AtlasOps

AI-Powered Job Application Manager & Resume Generator

Project Overview, Target Architecture, and Development Roadmap

Prepared for: AtlasUniversalis (existing site migration)
Generated: January 11, 2026
Scope: MVP through v2 feature set (job parsing, company deep dive, profile enhancement, tailored resumes, application lifecycle, follow-ups, interview prep, continuous improvement)

This document translates your concept into an implementable plan based on your current FastAPI/Jinja/Vue + PostgreSQL deployment on DigitalOcean.

Table of Contents

1. Starting Point: Your Current Web Architecture

Your existing site is already a solid foundation for an application: it has a Python backend (FastAPI), server-side templates (Jinja2), a Vue 3 component for navigation, a PostgreSQL database, and automated deployment to a DigitalOcean droplet behind Nginx. The current system primarily serves content pages stored in a pages table, and exposes basic JSON APIs for page CRUD operations.

Why this matters: AtlasOps can be added as a new authenticated, database-driven product area in the same stack (same droplet, same CI/CD, same database engine) without needing a full rewrite on day one.

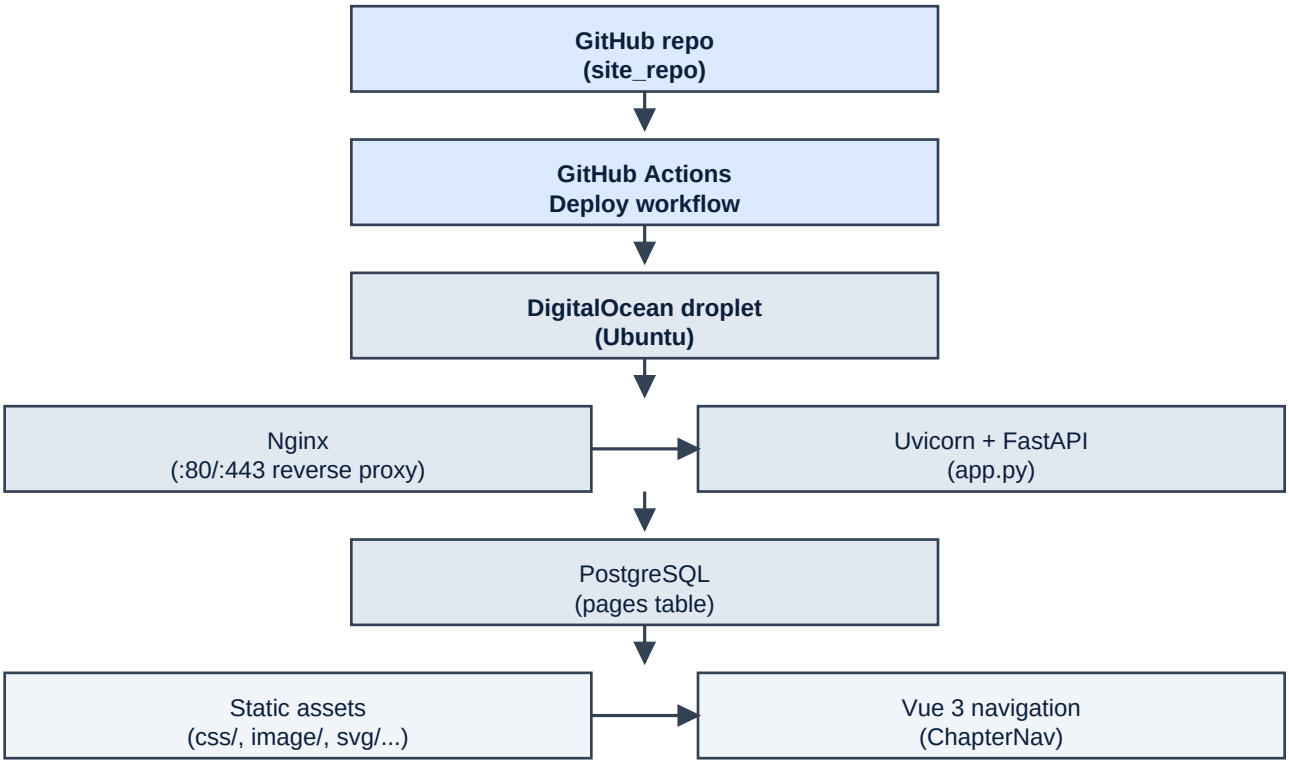

Figure 1. Current deployment and runtime architecture (summarized from your existing stack overview).

