ChatGPT

name: "Post Scheduler Expo App Specification PRP v3" description: | Transform the existing `kamo` MiniKit template into a **cross-platform Expo app** that fully implements the Post Scheduler mini-app for Farcaster and Base.
Building on the previous scaffold, this PRP adopts the Expo SDK and leverages the stickersmash example to provide a polished, Instagram-like image editor and post scheduling workflow on mobile. Users should be able to pick or capture photos, edit them with cropping, filters, and stickers using Expo's `ImagePicker` and `ImageManipulator` APIs, compose captions, schedule posts with a friendly date/time picker, and publish them to Farcaster via Neynar (with optional on-chain triggers on Base).
The document defines goals, deliverables, success criteria, required context, design blueprint, and a dependency-ordered task list to convert the Kamo codebase into a mobile app while retaining backend functionality for scheduling and publishing.

Goal

Feature Goal: Convert the Kamo MiniKit template into a **cross-platform Post Scheduler app built with Expo**. The mobile app should feel native on iOS and Android and replicate Instagram's editing UX using Expo's toolkit. Users will be able to pick or capture a photo, crop/rotate/resize it, adjust brightness and contrast, apply artistic filters, add stickers or overlays, and then compose a caption and schedule the cast for future publication. Scheduling should support off-chain posting via Neynar and optional on-chain triggers on Base. The app must also retain a backend (Next.js or Node) for storing schedules and publishing posts.

Deliverables:

- **Expo mobile project** configured with dependencies for image picking and editing (`expo-image-picker`, `expo-image-manipulator`, `expo-file-system`), date/time picking (`@react-native-community/datetimepicker`), navigation (`@react-navigation/native` or `expo-router`), modal dialogs (`react-native-modal`), Farcaster integration, and backend API interactions.
- **Farcaster mini-app manifest and Base metadata** packaged in the mobile app and served from the backend so casts embed frames correctly on Farcaster [1].
- **Functional screens**:
- `EditorScreen.tsx`: Uses `expo-image-picker` to select or capture photos; implements cropping, rotating, zooming, and filter adjustments via `expo-image-manipulator`; allows adding stickers/overlays (e.g., using an `Image` component with pan/pinch gestures); and returns the edited image URI.
- `ComposeScreen.tsx`: Accepts the edited image, provides a caption input, includes a date/time picker, and toggles for off-chain vs on-chain scheduling. Integrates modals for choosing filters or stickers.
- `ScheduleQueueScreen.tsx`: Displays scheduled posts with thumbnails, captions, and scheduled times; enables editing or canceling schedules.
- **Backend API**:

- `/api/schedule`: A Next.js or Node endpoint (running in the same repository) that accepts requests from the mobile app, validates them, uploads the image to IPFS via Pinata, stores schedule data in Redis or Upstash, and returns a schedule ID.
- `/api/publish`: A cron-protected endpoint that reads due schedules, posts to Farcaster via Neynar, and optionally triggers a Base smart contract for on-chain scheduling【285777756637392†L90-L147】.
- **Utility modules**: Shared code (TypeScript) for interacting with Neynar, Pinata, Farcaster metadata, Base chain, and validation schemas; reusable between the mobile app and backend.
- **Environment configuration**: `.env.example` specifying keys for NEYNAR, Pinata, Redis, Base chain networks, Expo project config (e.g., scheme, deep link), and CRON_SECRET.
- **Tests**: Unit tests for image editing functions (cropping, filters), form validation, API endpoints, and scheduling logic; integration tests for the full mobile-to-backend workflow.
- **Documentation**: Comprehensive README explaining how to install dependencies (`npm install`), start the Expo app (`npx expo start`), run the backend (`npm run dev`), configure environment variables, test the app, and deploy to platforms like EAS (Expo Application Services) and Vercel. Include guidance on mimicking Instagram UX using Expo components.

