ChatGPT

Audit of Nutri-GPT Assistant Backend (GitHub: `nutri-gpt-assistant`)

Feature Implementation vs. Plan Requirements

Planned Feature (v2.1 & v3)	Implementation Status
Auth & Tiered Access – JWT auth middleware with roles for Free/ Limited/Unlimited plans (and admin)	**Partially implemented.** A JWT-based auth middleware is in place [1]. It verifies bearer tokens and attaches `req.user` (with `id` and `plan` claims). The code checks `req.user.plan` for `"unlimited"` to bypass quotas [2] [3]. However, **no explicit admin role check** is implemented – endpoints meant for admins (dataset upload, settings update) are only protected by regular login auth (any user token can call them) and not restricted to an admin user, which deviates from the plan.
Quota & Credits – Per-user `remainingCredits` tracked, decrement on chat unless plan is unlimited	**Implemented.** The database schema added a `remaining_credits` field to users and a Postgres function `decrement_credit(uid)` [4] [5]. A `quota` middleware checks the user's remaining credits via Supabase on each request and calls the RPC to decrement it [6] [7]. If credits are 0, it returns `403 QUOTA_EXCEEDED` [3]. Unlimited-plan users skip the credit check/deduction entirely [8]. **Gap:** The JWT token itself is not reissued with updated credits – the system always queries the DB for current credits, which is fine, but the plan's idea of embedding `remainingCredits` in JWT is not realized.
Stripe Billing Webhook – Handle subscription checkout events to upgrade user plan & credits	**Partially implemented.** A Stripe webhook endpoint exists (`POST /api/billing/webhook`) to process `checkout.session.completed` events [9]. It reads `event.data.object.customer` (Stripe customer ID) and `metadata.plan` ("limited" or "unlimited") from the Stripe event [10]. It then updates the user's `plan` and sets `remaining_credits` (1000 for limited, `null` for unlimited) where `stripe_customer` matches the event's customer ID [11]. This covers applying the purchased plan. **Gaps:** There is no code to create Stripe checkout sessions or to ensure each user's `stripe_customer` ID is set in the database – presumably this must be handled elsewhere or manually. Also, the webhook blindly overwrites credits (e.g. resets to 1000 for limited) rather than accumulating; repeated purchases or subscription renewals might not increment credits. The code doesn't handle other Stripe events (e.g. subscription renewal invoices) beyond ignoring non-checkout events [12] [13].

Planned Feature (v2.1 & v3)	Implementation Status
Dataset File Upload to Supabase – Admin-upload of reference data files (with 10 MB size cap)	**Mostly implemented (user-scoped, missing size check).** There is an endpoint `POST /api/datasets/upload` that saves a file to a Supabase Storage bucket and records it in a `datasets` table [14] [15]. It expects a `filename` and base64 `fileData` in the request body [16], and it stores the file under a path namespaced by the user's ID, then inserts a DB record [17] [15]. The response returns the new dataset's ID, filename, URL, and timestamp [18]. **Gaps:** The plan's 10 MB file size guard is **not enforced** – there's no check on `fileData` length, so a large upload could exhaust memory or storage. Also, this endpoint is not restricted to admins; any authenticated user can upload (the code does require auth and even applies the quota middleware [19], meaning uploads consume a credit, though the plan didn't specify that). There is no validation of file content (only filename and data presence).
Assistant Settings Persistence – Save & retrieve configurable prompts (strict/creative) and params	**Implemented (per-user instead of global).** The app provides `GET/PUT/PATCH/DELETE /api/assistant/settings` endpoints (mounted at `/api/assistant/settings` and requiring auth) [20] [21] [22] [23] [24]. These use a Supabase table `assistant_settings` (keyed by user) to store each user's custom `strict_prompt`, `creative_prompt`, task lists, and temperature settings [25]. On GET, if no record exists for that user, default values (null or presets) are returned [26]. PUT upserts the settings for the user (inserting or updating the row) [27], PATCH updates specific fields, and DELETE resets (deletes the row) [28]. **Deviation from plan:** In the roadmap v3, these settings were envisioned as **admin-managed global prompts**. The implementation instead treats them as user-specific preferences (each user can have their own custom prompts), and no admin check is required to modify them. This is a functional choice difference, but not necessarily a bug – it adds flexibility, though it means there's no single "global" prompt repository as originally described.
Centralized Error Handling – Uniform error responses and logging	**Implemented.** A catch-all error middleware is registered last in the Express app to log the error and return HTTP 500 with a JSON error message [29]. In practice, many routes also have their own `try/catch`. For example, the chat route catches exceptions and returns an `'UPSTREAM_ERROR'` for OpenAI failures [30], and other routes return generic `{ error: "Internal server error" }` on exceptions [31] [32]. All unhandled errors would flow into the central handler (which uses a generic `"INTERNAL_SERVER_ERROR"` message) [29]. Logging is done via `console.error` in both the route catches and central handler. This satisfies basic stability requirements, though there is no more granular error differentiation beyond the codes above, and no external logging service integration.