1.1 Current codebase characteristics

- Backend: FastAPI in **app.py**, using async SQLAlchemy and Jinja2 templates.
- Frontend: Vue 3 + Vite (currently used for the chapter navigation component).
- Database: PostgreSQL (currently includes a *pages* table storing slug/title/html).
- Deployment: GitHub Actions SSH deploy to a DigitalOcean droplet; systemd manages the Uvicorn service; Nginx reverse proxies HTTP/HTTPS.

1.2 Design constraints to preserve

- Keep the existing content pages working (no breaking changes to /pages/{slug}).

- Preserve the current deployment model (GitHub Actions -> droplet) while introducing staged improvements (migrations, background workers, observability).
- Add new product functionality under a clear namespace (e.g., /app, /dashboard, /api/v1) to avoid route collisions and template confusion.

2. Product Vision: What AtlasOps Will Do

AtlasOps is a job-application workflow system centered on two data assets: (1) structured job postings + company research, and (2) an AI-enhanced user profile. With those in place, the system can generate job-specific resumes, track application outcomes, and create follow-ups and interview prep materials.

2.1 Primary user journey

- **Capture job posting(s):** user pastes a URL or list of URLs; the system extracts key job data and creates dashboard entries.
- **Company deep dive:** for each entry, the system generates a one-page company profile (culture, size, location, revenue, reputation signals, and role-specific insights).
- **Profile onboarding + enhancement:** user enters work history, education, skills, projects; the system expands and normalizes this into an exhaustive skills inventory.
- **Tailored resume generation:** user selects a job entry and a template; the system produces an ATS-friendly resume and (optionally) a CV variant.
- **Lifecycle management:** user marks applied, schedules follow-ups, updates interview status, records outcomes; system auto-updates stale entries after 30 days.
- **Continuous improvement:** for rejections or inactivity, system explains likely gaps and recommends learning/practice projects.

2.2 Key entities (what gets stored)

- **User profile:** structured facts + AI-generated expansions, plus an audit trail of what was generated vs. user-supplied.
- **Job posting:** extracted structured fields and the original source URL + raw page snapshot.
- **Company deep dive:** a cached research report and citations to sources.
- **Generated resume:** chosen template, versioned content, match score, export file references.
- **Application timeline:** status changes, timestamps, reminders, follow-up messages, interview prep artifacts.