Success Definition:
A developer can clone the repository, install dependencies, and run the Expo app (`npx expo start`) alongside the backend server (`npm run dev`). The mobile app launches on iOS and Android simulators (or devices), displaying a polished UI that allows users to pick or capture a photo, edit it with cropping/filters/stickers, compose a caption, select a future time, and schedule the post. Schedules appear in a queue screen, and an automated backend job publishes due posts to Farcaster via Neynar (optionally interacting with a Base contract). The editing experience mimics Instagram, the project passes linting and formatting checks, tests run successfully, and the deliverables satisfy all goals and success criteria.

User Persona

Target User: Developers building a Farcaster mini app that enables creators and brands to schedule polished posts with images.
Use Case: The developer uses this baseline to implement the image editor, post composer, scheduling logic, and optional on-chain features without worrying about initial scaffolding or project configuration.

Why

- **Foundation for Feature Development**: Provides a well-structured starting point so that subsequent features (image editing, scheduling, on-chain integration) can be built incrementally without re-architecting the project.
- **Consistency**: Aligns with established patterns and templates to ensure that the PRP agentic workflow remains repeatable and maintainable.
- **Developer Velocity**: Reduces setup time by installing necessary packages and creating placeholder files and functions that follow the planned architecture.

What

At this stage the app should implement **complete cross-platform functionality**, not just scaffolding:

- **Mobile front-end**: Render a polished Expo app with screens for image editing, composing, and scheduling. Include correct Farcaster mini-app metadata and Base manifest (served via the backend) so that casts embed frames correctly [1]. The app should request permissions for camera and media library, handle errors gracefully, and feel responsive on both iOS and Android.
- **Image editing**: Present a full-fledged editor screen using `expo-image-picker` and `expo-image-manipulator`. Users can select or capture a photo, crop and rotate it, adjust brightness/contrast/saturation, apply artistic filters, and add stickers/overlays. Editing tools should be organised in modals or toolbars reminiscent of Instagram's editing flow, enabling intuitive navigation.
- **Post composition**: Provide a compose screen that integrates the edited image, allows the user to input text (respecting Farcaster's character limit), and choose a date/time for scheduling via a native date/time picker. Include toggles to choose off-chain vs on-chain scheduling and modals to adjust filters or stickers. Validate inputs and display helpful error messages.
- **Schedule management**: Display a schedule queue screen showing upcoming posts with thumbnails, captions, scheduled times, and statuses. Users can edit or cancel scheduled posts. The screen periodically refreshes or subscribes to updates via WebSocket or polling.
- **Backend API**: Implement `/api/schedule` and `/api/publish` endpoints that perform real validations using Zod, store schedules in Redis/Upstash, upload images to IPFS via Pinata, and call Neynar for posting at the appropriate time【285777756637392†L90-L147】. When on-chain scheduling is enabled, interact with a Base smart contract (stub until contract is implemented). `/api/publish` must be protected by a cron secret.
- **Utility modules**: Provide working functions for uploading images to IPFS, posting casts via Neynar, generating Farcaster metadata, interacting with Base networks, validating input data, and managing scheduled jobs. These modules should be shared between the mobile app and backend.
- **Manifest and metadata**: Include a `.well-known/farcaster/mini-app-manifest.json` file with real values (name, description, icon URL, actions). Ensure the backend serves this file and sets appropriate headers.
- **Quality and testing**: Ensure the project builds, lints, and runs unit and integration tests cleanly. `.env.example` lists all required variables for NEYNAR, Pinata, Redis, Base networks, Expo scheme, and CRON_SECRET.

Success Criteria

- [ ] The Expo app starts with `npx expo start` and displays the expected screens without runtime errors on both iOS and Android simulators/devices.
- [ ] Farcaster clients recognize the app via proper OpenGraph and `fc:frame` metadata and manifest settings served by the backend.
- [ ] Users can pick or capture an image and use the editor to crop, rotate, adjust brightness/contrast/saturation, apply filters and overlays, add stickers, and preview the changes.
- [ ] The Compose screen accepts a caption, displays the edited image preview, provides a native date/time picker, and validates that the scheduled time is in the future and the caption is non-empty.
- [ ] Users can schedule a post; it appears in the schedule queue with the correct thumbnail, caption, and scheduled time. Users can edit or cancel scheduled posts.