Planned Feature (v2.1 & v3)	Implementation Status
Unit Tests & Coverage – Automated tests for auth, quota, upload, billing, settings, etc.	**Partially implemented.** The repository includes a test suite using Jest (`/tests/` directory). There are tests covering the chat endpoint behavior (modes, JSON vs. text output, allergen guard, missing fields) [33] [34] [35], the quota middleware (credits allowed vs. exhausted vs. unlimited) [36] [37] [38], the settings routes (get defaults vs. existing, create/update settings) [39] [40], and the Stripe webhook (valid vs. invalid signatures and event types) [41] [12]. These tests confirm that core logic works as expected (e.g. returning 422 on allergen conflict [42] [43], 403 on no credits, etc.). **Gaps:** Tests for the dataset upload route were not found – the upload functionality might not have dedicated test coverage, which is a notable omission given its complexity. Also, no standalone tests for the auth middleware (e.g. ensuring `auth(false)` allows through when token missing) are present, though auth is indirectly exercised in other tests via requiring a valid token. Overall unit test coverage is decent for chat/quota/settings/billing, but a few endpoints are untested.

Architecture Alignment with Roadmap

The backend's overall architecture and tech stack align well with the planned **v2.1/v3 design**. Notable points of alignment:

- **Node.js + Express stack:** The project is structured as a Node/Express application, as intended [44]. The folder layout (server/routes, server/lib, etc.) matches the plan's scaffold for v2.1 [45] [46]. Key components like `openaiClient.js` for OpenAI integration, `buildPrompt.js` for mode-specific prompt construction, and `guardRails.js` for post-checks are all present and used exactly as designed [47] [48] [49].

- **Dual chat modes & JSON schema:** The strict vs. creative mode behavior is correctly implemented. The code uses an OpenAI function call with the defined `MealPlanSchema` when `mode === "meal_plan_strict"` to enforce structured JSON output [50] [51]. For `goal_motivation` mode it provides a more conversational system prompt and higher temperature [52] [53]. This matches the spec which required mode-aware output discipline (structured JSON vs. free-form) [54] [55]. The function-calling approach ensures the meal plan JSON adheres to schema, fulfilling the "valid MealPlanSchema" contract.

- **Supabase integration:** The plan called for using Supabase (Postgres) both for persistent data (user quota, settings, datasets) and file storage [56]. The implementation indeed uses Supabase's JS client (`server/lib/supabase`) to perform DB operations and uploads to storage. For example, the quota middleware queries the `users` table and invokes a Postgres RPC via Supabase JS SDK [57] [7], and the dataset route uses `supabase.storage.from('datasets').upload(...)` to store files [58]. The DB schema migration aligns perfectly with the roadmap: the `users` table is extended with `plan` and `remaining_credits`, and new tables for `datasets` and `assistant_settings` are created [4] [59] [60], exactly as listed in the v3 plan [56]. This shows strong alignment between the intended data model and the actual implementation.

- **Deployment and configuration:** The codebase includes a `/api/healthz` endpoint for uptime checks [61], consistent with using it for health monitoring. Environment variables like `JWT_SECRET` and `STRIPE_SIGNING_SECRET` are used as expected for security [62]. The presence of a `SIGTERM` and `SIGINT` handler in `index.js` for graceful shutdown [63] [64] is a good practice, ensuring alignment with a production environment (likely Replit autoscaling or similar). The plan's mention of p99 ≤ 3s @ 25 RPM and autoscale is more about performance; this isn't directly verifiable in code, but the non-blocking design (async/await and external API calls) is consistent with achieving those targets.

Areas of misalignment or deviation:

- **Admin authorization model:** The plan explicitly noted some endpoints as "admin only" (dataset upload, assistant prompt updates) [65]. The code, however, does not implement an admin role or any differentiation beyond the `plan` field. Currently any authenticated user can hit those endpoints. This is a significant deviation in terms of security assumptions. If the intention is to restrict those actions to privileged users, the implementation will need an update (e.g. a flag in JWT or a user role field).

- **Token-based auth vs. Supabase Auth:** The system uses a custom JWT secret and manual auth middleware, whereas using Supabase Auth (which issues its own JWTs with a built-in `authenticated` role) might have been expected. The approach is fine, but it means the app is somewhat "rolling its own" auth. The token is expected to carry `id` and `plan`, which implies the token must be custom-issued by the app or configured via Supabase JWT claims. Currently, there's no endpoint to log in or obtain such a token in this repo. This could be an omission (perhaps handled outside the repo), but it's a gap in alignment – the plan assumed an existing bearer token mechanism and possibly a single shared token for Replit in v2.1 [66] [67], whereas v3 moves to user-specific JWTs. The code is in between: it expects user-specific JWTs but doesn't show how they're created.

- **Lack of rate-limit 429 response:** The plan's error contract mentioned HTTP 429 for rate limiting (perhaps if Supabase triggers or policies enforced it) [68]. In the implemented code, hitting quota returns 403 `QUOTA_EXCEEDED` (not 429) [69], and there's no other rate limit logic. This is a minor mismatch in semantics (403 vs 429) and suggests no explicit rate-throttling besides the credit system.

- **Test coverage vs. milestones:** The roadmap milestones R1–R5 correspond to implementing auth, quota, webhook, upload, settings with tests [70]. Most of these have tests, but as noted, the upload route lacks a dedicated test, which means R4 ("jest upload.test.js green" [71]) may not have been fully realized. It's possible this was an oversight or the test exists in a different form not easily found. All other milestones (auth, chat mode, quota, settings) have passing tests, aligning with the plan's done criteria of "all endpoints pass unit tests" [72].

In summary, the backend's design is largely consistent with the planned architecture. The core features (multi-mode chat, credit-based usage tiers, Stripe integration, persistent settings) are either in place or in progress. The primary misalignments are in **access control** (admin vs. user) and some missing auxiliary pieces (token issuance, file size limits, etc.), which we detail in the next section.

Gaps and Missing Functionality

Below is a list of functionality gaps or deviations, with an assessment of their impact:

- **Missing Admin Privileges** (**High Severity**): Endpoints intended for admin use are not actually restricted. For example, `/api/datasets/upload` and `/api/assistant/settings` updates can be invoked by any authenticated user (no check on user's plan or role) [73] [22]. This is a security issue if certain features (like uploading reference datasets or changing prompt templates) should be limited to internal or paid users. Implementing role-based auth (e.g. an `isAdmin` flag or treating `plan: 'unlimited'` as admin) is needed to fill this gap.

- **No User Authentication Flow** (**High Severity**): The system expects JWTs but provides no API to log in or obtain them. There's no `/login` or Supabase auth integration shown. Without integration, new users cannot easily get a valid `JWT_SECRET`-signed token to use the API. If Supabase Auth is being used behind the scenes, the code would need to use the Supabase JWT secret and include user claims. This gap could block any new user from actually using the system unless tokens are manually issued, so it's critical to address (either by documenting that Supabase handles auth or by adding endpoints for login/signup).

- **Stripe Customer Linking & Purchase Flow** (**Medium Severity**): While the webhook updates user records, the flow to create a Stripe Checkout Session and record the `stripe_customer` ID for a user is missing. Without it, the webhook can't find which user to upgrade (unless the Stripe customer ID was manually added to the user row). This is a functional gap – implementing an endpoint like `POST /api/billing/checkout` to initiate a payment (returning a Stripe session URL or ID) and creating/linking the Stripe customer to the user would complete the billing integration. Also, if the pricing model involves recurring subscriptions, handling subscription renewal events (e.g. `invoice.paid` events to re-up credits each billing cycle) might be needed down the line – currently such events are ignored [12].

- **File Upload Size & Safety Checks** (**Medium Severity**): The dataset upload endpoint does not enforce the 10 MB file size limit from the plan, nor any type validation. This means a malicious or careless user could attempt to upload a huge file (dozens of MBs in base64 JSON), potentially exhausting API memory or storage. The absence of `multipart/form-data` handling (it expects base64 in JSON) also means the entire file is read into memory via `Buffer.from(...)` [58], which can crash the process for very large files. This needs a safeguard (check `fileData` length and reject >10 MB with 413 or 400 error). Additionally, there's no scanning for content (which might be okay for now, but admins might want to ensure no personally identifiable or dangerous content is uploaded).

- **Content Moderation & Abuse Prevention** (**Medium Severity**): Beyond allergen/medication conflicts, the assistant has **no content filter**. Users could potentially prompt it with inappropriate or harmful requests outside nutritional scope. Production-grade AI apps usually integrate content moderation – e.g., OpenAI's free moderation endpoint can detect and flag hateful, self-harm, or violent content [74]. Nutri-GPT currently lacks such filters. Likewise, there's no mechanism to detect prompt injection or misuse (the model instructions are somewhat fixed per mode, but a crafty user could still try to manipulate it). Prompt abuse or off-label use isn't checked, representing a potential risk (the assistant might give medical advice or other disallowed content without checks). Addressing this would involve adding a middleware to run inputs (and possibly outputs) through a moderation API or a custom filter list.

- **Rate Limiting & Request Throttling** (**Low/Medium Severity**): The credit quota system provides a basic **usage limit**, but there is no short-term rate limiting (e.g. requests per minute). A user with credits could still spam the API with rapid requests. Industry practice is to enforce rate limits per minute or per IP to prevent abuse and manage cost spikes [75]. Implementing such limits (e.g. using an in-memory counter or using an API gateway rule) would harden the service. Also, the app returns `403` for exhausted credits – using `429 Too Many Requests` in such cases (or when adding rate throttle) would be more standard. This is a moderate concern; at low scale it may not matter, but as usage grows, not having any burst control could impact stability.

- **Logging and Monitoring** (**Low Severity**): Aside from console logs and the health check, there's no dedicated logging of user activity or error monitoring service integrated. For example, each chat request and response is not saved anywhere. Lack of audit logs means troubleshooting or analyzing usage (e.g. to see why a response was malformed or to gather feedback) is harder. Moreover, critical errors are only printed to console – in a production deployment, one would typically pipe these to an external monitor (like Sentry or CloudWatch). The current approach meets minimal requirements but leaves a visibility gap. Introducing structured logging (with user ID, timestamps, etc.) and an error alerting system would be valuable as the platform matures.

- **Test Coverage Gaps** (**Low Severity**): As noted, the dataset upload route has no test, and the auth middleware isn't directly tested for edge cases (e.g. invalid token structure). While not immediately user-facing, these untested parts could hide bugs. For instance, if Supabase returns an error during file upload, does the code handle it gracefully? (It does check `uploadError` and `dbError` and returns 500 with details [76] [77], which is good). Writing tests for file upload (perhaps with a small base64 string) and for auth required vs optional routes would shore up confidence and catch regressions in these areas.

- **Minor Implementation Quirks** (**Low Severity**): There are a few minor issues in code that are not breaking anything now but are worth noting. For example, duplicate route mounting (`server/app.js` vs `server/server.js`) – it appears `server/server.js` supersedes `app.js` by including all routes and middleware [78], and `index.js` uses `server/server` [79], so `app.js` might be a leftover. This could confuse new contributors. Also, the Stripe webhook sets `remaining_credits: null` for unlimited plan [11]; this relies on code treating `null` as effectively infinite. The quota middleware doesn't actually handle a null value explicitly – it fetches `remaining_credits` [57] and checks `< 1` [69]. If `remaining_credits` is `null`, the `< 1` check might throw or be truthy in unexpected ways. In practice, it likely comes through as `data: { remaining_credits: null }` and that comparison would fail (probably causing an error or always allowing?). This edge case could be handled more cleanly (e.g. set unlimited users' credits to a very large number or make the query condition `remaining_credits IS NOT NULL`). These issues are not immediate show-stoppers but should be cleaned up to avoid bugs as the codebase grows.

Industry Comparison – Missing “Production-Grade” Features

Beyond the immediate roadmap, Nutri-GPT’s current backend is missing several features commonly found in mature AI assistant platforms (such as OpenAI’s own ChatGPT API stack, Jasper.ai, Copy.ai, etc.). Implementing these will help the project catch up to industry standards:

- **Administrative Tools & Analytics:** In production services, administrators typically have **dashboards or APIs for monitoring usage** and managing users. Nutri-GPT lacks any admin UI or endpoints for this. For example, Jasper’s admin role can view **billing, settings, and usage statistics** for the account [80]. We recommend adding features like:
- **Usage logs and metrics:** Track number of messages, tokens consumed, daily active users, etc. This could be as simple as a “usage” table that records each chat request (timestamp, user, tokens used) or integration with an analytics service.
- **Manual credit management:** Provide an admin-only endpoint or console script to adjust a user’s remaining credits (for support or compensation cases). Right now, credits can only change via Stripe webhook; admins cannot top-up or deduct credits easily.
- **User management:** If using a database of users, allow listing users, banning users, or toggling roles (e.g. promote a user to admin or give someone free unlimited access). Currently, none of these controls exist in-app.

- **Billing dashboard:** Although Stripe handles payments, exposing some info (like a list of subscriptions or last payment date per user) to admins can be useful. At minimum, ensure there’s a way to correlate Stripe customers to app users (possibly by storing `stripe_customer` as done, and maybe the subscription status).

- **Safety & Abuse Prevention:** As mentioned in gaps, content moderation is a key missing piece. **Production AI assistants integrate safety nets** to prevent misuse. Some recommendations:

- **Content Moderation API:** Leverage OpenAI’s Moderation endpoint (which is free and fast) to screen user prompts and AI outputs for disallowed content [74]. For example, if a user asks for medical advice beyond nutrition or uses hateful language, the system should refuse or filter the response according to a policy. Right now, Nutri-GPT would attempt to answer any prompt not containing an allergen. Adding a middleware before the OpenAI call to check the prompt against moderation categories (hate, self-harm, violence, sexual, etc.) and respond with an error or safe completion is advisable.
- **Prompt filtering & sanitization:** Define a set of banned or discouraged inputs. For instance, ensure the user is actually asking about nutrition – if not, perhaps warn or refuse. This could protect against prompts that could lead the model off-track or into problematic territory.
- **Rate limiting & spam prevention:** (As discussed, a per-minute limit). Additionally, consider **CAPTCHA or email verification** for sign-ups (if an open signup model is planned) to prevent bot abuse. Currently, none of these are in place.

- **Monitoring for model misuse:** In admin tools, it’s useful to have logs or alerts for when the model returns a potentially sensitive output (the moderation API can flag content – those events should be logged and reviewed). Nutri-GPT's allergen guard is a domain-specific safety check, which is good, but general AI safety is missing. Implementing a broader safety framework will be important for public-facing deployment.

- **Reliability & Infrastructure Stability:** To operate at scale, the backend would benefit from:

- **Robust error logging:** Instead of just `console.error`, integrate an error tracking service. For example, use Sentry or a similar service to capture exceptions (with stack traces, request info, user ID) whenever a 500 occurs. This will help diagnose issues in production that may not be caught in tests.
- **Performance monitoring:** Track response times and throughput. Replit autoscaling might handle scaling automatically, but having insight into latency (whether the 3s p99 target is met [72]) is useful. This could be as simple as logging response time per request or using an APM tool.
- **Uptime monitoring:** The `/api/healthz` endpoint exists [61] – ensure it's hooked into an external uptime checker. Many services will ping this endpoint periodically and alert if it's down. This isn't a code change but an ops setup; still worth mentioning.
- **Graceful degradation and retries:** In case the OpenAI API is slow or returns errors, consider implementing a retry mechanism (perhaps retry once on failure, since some OpenAI errors are transient) or queueing system for high load. Presently, a failed OpenAI call immediately returns a 500 `UPSTREAM_ERROR` [81] [82]. In a polished product, one might catch specific errors (e.g. timeouts vs. invalid requests) and handle them (maybe return a friendly message to user or try an alternative approach).

