner function to update it, which is uncommon |

for simplicity). So double-check the addresses (the agent’s or owner’s addresses for revenue) are correct in the code or deployment parameters. 3. **Deploy to Testnet:** Deploy the contract to testnet using the deploy script or CLI. Note the **testnet contract address** that results. Perform basic **testing**: send a small amount of test TON to the contract and verify that it immediately forwards the correct portions to each payee address. If possible, write or use a unit test that simulates this (the repository may contain unit tests or you can call the contract’s get-methods to confirm the configuration). 4. **Security Check:** Make sure the contract code has been reviewed or tested for security, as it will handle funds. If not already done, perform a code review or use known templates (OpenZeppelin’s payment splitter logic on EVM is a good reference; TON’s equivalent should be carefully implemented). 5. **Deploy to Mainnet:** Once confirmed on testnet, deploy the contract on **TON Mainnet**. Use the mainnet wallet (with sufficient TON balance) for this. After deploying,

|  |  |
| --- | --- |
| record the **mainnet contract address** | <PAYMENT\_SPLITTER\_MAINNET\_ADDRESS> and verify it’s |

functioning (perhaps by sending a tiny amount of Toncoin to ensure the split occurs as expected). 6.

4

**Integrate with Game:** Update the webapp or backend with the contract address. For example, if the front-end needs to know where to send payments or call for rewards, configure it to use the mainnet address. This could be in a config file or environment variable like REACT\_APP\_CONTRACT\_ADDRESS . Similarly, if a **Jetton** (fungible token on TON) is used as the in-game currency, ensure its smart contract (jetton master address) and wallet addresses are set in the config. 7. **Maintain Keys:** The agent should securely store any keys related to the contract. If the contract has an admin key (to pause, update, or emergency withdraw funds), that private key must be backed up. If the contract is truly trustless with no admin, then the main keys of concern are the wallets controlling the funds.

**Royalty Enforcement:** If the context includes NFTs or other assets (for example, if high-scoring players earn NFTs, or if game sessions themselves are tokenized), there might be additional contracts (like an NFT contract) tied into the system. Document any such contract similarly: - NFT contract address and how it’s deployed (e.g., via a standard like TIP-4/TIP-6 on TON). - How the payment splitter ties in (possibly as the royalty receiver for NFT sales). - Any deployment pipeline for those assets (e.g., minting NFTs to players for certain achievements).

In summary, the smart contract aspect ensures all financial transactions in the game are handled on-chain. The new agent should gain familiarity with TON’s contract deployment (which differs from Ethereum but follows similar principles of compile -> deploy -> test). After the handoff, the agent will be responsible for updating the contract if royalty recipients change (which might require deploying a new contract if it’s immutable) and for monitoring contract activity to ensure payouts are happening correctly.

**Webapp Deployment & Codebase Layout**

The web application is the front-end and potentially backend components that make up the game and its supporting services. This section describes how the webapp is structured and how to deploy it, along with the agent’s responsibilities for maintaining it.

**Deployment Stack Overview:** The Flappy Bird webapp is deployed as a **web service** accessible via HTTPS (required for Telegram integration). The likely stack is: - **Front-end:** A React single-page application (SPA) that contains the game UI and logic. This can be hosted on a static hosting platform or served via a Node.js server. The front-end communicates with TON (for wallet transactions) either directly from the browser (using TON wallet APIs) or via calls to the backend. - **Backend:** (If applicable) A Node.js/Express server that might serve the content and handle any server-side logic, such as communicating with Telegram Bot API, orchestrating blockchain operations (minting tokens/NFTs), or storing user data. The official TON example uses a combined server for Telegram auth, NFT minting, etc., so check if this project has a server component (e.g., a server or api directory). If the game is purely client-side except for the blockchain, the backend might be minimal or only for the Telegram bot webhook.