- [ ] `/api/schedule` and `/api/publish` endpoints validate input via Zod and perform the necessary actions: storing schedule data, uploading images to IPFS, and publishing due posts via Neynar (on-chain path stubbed).
- [ ] Utility modules (`farcaster.ts`, `neynar.ts`, `pinata.ts`, `viem.ts`, `validation.ts`) contain working helper functions and type definitions shared by the mobile app and backend.
- [ ] Environment variables include keys for NEYNAR, PINATA_JWT, Base chain IDs, Redis, Expo scheme, and CRON_SECRET, and the app loads them correctly in mobile and backend environments.
- [ ] Linting (`npm run lint`) and formatting (Prettier) produce no errors for both mobile and backend codebases.
- [ ] Unit tests cover the editor logic (including cropping and filters), form validation, API routes, and scheduling functions, with all tests passing.
- [ ] Integration tests confirm that scheduling and publishing flows work end-to-end between the mobile app and backend.

All Needed Context

Context Completeness Check

Before implementing this PRP, ensure that all referenced documentation and files are available. Key context includes:

- **Post Scheduler Implementation PRP** – describes the high-level features, success criteria, and desired code structure for the full mini app [1].
- **Post Scheduler Planning PRP** – provides detailed milestones and dependency-ordered tasks for building the mini app【285777756637392†L90-L147】.
- **MiniKit Template** – the Kamo directory is currently a MiniKit scaffold with demo components, OnchainKit integration, and a README describing environment variables [2].
- **postschedule-appbuild.md and postschedule-vision.md** – offer high-level guidance on integrating TOAST UI, scheduling logic, and Base chain interactions [3] [4].
- **PRP Templates** – `prp_base.md`, `prp_spec.md`, and `prp_task.md` define the structure and expectations for base, spec, and task PRPs. Use them as references for formatting and content density.

Documentation & References

```
# MUST READ
- file: PRPs-agentic-eng/PRPs/post-scheduler-mini-app.md
  section: Implementation Tasks
  why: Provides the features, success criteria, and desired code structure [5].
- file: PRPs-agentic-eng/PRPs/post-scheduler-planning.md
  section: Phase 1 – Core Mini App & Off-chain Scheduling
  why: Lists dependency-ordered tasks for building the mini app
【285777756637392†L90-L147】.
- file: PRPs-agentic-eng/PRPs/templates/prp_spec.md
  why: Template for writing specification PRPs (structure and sections).
- file: PRPs-agentic-eng/PRPs/templates/prp_task.md
  why: Template for writing task PRPs that break down a specification into
```

```
executable tasks.
- file: postschedule-appbuild.md
  why: High-level guidance on integrating the image editor and scheduling logic
[3].
- file: postschedule-vision.md
  why: Provides success criteria and considerations for the Post Scheduler mini
app [4].
- file: kamo/README.md
  why: Describes the MiniKit template configuration and environment variables
[2].
```

Current Codebase Overview

The `kamo` directory is a MiniKit template that includes:

- A Next.js project with `app/` and `app/components/DemoComponents.tsx` demonstrating basic UI elements.
- OnchainKit integration for wallet connections and notifications.
- Minimal pages and components; there is no image editor, post composer, or scheduling functionality.
- `package.json` dependencies including `@coinbase/onchainkit`, `@farcaster/frame-sdk`, `@upstash/redis`, `next`, `react`, `wagmi`, but lacking `tui-image-editor`, `react-datepicker`, Neynar or Pinata SDKs [6].

Desired Codebase Tree (Baseline)

```
project-root/
  mobile/
    App.tsx                       # Entry point for the Expo app
    app.config.js                 # Expo configuration (scheme, icons, deep
linking)
    screens/
      EditorScreen.tsx
# Image editor implementing cropping, filters, stickers
      ComposeScreen.tsx           # Post composer with caption and scheduling
      ScheduleQueueScreen.tsx     # Manage scheduled posts
    components/
      StickerSelector.tsx         # Component for choosing stickers (optional)
    assets/
      icons/                      # App icons and stickers
      images/                     # Default or sample images
    navigation/
      index.tsx                   # Navigation setup (stack or tabs)
    test/
      editor.test.tsx             # Tests for editing logic
      compose.test.tsx            # Tests for compose form validation
```

```
      queue.test.tsx              # Tests for schedule queue interactions
    .env.example                  # Mobile environment variables (NEYNAR,
PINATA_JWT, etc.)
    package.json                  # Mobile dependencies and scripts
  backend/
    api/
      schedule.ts                 # API route to accept schedule requests
      publish.ts                  # API route for publishing (cron)
      ping.ts                     # Health check (optional)
    lib/
      farcaster.ts                # Farcaster metadata helpers
      neynar.ts                   # Neynar API client helpers
      pinata.ts                   # IPFS upload helpers
      viem.ts                     # Base chain helpers
      validation.ts               # Zod schemas for request validation
    pages/
      index.tsx                   # Landing page with manifest and frame
metadata
    .well-known/
      farcaster/mini-app-manifest.json  # Farcaster mini-app manifest
    test/
      api.schedule.test.ts        # Tests for schedule API
      api.publish.test.ts         # Tests for publish API
    .env.example                  # Backend environment variables
    package.json                  # Backend dependencies and scripts
    next.config.js                # Next.js configuration (if using Next.js)
  reference/
    stickersmash/                 # Cloned or copied Expo StickerSmash example
for reference
  README.md
# Project setup, run instructions, and feature overview
```

Known Gotchas & Library Quirks

```
# CRITICAL: Farcaster has no native scheduling; scheduling must be handled via a
backend using delegated signers via Neynar [7].
# CRITICAL: Base chain IDs are 8453 (mainnet) and 84532 (Sepolia).  Always
switch networks explicitly when on-chain scheduling is implemented [8].
# CRITICAL: Images must be uploaded to a public service (e.g., IPFS via Pinata)
because Farcaster clients cannot load local references [9].
# CRITICAL: Mini-app metadata (OpenGraph and `fc:frame` tags) must be included
in page headers and manifest for correct embedding [10].
# CRITICAL: Avoid storing private keys in the client; use signer delegation and
wallet providers for any signing operations [11].
# CRITICAL: At this stage we provide stub implementations only; actual logic
will be implemented in subsequent PRPs.
```

```
# NOTE: Expo's `ImageManipulator` supports basic operations (crop, rotate, flip)
but not advanced filters out-of-the-box; custom filters may require third-party
libraries or shaders.  Be cautious when adding complex filters to ensure
performance on mobile devices.  Sticker dragging and pinch gestures may require
proper configuration of `react-native-gesture-handler` and `react-native-
reanimated`.
```

Implementation Blueprint

Data Models and Schemas

Define basic Zod schemas in `lib/validation.ts` for schedule requests and responses. For example:

```typescript
import { z } from 'zod';

export const ScheduleRequestSchema = z.object({
  text: z.string().min(1).max(320),
  imageUrl: z.string().url().optional(),
  timestamp: z.string(), // ISO8601
  fid: z.number().int().positive(),
  onChain: z.boolean().default(false),
});

export type ScheduleRequest = z.infer<typeof ScheduleRequestSchema>;
```

