ChatGPT

Codex Automation System for Next.js Bitcoin Dashboard

This documentation provides a complete setup for an autonomous ChatGPT Codex agent to develop and maintain a lightweight TypeScript/Next.js Bitcoin dashboard web app. The system is designed for minimal infrastructure (no dedicated servers or backends) and uses the GitHub repository itself (files and commit history) to coordinate tasks and preserve context. Below are all the required files, configurations, and examples – each in a copy/paste-ready format – to implement this Codex automation system.

AGENTS.md

```
# Codex Agent Guidelines

This project uses an AI coding assistant (ChatGPT Codex) acting as a junior
developer agent. The following guidelines outline the agent's roles, workflow,
commit conventions, and constraints to ensure safe and efficient autonomous
operation.

## Roles and Responsibilities

- **Developer Agent** – The primary AI agent that implements features, fixes,
and improvements. It reads tasks from `TASKS.md`, writes code or documentation
for each task, and commits changes. It should follow project standards and only
tackle one task at a time.
- **Reviewer Agent** (optional) – An AI agent (or human) that reviews code
changes. It ensures quality, catches errors, and verifies that each task is
properly completed. Code review may be required via pull requests (enforced by
CODEOWNERS).

*(The Developer Agent is the default mode; the Reviewer role may be invoked
separately for quality checks.)*

## Task Execution Workflow

1. **Read Tasks** – Begin by reading `TASKS.md` to identify the next incomplete
task (the first `[ ]` checkbox).
2. **Plan and Implement** – Focus only on that single task. Determine which
files or sections of code need changes. Make concise, incremental changes to
implement the task's requirements.
3. **Update Files** – Apply code changes or create new files as needed. Adhere
to the existing code style and structure of the Next.js TypeScript project (e.g.
```

````
use functional React components, hooks, etc. where appropriate).
4. **Local Verification** – If possible, run checks (type-checking, build, or unit tests) in the repository's context to ensure the change doesn't break the build. Address any errors before committing.
5. **Mark Task Complete** – In `TASKS.md`, mark the task's checkbox as complete (`[x]`) and optionally add a brief note if needed.
6. **Commit Changes** – Commit the code and `TASKS.md` update together. Follow the commit message guidelines below.
7. **Repeat** – Proceed to the next task in `TASKS.md` and continue until all tasks are completed. Do not skip tasks. If a task seems unclear or blocked, pause and seek clarification rather than guessing.

## Commit Rules and Message Format

- **One Task, One Commit**: Each commit should correspond to completing one atomic task from `TASKS.md`. Do not mix multiple tasks in one commit.
- **Structured Commit Message**: Start the commit message with the task identifier and a short summary. For example:
  ```text
  Task 3: Add agent interface and types
  ```
  Follow the summary with a more detailed description in the commit body (after a blank line), if necessary, to explain *what* was done and *why*. Keep the tone factual and concise.
- **Include Context**: In the commit body, mention key changes or important decisions. This helps the commit history serve as a memory log. For instance, note affected modules, reasons for specific implementations, or any limitations. Example:
  ```text
  Task 3: Add agent interface and types

  - Created `types/agent.ts` defining AgentMessage interface and AgentRole type.
  - This establishes a schema for agent communications (developer vs reviewer roles).
  ```
- **Commit Message Length**: Limit the subject line to ~50 characters if possible, and wrap body lines at ~72 characters for readability. Use a blank line to separate the subject and body.
- **Consistent Prefix**: Always prefix with "Task X:" (where X is the task number or a brief label) to trace commits back to tasks. This convention makes it easy to review progress and for the agent to recall context from the git log.
- **No Secrets or Sensitive Data**: Never include sensitive information (API keys, passwords, personal data) in commits. Use placeholders or environment variables (`.env.local`) for any secrets and ensure they are git-ignored.
- **Atomic Changes**: Ensure each commit builds and passes any tests/lint. If the CI pipeline (see `.github/workflows/ci.yml`) fails due to a commit, address the issue in a follow-up commit promptly.
````

```
## Constraints and Best Practices

- **Offline & Deterministic**: Operate within the repository and provided tools only. Do not rely on internet access or external services at runtime (except calling public APIs from the app itself, which is done client-side). The Codex agent should perform all actions through file edits and git commits via the ChatGPT interface.
- **Minimal Compute**: Avoid heavy computations or long-running processes. This Next.js application is lightweight and does not use a custom server – all data (e.g. Bitcoin prices) should be fetched from public APIs on the client side or during build (using Next.js API routes or getStaticProps). Do not introduce new backend servers or databases.
- **File-Based Memory**: Use the repository files and commit history as the source of truth for context. On each new session or task cycle, re-read `AGENTS.md` (this guide), the latest `TASKS.md`, and recent commit messages to recall the project state. Do not assume any conversational memory beyond what's in the repository.
- **Coding Style**: Follow common TypeScript and Next.js conventions. Use proper types, avoid any `any` unless necessary, and keep code clean and commented if complexity is high. If the project has linters or formatters configured (ESLint, Prettier), ensure your changes conform to those (the CI will run checks).
- **Error Handling**: If unsure about a requirement or encountering errors you cannot resolve with given context, do not proceed blindly. Mark the task as needing clarification (e.g., leave it unchecked and include a note in the commit or `signals.json` if applicable), so that a human can intervene.
- **Environment Variables**: Any keys or URLs needed for functionality (e.g. API keys for Bitcoin data) should be referenced via `process.env` and defined in `.env.local` (never committed). Use the provided `.env.local.example` as a template for required env vars.
- **Autonomy and Approval**: The Codex agent operates with a high degree of autonomy on feature branches. However, all changes should be peer-reviewed. The `CODEOWNERS` file ensures that at least one human (the repository owner) reviews pull requests before merging. The agent should create a pull request when a batch of tasks (or the entire list) is complete, rather than pushing directly to main.

## Memory & Context Persistence

The commit history serves as the agent's long-term memory:
- Before starting work each session, the agent should scan recent commits to understand what has been done (e.g., which tasks were completed, any notes about issues).
- Commit messages are crafted to include context, so they double as reminders. The agent can open `TASKS.md` to see remaining tasks and which are done (checked `[x]`).
- A supplementary `signals.json` file is available to store any additional state or signals between runs (if needed). This could include flags like an error indicator or the last completed task ID, but it's optional. The agent can read/
```

```
write `signals.json` for coordination if the need arises.

By following these guidelines, the Codex Developer Agent will behave predictably
and safely, like a diligent junior developer working through the task list under
the supervision of the repository owner and CI checks.
```

TASKS.md

```
# Project Tasks

<!--
This file is used by the Codex developer agent to track work.
Each task is atomic and should be completed and checked off by the agent via
commits.
-->

- [ ] **Task 1:** Create `AGENTS.md` with guidelines for Codex agents (roles,
commit rules, workflow).
- [ ] **Task 2:** Add a `CODEOWNERS` file to enforce that all code changes
require review (assign default owner).
- [ ] **Task 3:** Set up CI workflow (`.github/workflows/ci.yml`) to run build
and lint checks on pull requests.
- [ ] **Task 4:** Implement TypeScript definitions for the AI agent (`/types/
agent.ts` with AgentMessage interface and roles).
- [ ] **Task 5:** Ensure project structure is in place for automation:
  - Create `/lib/agents/` directory (for agent-related modules).
  - Create `/config/` directory (for configuration files).
  - Create `/logs/` directory (for any logs or notes by the agent).
  - (Add placeholder README.md or `.gitkeep` files in each as needed so they
exist in git.)
- [ ] **Task 6:** Add a safe environment config example `.env.local.example` (no
secrets, just placeholders for Bitcoin API keys/URLs).
- [ ] **Task 7:** Add a `signals.json` file to the project with an initial empty
structure (used for storing agent state or flags if needed).
```

CODEOWNERS

```
# Define code owners to auto-assign reviewers for all changes
* @YourGitHubUsername
```

.github/workflows/ci.yml

```yaml
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 18

      - name: Install dependencies
        run: npm install

      - name: Build project
        run: npm run build

      - name: Lint project
        run: npm run lint
```

/types/agent.ts

```ts
/**
 * Defines the structure of messages exchanged with or between AI agents,
 * and enumerates agent roles in the system.
 */

export type AgentRole = "developer" | "reviewer" | "system" | "user" |
"assistant";

export interface AgentMessage {
  /** Role of the entity sending the message (agent or chat role) */
  role: AgentRole | "user" | "assistant";
  /** The textual content of the message */
```

```
  content: string;
  /** Timestamp of the message creation */
  timestamp: Date;
  /** Optional identifier for related task or context */
  taskId?: string;
}
```

Example Directory Structure

```
📦 bitcoin-dashboard-app/   # Root of the Next.js project repository
├── AGENTS.md               # Codex agent guidance and rules (for AI reference)
├── TASKS.md                # List of development tasks for the Codex agent
├── CODEOWNERS              # Code owners for auto-assigning PR reviewers
├── signals.json            # JSON file for agent signals/state persistence
├── .env.local.example      # Example environment variables (no sensitive data)
├── /types
│   └── agent.ts            # Type definitions for agent messages and roles
├── /lib
│   └── agents/             # (Placeholder directory for future agent logic
modules)
│       └── README.md       # (Placeholder file describing this directory)
├── /config/                # (Placeholder directory for config files if needed)
│   └── README.md           # (Placeholder file for directory retention in git)
├── /logs/                  # (Placeholder directory for logs or notes generated
by agent)
│   └── README.md           # (Placeholder file for directory retention in git)
├── /pages/                 # Next.js pages (UI of the dashboard)
│   └── index.tsx           # Example main page (Bitcoin dashboard UI)
├── /components/            # React components for the dashboard UI
├── /public/                # Static assets (if any)
├── /styles/                # CSS/SCSS files
├── /node_modules/          # (Node dependencies - not committed)
└── .github/
    └── workflows/
        └── ci.yml          # GitHub Actions workflow for CI (build and lint)
```

(Directories `lib/agents`, `config`, and `logs` contain placeholder README files or `.gitkeep` to ensure they are tracked by git. These can be populated with actual content as the project grows.)

Example Commit Messages

```
Task 1: Add AGENTS.md with Codex guidelines
- Created `AGENTS.md` describing agent roles, task workflow, commit conventions,
and constraints for the AI developer.
```

```
- This provides the Codex agent with project-specific instructions and rules of
engagement.

Task 2: Add CODEOWNERS for auto-review
- Added a `CODEOWNERS` file that makes @YourGitHubUsername the default reviewer
for all changes.
- Ensures all AI-generated pull requests require human review before merging.

Task 3: Setup CI workflow for build and lint
- Created `.github/workflows/ci.yml` to automatically install dependencies,
build the Next.js app, and run lint checks on every push/PR to main.
- This will catch syntax/type errors or lint issues in AI commits, enforcing
code quality.

Task 4: Define AgentMessage interface and roles
- Added `/types/agent.ts` with an `AgentMessage` interface and `AgentRole`
type.
- Establishes a structured format for messages and roles (developer, reviewer,
etc.), laying groundwork for multi-agent communication if needed.

Task 5: Create base project directories for agents, config, logs
- Created empty `README.md` files in new directories: `lib/agents`, `config`,
`logs`.
- This scaffolds the project for future agent-related modules, config files, and
any logs, and ensures these folders exist in version control.

Task 6: Add .env.local.example for configuration
- Added an example env file `.env.local.example` with placeholders for
`NEXT_PUBLIC_BITCOIN_API_URL` and `NEXT_PUBLIC_API_KEY`.
- Allows developers (or the AI agent) to know what env vars are needed for
Bitcoin data fetching, without exposing real secrets.

Task 7: Introduce signals.json for agent state
- Added `signals.json` with an initial structure for storing agent state or
signals between runs (e.g., last task completed, error flags).
- This file can be used by the Codex agent to persist any needed context or
coordination info beyond commit messages.
```

(Each commit message begins with the task reference and provides a clear summary of changes. The bullet points (here as hyphens) in the body give additional context. This structure ensures that anyone (or any AI agent) reading the git history can understand what was done and why, aiding memory and review.)