**Codebase Layout:** The repository is organized into directories. Here is a guide to the key parts of the codebase: - **Front-end Code ( /src or similar):** This contains React components, assets, and logic for the game. - src/components/FlappyGame.jsx – The main game component implementing the Flappy Bird gameplay. It uses React state and hooks for game state, score, etc. The code defines constants for game

|  |  |  |
| --- | --- | --- |
| physics (gravity, jump impulse, pipe gap, etc.) | [2](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L14-L19) | and manages the render loop and collision detection. |

(Comments in the code are in Russian, but function names and logic are clear.) The agent should review this file to understand how the game updates frames, detects pipe collisions, and updates score. - src/

5

App.jsx or src/index.jsx – The entry point of the React app. Likely sets up the main game screen and integrates any wallet connection UI or menus. - src/styles/ – CSS files for styling (e.g., FlappyGame.css for game visuals). - Other components or util files – e.g., src/components/ Scoreboard.jsx (if any), or context providers for wallet integration, etc. - **Smart Contract Integration:** Look for any config or code where the front-end interacts with the blockchain: - There might be a config file or constants file containing the payment splitter contract address and perhaps a Jetton address. - If using a TON wallet SDK, there may be code to connect to TON (for example, integration with **Tonkeeper** or **TON Wallet** browser extension via TonConnect). - If the project uses **Jettons** (fungible tokens) as rewards, there might be code to listen for rewards or to display token balances. - Check for any mention of **TonWeb** or **ton** in the code, which would indicate blockchain calls. Also, in package.json, dependencies like tonweb or ton-core would confirm this. - **Backend/Bot Code:** If a backend exists: - Possibly under server/ or similar. It could be a Node.js app that does a few things: hosts the React build for production, provides an API for high scores or rewards, and contains the Telegram bot logic (using the <TELEGRAM\_BOT\_TOKEN> ). -For Telegram: The bot might handle WebApp initialization or verification. In Telegram WebApp, upon launch, Telegram provides the web app with user info (via window.Telegram.WebApp ). The backend can verify this data using the bot token if needed. There might also be endpoints for the bot to send messages (like sending a user their score or reward after game over, as a Telegram message). - If players are rewarded with NFTs or Jettons after certain games, the backend likely triggers the smart contract calls (since the private keys to mint or distribute would be kept server-side). For example, awarding an NFT for the first

|  |  |  |
| --- | --- | --- |
| time a player scores, etc., as hinted by similar projects | [11](https://github.com/ton-community/flappy-bird#:~:text=The%20example%20contains%3A) | . - Check for any database usage (the ton- |

community example used TypeORM and migrations, suggesting a database to keep track of users and rewards). If a database is used (PostgreSQL, etc.), configuration for it (connection string, etc.) should be identified and credentials provided.

**Build and Deployment Process:** - For **development**, the front-end can be run with npm start (or npm run dev ). This starts a local dev server (typically on http://localhost:3000 ). If a backend exists, it might run on another port (e.g., 3001) and you’d run it with a separate command (like npm run server or using a concurrently script). - For **production deployment**, you will likely: 1. Build the React app: npm run build . This generates static files (HTML, JS, CSS) in a build/ directory. 2. Deploy these static files to the hosting environment. If using a cloud service (e.g., Vercel, Netlify, GitHub Pages), connect the repo or upload the build directory. Ensure the site is served over HTTPS at the domain configured in Telegram. 3. Deploy/Run the backend: If on a PAAS (Platform as a Service) or your own server, start the Node.js server with npm start (after setting environment variables like the bot token, API keys, contract addresses, etc.). If Docker is used, build the Docker image ( docker build ) and run containers as per provided

|  |  |  |  |
| --- | --- | --- | --- |
| compose files. For instance, there might be a docker-compose.prod.yaml | |  | | --- | | 12 | | to set up the client and |

server. The agent should update any environment variables in those Docker files (like tokens and secrets). 4. **Telegram Webhook (if applicable):** If the bot uses webhooks (BotFather might have a webhook URL for the bot), set it to point to your server (e.g., <YOUR\_DOMAIN>/bot<token> route) and ensure the server’s route is handling incoming messages. Alternatively, the bot could poll messages – see project documentation for how the bot is started. The new agent should verify the bot is running by sending it a test command or by launching the Telegram game.

**Agent’s Deployment Responsibilities:** Going forward, the agent will: - Maintain the hosting environment: ensure the site’s SSL certificate is valid, and the domain remains accessible. Renew or update DNS as needed. - Monitor the webapp performance and availability. Use uptime monitoring for the domain/ endpoint. - Apply updates or bug fixes to the code: after making changes in the repository, rebuild and redeploy the app. - Manage environment configuration: update API keys, contract addresses, or

6

percentages in the code when needed (for example, if royalty splits change or if migrating to new contracts). - If using cloud services (like a server instance or container hosting), keep them secure and updated. - If a **database** is used for storing any game state or user records, manage backups and migrations for that database.