Implementation Tasks (Ordered by Dependencies)

```
Task 1: INITIALIZE mobile and backend project structure
  - IMPLEMENT: Create an Expo app inside the `kamo` repository using `npx
create-expo-app post-scheduler-mobile` (or similar).  Retain the existing
Next.js backend (`post-scheduler-backend`) for API routes.  Organize the
repository into `mobile/` and `backend/` directories.
  - ADD: dependencies to the mobile project: `expo-image-picker`, `expo-image-
manipulator`, `expo-file-system`, `@react-native-community/datetimepicker`,
`@react-navigation/native`, `@react-navigation/stack`, `react-native-gesture-
handler`, `react-native-reanimated`, `react-native-modal`, `zod`, `axios`, and
`expo-constants`.  Add development tooling like ESLint and Prettier configured
for React Native.
  - ADD: dependencies to the backend: `@farcaster/miniapp-sdk`, `@coinbase/
onchainkit`, `@upstash/redis`, `viem`, `wagmi`, `zod`, `axios`, `dotenv`,
`eslint`, and `prettier`.
  - VALIDATE: Run `npm install` in both `mobile/` and `backend/` directories
without errors; launch the Expo development server to ensure the bare app runs.

Task 2: CLONE and INTEGRATE StickerSmash Example
```

```
  - IMPLEMENT: Clone the `expo/examples/stickersmash` repository (or copy its
relevant files) into a `reference/` folder within the project.  Study its
implementation of image picking, manipulating, and overlaying stickers on
canvas.
  - IDENTIFY:
Components and functions that can be reused for our `EditorScreen` (e.g.,
sticker dragging, pinch-to-zoom, use of `ImageManipulator`).
  - VALIDATE:
Ensure no unused code or images remain after integration; keep the project size
lean.

Task 3: ADD Farcaster mini-app metadata & Base manifest
  - IMPLEMENT: Create a `.well-known/farcaster/mini-app-manifest.json` in the
backend with real values (`name`, `description`, `icon`, `actions`, `version`).
Serve this file via an API route so Farcaster clients can fetch it.
  - UPDATE: In `backend/pages/index.tsx` (if using Next.js) or equivalent,
ensure proper OpenGraph tags and `fc:frame` tags are set.
  - VALIDATE: Cast a link to the backend URL in a Farcaster client to confirm
the frame renders correctly [3] .

Task 4: SET UP environment configuration and app settings
  - IMPLEMENT: Create `.env.example` in both mobile and backend directories
specifying required keys: `NEYNAR_API_KEY`, `PINATA_JWT`, `REDIS_URL`,
`REDIS_TOKEN`, `BASE_CHAIN_ID`, `CRON_SECRET`, `EXPO_APP_SCHEME`, and any others
(e.g., `APP_NAME`, `APP_ICON`).
  - IMPLEMENT: Configure `app.json` or `app.config.js` in the mobile project
with the Expo scheme, deep linking configuration, and icons/splash images.
  - VALIDATE: Running the app picks up environment variables via `expo-
constants` or `react-native-dotenv`.

Task 5: BUILD the Image Editor (`EditorScreen.tsx`)
  - IMPLEMENT:
Create `screens/EditorScreen.tsx` that lets users pick or capture an image with
`expo-image-picker`, then displays it for editing.  Use `expo-image-manipulator`
to apply cropping, rotation, and filter operations.  Implement a toolbar or
modal allowing users to adjust brightness, contrast, saturation, and apply
pre-set filters.  Integrate sticker overlays with drag and pinch gestures
(drawing inspiration from the StickerSmash example).
  - IMPLEMENT:
Provide options to reset edits or discard the image.  Persist the edited image
URI in React state or context so it can be passed to the compose screen.
  - VALIDATE: Users can perform editing operations smoothly; images save to a
temporary directory (`FileSystem.cacheDirectory`) or memory.
  - TEST: Write unit tests for editing functions and integration tests (where
feasible) to ensure editing operations produce expected results.

Task 6: BUILD the Compose Screen (`ComposeScreen.tsx`)
```

```
  - IMPLEMENT:
Create `screens/ComposeScreen.tsx` that displays the edited image preview and
allows the user to enter a caption (with a character limit), choose a scheduled
date and time via `@react-native-community/datetimepicker`, and select between
off-chain and on-chain posting.
  - IMPLEMENT: Use `zod` to validate the form (caption non-empty, date in the
future).  Use modal dialogs (`react-native-modal`) to confirm scheduling or
adjust filters.
