Here’s a high-level analysis of your current codebase, followed by concrete recommendations for how to decompose it into smaller,

domain-focused services and modules, and finally a suggested multi-phase dev plan you can pick up in another thread to execute the

work.

1. Current State and Pain-Points

• api.py is a ~600-line “god” module that mixes

– HTTP routing (FastAPI)

– authentication & config

– RAG queries

– LLM orchestration (goal-setter, critique, follow-up)

– session- & user-memory persistence (direct DB calls)

– periodic background tasks

• query.py is a minimal RAG router, but still bundles

– embedding lookup

– vectorstore client

– HTTP endpoint and business logic

• onboarding\_routes.py contains all trait-bundle logic, storage calls, fallback code, etc., in one file

• prompt\_engine/, agents/ and maintenance/ are lightly-coupled libs, but not formally packaged or tested

• There’s no clear service-to-service boundary, no API contracts, and no CI/CD split

2. Proposed Service & Module Boundaries

A “microservice” here is just a separately deployed FastAPI app (or gRPC) with its own repository, Docker image, schema and

tests.

2.1. Query Service (“vector-service”)

• Responsibility: embed user text, hit Chroma (or other) vector store, return top-K docs

• Public API (REST or gRPC):

– GET /search?prompt=…&top\_k=…&filter\_tags=…

– POST /search { prompt, top\_k, filters } → { documents: [{ text, metadata }, …] }

• Internals:

– embeddings.py → get\_embedding()

– vectorstore.py → Chroma client wrapper

– router.py → just HTTP → run\_query()

2.2. Memory Service (“session-service”)

• Responsibility: store & retrieve user profiles and session logs (Postgres/Redis)

• Public API:

– GET /users/{uuid} → UserProfile

– POST /users → create/update UserProfile

– GET /sessions/{uuid} → [PromptExchange,…]

– POST /sessions → append exchange & plan snapshot

• Internals: storage models (Pydantic + SQLAlchemy), storage‐only routes, background expiration task

2.3. Onboarding Service

• Responsibility: handle onboarding flows, trait bundles, trait evaluation and persistence

• Public API:

– POST /onboarding/trait-bundle (TraitBundleRequest → TraitBundleResponse)

– GET /onboarding/trait-bundle-data?trait\_name=…

– GET /onboarding/next-question?user\_id=…

– POST /onboarding/respond /feedback /complete /reset

• Internals:

– flow manager (onboarding/flow.py)

– trait\_bundle loader + evaluator + storage calls

– router split into submodules (trait\_bundle\_router, user\_progress\_router, feedback\_router)

2.4. Conversation Service (“chat-service”)

• Responsibility: orchestrate a single “chat turn”:

1. auth & parse request

2. fetch profile & history from Memory Service

3. select system prompt via Prompt-Engine lib

4. call Query Service for RAG context

5. call OpenAI chat completion

6. call CritiqueAgent & FollowUpAgent (could live as libs, or as separate “agent” service)

7. push new exchanges/updates to Memory Service

8. return { answer, sources, plan, critique, followup }

• Public API: POST /chat or /generate

• Internals: minimal FastAPI app, depend on shared-libs and call out to other services

2.5. Shared Libraries (pip packages)

• prompt\_engine (templates + select\_prompt)

• agents (goal\_setter, critique\_agent, follow\_up\_agent)

• maintenance scripts → move into a separate “admin‐cli” or scheduler

2.6. API Gateway / Edge

• A thin FastAPI/Nginx/OpenAPI gateway providing:

– authentication, rate-limit, metrics

– route-forwarding: /search → Query Service, /chat → Chat Service, /onboarding → Onboarding Service

– CORS, logging, monitoring

3. Development Plan (phased)

Phase 1: Extract Query Service

• Create new repo/service “vector‐service”

• Copy query.py → service code; split embedding, client, router

• Environment‐driven config (DB path, API port)

• Add OpenAPI schema, unit tests for run\_query, mocking Chroma

• Dockerfile, Docker-Compose stub

• CI: lint, pytest, build image

Phase 2: Extract Memory Service

• Create “session‐service” repo

• Reconstitute all storage\_service.memory.\* & storage\_service.session.\* logic

• Define Pydantic/SQL models, CRUD routes as above

• Background purge as its own startup task

• Unit/integration tests (use sqlite in-memory)

• Container + CI

Phase 3: Extract Onboarding Service

• New “onboarding‐service” repo

• Break onboarding\_routes.py into submodules:

– trait\_bundle\_router.py

– user\_progress\_router.py

– feedback\_router.py

• Isolate flow.py + agent logic; inject storage via Memory Service API instead of direct import if possible

• Tests + Docker + CI

Phase 4: Build Conversation Service

• New “chat-service” repo

• Minimal FastAPI, routers for /chat & /query-legacy if needed

• Inject dependencies:

– vector‐service client library or REST client

– session‐service client

– prompt\_engine & agents libraries

• Move all direct OpenAI calls here; refactor API key handling

• Tests: end-to-end with mocks

• Docker + CI

Phase 5: Package & Publish Shared Libraries

• Split prompt\_engine/, agents/ into versioned pip packages

• Publish to private PyPI or include via Git URLs

• Update all services to depend on them

Phase 6: API Gateway & Deployment

• Build a gateway application:

– unified CORS, auth, routing

– OpenAPI aggregation

– TTL caching, rate lim­its

• Compose all services via Docker-Compose or Kubernetes

• Set up environment variables, secrets

• Health checks, readiness probes

Phase 7: Observability & Hardening

• Centralized logging/metrics (Prometheus, Grafana, ELK)

• Request tracing across services

• Add authentication between services (mTLS or tokens)

• Security review

1. Interface Design Sketch

4.1. Query Service OpenAPI (excerpt)

/search:

GET

query: prompt (str), top\_k (int), filter\_tags (str)

200 → { documents: [ { text: str, metadata: dict }, … ] }

4.2. Memory Service (excerpt)

/users/{uuid}:

GET → UserProfile

POST → create/update → UserProfile

/sessions/{uuid}:

GET → [ PromptExchange ]

POST → { uuid, exchanges: [ … ], plan\_snapshot, … } → 201

4.3. Chat Service

POST /chat { prompt, user\_id, top\_k, … } →

{ answer: str, sources: [ … ], plan: dict, critique: str, followup: str }

4.4. Onboarding Service

POST /onboarding/trait-bundle → TraitBundleResponse

GET /onboarding/trait-bundle-data?trait\_name=…

GET /onboarding/next-question?user\_id=…

POST /onboarding/respond, /feedback, /complete

Each phase can be broken into 1–2 week sprints, with clear pull-request boundaries and automated tests. Let me know which phase

you’d like to pick up first!

It looks your recommending a different structure of services than the current refactored microservices I've created, auth, db, rag,

share, stats, and storage?

codex

Got it—I’ll align the plan to the six services you already have (auth, db, rag, share, stats, storage) instead of proposing new

ones. Here’s how I’d remap and further decompose your existing microservices, plus a phased rollout plan:

1. Service‐to‐Domain Mapping

• auth

– Responsibility: all API key / user-token validation, rate-limit checks, and identity lookup.

– Pull out from api.py: the `X-XAVIGATE-KEY` header check and any other auth helpers.

• db

– Responsibility: raw relational DB access for persistent entities.

– Keep the psycopg2 logic here (session\_memory, user\_memory tables), plus any SQL migrations.

– Expose only CRUD endpoints for user and session records.

• storage

– Responsibility: in-memory or file-based profile/session caching (what you currently have under

`storage\_service.memory`).

– Expose endpoints for load/save user profile, load/save session history.

• rag

– Responsibility: your RAG/query logic.

– Move `query.py` (and its dependencies on embeddings & Chroma) here.

– Expose GET /search or /query only.

• share

– Responsibility: all shared libraries/clients:

– prompt\_engine, agents (goal\_setter, critique, follow\_up), onboarding flow code, Pydantic models.

– Package these into a shared pip/helm chart or Git submodule.

• stats

– Responsibility: metrics, analytics, event logging.

– Hook into every service via a small client lib in share and report counters, latencies, plan success rates, onboarding

funnel metrics, etc.

2. File & Logic Refactor Recommendations

• api.py (currently 600+ lines) → chat-orchestrator service

– Strip out:

• auth checks → call auth service

• direct DB calls (psycopg2) → call db service

• memory imports → call storage service

• run\_query logic → call rag service

Leave only:

1. assemble request

2. fan-out calls (storage, rag, share.agents)

3. call OpenAI (or wrap this in share.agents?)

4. package result + push session log to db/storage

• query.py → rag service

– Rename router to `/search`

– Keep only embedding lookup, vectorstore client, filtering & HTTP endpoint

• onboarding\_routes.py → storage or share

– Ideally move all onboarding REST handlers into a dedicated “onboarding service” but if you want to keep it under

storage:

• Split its router into three sub-routers (trait-bundle, progress, feedback).

• Replace direct storage calls with HTTP calls to storage service.

• storage\_service.memory

– Refactor into storage service repo:

• models.py (Pydantic)

• storage.py (load/save)

• routes.py (FastAPI CRUD)

• storage\_service.session

– Same: session models + conversation manager should live in share or storage, not in api.py

• share libs (prompt\_engine, agents)

– Package as a versioned library that rag, chat and onboarding all depend on.

– Add unit tests for select\_prompt, goal\_setter, critique\_agent, follow\_up\_agent.

3. Phased Dev Plan

Phase 1: Harden “share”

• Extract prompt\_engine/ and agents/ into a standalone library.

• Add tests for each module. Publish to your internal PyPI or git.

Phase 2: Carve out rag service

• Move query.py → new rag microservice repo.

• Implement GET /search, wire in share.get\_embedding, share.vectorstore client.

• Update chat-orchestrator to call rag service over HTTP.

Phase 3: Carve out storage service

• Move storage\_service.memory.\* (and session.\*) into a storage repo.

• Build CRUD routes: /profiles, /sessions.

• Remove direct filesystem or DB imports from api.py, point to storage service.

Phase 4: Carve out db service

• Isolate all psycopg2 / Postgres table logic into db service.

