Executive overview

- **ResumePilot** is an AI-assisted resume and cover-letter platform with live ATS-style feedback, batch cover-letter generation, and PDF exports. It supports importing resumes, parsing job descriptions (JDs), producing cover letters tailored to specific roles, and scoring resumes against JDs.
- The system is a full-stack TypeScript app:
    - **Backend**: Express on Node.js with HTTP + WebSocket, SQLite via `better-sqlite3` for persistence, domain logic for resume/JD management, ATS scoring, cover letter generation/improvement, batch processing, and export routes.
    - **Frontend**: React + Vite + Tailwind (client directory) provides resume editor, JD viewer, cover-letter viewer, ATS panel, and realtime batch status updates via WebSockets.
    - **Shared**: Drizzle schema and Zod validators shape data integrity across server and client.
- **AI integration**: The code references Gemini-enhanced and OpenAI-style embedding logic. It currently computes ATS scores deterministically/mocked using resume depth and randomization, while allowing future swap-in of embedding/LLM-powered scorers and generators.
- **Realtime updates**: Batch jobs broadcast progress via channel-based WebSockets that the client subscribes to.
- **Rate limiting**: API and LinkedIn scraping endpoints are protected with express-rate-limit.

Problem statement and goals

- Job seekers must tailor resumes and cover letters to specific roles to get through ATS filters and impress hiring managers. This is time-consuming and often guesswork-heavy.
- ResumePilot aims to:
    - Ingest resumes and job descriptions robustly (paste/upload/import).
    - Parse JDs, identify key competencies/keywords, and structure them.
    - Generate tailored cover letters (single or batch) aligned to JD context and resume content.
    - Provide ATS-style scoring with actionable suggestions to improve resume match.
    - Export professional PDFs for resumes and cover letters.
    - Offer a clean, modern UI with a cohesive editing and preview experience and live feedback.

High-level architecture

- Server composition:
    - `server/index.ts` configures Express, JSON body parsing, request logging, error handling, and environment bootstrapping. It mounts routes and conditionally serves Vite dev or static assets in production. Web server listens on `PORT` (defaults to 3001).
    - `server/routes.ts` defines HTTP and WebSocket endpoints: resume CRUD, JD CRUD and parsing/upload, cover letter generation/improvement, ATS scoring (on-demand and retrieval), batch cover-letter jobs, job status query, and PDF exports.
    - `server/storage.ts` implements `SQLiteStorage` with `better-sqlite3`, wraps SQL tables, and provides domain-centric CRUD methods. It maps snake_case DB columns to camelCase fields consistently.
    - `shared/schema.ts` defines DB schema using Drizzle and data validators via Zod, ensuring type safety across boundaries.
    - `server/services/*` encapsulates AI integrations, JD parsing, PDF generation, and scraping. The code references modules for Gemini/OpenAI/LinkedIn scraping, with graceful fallbacks.
- Client composition:
    - React app under `client/` uses components for resume editing, JD viewing, cover-letter viewer, ATS panel, modals for job uploads and batch operations, and `websocket.ts` for

realtime updates.
    - UI components and patterns (e.g., dialog, sheet, cards, forms) are curated under `client/src/components/ui/` .

Backend in detail

Server bootstrap and middleware

- `server/index.ts` loads `.env` and initializes Express with JSON parsing, a concise API logger, centralized error handling, and environment-aware asset serving.
- The request logger captures API performance and truncates long JSON payloads to maintain readable logs. Errors are normalized to JSON.

Routes and WebSockets

- `registerRoutes` in `server/routes.ts` starts an HTTP server, attaches `ws` WebSocket server at `/ws` , and manages channel subscriptions in-memory via `Map<string, Set<WebSocket>>` .
- Channel-based broadcasting supports ATS score channels `ats:${resumeId}:${jdId}` and batch job channels `batch:${jobId}` .
- Rate limiting: 100 requests/min for general API; 5 req/15 min for LinkedIn scraping.

Core endpoints

- Resumes:
    - `GET /api/resumes` list, `GET /api/resumes/:id` fetch
    - `POST /api/resumes` create (validated by `resumeJsonSchema`)
    - `PATCH /api/resumes/:id` update
- Job descriptions:
    - `GET /api/jds` , `GET /api/jds/:id`
    - `POST /api/jds` parse and store with optional embeddings
    - `POST /api/jds/upload` upload text file (10MB limit)
    - `DELETE /api/jds/:id`
- Cover letters:
    - `POST /api/jds/:id/cover-letter`
    - `POST /api/covers/generate`
    - `POST /api/covers/:id/improve`
    - `GET /api/covers` , `GET /api/covers/:id` , `DELETE /api/covers/:id` , `DELETE /api/covers/bulk/all`
- ATS scoring:
    - `POST /api/ats/score` compute and broadcast partial/final, persist
    - `GET /api/ats/score/:resumeId/:jdId` fetch persisted score
- Export:
    - `POST /api/export/resume-pdf`
    - `POST /api/export/cover-pdf`
- Jobs:

  - `POST /api/covers/batch` create background job; `GET /api/jobs/:id` fetch status

Storage and persistence

- `server/storage.ts` provides a typed storage layer using `better-sqlite3`. Tables: `users`, `resumes`, `job_descriptions`, `ats_scores`, `cover_letters`, `jobs`.
- JSON fields are serialized/deserialized at the boundary; snake_case mapped to camelCase in DTOs.
- Ids via `randomUUID`; timestamps stored as UNIX epoch.