**Codebase Familiarization:** The agent should take time to read through the key parts of the code: -Understand how the game loop works in FlappyGame.jsx , how state like score and highScore are managed, and how the game transitions from playing to game over. - Note any **reward logic** in code – e.g., after game over, does the front-end send a request to the server to credit the player’s wallet? Look for functions or API calls that might indicate that (for example, an axios/fetch call to an endpoint or a direct Toncoin transfer using the wallet). - Check the **configuration files**: .env or similar, to see what values are expected (common ones might be REACT\_APP\_TON\_API\_KEY , BOT\_TOKEN , CONTRACT\_ADDRESS , etc.). Ensure these are documented and provided. - Review the **README.md** (if present) and any documentation files for any additional context (like architecture decisions, known issues, TODOs, etc.).

By fully understanding the codebase layout and deployment process, the agent will be prepared to run the webapp and make iterative improvements.

**Game Logic & Reward Mechanics**

This section highlights the core **game mechanics** and the additional **Web3 reward features** so the new agent grasps how the game operates and what to monitor or tweak.

**Flappy Bird Core Mechanic:** The game itself replicates the well-known Flappy Bird gameplay: - The player controls a bird that automatically falls due to gravity. Each tap/click causes the bird to flap upwards (a sudden upward velocity). In the code, this is modeled by constantly increasing the bird’s downward velocity and resetting it when a jump occurs (gravity is a positive acceleration, jump strength is a negative impulse)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | [2](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L14-L19) | | . For example, gravity might be set around 0.3 (adding to downward speed each frame) and each flap | | | |
| gives an upward velocity of -5 units | | |  | | --- | | [2](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L14-L19) | | . - The bird must navigate through gaps between pipes. Pipes are | |
| generated at intervals and move from right to left across the screen. The gap height is fixed (e.g. 100px) | | | | |  | | --- | | [13](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L22-L28) | |

but the gap’s vertical position is randomized. The code continuously spawns pipe objects and moves them in each frame update. - Collision detection is implemented to end the game if the bird hits a pipe or the

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ground/ceiling. This uses the bird’s hitbox and pipe positions to check overlap on each frame | |  | | --- | | [14](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L280-L289) | | [15](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L291-L300) | . - |

Scoring: Each time the bird successfully passes a pipe (i.e., the bird crosses the vertical line of the pipe gap

|  |  |  |
| --- | --- | --- |
| without collision), the score increments | [16](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L310-L319) | . In the code, there’s a check for when a pipe’s right edge passes |

the bird’s x-position for the first time to increment the score. - Game States: Typically, there are at least three states – idle (waiting to start), playing, and game over. The user might have to click to start the game (transition from idle to playing), and on game over the game stops and shows final score. The component likely calls an onGameOver callback or similar when the game ends, which could trigger other processes (like sending score to backend, or showing a menu). - Difficulty: The game might gradually increase difficulty by speeding up pipes over time. For instance, there could be a speed multiplier in the game loop

|  |  |  |
| --- | --- | --- |
| (some code shows adapting speed to frame rate and possibly capping it) | [17](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L247-L255) | . The agent can adjust |

parameters like pipe speed or gap if needed to tune difficulty.

**Reward Multipliers:** The term “reward multipliers” refers to any logic that increases rewards (points or tokens) under certain conditions. In the current implementation, the score increases by 1 per pipe by

7

|  |  |  |
| --- | --- | --- |
| default | [16](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L310-L319) | . If a multiplier system is planned, it might work as follows: - **Score Multipliers:** e.g., every 10 |

pipes passed without collision could multiply the points earned (or give bonus points). The code does not explicitly show this yet, so it may be a planned feature. The agent could implement this by checking the score and applying a bonus when certain thresholds are hit. - **Token Reward Multipliers:** If the game rewards cryptocurrency (Jettons or Toncoin) for high scores, a multiplier might mean higher payouts for higher difficulty or combo. For example, a base reward of X tokens per pipe could be increased by 2x if the player has an unbroken streak of N pipes, etc. These rules would be defined either in the front-end or the back-end reward logic. - **Power-ups:** It’s also possible “multipliers” refer to in-game power-ups (not explicitly seen in code). For instance, a token or NFT could give a player a score multiplier power-up during gameplay. The new agent should clarify if such mechanics exist or are planned, and if so, ensure the logic for applying multipliers is correctly implemented and tested.