- **Scalability considerations:** If usage grows, using Supabase (a hosted Postgres) is fine, but keep an eye on rate limits for Supabase itself (the current design makes a DB query for every single chat request via the quota middleware [57] – at high QPS that could be a bottleneck or cost issue). Caching the `remaining_credits` in memory for short spans or using a local counter might reduce DB load. Similarly, file uploads should stream to storage instead of buffering entirely in memory for scalability.

- **CI/CD and DevOps Practices:** The project would benefit from some enhancements in its development workflow:

- **Automated Testing Pipeline:** Set up GitHub Actions (or Replit CI) to run the `npm test` on each commit/PR. This ensures that the tests (which are currently mostly green) stay that way and catch regressions. It appears no CI is configured yet (no YAML workflows in the repo).
- **Code Style and Linting:** There is a `.prettierrc` file for consistent formatting and some evidence of ESLint fixes in the commit logs. Enforcing lint rules (e.g. via an npm script or CI check) would maintain code quality. Minor issues like the potential null handling in quota could be caught by static analysis.
- **Continuous Deployment:** The plan mentions auto-deploy on push (likely using Replit's integration) [83]. It would be wise to ensure this pipeline only deploys after tests pass. Incorporating a staging environment for testing new features (especially around billing or external API calls) before production deploy is also a common practice Nutri-GPT could adopt as it grows.
- **Webhook testing and security:** The Stripe webhook is currently not authenticated besides signature verification. This is correct practice (using Stripe's signing secret). It's good that the test covers invalid signatures [84] [85]. To be extra safe, ensure the route is not exposed to anything but

Stripe (e.g. if behind a firewall or using a secret path). Also consider end-to-end testing of the webhook with Stripe's test mode (right now it's unit-tested with mocks, which is fine).

In comparison to established AI assistants, Nutri-GPT has the core functionality for its niche (nutrition coaching) but lacks many "enterprise" or **polish features**. For example, Jasper.ai provides team management and brand settings on top of the AI; OpenAI's own platform enforces strict content policies and provides tooling around that. Nutri-GPT should prioritize building out admin controls and safety mechanisms as outlined above. These will not only close the gap with industry standards but also ensure the platform is robust and trustworthy for users.

Recommendations and Next Steps

Based on the audit findings, here is a list of recommended next steps to align the Nutri-GPT assistant backend with its roadmap and industry best practices:

1. **Implement Role-Based Access Control (RBAC):** Introduce an admin role. This could be done by adding an `is_admin` boolean in the users table or designating a plan value (e.g. plan `"admin"` or a separate claim) for admin users. Update the auth middleware to set `req.user.isAdmin` from the token claims, and then guard the dataset and settings routes (and any future admin functions) to allow only admins. This closes the unauthorized access gap for sensitive endpoints immediately.

2. **Provide an Authentication Mechanism:** If using Supabase Auth, configure it so that the JWT it issues includes the custom claims (plan and remaining_credits) – Supabase allows JWT triggers or claim functions for this. Alternatively, create a simple `/api/login` route that verifies user credentials (if any user management exists) and returns a signed JWT containing the user's ID, plan, and credits. This way, users can actually obtain a token to use the API. In the interim, for testing, documentation should note how to manually get a JWT (currently tests sign their own token with `JWT_SECRET` [86], but real users would not do that).

3. **Complete the Stripe Integration:** Develop an endpoint such as `POST /api/billing/checkout` that the frontend can call to initiate a Stripe Checkout Session for a user's upgrade. This endpoint should create a Stripe customer (if not already stored for the user) and a checkout session with the appropriate plan metadata, then return the session URL or ID. Store the `stripe_customer` ID in the user record so that the webhook can match it. Also handle edge cases: if a user already has an active subscription, decide whether to prevent creating another, or allow multiple top-ups (if the model shifts to one-off credit purchases). Additionally, test the full flow in Stripe's test mode (ensure that when a test checkout completes, the webhook updates the user as expected). Over time, consider handling subscription renewals by listening to `invoice.payment_succeeded` events – e.g., if plan is limited and subscription renews monthly, you might want to reset or increment credits.