Data model and validation

- Defined in `shared/schema.ts` (Drizzle + Zod) with types for Users, Resumes, JobDescriptions, CoverLetters, ATSScore, Jobs.
- `resumeJsonSchema` enforces structure for `personalInfo`, `summary`, `experience[]`, `skills`, optional `education[]`.

AI features and approach

- JD parsing: deterministic extraction; extensible to NLP/LLM-powered parsing.
- Embeddings: `generateEmbedding` attempted; non-fatal on failure.
- Cover letters: `generateCoverLetter` (tone/length control), `improveCoverLetter` returns improved content + suggestions.
- ATS scoring: heuristic/deterministic (65–80) based on resume richness; broadcasts partial then final; persisted in `ats_scores`. Designed to be replaced with embedding + LLM hybrid.

Realtime and batch processing

- WebSockets: subscribe by sending `{ channel }`; messages are `{ channel, data }`.
- Batch cover letters: background processor iterates JDs, broadcasts progress to `batch:${jobId}`, saves artifacts, and marks completion.

Security, reliability, and compliance notes

- Rate limiting for API and scraping.
- Input validation with Zod; add further size/content constraints as needed.
- Error handling returns JSON with correct HTTP codes.
- DB is local SQLite; for scale, migrate to Postgres and add a queue for jobs.
- Auth: currently `user-id` header fallback. Add real auth (JWT/OAuth) and enforce per-user access.
- Secrets via `.env`; restrict logging of sensitive data.
- LinkedIn scraping: ensure ToS compliance and user consent.

Frontend in brief

- React app offers resume editor/preview, JD list/panel with paste/upload, cover-letter viewer/editor, ATS panel with live updates, and batch modal.
- State via local state and stores in `client/src/store/`, server state via React Query, realtime via `lib/websocket.ts`.
- UI primitives under `client/src/components/ui/` ensure consistent design and accessibility.

Data flows and user journeys

1. Create resume → POST `/api/resumes` → validate → persist → display.
2. Add JD → POST `/api/jds` or `/api/jds/upload` → parse → embed → store.

3. ATS score → POST `/api/ats/score` → partial+final via WebSocket → persist → GET for retrieval.
4. Generate cover letter → POST `/api/jds/:id/cover-letter` or `/api/covers/generate` → store → edit.
5. Improve cover letter → POST `/api/covers/:id/improve` → update + suggestions.
6. Batch letters → POST `/api/covers/batch` → subscribe `batch:${jobId}` → progress → artifacts.
7. Export → POST `/api/export/*-pdf` → return document structure + filename.

Algorithms and heuristics (current and future)

- JD parsing: rules/regex now; consider section segmentation, NER, TF-IDF.
- Embeddings: enable semantic similarity (cosine) for resume–JD relevance.
- ATS scoring: replace heuristic with hybrid of keyword lexicon + semantic similarity; readability (Flesch-Kincaid); formatting checks.
- Cover letters: prompt templates emphasizing JD alignment, quantified achievements, and clear closing; tone/length controls.

Testing and tooling

- Scripts: `test-api-*.sh` , `test-gemini*.js` , `test-openai.js` , `test-linkedin.js` for manual/scripted validation.
- Local DB `data.db` ; Vite dev server for frontend HMR; TypeScript + Zod/Drizzle for safety.

Deployment and environment

- `.env` for keys and config; one-port model serving API + static client; Vite in dev.
- Production: build client, serve static assets, proxy SSL, forward WebSockets at `/ws` , persist DB volume or migrate to hosted SQL.

Performance and scalability

- For multi-instance: move to Postgres, add Redis/pub-sub for WebSockets, introduce a queue for jobs, and caching (JD embeddings, ATS scores).

Reliability and observability

- Structured logging, request IDs, metrics; error tracking (Sentry); health checks.

Risks and limitations

- ATS scoring is heuristic; communicate as guidance.
- Scraping risks; adhere to ToS.
- No auth yet; add before production.
- Privacy: resumes contain PII; consider encryption and deletion tooling.

Roadmap highlights

- Authentication and row-level security.
- Embedding + LLM ATS scoring; advanced JD parsing.
- Cover-letter presets, inline AI edits, versioning.
- Bulk ingestion; industry templates.
- Analytics on ATS improvements and resume changes over time.

Competitive positioning

- Differs from generic AI writers by structured resume JSON + JD parsing + live ATS feedback + batch workflows.
- Differs from template builders by focus on match optimization and AI-tailoring.

Literature review pointers

- ATS and keyword matching: TF-IDF, BM25, transformer embeddings (Sentence-BERT).
- LLM prompting for resume/cover tailoring; grounding to avoid hallucinations.
- Readability (Flesch-Kincaid/SMOG), quantification heuristics; professional tone.
- Ethics: consent, bias, privacy; evaluation with human raters vs proxy metrics.

README skeleton (copy-ready)

- **Title + tagline**
- **Features**
- **Quick start**: prerequisites, install, run
- **Architecture**: backend, frontend, shared validation
- **API reference**: resumes, JDs, ATS, covers, jobs, export
- **Realtime events**: WebSocket channels
- **Configuration**: `.env` variables
- **Security notes**
- **Contributing**
- **License**