**Tokenized Progression:** The project incorporates blockchain tokens to enhance progression: - Players can **earn tokens** for playing. The design likely uses a **Jetton** (a fungible token on TON) as the game’s currency

|  |  |  |
| --- | --- | --- |
| or reward coin | [11](https://github.com/ton-community/flappy-bird#:~:text=The%20example%20contains%3A) | . For example, each pipe or each game could reward a small amount of |

“FlappyCoins” (just as a concept) which are actually Jettons credited to the player’s wallet. The agent should confirm what token (if any) is used – it could be an existing token or one specifically created for the game.

|  |  |  |
| --- | --- | --- |
| The ton-community example explicitly creates a Jetton as a game currency | [18](https://github.com/ton-community/flappy-bird#:~:text=The%20example%20contains%3A) | . - **Wallet Connection:** To |

facilitate token rewards, the game allows users to connect their TON wallet. The interface might have a“Connect Wallet” button. When connected, the app knows the user’s address and can send rewards there

|  |  |
| --- | --- |
| [19](https://github.com/ton-community/flappy-bird#:~:text=%2A%20Creating%20jetton%20,game%20props%20in%20the%20shop) | . Ensure that the front-end uses a TON wallet integration (Tonkeeper deep link, TonHub, or the Telegram |

Wallet if playing in Telegram) and that it’s tested. The agent should test connecting their own wallet in the game to see the flow. - **Progress tied to Tokens/NFTs:** Certain milestones could be represented by NFTs (non-fungible tokens). For example, the game might award an NFT badge when a player plays their first

|  |  |  |
| --- | --- | --- |
| game, or reaches a score of 5, etc. | [19](https://github.com/ton-community/flappy-bird#:~:text=%2A%20Creating%20jetton%20,game%20props%20in%20the%20shop) | . If such features exist: - Identify the NFT contract and how minting is |

triggered. Possibly the backend mints an NFT (via the bot or server) to the player’s wallet. - These NFTs can serve as proof-of-achievement and might even be used to unlock new game content (e.g., different skins or“levels”). - The agent should check if there is a **“Shop” or in-game purchases** functionality (the example

|  |  |  |
| --- | --- | --- |
| mentions buying game props | [20](https://github.com/ton-community/flappy-bird#:~:text=,game%20props%20in%20the%20shop) | ). If implemented, players might spend the tokens they earned to buy |

cosmetic items or gameplay boosts. This would involve smart contract calls or at least updating some state. The new admin will need to manage the inventory of items and ensure the smart contract (or backend logic) deducts tokens and grants the item.

**Smart Contract Payout Logic:** When the game is connected to blockchain features, there are a few logical flows for payouts: - **Direct Payouts:** The game could directly trigger the payment splitter contract when a payout is due. For instance, if a player wins 10 TON from a tournament or achieves a reward threshold, the application (or a backend script) calls the payment splitter contract’s function to distribute that 10 TON (so a portion goes to the player and a portion to the developers per the royalty settings). The agent should know how to invoke the contract – possibly via a tonweb call or a function call through the connected wallet. -**Periodic Payouts:** Alternatively, the contract might accumulate funds (e.g., from many micro-transactions or from an income stream like advertisements or entry fees) and then the agent (or an automated script) calls a distribute() function periodically. The agent should schedule or manually perform such calls as needed (unless it’s fully automatic upon each payment). - **Player Rewards:** If Jettons are awarded per game, how are they given? One approach: the game’s Jetton master contract could allow **minting** to players up to a certain amount. The backend might hold the authority to mint these reward tokens. So after each game, the server could call the Jetton contract to mint X tokens to the user’s wallet. The agent must ensure they have the keys to the mint authority if that’s the case, or know the limits (maybe each user can only claim via

8

an airdrop mechanism). - **Auto-fly / Paid Feature**: If there’s an **“auto-fly”** feature, it could tie into payouts. For example, maybe players can pay a small amount of tokens to activate “auto-fly” (an automated mode where the bird navigates obstacles on its own) for a short period. This would be a usage of tokens within the game to gain an advantage. The agent should examine if such a mechanism exists: - Look for any reference in the code to an auto-play or AI control. If not present, it might be a planned feature. - If implemented, clarify how it’s triggered. Possibly a button that appears when you have a certain NFT or after watching an ad, etc. - **Auto-fly system notes:** This system would keep the bird afloat without input. It might be implemented by temporarily disabling gravity or automatically sending jump commands. The agent should test any auto-fly mode to ensure it doesn’t break the game balance. If it’s a developer tool (to demo the game), then it might be toggled via a secret key or config.

**Integration of Game Logic and Blockchain:** The agent will be responsible for maintaining the balance between fun gameplay and the crypto features: - Ensure that the game remains skill-based and not easily exploitable even with Web3 elements. For instance, if token rewards are too high or easy to get (or if someone could script the game to farm tokens), it could be problematic. - If any changes are made to game physics or reward formula, consider how that affects the token economy. (E.g., making the game easier would cause more tokens to be won by players.) - Monitor the **high score system** and consider linking it with the rewards: perhaps the highest scorers get extra payouts or a leaderboard NFT. This isn’t explicitly stated, but as an admin, the agent could propose or implement such ideas.

**Session Persistence & Browser Handling**

The current implementation of the game uses basic session persistence for certain data, and the agent might need to extend this as needed.

**Local Storage for High Scores:** The game stores the player’s high score in the browser’s local storage. On

|  |  |  |  |
| --- | --- | --- | --- |
| component mount, it loads flappyHighScore from localStorage and updates the state | |  | | --- | | [21](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L93-L101) | | . When a |
|  | | |

game is over and the score exceeds the current high score, it saves the new high score back to local storage

|  |  |
| --- | --- |
| [22](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L229-L237) | . This allows a returning player (on the same device & browser) to see their record. However, this is |

**client-side only** – if the player clears their cache or moves to another device, the high score will not carry over.

For the agent: - Keep this localStorage feature intact, as it’s a simple way to motivate players with personal

|  |  |  |
| --- | --- | --- |
| bests. The code for this is in FlappyGame.jsx (see the useEffect that calls localStorage.getItem and | | |
| the part where it calls localStorage.setItem ) | |  | | --- | | [22](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L229-L237) | | . - If a more persistent or universal tracking of high |
|  | | |

scores is desired, consider storing scores on a backend or on-chain. For example, you could maintain a leaderboard on the server or even use a TON smart contract to record top scores globally. This would be an enhancement and requires careful design (to prevent cheating, one would need to verify score submissions). - For now, note that **session = browser session** in the current setup. There is no login system apart from wallet connection. Wallet connection itself doesn’t automatically store game state, it just gives an identity. If needed, one could tie the high score to the wallet address and store it in a backend, but that’s not implemented yet.

**Browser Session Handling:** Since the game runs in a browser (either standalone or within Telegram’s in-app browser), there are a few considerations: - The game resets on page refresh. If a player accidentally refreshes or navigates away, their current game is lost (which is typical for games). There is no persistent game state across sessions (except the high score). - **Advice:** Don’t attempt to persist ongoing game state in

9

localStorage (like current score or position) because the game is quick and stateless beyond each round. It’s acceptable that a game in progress cannot be resumed after closing the tab – Flappy Bird rounds are meant to be short. - The agent should ensure that the **localStorage usage doesn’t exceed quotas** (just storing one integer for high score is fine). - **Cookies** are not explicitly used. If the backend sets any session cookies (for example, if a backend keeps track of user sessions or Telegram auth), ensure cookie policies (secure, same-site if needed for Telegram, etc.) are correctly set. For instance, Telegram Web Apps can pass an auth\_date and hash – some apps verify this on backend and might set a session cookie for the user. If that’s the case, confirm the cookie life and security. - If the game ever introduces a **user account system** (say, linking an email or username), then session persistence would involve authentication tokens. Currently, using the Telegram identity or wallet address as the user identifier is the approach. The agent might consider leveraging those: - *Telegram ID:* When the game is launched via Telegram, you get the user’s Telegram ID and name. You can use that to load/save user-specific data on the backend (for example, fetching their total tokens earned or NFTs). - *Wallet address:* Similarly, if a user connects a TON wallet, that address could key their data (e.g., query how many game tokens they have, or store their all-time stats).

**Multi-Device or Multi-Session:** If a user plays on multiple devices, presently their high scores won’t sync. One way to improve this (post-handoff) would be to implement a cloud sync: - A simple method: if the user has connected a wallet, use the wallet address to retrieve a stored high score from a server or contract. The agent can consider adding this so that the high score becomes independent of device. - Alternatively, if using Telegram, the backend could store each Telegram user’s best score in a database. Then every time the WebApp starts (Telegram provides a user unique ID), the app could fetch that and compare to local high score. - These enhancements would require backend endpoints (e.g., /api/highscore?userId=... ) and are beyond the current localStorage-only approach, but are worth noting if persistence becomes important.

**Handling Browser Compatibility:** Ensure the game runs well in modern browsers and in the Telegram in-app browser: - It uses standard Canvas or DOM for graphics (likely <canvas> or moving DIVs for the bird/ pipes). Test on mobile WebView (Telegram on Android/iOS) to verify performance and input (touch events) are working. The code includes event handlers for pointer/touch events, which is good for mobile. - If any issues are found (like scaling on different screen sizes), the agent should address those via responsive design or adjusting constants (the game width/height are set to 390x400 which may need scaling on high-res devices). - **Session timeouts:** If the game or bot uses any session timeout (unlikely in this context, since it’s mostly stateless except for playing), be aware of them. For example, if the Telegram web app has a time limit (some bots require re-login after a certain time), know how to re-init if needed.

In summary, the current session persistence relies on the browser’s local capabilities for a single user on one device. The agent can maintain this and optionally expand it using the available user identity info if a more robust system is desired. At minimum, the agent should confirm that high scores save and load correctly and that wallet connections persist as needed (Tonkeeper/extension might keep the connection alive across refresh depending on implementation).

**Operational Workflow & Next Steps**

To ensure the new agent can run the entire system solo, here is a step-by-step workflow of all major tasks from setup to ongoing operations:

1. **Access the Code Repository & Set Up Environment:**

10

2. Accept the invitation to the GitHub repository <REPO\_URL> and confirm you have proper access (can push to main/master, manage issues, etc.).

3. Clone the repository to your development machine. Install Node.js (the version specified in .nvmrc or documentation, if provided) and run npm install (or yarn install ) to fetch all   
dependencies.

|  |
| --- |
| 4. Review the project README and this guide to ensure all prerequisites (Node, TON tools, etc.) are |

installed. Set up any required global tools (for example, func compiler if using FunC, Docker if needed for deployment, etc.).

5. **Insert Credentials and Configure Environment:**

6. Create a copy of the example environment file (e.g., .env.example ) as .env (if applicable) in both the front-end and backend directories. Fill in the placeholders with actual values:  
 ◦ REACT\_APP\_TON\_CONTRACT\_ADDRESS=<PAYMENT\_SPLITTER\_MAINNET\_ADDRESS> (or testnet address if testing).

|  |  |
| --- | --- |
| ◦ | REACT\_APP\_JETTON\_ADDRESS=<GAME\_TOKEN\_ADDRESS> (if a game token is used for |

rewards).  
◦

|  |
| --- |
| Backend environment: BOT\_TOKEN=<TELEGRAM\_BOT\_TOKEN> , |

|  |
| --- |
| DB\_CONNECTION=<DATABASE\_URL> (if a database is used), PINATA\_KEY , |

|  |
| --- |
| PINATA\_SECRET , etc. |

7. For local development, also configure the wallet or test settings: you might use the testnet values during development. For example, point the app to testnet by setting an API endpoint if needed (some TON SDKs need to know to use testnet vs mainnet).

8. **Do not commit** any .env files with real secrets to GitHub. These should remain local or stored securely (or in GitHub Actions secrets if CI/CD is used).

9. **Run the Application Locally (Dev Mode):**

10. Start the front-end: npm start (within the project’s client directory if monorepo, or root if single package). Verify that the game loads in your browser at http://localhost:3000 and you can play it. Check the console for any errors (especially related to missing config or wallet connectivity). 11. Start the backend/bot (if separate): npm run server or relevant command. Ensure it connects to Telegram (watch logs for “bot started” or similar message). If the bot uses webhooks locally, you might need to run Ngrok and set the webhook to the Ngrok URL for testing.

12. Test core features in dev: play a game until game over, see that high score is stored. If possible, connect your TON wallet in the game and see that the address is recognized. This might be limited on localhost due to Telegram requiring HTTPS – you can test wallet connect in a normal browser setting, but for Telegram WebApp testing, use Ngrok to expose your local server and open the Ngrok URL via Telegram.

13. **Deploy the Smart Contract (if not already deployed):**

14. Using the testnet credentials, deploy the payment splitter contract to TON testnet. Follow the deployment steps from the **Smart Contract** section. Use func or the deployment script, and fund

11

the deploying wallet with test TON. Once deployed, run tests: e.g., send 1 test TON to the contract and observe that it forwarded (e.g., 0.5 to address A, 0.5 to address B if that’s the split).

15. Update the front-end config to use the testnet contract address and switch any network indicators to testnet (if applicable).

16. When ready, deploy the contract to mainnet. Double-check all payee addresses in the code before deploying (they should likely include the project’s revenue address and possibly a community or charity address depending on design). After deploying on mainnet, record the address and set it in the production config.

17. If the game token (Jetton) is not yet created, deploy that as well (the ton-community example’s setup script might handle creating a Jetton master). Ensure the agent has the mint authority for it.

18. **Document** the contract addresses: Add them to this document or a secure vault for quick reference. Example: “PaymentSplitter deployed at <TON\_CONTRACT\_ADDRESS\_MAINNET> on TON mainnet on YYYY-MM-DD.”

19. **Production Deployment of Webapp:**

20. Build the production bundle: npm run build . Ensure the build completes without errors. The output should be ready to serve.

21. Upload or deploy this build to the hosting environment:  
 ◦ If using a cloud service (e.g., Vercel), connect the GitHub repo and configure environment variables in the dashboard (for example, REACT\_APP\_CONTRACT\_ADDRESS , etc.). Then trigger a production build. Confirm the site is live at the intended domain   
 ( <YOUR\_DOMAIN> ).

◦ If self-hosting on a VM or container, transfer the build files or set up the Node.js server to serve them. You might run npm install && npm run prod on the server to start the app (depending on how the server is set up).

◦ Set up **HTTPS**: This is critical. Use Let’s Encrypt or the hosting provider’s SSL solution to get a certificate for <YOUR\_DOMAIN> . Verify that https://<YOUR\_DOMAIN> loads the game. No mixed content errors should be present.

22. For the backend/bot in production:  
 ◦ Deploy the Node.js bot server. This could be on the same host or a separate one. Set the environment variables on the server (never expose the bot token or wallet keys publicly).

Start the service (e.g., using a process manager like PM2 or as a Docker container). Ensure it’s◦   
running continuously.

◦ Set the Telegram bot’s webhook to the production URL ( https://<YOUR\_DOMAIN>/BOT if that’s the path). You can do this via BotFather or via an API call. Confirm Telegram is sending updates (maybe send a /start to the bot and see if your server logs it).

23. Once everything is up, **test the live system**: Open the Telegram game URL (provided by BotFather earlier) on a phone or PC. It should load the webapp within Telegram. Play a round, connect a wallet if possible through Telegram’s own TON wallet, and see that no errors occur. This end-to-end test confirms the integration is successful.

24. **Handoff Verification and Monitoring:**

12

25. The agent should now be effectively running the system. It’s wise to do a joint review with the project owner: walk through the deployed app, maybe a quick code tour, and confirm all credentials have been transferred.

26. Set up monitoring:  
 ◦ Uptime monitor for the domain (so you get alerted if the site goes down).

◦ If the bot is critical, consider a heartbeat or error logging for it. You might integrate with a logging service or at least have it output to a log file.

◦ Blockchain monitoring: You can use a TON explorer or API to track the contract’s balance and transactions. This helps ensure the splitter contract is receiving and sending funds correctly. Any irregularity (like funds stuck in contract) would show up there.

27. Security check: Make sure the **private keys and tokens** are stored securely by the agent. Any local .env files with secrets should be secured. Consider rotating any keys that the previous maintainer had access to (for instance, generate a new BotFather token if ownership of the bot fully transfers, and update the server with it).