4. **Harden the File Upload Endpoint:** Add input validation for `/api/datasets/upload`. At minimum, check the `fileData` size (after base64 decoding) and reject files above 10 MB (or some configured limit) with a clear error (HTTP 413 Payload Too Large or 400 with a message). Also, verify the file extension/ MIME type if certain types are expected (the code currently uses a generic `application/{ext}` content type [58], which is okay given various data files). You might also implement streaming uploads instead of buffering – e.g., accept multipart form data to stream directly to Supabase storage. If the use-case is admins uploading large datasets (like USDA nutrition

database), streaming will be more memory-efficient. Finally, consider whether these datasets should be accessible to all users – currently the files are uploaded with a public URL [87]. If some datasets are meant to be private or only used by the AI internally, adjust the storage bucket's privacy settings or do not expose the URL in the response.

5. **Integrate Content Moderation:** To safeguard the system, integrate OpenAI's Moderation API or a similar content filter. This can be done by calling `openai.Moderation.create()` on the user's prompt (and possibly on the AI's draft response) before finalizing the chat response. Define a set of rules: e.g., if moderation flags the prompt/response as hateful, sexual, self-harm, or violence (according to OpenAI policy categories), then refuse the request with an error like `{ error: "CONTENT_NOT_ALLOWED" }`. Because the moderation endpoint is free and fast, this won't add significant cost or latency. This feature will protect both the users (from getting unsafe output) and the platform (from being used for disallowed purposes). Document these usage policies in the README or docs as you enforce them.

6. **Add Rate Limiting Controls:** Implement a basic rate limit to prevent abuse even if credits are available. For example, limit each user to, say, 5 requests per minute on the `/api/chat` endpoint (this is arbitrary; choose a sensible number). You could use an in-memory counter or a small in-memory cache keyed by `req.user.id` (or by IP for unauthenticated routes like if you later allow some open access). On each request, increment the count and if it exceeds the threshold, respond with HTTP 429 `{"error": "TOO_MANY_REQUESTS"}`. Reset the counts every minute. Libraries like `express-rate-limit` could help, or since you have Redis via Supabase (if enabled) you could use that for a distributed counter. Given the expected scale (not extremely high yet), an in-memory solution might suffice initially. This will prevent scenarios like a user scripting rapid-fire requests to exhaust their 1000 credits in a minute (which could spike your OpenAI API usage suddenly). It also helps guard against denial-of-service attacks.

7. **Enhance Monitoring & Logging:** Introduce more robust logging. For instance, log each chat interaction result (could be as simple as `console.log` statements now, but structured) including user ID, mode, whether it was successful or errored, and tokens used (if OpenAI API returns usage info). Over time, pipe these logs to a file or external service. Setting up Sentry (just an npm package and DSN config) would automatically capture exceptions with stack traces; this is very useful for catching issues in production that weren't seen in testing. Also ensure the `console.error` in the central error handler [29] doesn't just vanish in a deployed environment – on Replit, console output is saved, but using a persistent logging solution would be more reliable. Monitoring-wise, use the health check in an uptime service (if not already). If possible, track response times for each request (you can add middleware to record `Date.now()` at start and end). These data will inform future optimizations and ensure you meet performance SLAs.

8. **Improve Testing & CI:** Augment the test suite to cover remaining areas:

9. Write a test for the dataset upload (simulate a small file upload and verify it calls Supabase correctly and returns the expected JSON structure). You can mock `supabase.storage.upload` similar to how other tests mock supabase functions [88]. Also test the 400 error when `filename` or `fileData` is missing [89].
10. Add tests for the auth middleware behavior: e.g., when a valid token is present vs missing vs invalid signature. This can be done by calling a dummy route that uses `auth(required=true)` and

checking responses. Similarly test `auth(false)` if you intend to allow optional auth on any future routes.