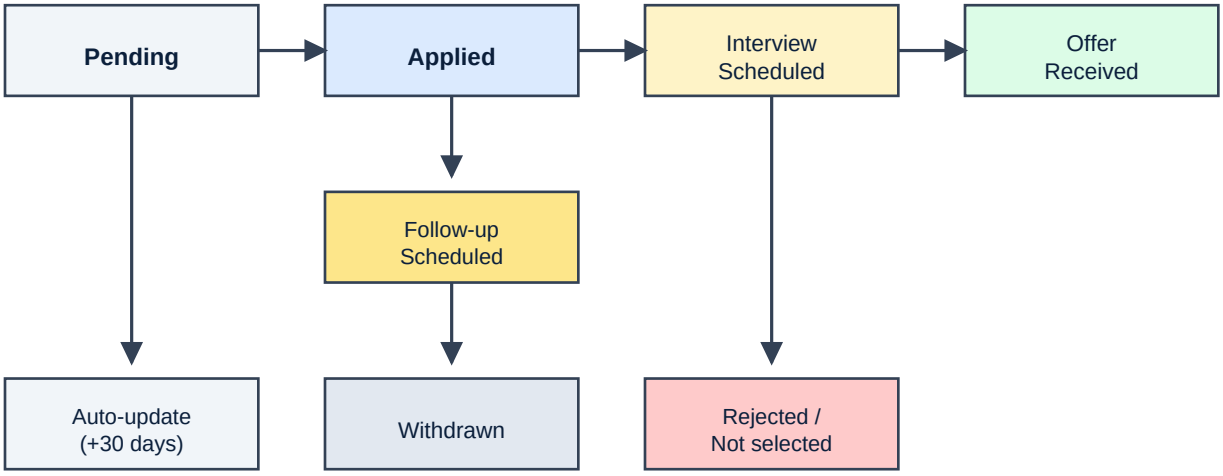

Figure 2. Recommended application lifecycle states and automated transitions.

3. Target Architecture: Evolving Your Stack Into an Application Platform

A practical approach is a modular monolith at first: keep FastAPI as the single backend service, but separate code into internal modules (auth, jobs, profiles, resumes, agents, billing, notifications). Add a background worker tier for scraping and AI-heavy tasks.

3.1 Proposed system components

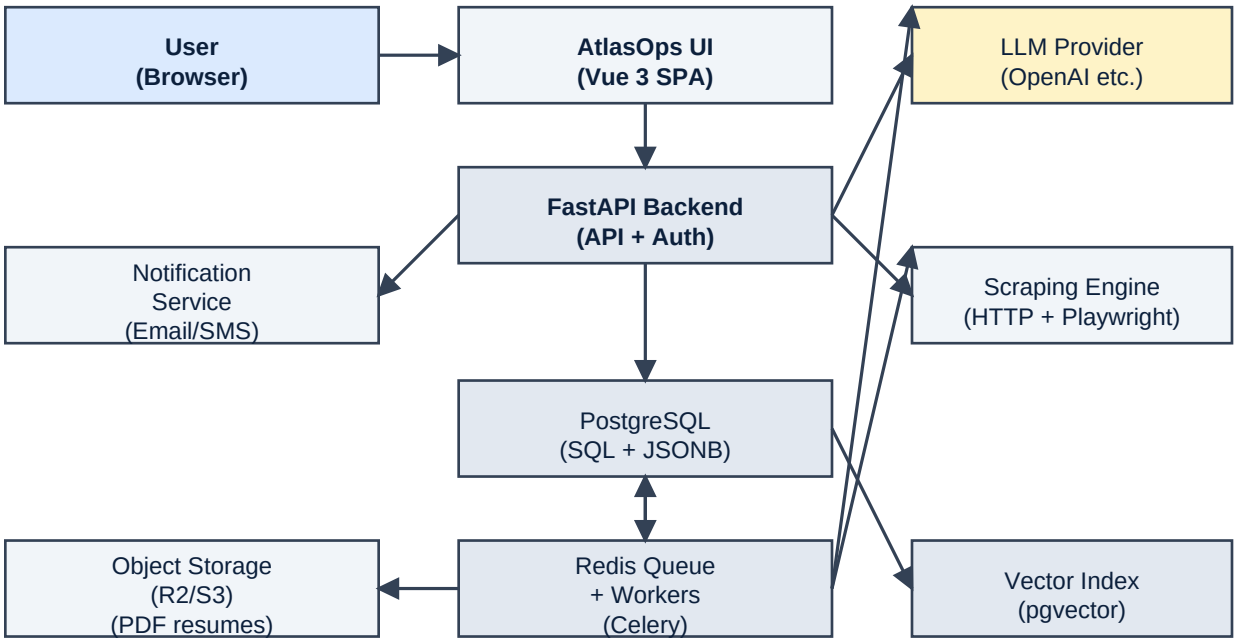

Figure 3. Target logical architecture: UI + API + background workers + data/AI integrations.