• Expose only HTTP CRUD endpoints for user\_memory and session\_memory.

• Chat-orchestrator and storage service call it as needed.

Phase 5: Extract auth service

• Centralize X-XAVIGATE-KEY logic, JWT or API-key store, into auth.

• All other services call /auth/verify before handling requests (or use FastAPI middleware).

Phase 6: Stats & Observability

• Build stats endpoints in stats service.

• Instrument all services (share lib) to emit events.

Phase 7: Chat-Orchestrator (api.py refactor)

• Strip api.py to its essence: one FastAPI server that:

1. Validates auth via auth service

2. Loads profile & history via storage service

3. Fetches contexts via rag service

4. Builds system prompt via share library

5. Calls OpenAI via share.agents or direct client

6. Critiques & follow-ups via share.agents

7. Persists new logs via db/storage services

8. Emits metrics to stats service

• Add tests and CI.

This way you keep the six services you’ve already stood up, but give each a clear, single responsibility and remove the

intermingled logic in api.py, query.py, and onboarding. Let me know if you’d like to drill into a specific phase or need interface

specs for the HTTP contracts!

user

eventually we are going to have to move to a more secure backend infrastructure using lambdas and sockets. would this structure set us

up for that transition down the road?

codex

Absolutely—by carving your code into small, single-purpose services with clean HTTP contracts, you’ll be in great shape to migrate

each piece onto AWS Lambda (or another serverless platform) and to adopt socket-based protocols later. Here’s why and what to

watch for:

1. Why this “microservice + shared-lib” layout maps well to Lambdas & sockets

• Fine-grained deployment units

– Each service (auth, db, rag, storage, stats, share) becomes its own Lambda function or small group of functions behind

API Gateway.

• Clear, stateless HTTP APIs

– Lambdas are ephemeral: you must externalize all state. Having each service expose only well-defined REST (or WebSocket)

endpoints means you can hook them up to API Gateway (HTTP API or WebSocket API) almost unchanged.

• Shared libraries packaged independently

– Your “share” repo (prompt\_engine, agents, models) becomes a versioned layer (Lambda Layer or internal PyPI). You don’t

duplicate code, but each Lambda imports exactly what it needs.

• Independent scaling & security boundaries

– You can assign each Lambda its own IAM role, VPC/subnet, concurrency limits, secrets in AWS Secrets Manager—and nothing

in one function bleeds into another’s environment.

• Event-driven hooks

– Stats and background tasks become lightweight event→Lambda subscribers (SNS/SQS). Your periodic purge can be a

CloudWatch Event Rule targeting a small “purge” Lambda rather than a thread in a monolith.

2. Key migration tweaks you’ll need

• Peel off any in-Lambda filesystem or in-memory state

– Move user/session storage from local disk or process memory to DynamoDB (or RDS) behind your storage & db services.

• Swap Chroma “PersistentClient” for a managed vector DB

– Hosted Chroma Cloud, Pinecone, or AWS OpenSearch with k-NN plugin works better in serverless; that way your rag Lambdas

never rely on local SQLite.

• Centralize secrets & config

– API keys, DB URLs, vector-store creds should live in AWS Secrets Manager or Parameter Store, injected via environment

vars or Lambda layers.

• Adopt Lambda warm-start best practices

– Initialize SDK clients (OpenAI, Dynamo, vector DB) at top-level so cold starts only pay once per container.

• WebSockets via API Gateway

– If you later need real-time chat, map your chat-service’s “connect/sendMessage/disconnect” routes to API Gateway

WebSocket routes, each backed by a tiny Lambda that routes to storage or invokes downstream Lambdas.

3. Example Lambda-based service breakdown

• auth-lambda

– /verify-key (Authorizer Lambda), issues JWTs or validates API-keys.

• rag-lambda

– GET /search → (get\_embedding + vectorstore.query) → return docs.

• storage-lambda

– GET/POST /profiles, /sessions → backed by DynamoDB.

• db-lambda

– (If you still need RDS) GET/POST /user\_memory, /session\_memory → backed by RDS Proxy for connection pooling.

• chat-lambda

– POST /chat → orchestrates: auth (via JWT Authorizer), storage-lambda (fetch history), rag-lambda (fetch context),

prompt\_engine + OpenAI call, storage-lambda (save exchange), returns answer.

• stats-lambda

– Ingests metrics/events from all other Lambdas (via SNS or direct HTTP call) and writes to CloudWatch or a time-series

DB.

4. Next steps to prep for serverless

1. Refactor each service to remove any local-disk or long-lived threads.

2. Extract shared libs into Lambda Layers or internal packages.

3. Codify each HTTP contract in OpenAPI or AsyncAPI (for WebSockets).

4. Build CI pipelines to package & deploy each service as a separate Lambda (or container image).

5. Experiment with an API Gateway + one Lambda (e.g. rag) proof-of-concept—validate cold start, config, secrets.

With this modular, stateless design in place, moving onto Lambdas and API-Gateway-powered WebSockets will be a straightforward

packaging/deployment exercise rather than a rewrite. Let me know which service you’d like to containerize first, or if you’d like

a deeper dive on any of the serverless patterns above!