  - IMPLEMENT: When the user submits, send a POST request to `/api/schedule`
with the image URI (or upload the image via `pinata.ts` first), caption,
timestamp, FID, and on-chain flag.
  - VALIDATE: Display success/error messages; navigate the user to the schedule
queue on success.
  - TEST: Write tests for form validation and API interactions.

Task 7: BUILD the Schedule Queue Screen (`ScheduleQueueScreen.tsx`)
  - IMPLEMENT: Create `screens/ScheduleQueueScreen.tsx` that fetches scheduled
posts from the backend, displays them in a list or grid with thumbnail previews,
captions, and scheduled times, and allows the user to tap to edit or cancel.
Use `FlatList` for efficient rendering.
  - IMPLEMENT: Provide editing: tap a scheduled post to navigate back to the
compose screen with pre-filled data; allow canceling by sending a DELETE or POST
request to `/api/schedule` with an action flag.
  - VALIDATE: Ensure the queue updates when changes occur; handle empty states
gracefully.
  - TEST: Write tests for queue rendering and interactions.

Task 8: IMPLEMENT backend API routes
  - CREATE: In the `backend/` project, implement `api/schedule.ts` (or Next.js
route) with a POST handler.  It should accept JSON input, validate using Zod,
upload the image to IPFS via `pinata.ts` if necessary, store the schedule in
Redis/Upstash with metadata (fid, caption, image URL, timestamp, on-chain flag),
and return a unique ID.
  - CREATE: Implement `api/publish.ts` with a handler that requires
`CRON_SECRET`.  On each invocation (triggered by a cron job), fetch schedules
due at or before the current time, post them to Farcaster via Neynar, mark them
as published, and, if on-chain is true, call a Base smart contract function
(stub until implemented).
  - VALIDATE:
Use curl or Postman to call these endpoints; verify that schedules are stored
and published accordingly【285777756637392†L90-L147】.  Handle errors gracefully.

Task 9: IMPLEMENT utility modules
  - IMPLEMENT: Shared TypeScript modules in `packages/` or `backend/lib/` for
Farcaster metadata generation, Neynar API calls, Pinata uploads, Base chain
interactions, schedule persistence helpers, and validation schemas.  Provide
both mobile and backend exports where appropriate.
```

```
  - IMPLEMENT: `pinata.ts` with an `uploadImage` function that takes a local
file URI (from Expo's FileSystem) and uploads it to Pinata using the JWT.
  - IMPLEMENT: `neynar.ts` with functions to post casts and read user info.
  - IMPLEMENT:
`validation.ts` with Zod schemas for schedule requests/responses, capturing
mobile and backend fields.
  - VALIDATE: Import these functions in mobile and backend code; ensure
TypeScript types align.  Write unit tests for these helpers.

Task 10: WRITE tests
  - IMPLEMENT: Set up Jest (and possibly Detox or Expo E2E) in the mobile
project and Jest in the backend.  Write unit tests for image editing functions
(e.g., cropping, filter application), form validation, API request/response
handling, and scheduling logic.  Write integration tests that simulate
scheduling from the mobile app and publishing via the backend.
  - VALIDATE: All tests pass locally and in CI.  Cover edge cases (invalid
images, oversized files, network failures).

Task 11: LINT & FORMAT
  - IMPLEMENT: Configure ESLint and Prettier for a monorepo with mobile and
backend projects.  Use React Native and Next.js ESLint configs as appropriate.
Consider Husky to enforce linting on commit.
  - VALIDATE: Running `npm run lint` and Prettier across both projects produces
no errors.

Task 12: UPDATE Documentation
  - IMPLEMENT: Write or update `README.md` explaining repository structure
(`mobile/`, `backend/`), dependencies, environment variables, and instructions
to run mobile and backend together.  Describe how to clone and reference the
StickerSmash example, how to develop editing features, and how to deploy both
apps (e.g., using EAS for mobile and Vercel for backend).
  - INCLUDE:
A section on environment variables, obtaining and storing API keys, and running
cron jobs (e.g., via Vercel Cron or GitHub Actions).  Provide a migration guide
from the previous Next.js scaffold to the Expo project.
  - VALIDATE: Ensure the README accurately reflects the current state and
includes screenshots or screen flows to illustrate the UX.
```