3.2 Responsibilities by layer

Layer	Responsibilities	Suggested implementation
Web UI	Dashboard, profile editor, job importer, resume preview, status updates, reminders UI	Vue 3 SPA (TypeScript) served by FastAPI or a separate frontend build
API (FastAPI)	Auth, CRUD for profiles/jobs/resumes, orchestration endpoints, webhooks, rate limiting	FastAPI + Pydantic + SQLAlchemy + Alembic
Workers	Scraping, deep-dive research, profile enhancement, resume generation, scheduled reminders	Celery workers + Redis (or RQ/Arq)
Data	Transactional storage, auditing, vector search	PostgreSQL (JSONB + pgvector)

Layer	Responsibilities	Suggested implementation
Storage	Generated PDFs, possibly raw HTML snapshots and exports	Cloudflare R2 / S3-compatible bucket (or local disk in MVP)
AI Gateway	Prompt versioning, structured outputs, caching, cost controls	Internal Python module wrapping LLM provider(s)

3.3 Why background workers are non-negotiable

- Scraping and headless browser work are slow and unreliable to run in an HTTP request/response cycle.
- LLM calls can be expensive; they need retries, caching, and rate controls.
- Company deep dives and profile expansions are best run asynchronously with progress updates.

4. Data Model & Database Design

Your current PostgreSQL usage is minimal (a pages table). AtlasOps requires a richer, auditable schema. The design below favors: (1) clear ownership boundaries, (2) immutable event history for critical actions, and (3) JSONB fields for flexible AI-generated artifacts.

4.1 Core tables (recommended)

Table	Purpose	Key fields (examples)
users	Accounts + subscription state	id, email, password_hash / oauth_id, created_at, plan
user_profile	Canonical user profile	user_id, headline, location, preferences, raw_input_json, enhanced_profile_json
profile_items	Normalized profile elements	id, user_id, type(work|project|education|skill), content_json, embedding
job_postings	Parsed job entries (one per URL)	id, user_id, source_url, company_name, title, location, pay_min/max, description, requirements_json
job_sources	Evidence + snapshots	job_posting_id, fetched_at, raw_html, extracted_text, jsonld_blob, checksum
company_deep_dives	Cached company research report	job_posting_id, summary_md/pdf, sources_json, last_updated
resumes	Generated resumes (versioned)	id, user_id, job_posting_id, template_id, resume_json, match_score, file_url, created_at
applications	Lifecycle state + timestamps	job_posting_id, status, applied_at, followup_at, interview_at, offer_at, rejected_at
messages	Follow-ups + recruiter messages	application_id, channel, subject, body, scheduled_for, sent_at
interview_prep	Prep sheets	application_id, questions_json, study_plan_json, generated_at
improvement_suggestions	Post-rejection analysis	application_id, gaps_json, recommendations_json, project_ideas_json
events	Audit log for compliance/debugging	user_id, entity_type, entity_id, action, payload_json, created_at

4.2 PostgreSQL features to leverage

- **JSONB:** store AI artifacts (structured extractions, enriched profiles, deep dive reports) while keeping a canonical, normalized subset in columns for querying.
- **pgvector:** store embeddings for semantic search and matching (profile items versus job requirements).
- **Row-level security (optional):** if you ever expose direct SQL access via a service layer, RLS can provide an extra defense-in-depth layer.

4.3 Migration strategy from your current DB

- Keep the existing pages table untouched to avoid breaking content pages.
- Introduce new tables via Alembic migrations (do not hand-edit schema in production).
- Add a new database schema namespace (e.g., atlasops) if you want a hard separation from content tables.

5. AI Integration & Agent Architecture

Your product is essentially an orchestration layer over several repeatable AI workflows. The safest and most reliable pattern is: **deterministic extraction where possible** (for example, JSON-LD JobPosting), plus **LLM structured outputs** as the flexible fallback/augmenter. Every AI output should be versioned, cached, and attributable to sources.

