* Documenting

every live endpoint (with its inputs, outputs, auth surface, and service‐level responsibilities) gives us a concrete map we can use to

prioritize the refactoring phases.

Below is a per-service cheat-sheet of all the HTTP endpoints you have in place today, their purpose, how to call them, and any special

notes. I’ve also included ports and key environment variables so a new developer can spin up the whole stack end-to-end.

–––

SERVICE 1: Auth Service

• Location: microservices/auth\_service

• Default port: 8014

• Env vars: none (just spins up)

• Endpoints:

– GET /health

• Returns

{ "status": "ok", "service": "auth" }

• