A FULL-STACK LANGUAGE-LEARNING PLATFORM

HSK Trainer

Spaced repetition (FSRS) and LLM-powered
written-expression grading for the Chinese HSK exams

Author Ivann Vasic

Stack FastAPI · Next.js · Postgres · Anthropic Claude

Type Technical project report

Abstract

HSK Trainer is a web application that helps learners prepare for the official *HSK* Chinese proficiency exams. It pairs a modern spaced-repetition scheduler (FSRS) for vocabulary review with an LLM grader that returns structured, actionable corrections on free-form written production. This report describes the system's purpose, its decoupled architecture, the engineering decisions behind it, and how it is tested and deployed at near-zero cost. The project began as a single Streamlit prototype and was deliberately rebuilt as a typed FastAPI service with a separate Next.js front end to demonstrate production-oriented software, data, and AI-integration practices.

Contents

1 Introduction and Motivation

Preparing for the HSK exams requires two distinct skills: *recognising* vocabulary quickly and *producing* correct written Chinese. Generic flashcard apps address the first poorly (fixed intervals, no production practice) and the second not at all.

HSK Trainer was built to solve a concrete personal need — a genuinely useful study tool — while simultaneously serving as an engineering portfolio piece. That dual purpose shaped every decision: features were only kept if they passed the "would I use this every day?" test, and the architecture was chosen to make production practices visible (typed APIs, migrations, streaming LLM integration, CI, infrastructure-as-code).

The application targets HSK levels 1–3 (roughly 600 words). Two capabilities form its core:

- **Comprehension review** — an FSRS scheduler selects the words a learner is about to forget, mixing in new vocabulary, and adapts to self-assessed recall.
- **Written-expression grading** — the learner writes a short text in Chinese on a chosen topic; Anthropic Claude streams back a structured correction (score, corrected version, pinyin, translation, a typed list of mistakes, strengths, and a next-step recommendation).

A closing feedback loop ties the two together: vocabulary that Claude flags as misused is automatically pushed back into the FSRS review queue.

2 System Overview

From the user's perspective the application exposes four areas: a landing/auth flow, a comprehension quiz, a written-expression workspace, and a personal dashboard.

- **Comprehension quiz** — two modes (self-assessment and pinyin input), keyboard shortcuts (1–4 for the FSRS ratings), progressive hints (pinyin first, then full solution), and instant card transitions.
- **Written expression** — pick a curated subject, have Claude generate one, or write your own; submit Chinese text and watch the correction stream in token by token.
- **Dashboard** — a mastery-distribution chart, a 90-day activity heatmap, quiz and expression progression curves, and browsable lists of *mastered*, *known*, and *to-review* words.
- **Cross-device** — stateless JWT auth and a shared Postgres database mean the same account works on a laptop and a phone; the front end is installable as a PWA.

3 Architecture

The system is a **monorepo** with two independently deployable services that communicate over HTTPS using JSON and, for the grading endpoint, a Server-Sent-Events (SSE) stream.
The repository layout reflects this separation:

```
backend/ FastAPI service (Python 3.12)
  app/api/v1/ auth, quizzes, srs, mastery, expression, account
  app/services/ srs_service, llm_service, expression_service, ...
  app/models/ SQLAlchemy ORM
  app/schemas/ Pydantic v2 request/response models
  alembic/ migrations (0001 baseline)
  data/ HSK1-3 vocabulary CSVs + curated writing subjects
  tests/ pytest (Anthropic mocked)
frontend/ Next.js 16 app (TypeScript)
  app/ routes (landing, auth, comprehension, expression, account)
  components/ quiz, expression, charts, mastery, ui primitives
  lib/ typed API client, auth context, SSE consumer, hooks
  tests/ Vitest unit tests
```

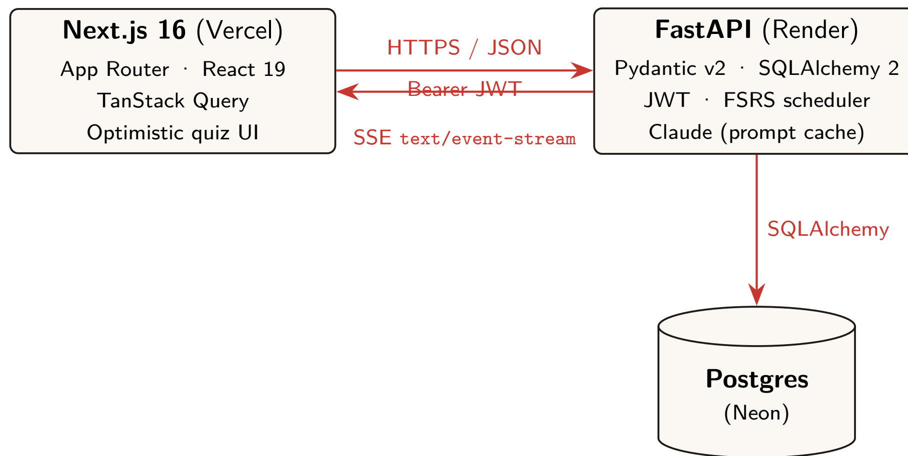

Figure 1: High-level service topology. The front end and back end deploy independently; the database is shared.

3.1 Request lifecycle

Every authenticated request carries a JSON Web Token in an `Authorization: Bearer` header. A reusable FastAPI dependency decodes and validates it, resolves the user, and — on quota-bearing endpoints — enforces a per-user daily limit before the handler runs.

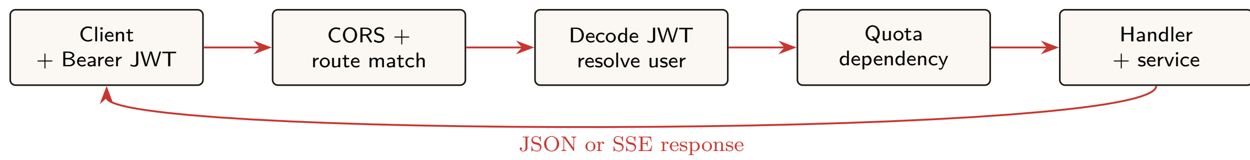

Figure 2: Request pipeline. Auth and quota are dependencies, so handlers stay free of cross-cutting concerns.

4 Technology Stack

Layer	Choice	Rationale
API	FastAPI + Pydantic v2	Typed I/O, auto OpenAPI, async SSE
ORM	SQLAlchemy 2.x	Mature, explicit, Postgres + SQLite
Migrations	Alembic	Versioned schema, baseline stamping
Auth	JWT (HS256) + PBKDF2	Stateless, no session store
Scheduler	FSRS (py-fsrs)	ML-based, outperforms SM-2
LLM	Claude Haiku 4.5	Cheap, fast, prompt-cache friendly
Front end	Next.js 16 / React 19	App Router, RSC, mature ecosystem
Server state	TanStack Query	Caching, retries, optimistic updates
Styling	Tailwind v4	CSS-first tokens, fast iteration
Forms	react-hook-form + zod	Typed validation shared with API types
Database	Postgres (Neon)	Free tier, autoscaling, branching
CI	GitHub Actions	pytest + Vitest + production build

Table 1: Stack and the reasoning behind each layer.

5 Backend Design

5.1 API surface

The API is versioned under `/v1` and fully described by an auto-generated OpenAPI document (Swagger UI at `/docs`). The endpoint groups are:

- `auth` — register, login, me
- `quizzes` — catalogue and entries
- `srs` — rate, due, stats
- `mastery` — per-word state
- `expression` — subjects, generate, correct, attempts, quota
- `account/overview` — dashboard aggregates

plus an unauthenticated `/healthz` liveness probe used by the keep-warm cron.

5.2 Spaced repetition service

The scheduler is isolated behind a service module. The public API speaks the standard SRS rating vocabulary (`again`/`hard`/`good`/`easy`); internally that maps to FSRS `Rating` values, and the FSRS card state is serialised onto the mastery row so scheduling survives across sessions.

Listing 1: FSRS rating, simplified — the scheduler owns interval/stability, the row is just persistence.

```
1  def rate_entry(db, *, user_id, entry_id, rating: RatingEnum) -> MasterySnapshot:
2      record = _get_or_create_mastery(db, user_id, entry_id)
3      card = _build_card_from_record(record) # rebuild FSRS state
4      new_card, _ = _SCHEDULER.review_card(
5          card, _RATING_TO_FSRS[rating], review_datetime=datetime.now(timezone.utc)
6      )
7      _write_card_into_record(record, new_card, rating) # stability, due, ...
8      return MasterySnapshot(entry_id=record.entry_id,
9                             next_review_at=record.fsrs_due_at,
10                            stability_days=record.fsrs_stability)
```

5.3 LLM integration: prompt caching and streaming

Two properties make the Claude integration production-grade. First, the system prompt is wrapped in an ephemeral cache block, so repeated corrections within the cache window pay only the cheap cache-read price rather than re-billing the full instruction every call.

Listing 2: Prompt caching — the static grading rubric is cached; only the learner's text varies.

```
1  message = client.messages.create(
2      model=get_model_id(), max_tokens=1200,
3      system=[{"type": "text", "text": CORRECTION_SYSTEM_PROMPT,
4               "cache_control": {"type": "ephemeral"}}],
5      messages=[{"role": "user", "content": user_payload}],
6  )
```

Second, the correction endpoint streams the model output to the browser as typed SSE events. The generator opens its own database session so it can persist the attempt and re-queue misused vocabulary *after* the stream completes, without holding a request-scoped transaction open for the whole response.

Listing 3: SSE generator, simplified — events: `start`, `partial*`, `complete`, `persisted` (or `error`).

```
1  def _stream_correction_response(*, user_id, level, subject, user_text):
2      db = SessionLocal()
3      try:
```

```
4         for event in llm_service.stream_correction(user_text, level=level,
5                                                    subject=subject):
6             yield _sse(event.type, event.data) # start / partial / complete
7         attempt = expression_service.record_attempt(db, ...)
8         relearned = queue_entries_for_relearn(db, user_id, misused_entry_ids)
9         db.commit()
10        yield _sse("persisted",
11                   {"attempt_id": attempt.id, "relearned_count": relearned})
12    finally:
13        db.close()
```

5.4 Data layer and migrations

The schema is described by an Alembic baseline migration. Because the project migrated from an earlier app that already owned a populated production database, the deployment procedure *stamps* that database at the baseline revision (recording it as applied without re-running DDL) rather than recreating tables — a small but important detail for zero-downtime evolution. Tests run against a throwaway SQLite database and never touch production.

6 Frontend Design

The front end is a Next.js App Router application. Auth state lives in a React context backed by `localStorage`; server state (quizzes, due cards, mastery, quota) is cached by TanStack Query. A small typed `fetch` wrapper attaches the Bearer token and mirrors the backend's Pydantic schemas.

6.1 Optimistic UI: the latency win

The single most impactful UX decision. In the original Streamlit prototype, every interaction triggered a full server rerun, so the gap between answering a card and seeing the next one was 2–5 seconds. The rewritten quiz prefetches the whole session, advances the local index *immediately* on a rating, and fires the persistence request in the background with retry.

Listing 4: Optimistic advance — the next card renders before the network call resolves.

```
1 const recordRating = useCallback((rating: RatingEnum, meta?) => {
2   setState(prev => ({
3     ...prev,
4     currentIndex: prev.currentIndex + 1, // advance now
5     history: [...prev.history, { entry: prev.questions[prev.currentIndex],
6                                  rating, ...meta }],
7   }));
8 }, []);
9 // the page then fires postRate() via a background mutation (retry x2)
```

The perceived transition dropped from seconds to well under 50 ms, with persistence failures surfaced non-blockingly through a toast.

6.2 Consuming the SSE stream

The browser's native `EventSource` cannot send an `Authorization` header, so the front end reads the stream manually with `fetch` and a `ReadableStream` parser.

Listing 5: Custom SSE reader — handles auth headers, which native EventSource cannot.

```
1 const reader = response.body.getReader();
2 const decoder = new TextDecoder();
3 let buffer = "";
4 while (true) {
```

```
5    const { done, value } = await reader.read();
6    if (done) break;
7    buffer += decoder.decode(value, { stream: true });
8    let i; while ((i = buffer.indexOf("\n\n")) !== -1) {
9      const block = buffer.slice(0, i); buffer = buffer.slice(i + 2);
10     handlers.onEvent(parseSseBlock(block)); // {event, data}
11   }
12 }
```

While Claude streams, the UI shows an "analysing" indicator with a live token counter; on the `complete` event it swaps to the fully structured correction with an animated reveal. The visual identity (a cinnabar/ink/paper palette inspired by Chinese seal art, with 学 as the mark) carries through to a PWA manifest so the app installs to a phone home screen.

7 Architecture Decisions and Trade-offs

Every significant choice was made explicitly. The most consequential are recorded below with the alternative that was rejected and why.

FSRS over SM-2 or a hand-rolled heuristic

The original prototype used a fixed-ratio bucket heuristic (45 % review / 50 % new). FSRS models per-card stability and difficulty and schedules each review just before predicted forgetting, which is the entire point of spaced repetition. SM-2 (classic Anki) was the simpler alternative but is empirically weaker; a bespoke algorithm would have been unjustifiable reinvention. FSRS is isolated behind a service boundary so it can be tuned or swapped without touching the API.

Optimistic UI over server-driven rendering

Keeping the quiz server-authoritative (the Streamlit model) was simplest but produced the 2–5 s stall that motivated the rewrite. Prefetching the session and advancing client-side trades a small amount of state-reconciliation complexity for an order-of-magnitude better perceived latency — the correct trade for a drilling interaction repeated hundreds of times per session.

Keep a Python backend; do not rewrite in TypeScript

A single Next.js full-stack app (API routes) would have removed a service and a deployment. It was rejected to preserve the Python data/AI stack (SQLAlchemy, Alembic, the FSRS library, the Anthropic SDK) — the part of the project most relevant to its goals — and to keep a clean, independently testable API boundary.

Render free tier + keep-warm cron over serverless

Free serverless (e.g. rewriting the backend for an edge runtime) avoids cold starts but would have meant porting the entire Python service. Instead the FastAPI app runs on Render's free tier and a 10-minute cron ping to `/healthz` prevents the idle spin-down. Same zero cost, no rewrite, and the spin-down behaviour is documented rather than hidden.

JWT in localStorage with a Bearer header

An `httpOnly` cookie is the stricter choice against XSS but adds cross-site cookie and CSRF complexity between a Vercel front end and a Render API. For a single-user study tool the localStorage + Bearer approach is a deliberate, scoped simplification, isolated in one auth module so it can be hardened later without touching call sites.

Typed SSE events over raw text or no streaming

A non-streamed response was simplest but loses the real-time feedback that makes a multi-second LLM call feel responsive. Streaming raw text would have pushed JSON parsing into the UI. Typed events (`start` / `partial` / `complete` / `persisted`) keep the transport self-describing and let the client switch cleanly from a progress state to the structured result.

8 Testing and Continuous Integration

The backend has a pytest suite that runs entirely offline: the Anthropic client is replaced by a fake that mimics both the one-shot `messages.create` call and the streaming context manager, so subject generation, the full SSE correction flow, quota enforcement, and the SRS re-queue loop are all covered without a network or a paid API call. The front end has Vitest unit tests for the pure logic (notably the tone-stripping pinyin comparator). Together the suites total 42 tests.

GitHub Actions runs both suites plus a production Next.js build on every push and pull request, so a type error or a broken test is caught before it can be deployed. A subtle bug surfaced and fixed this way — a SQLite write lock when the SSE generator opened a second session while a request-scoped transaction was still open — is exactly the class of issue an offline integration test is meant to catch.

9 Deployment

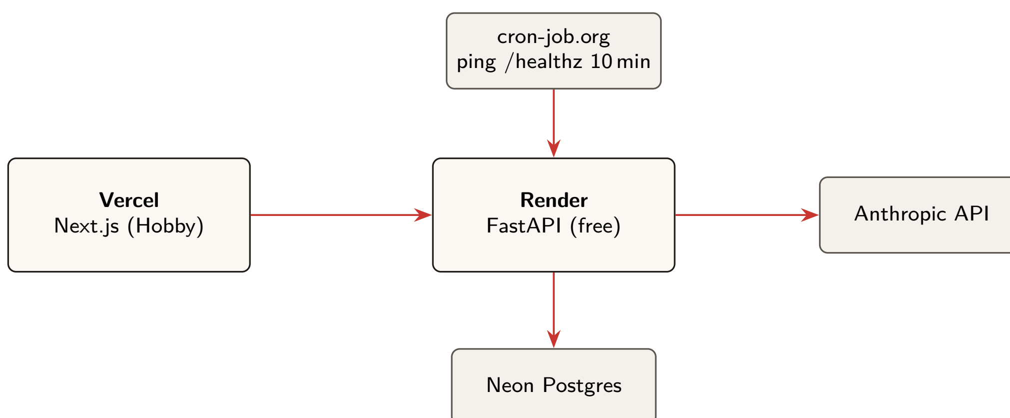

Figure 3: Deployment topology. Every component runs on a free tier; the cron ping keeps the free Render instance from sleeping.

The entire production stack runs at a recurring cost of approximately **€0**. The only variable cost is Anthropic usage: with prompt caching, a written-expression correction costs on the order of \$0.003. Infrastructure is declared as code in `render.yaml`, and both platforms redeploy automatically on a push to `main`.

10 Conclusion

HSK Trainer demonstrates a complete path from a quick prototype to a production-shaped application: a typed, versioned API; a migrated and version-controlled data layer; a streaming LLM integration with cost controls; an optimistic, mobile-first front end; an offline test suite with CI; and zero-cost infrastructure-as-code. The guiding constraint — that it had to be a tool worth using daily *and* a credible engineering artefact — pushed the design toward explicit, defensible trade-offs rather than defaults. The result is a focused application whose every layer can be explained and justified.

Ivann Vasic – HSK Trainer – FastAPI · Next.js · Postgres · Anthropic Claude