5.1 Recommended agent lineup

Agent	Purpose	Inputs	Outputs
Job Scraper	Fetch + extract job posting data	URL(s)	Structured job JSON + evidence snapshot
Company Research	Generate 1-page deep dive	Company name + site + job context	Company summary + citations
Profile Enhancer	Expand and normalize user profile	User-provided profile JSON	Enhanced profile + skill taxonomy
Resume Optimizer	Generate ATS-optimized resume	Job JSON + selected profile items + template	Resume content JSON + match score + export PDF
Follow-up Writer	Draft recruiter follow-ups	Application context	Message variants by channel
Interview Prep	Generate prep sheet	Job + company + resume	Likely questions + study plan + questions to ask
Career Advisor	Explain gaps + suggest projects	Rejection/inactivity + job requirements	Gap analysis + learning plan + practice project idea

5.2 Job ingestion pipeline (recommended)

Implement job parsing as a layered pipeline. Each stage produces artifacts stored in the database for debugging and reprocessing.

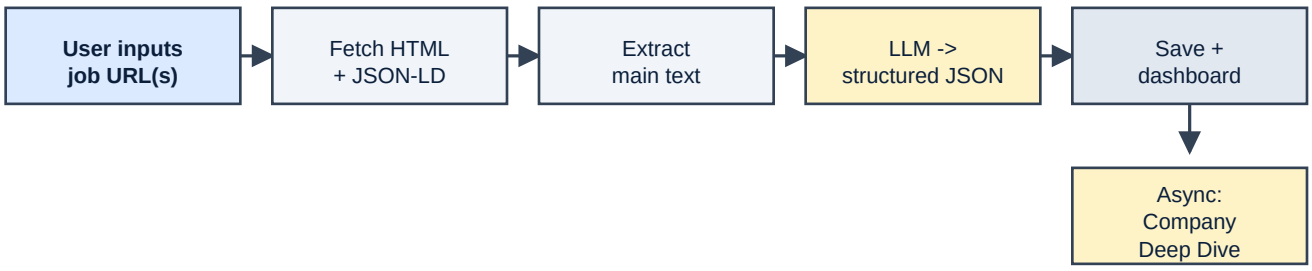

Figure 4. Job URL ingestion pipeline with an asynchronous company deep dive stage.

Implementation notes

- **Stage 1 - Fetch:** try plain HTTP GET first; fall back to Playwright for dynamic pages.
- **Stage 2 - Detect structured data:** many job sites embed schema.org JobPosting JSON-LD; parse it when present.
- **Stage 3 - Clean text:** extract the main readable content (remove nav/ads) for LLM consumption.
- **Stage 4 - LLM extraction:** enforce a strict JSON schema (company, title, requirements, pay, location, etc.).
- **Stage 5 - Validate:** run rule-based checks (e.g., title non-empty; pay numeric range) and mark fields with confidence scores.

5.3 Resume generation pipeline (recommended)

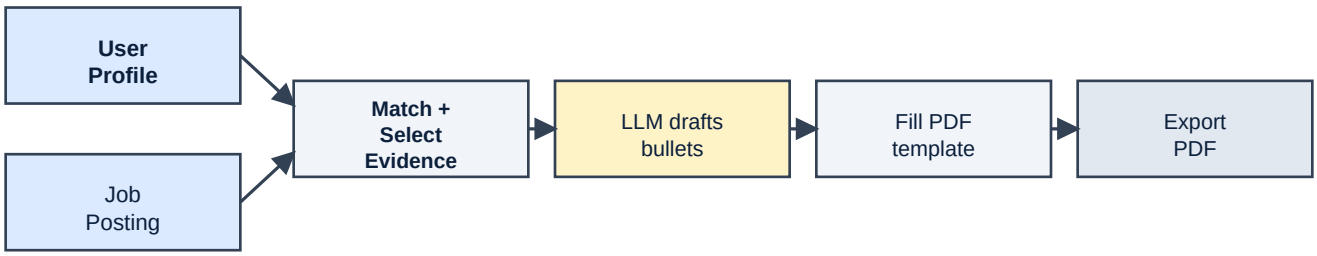

Figure 5. Resume generation: profile/job matching + LLM drafting + template rendering.

5.4 Matching algorithm (practical MVP)

- **Keyword layer:** extract keywords and required skills from the job posting (simple TF-IDF + curated skills dictionary).
- **Semantic layer:** embed each profile item (work bullet, project, skill) and embed the job requirements; use cosine similarity to rank.
- **Rules layer:** enforce constraints (e.g., only include experiences within last N years if user chooses; include at least one leadership bullet).
- **LLM layer:** draft and rewrite selected bullets to match the job language while staying truthful to user-provided facts.

5.5 Guardrails and quality controls

- **Truthfulness:** do not invent employers, degrees, or certifications. Treat user-supplied facts as the boundary; AI can rephrase and quantify only when supported.
- **Prompt versioning:** store prompt template versions and model version used for each artifact.

- **Evaluation harness:** create a small internal dataset of job postings plus expected extraction outputs; run regression tests on extraction quality.
- **Cost controls:** cache by URL checksum and by (job_id, profile_version, template_id).

6. Tooling & Language Recommendations

Because your current stack is already Python plus Vue on a single droplet, the most efficient path is to stay polyglot only where necessary. Use Python for orchestration, AI, and scraping, and TypeScript in the UI for maintainable front-end logic.

6.1 Recommended languages by subsystem

Subsystem	Language	Why
API backend + business logic	Python	You already run FastAPI; Python ecosystem is strong for scraping, NLP, and LLM orchestration.
Background jobs (scraping/AI)	Python	Reuse the same models/validators; easier shared code and deployments.
Web UI (dashboard)	TypeScript	Stronger correctness, safer refactors, better component contracts.
PDF template rendering	Python (HTML-to-PDF) or Node (Puppeteer)	Choose based on template complexity and hosting constraints; start with Python-first for MVP.
Infrastructure-as-code (optional)	Terraform	Codify droplet, firewall, managed DB, bucket, CDN for repeatable deployments.

6.2 Core libraries and services

Category	Recommendation	Notes
Auth	FastAPI Users or Auth0/Clerk	External auth reduces risk; self-hosted gives full control.
DB ORM + migrations	SQLAlchemy + Alembic	Add migrations before introducing multi-table app data.
Task queue	Celery + Redis	Battle-tested; supports retries, schedules, queues.
Scraping	httpx + BeautifulSoup + Playwright	Layered approach; Playwright for dynamic sites.
Text extraction	trafilatura or readability-lxml	Reduces HTML noise before LLM parsing.
LLM	OpenAI API (structured outputs)	Use JSON schema; keep provider wrapper for optional multi-model fallback.
Embeddings	pgvector in Postgres	Avoids extra infra; good enough for early scale.
File storage	Cloudflare R2 / S3	Store generated PDFs and optionally HTML snapshots.
Observability	Sentry + structured logging	Capture exceptions, task failures, latency.
Analytics	PostHog	Track feature adoption and funnel drop-off.

6.3 Frontend approach inside your existing repo

- Keep the existing Vue navigation build pipeline, but introduce a new Vue app for the dashboard (separate entry point).
- Adopt TypeScript, Pinia (state), and a UI kit (Tailwind or similar) for consistent dashboard styling.
- Serve the compiled dashboard bundle as static assets from FastAPI, similar to how you currently serve the ChapterNav bundle.

7. Development Roadmap (Phased Delivery)

The roadmap below is organized to keep the site deployable at every step. Each phase produces a working slice of functionality and reduces unknowns early (scraping reliability, template rendering, and profile matching).