28. **Ongoing Operations & Maintenance:**

29. **Solo Development:** The agent now owns development. Follow best practices: use a branching strategy for any new features (create feature branches, merge via PRs for record-keeping even if you’re sole dev), and test changes on testnet/staging before pushing to production.

30. **Community & Support:** If the game has users, the agent might handle support queries. Ensure the Telegram bot or game has a contact method (maybe via the bot or a linked group) so players can report issues. The agent should be prepared to respond and fix bugs as needed.

31. **Updates to Smart Contracts:** If any change is needed in the contract (e.g., adding a new payee or changing royalty rates), note that on TON this typically means deploying a new contract (since contracts are immutable unless specifically designed to be upgradable). Plan such changes carefully: deploy new contract, update addresses in the app, and possibly sunset the old contract (transferring out any remaining funds).

32. **Scaling & Performance:** Monitor if the server can handle the load (if thousands of players join, ensure the infrastructure is adequate). The game itself is client heavy, but if the backend issues tokens or writes to a DB on each game, keep an eye on those components.

33. **Future Enhancements:** With full ownership, the agent can implement new features (perhaps more games modes, leaderboards, etc.). Keep the smart contract addresses and any new contract deployments documented as part of the project’s knowledge base.

Finally, ensure all **documentation** is up to date for the project. This guide should be updated by the agent if anything changes, and the README in the repository should reflect the current setup and any usage instructions. With this handoff complete, the new agent should be equipped to run and evolve the Flappy Bird TON game project independently, providing continuity and hopefully guiding it to further success on the TON platform.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | [1](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L8-L16) | | |  | | --- | | [2](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L14-L19) | | |  | | --- | | [13](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L22-L28) | | |  | | --- | | [14](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L280-L289) | | |  | | --- | | [15](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L291-L300) | | |  | | --- | | 16 | | |  | | --- | | [17](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L247-L255) | | |  | | --- | | [21](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L93-L101) | | |  | | --- | | [22](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx#L229-L237) | | FlappyGame.jsx |

[https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/ FlappyGame.jsx](https://github.com/optimismfanboy/flappy-bird-bar/blob/0938a8cad30ea55e037c9c6a14f0309660d23fd6/src/components/FlappyGame.jsx)

13

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [3](https://github.com/ton-community/flappy-bird#:~:text=Ngrok) | |  | | --- | | [4](https://github.com/ton-community/flappy-bird#:~:text=Running) | | |  | | --- | | [5](https://github.com/ton-community/flappy-bird#:~:text=Telegram%20bot%20%26%20Telegram%20Web,App) | | |  | | --- | | [6](https://github.com/ton-community/flappy-bird#:~:text=Telegram%20bot%20%26%20Telegram%20Web,App) | | |  | | --- | | [8](https://github.com/ton-community/flappy-bird#:~:text=Pinata) | | |  | | --- | | 9 | | |  | | --- | | [11](https://github.com/ton-community/flappy-bird#:~:text=The%20example%20contains%3A) | | |  | | --- | | [12](https://github.com/ton-community/flappy-bird#:~:text=docker) | | |  | | --- | | [18](https://github.com/ton-community/flappy-bird#:~:text=The%20example%20contains%3A) | | |  | | --- | | [19](https://github.com/ton-community/flappy-bird#:~:text=%2A%20Creating%20jetton%20,game%20props%20in%20the%20shop) | | |  | | --- | | [20](https://github.com/ton-community/flappy-bird#:~:text=,game%20props%20in%20the%20shop) | | GitHub - ton-community/flappy-bird |

<https://github.com/ton-community/flappy-bird>

|  |  |
| --- | --- |
| [7](https://www.datawallet.com/crypto/get-ton-testnet-tokens#:~:text=Summary%3A%C2%A0You%20can%20get%20free%20TON,2%20tokens%20every%2060%20minutes) | How to Get TON Testnet Tokens |

<https://www.datawallet.com/crypto/get-ton-testnet-tokens>

|  |  |
| --- | --- |
| [10](https://www.cookbook.dev/contracts/royalty-splitter#:~:text=Royalty%20Splitter%20on%20Cookbook%20This,is%20specified%20is%20by) | Royalty Splitter on Cookbook |

<https://www.cookbook.dev/contracts/royalty-splitter>

14