Reusable Prompt Templates

Use the following prompt templates to interact with the Codex agent for common scenarios. These can be copy-pasted into the ChatGPT Codex UI to guide the AI:

- **Kick off Automated Task Execution** – Start the agent on the prepared task list:

```
We have a project with a `TASKS.md` outlining development tasks.
Act as a junior developer AI agent and begin working through the tasks one by one.
Open the `TASKS.md` file, identify the first incomplete task, and implement it.
Mark the task as done in `TASKS.md` and commit the changes with an appropriate message (following our commit guidelines).
Continue this process for each remaining task, making a separate commit for each task.
```

- **Focus on a Specific Task** – If you want the agent to do a particular task out of sequence or repeat a task:

```
Task 4 in our `TASKS.md` needs to be addressed next. Please focus only on Task 4 (“Implement TypeScript definitions for the AI agent”) now.
Open the relevant files (e.g., create or update `/types/agent.ts`) and complete the task.
Remember to mark Task 4 as done and commit with a message starting with "Task 4:" and details of the changes.
```

- **Add a New Feature (create new tasks)** – Have the agent plan out tasks for a new requirement and then execute:

```
A new feature request: display the current Bitcoin price on the dashboard page and update it in real-time.
Please add appropriate tasks to `TASKS.md` to implement this feature (such as fetching data from an API, creating a Price component, real-time updates via WebSocket or polling, etc.).
Then, follow the new tasks you added and complete them one by one with separate commits.
```

- **Code Review or QA** – Ask the agent (in reviewer mode) to analyze the changes or ensure quality:

```
Switch to the Reviewer Agent role and review the last few commits.
Check for any bugs, code style issues, or opportunities to improve the implementation.
```

```
If issues are found, list them or create new tasks in `TASKS.md` for the
Developer Agent to address.
```

(In practice, the first template or a variation of it will be used at the start of a session to have the agent consume the tasks list and begin working. Subsequent templates can target specific needs like focusing on a task, extending the task list, or initiating a self-review.)

.env.local.example

```
# Environment variables for the Bitcoin Dashboard app (example values).
# Copy this file to .env.local and fill in real values for local development.
# Do NOT commit actual secrets to the repo.

NEXT_PUBLIC_BITCOIN_API_URL="https://api.coindesk.com/v1/bpi/currentprice.json"
NEXT_PUBLIC_API_KEY=""  # Example: API key for a paid Bitcoin data provider (if
required; leave blank if not used)
NEXT_PUBLIC_DEFAULT_CURRENCY="USD"
```

signals.json

```
{
  "last_task_completed": null,
  "error_flag": false,
  "notes": "This file can store state or signals for the Codex agent between
runs. For example, 'error_flag' could be set true if the last run ended with an
error, or 'last_task_completed' can record the ID of the last finished task.",
  "version": 1
}
```

First Prompt (to initialize Codex)

```
You are an AI Developer Agent working on a Next.js Bitcoin Dashboard project.
The repository contains an `AGENTS.md` with guidelines and a `TASKS.md` with a
list of development tasks.

Please follow these steps:
1. **Initialize** – Load the project context. Open and read `AGENTS.md` to
understand your instructions, then open `TASKS.md` to see the to-do list.
2. **Execute Tasks** – Start with the first unchecked task in `TASKS.md`. Carry
out the required code changes for that task. When done, mark the task as
completed (replace `[ ]` with `[x]` in `TASKS.md`).
3. **Commit Changes** – Commit your changes (code + updated `TASKS.md`) with a
descriptive commit message. Begin the commit message with "Task <number>:"
```

```
followed by a brief summary, and include details of changes in the commit body
as per the guidelines.
4. **Iterate** – Move on to the next task and repeat the process until all tasks
are completed or you reach a task that cannot be completed without further
input.

Remember to work only on one task at a time, and follow the rules in `AGENTS.md`
for coding style, commit structure, and not introducing any disallowed changes.
Use the repository files and prior commits as your context and memory.

Begin now by confirming you have read the instructions and then proceed with
Task 1.
```