Integration Points & Configurations

```
CONFIG:
  - NEYNAR_API_KEY: Use to call Neynar API for posting casts.
  - PINATA_JWT: Use to upload images to IPFS via Pinata.
  - NEXT_PUBLIC_BASE_CHAIN_ID: Default Base chain ID (84532 for Sepolia in
development).
  - REDIS_URL, REDIS_TOKEN: For future notification and background queue
```

```
features (stubbed).
  - CRON_SECRET: Secret key for protecting the `/api/publish` endpoint.

ROUTES:
  - POST app/api/schedule/route.ts: Accepts schedule requests.  Validates input
and stores (stub).
  - POST app/api/publish/route.ts: Cron job endpoint to publish due posts
(stub).

PUBLIC ASSETS:
  - public/default.jpg: Default image used for initializing the editor
(placeholder).
```

Validation Loop

Level 1: Syntax & Style

```
npm run lint        # Run ESLint across the project; fix any errors.
npx prettier --check .  # Verify formatting; run with --write to fix.
```

Level 2: Unit Tests

```
npm test            # Run (currently empty) test suite to ensure tests compile.
```

Level 3: Integration Testing

```
npm run dev &
sleep 5
curl -X POST http://localhost:3000/api/schedule \
  -H "Content-Type: application/json" \
  -d '{"text":"Test","imageUrl":"","timestamp":"2025-08-14T12:00:00Z","fid":
123}'

# Expect: JSON response with a stub schedule ID.

curl -X POST http://localhost:3000/api/publish \
  -H "CRON_SECRET: your_secret_here"

# Expect: JSON response with { published: false }.
```

Level 4: Creative & Domain-Specific Validation

```
# Ensure the mini-app page renders without errors in a Farcaster client.
# Validate that the manifest and metadata meet mini-app requirements.
# Confirm environment variables are loaded and logged via process.env.* in the
dev server.
```

Final Validation Checklist

Technical Validation:

- [ ] Linting and formatting pass with no errors.
- [ ] The Next.js app builds and runs with stub components and routes.
- [ ] API endpoints accept JSON and respond with stubbed output.

Feature Validation:

- [ ] Placeholder components (`Editor`, `ComposeForm`, `ScheduleQueue`) render correctly.
- [ ] Farcaster metadata and manifest files are present and valid.
- [ ] Environment variable template includes all required keys.

Documentation & Deployment:

- [ ] README explains setup, environment variables, and running instructions.
- [ ] `.well-known/farcaster/mini-app-manifest.json` exists and is referenced in docs.
- [ ] All file paths match the desired codebase tree.

Anti-Patterns to Avoid

- Cutting corners on UX or functionality. The goal is a production-ready app; avoid leaving critical features half-implemented or ignoring validation.
- Committing API keys or sensitive credentials; use environment variables exclusively.
- Storing user or private keys on the client side or in the mobile app.
- Ignoring Farcaster or Base chain requirements for metadata, network IDs, or delegated signers.
- Introducing unnecessary dependencies or packages not outlined in this PRP, especially those that bloat the mobile bundle.
- Copying reference code blindly without understanding; adapt the StickerSmash example thoughtfully and remove unused assets.

---

1 5 7 9 10 11 post-scheduler-mini-app.md
https://github.com/mehulmehul1/post-scheduler-dev/blob/main/PRPs-agentic-eng/PRPs/post-scheduler-mini-app.md

2 README.md
https://github.com/mehulmehul1/post-scheduler-dev/blob/main/kamo/README.md

3 postschedule-appbuild.md
https://github.com/mehulmehul1/post-scheduler-dev/blob/main/postschedule-appbuild.md

4 postschedule-vision.md
https://github.com/mehulmehul1/post-scheduler-dev/blob/main/postschedule-vision.md

6 package.json
https://github.com/mehulmehul1/post-scheduler-dev/blob/main/kamo/package.json

8 post-scheduler-planning.md
https://github.com/mehulmehul1/post-scheduler-dev/blob/main/PRPs-agentic-eng/PRPs/post-scheduler-planning.md