7.1 Phase 0 - Foundations (no user-visible changes)

- Refactor backend into modules: atlasops/auth, atlasops/jobs, atlasops/profiles, atlasops/resumes, atlasops/agents.
- Introduce Alembic migrations; create initial AtlasOps tables without touching the existing pages table.
- Add Redis and a Celery worker process (systemd units) alongside the existing Uvicorn service.
- Add structured logging and error tracking for both API requests and worker tasks.

7.2 Phase 1 - Authentication + Profile MVP

- Add user accounts (signup/login/reset).
- Create a profile wizard UI for work history, education, skills, projects.
- Store raw profile input and a canonical normalized representation (JSONB plus normalized rows).
- Implement Profile Enhancer agent to expand bullets and normalize skill taxonomy (async task plus review UI).

7.3 Phase 2 - Job URL ingestion + dashboard entries

- Add a dashboard view listing job entries by status (Pending/Applied/etc.).
- Implement job URL import (single plus batch).
- Create Job Scraper pipeline with evidence snapshots and extraction confidence scores.
- Allow user edits to extracted job fields (because parsing will never be 100 percent).

7.4 Phase 3 - Resume generation MVP

- Add a small template library (2-3 templates).
- Implement matching plus ranking to select best profile items for a job.
- Generate resume content JSON, render to PDF, store export, and show a preview/download link.
- Record match score plus a missing requirements list to guide improvements.

7.5 Phase 4 - Application tracking + follow-ups

- Add status updates: user marks applied; store applied timestamp.
- Generate follow-up message variants; allow scheduling reminders (3/5/7 days).
- Implement scheduled jobs (Celery beat) to trigger reminders and update dashboard state.
- Maintain an event history for every status change and generated message.

7.6 Phase 5 - Company deep dive + interview prep

- Implement Company Research agent as an async job triggered after job ingestion.
- Generate a 1-page deep dive report with citations and store as markdown and/or PDF.
- When status becomes Interview Scheduled, generate an interview prep sheet and let user iterate on it.

7.7 Phase 6 - Continuous improvement loop

- Auto-update applications after 30 days with no progress to Not selected by employer (with the ability to override).
- Generate gap analysis: compare job requirements against the user's profile inventory and resume used.
- Recommend concrete learning resources and propose a practice project scoped to the missing skills.

8. API Design: Key Endpoints & Contracts

Design the AtlasOps API as a clean versioned surface (for example, /api/v1). The frontend should not depend on database structures directly; it should depend on stable DTOs (Pydantic models).

8.1 Suggested endpoint map (v1)

```
GET /api/v1/me
POST /api/v1/auth/login
POST /api/v1/auth/logout
POST /api/v1/profile
GET /api/v1/profile
POST /api/v1/jobs/import
GET /api/v1/jobs
GET /api/v1/jobs/{job_id}
PATCH /api/v1/jobs/{job_id}
POST /api/v1/jobs/{job_id}/deep-dive
POST /api/v1/jobs/{job_id}/resumes
GET /api/v1/resumes/{resume_id}
GET /api/v1/resumes/{resume_id}/download
PATCH /api/v1/applications/{application_id}
POST /api/v1/applications/{application_id}/followups
POST /api/v1/applications/{application_id}/interview-prep
POST /api/v1/applications/{application_id}/improvements
```

8.2 Long-running tasks pattern

- For scraping, deep dives, profile enhancement, and resume generation: API returns a task_id immediately.
- UI polls /api/v1/tasks/{task_id} or listens via WebSocket/SSE for progress events.
- Tasks write intermediate artifacts to DB so partial results can still be reviewed.

9. Resume Templates & PDF Generation Strategy

Resume generation should be template-driven. Avoid drawing PDFs from scratch for every resume; instead use a small set of professionally designed templates that can be filled with structured content.

9.1 Practical template formats

- **HTML + CSS templates:** easiest to iterate and preview in-browser; render to PDF using WeasyPrint or headless Chromium.
- **DOCX templates:** great for users who want editable output; can generate DOCX and optionally convert to PDF.
- **LaTeX templates:** best typography, but heavier toolchain and harder for non-technical edits.

