AI Task-Reminder Bot: Bounded, Safe LLM Generation (RUDE.AI)

Abstract

RUDE.AI is a live Telegram-based AI reminder product built around bounded LLM generation. The system turns a single user task into a scheduled sequence of escalating reminders across a 24-hour lifecycle, while enforcing safety, privacy, billing, and scheduling constraints outside the model. The technical challenge was not only generating reminders, but keeping the LLM inside a narrow product loop: safe input handling, deterministic refusal paths, timezone-aware scheduling, idempotent billing, and predictable data retention.

Business Problem

Small, low-effort tasks often carry a disproportionate cost when forgotten: replying to one message, sending a follow-up, making a quick request, or completing a small personal commitment. They are too minor to justify a full productivity workflow, but still important enough that missing them can create friction, embarrassment, lost opportunity, or relationship damage.

RUDE.AI addresses this narrow gap with a lightweight task-reminder loop: a user logs one task, and the bot sends escalating reminders until the task is marked done or expires after 24 hours. The product is intentionally designed for users who do not respond well to gentle, generic productivity language. Its voice is blunt and comic by design, but the interaction model is bounded: safety, refusal handling, data retention, and content limits are treated as core system requirements rather than brand afterthoughts.

System Logic

`Task intake → safety gate → task storage → schedule computation → LLM generation → Telegram delivery → completion/expiry → retention cleanup`

Each user holds one active task at a time. A subscription/trial gate controls *who* may create tasks and is kept entirely independent of the reminder engine, so billing state never affects an in-flight task.

Safety & Data Handling

Safety here spans three distinct concerns, each handled separately.

Interaction safety — what the bot will act on and say. A fail-closed input-moderation gate runs before any task is stored, scheduled, or sent to the generation model: tasks involving harm to self or others, or sexual content, are refused in code, not by prompt instruction. The check fails *closed* — a moderation error or timeout refuses the task rather than letting it through. Self-harm is handled distinctly from other refusals: instead of a refusal or a roast, the user gets a brief, plain redirect to an international helpline directory. Generation itself carries explicit, verbatim content boundaries — sharp about the task and the avoidance, never the person; no personal attacks, no references to self-harm or violence; automatic de-escalation the moment a user signals distress.

Content & data safety — what is stored, and for how long. Task text is treated as personal data: the application does not intentionally log task or chat message content. Free-text chat messages are not stored at all, only a per-task message count is kept. Completed and expired task text is automatically purged after 30 days while anonymous counts are retained; active tasks are never touched. A single command (`/forget`) deletes a user's tasks and account outright. Payments run through a Merchant of Record, so no card data ever reaches the system.

Prompt-injection robustness — keeping the persona inside its limits. The threat model is stated explicitly: the model has no tools, never sees secrets or other users' data, and its output returns only to the user who typed the input, so a successful injection can neither escalate privilege nor exfiltrate data, leaving one residual risk: a user coaxing the bot past its content boundaries. Three structural defenses contain that risk, each locked by regression tests: the content boundaries appear verbatim in the system prompt for every mode; all user-controlled text is confined to the user turn and never the system prompt, so injected text is framed as data rather than competing instructions; and all output is delivered as plain text, so markup or link injection in a task name or model reply cannot render. The tests exercise adversarial payloads including forged system tags, "ignore all previous instructions," and right-to-left override control characters.

Architecture & Technical Decisions

- **Schedule as a pure function.** All reminder times are produced by a side-effect-free function (`created_at` → an ordered list of timestamped events), kept separate from the scheduler and the delivery layer. This isolates the genuinely tricky logic — local-timezone quiet hours while the scheduler itself runs in UTC, minimum spacing between adjacent messages, an inviolable 24-hour expiry, and an accelerated test mode — so it can be unit-tested exhaustively without any infrastructure.
- **One-shot jobs, fixed at creation.** Reminders are scheduled as individual one-shot jobs rather than a recurring interval, so each message's exact send time — after quiet-hours and collision adjustments — is locked when the task is created. Active tasks recompute and reschedule themselves automatically on restart.
- **Prompt as content, not code.** All voice and boundary language lives in a single markdown file; the application only selects and assembles sections at runtime. The

persona can be tuned without touching code, and the safety boundaries live as load-bearing text that regression tests require to be present verbatim in every assembled prompt.
- **Unsafe inputs blocked before generation.** Input safety is enforced as an application gate before storage, scheduling, or generation. Generated reminder copy is then bounded through prompt structure, role separation, token ceilings, plain-text delivery, and regression-tested prompt boundaries.
- **Idempotent billing webhook.** Each payment-provider webhook delivery is recorded by a hash of its raw body, written in the same database transaction as the account update, so a retried, byte-identical delivery hits the primary key and short-circuits instead of double-applying. Signatures are verified with constant-time HMAC comparison.
- **Billing gate decoupled from the engine.** Subscription state controls only who may create a task; it never touches scheduling, so any task already running completes regardless of trial or payment state.
- **One active task per user.** A deliberate scope constraint that removes whole classes of complexity — no cross-task scheduling collisions, no list views to maintain — and keeps the product's single loop sharp.

Reliability & Operations

- PostgreSQL with versioned schema migrations; destructive schema changes handled with pre-migration backup precautions.
- Webhook authentication via constant-time HMAC comparison (for both the messaging and the billing providers), IP- and per-user rate limiting, request body-size limits, and secrets read only from the environment.
- A shared, retrying async API client used by both chat handlers and scheduled jobs; per-message hard token ceilings cap both cost and verbosity.
- An automated test suite (122 tests) covering the scheduling edge cases, the safety gate, and the prompt-injection defenses.
- Deployed on Railway; subscription billing via Lemon Squeezy (Merchant of Record).

Tools

Python, FastAPI, python-telegram-bot, APScheduler, PostgreSQL, SQLAlchemy, Alembic, OpenAI API (generation and moderation), Lemon Squeezy (Merchant of Record), Railway, pytest.

Outcome

A deployed, operating AI product demonstrating end-to-end ownership of bounded LLM application design: fail-closed input moderation, prompt-role separation, timezone-aware scheduling, transactional billing webhook handling, privacy-conscious data retention, authenticated webhooks, and regression-tested safety and scheduling behavior. The controlled,

on-brand voice is constrained through narrow generation modes, token ceilings, plain-text delivery, and regression-tested prompt boundaries.