11. If moderation is added, test that certain disallowed content yields the expected error.
12. Set up GitHub Actions to run these tests on each push. A simple Node CI workflow that installs deps and runs `npm test` will prevent broken code from being merged. Given that the project already has a `jest.setup.js` and uses standard commands, this should be straightforward.
13. Optionally, add an ESLint config and run `npm run lint` in CI as well. This will catch undefined variables or common mistakes early.

14. Aim for near 100% coverage on critical logic (auth, quota, chat, billing). This ensures confidence in any refactoring or new features.

15. **Documentation and Config Updates:** Update the README and docs to reflect the new features as they are added:

16. Include usage examples for the new admin-only endpoints (once secured) and clarify who should use them.
17. Document the moderation rules and any rate limits so users know the boundaries (for instance, if you implement content filtering, state that certain requests will be rejected by design).
18. If a login or auth method is provided, clearly explain how to obtain a token (e.g. "Use Supabase OAuth or call our login endpoint to get your JWT"). Right now, the Quick Start in the README only mentions setting `OPENAI_API_KEY` and starting the server [90], but not how auth works – fleshing this out will reduce user confusion.
19. Double-check environment variable needs. The `.env.example` lists JWT_SECRET and Stripe secret [62]. If Supabase URL and service key are needed, add them to the example file for completeness.
20. Provide cURL examples for **all endpoints**. The `docs/endpoint_examples.md` currently only shows the chat endpoint usage [91] [92]. Expanding this to show, say, how to upload a dataset or update settings (and including an example admin token if needed) will be very helpful for testers and early users.

By addressing the above steps, Nutri-GPT's backend will not only meet its v3 roadmap goals but also gain the robustness and feature breadth expected of a production-ready assistant platform. The key priorities are **security (admin controls, auth)** and **safety (content moderation)**, followed by improvements in **usability (auth flow, docs)** and **maintainability (tests, CI)**. With these in place, Nutri-GPT would be much more aligned with industry leaders, providing a secure, reliable, and user-friendly experience.

---

[1] auth.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/middleware/auth.js

[2] [3] [6] [7] [8] [57] [69] quota.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/middleware/quota.js

[4] [5] [25] [59] [60] 20250730_r1.sql
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/db/migrations/20250730_r1.sql

[9] [10] [11] billing.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/routes/billing.js

12 13 41 84 85 billing.test.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/tests/billing.test.js

14 15 16 17 18 19 31 58 73 76 77 87 89 datasets.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/routes/datasets.js

20 29 78 server.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/server.js

21 22 23 24 26 27 28 32 settings.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/routes/settings.js

30 chat.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/routes/chat.js

33 34 35 42 43 81 82 chat.test.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/tests/chat.test.js

36 37 38 88 quota.test.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/tests/quota.test.js

39 40 86 settings.test.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/tests/settings.test.js

44 45 46 54 55 66 67 68 replit_backend_plan_v_2_1.md
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/docs/replit_backend_plan_v_2_1.md

47 48 openaiClient.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/lib/openaiClient.js

49 guardRails.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/lib/guardRails.js

50 51 52 53 buildPrompt.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/lib/buildPrompt.js

56 65 70 71 72 83 Pasted--1-Create-docs-backend-plan-v3-md-supersedes-v2-1-cat-docs-backend-plan-v3-md-EOF-Nutr-1753812672052_1753812672053.txt
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/attached_assets/Pasted--1-Create-docs-backend-plan-v3-md-supersedes-v2-1-cat-docs-backend-plan-v3-md-EOF-Nutr-1753812672052_1753812672053.txt

61 health.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/server/routes/health.js

62 .env.example
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/.env.example

63 64 79 index.js
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/index.js

74 10 Best Practices for Managing User-Generated Content ... - Skim AI
https://skimai.com/10-best-practices-for-managing-user-generated-content-with-openais-api/

75 Rate Limiting OpenAI Requests with an API Gateway | Zuplo Blog
https://zuplo.com/blog/2023/10/11/rate-limiting-openai-requests

[80] Admin Role – Jasper Help Center
https://help.jasper.ai/hc/en-us/articles/27810624479131-Admin-Role

[90] README.md
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/README.md

[91] [92] endpoint_examples.md
https://github.com/nutri-org/nutri-gpt-assistant/blob/e5bb499f52e6ab9179aa8aa5b305e52572740b42/docs/endpoint_examples.md