9.2 Recommended approach for your stack

- Start with HTML plus CSS templates stored under templates/resume_templates/.
- Render using a worker task (not in the web request).
- Store the final PDF in object storage and keep only metadata plus URL in Postgres.
- Optionally export a DOCX version for editing, using the same resume content JSON.

10. Deployment & Operations Plan

You can keep your current deployment pipeline (GitHub Actions plus SSH pull plus systemd restart) while evolving the runtime into a multi-process app (web plus worker plus scheduler). The key is to make processes explicit and observable.

10.1 Recommended process layout on the droplet

- **atlas_web.service:** Uvicorn + FastAPI (existing).
- **atlas_worker.service:** Celery worker (new).
- **atlas_scheduler.service:** Celery beat (new) for reminders and 30-day auto-updates.
- **redis-server:** local or managed Redis for queue backend.

10.2 CI/CD improvements worth adding early

- Run unit tests and linting in GitHub Actions before deploy.
- Run DB migrations automatically during deploy (carefully; require backup plus rollback plan).
- Build Vue bundles in CI and deploy compiled artifacts (avoid building on the droplet).

11. Security, Privacy, and Compliance

AtlasOps will store sensitive personal data (employment history, education, potentially contact info) and generates content used in real applications. Security should be designed in from the start.

11.1 Must-have security controls

- Encrypt data in transit (TLS) and at rest (PostgreSQL disk encryption; encrypt object storage).
- Use strong password hashing (Argon2/bcrypt) or delegated OAuth provider.
- Implement rate limiting on auth and scraping endpoints.
- Separate secrets from code (env vars plus secret manager if available).
- Maintain audit logs for profile edits and generated artifacts.

11.2 Scraping legality and terms-of-service risks

Company reviews and ratings (for example, Glassdoor) can have restrictive terms. Avoid brittle scraping of protected sites. Where possible, rely on official APIs, licensed datasets, or user-provided data. For public sources (company websites, SEC filings, Wikipedia, press releases), store citations and keep deep dives clearly attributable.

12. Key Risks & Mitigations

Risk	Impact	Mitigation
Scraping failures / anti-bot blocks	Job ingestion unreliable	Layered scraping (HTTP -> Playwright), caching, user-editable fields, allow manual paste of job text.
LLM hallucinations	False resume claims	Treat user input as ground truth; require user confirmation; store provenance; add red-flag detection.
High API costs	Poor unit economics	Caching, smaller models for extraction, batching, daily quotas per tier, async processing.
Template quality issues	Ugly or non-ATS resumes	Start with a few tested templates; validate with ATS simulators; user preview; export DOCX as fallback.
Data privacy concerns	User distrust, compliance exposure	Clear privacy policy, data export/delete tools, least-privilege access, encryption.

Appendix A. Suggested Repository Structure (Monorepo-Friendly)

```
/ (repo root)
app.py # FastAPI entrypoint (thin)
atlasops/
__init__.py
config.py
db/
session.py
models.py
migrations/ # Alembic
auth/
routes.py
service.py
jobs/
routes.py
scraper.py
schemas.py
service.py
profiles/
routes.py
enhancer.py
schemas.py
service.py
resumes/
routes.py
templates/
renderer.py
service.py
agents/
llm_client.py
prompts/
orchestration.py
tasks/
celery_app.py
jobs.py
profiles.py
resumes.py
reminders.py
templates/ # Jinja templates (public pages + app shell)
frontend/
dashboard/ # new Vue app
nav/ # existing ChapterNav (optional split)
static/
css/
javascript/
```

Appendix B. References

1. *AtlasUniversalis Web Stack - Architectural Overview* (provided in project attachments).
2. *AtlasOps: AI-Powered Resume Generator System - Project Overview Document* (Generated Jan 11, 2026; provided in project attachments).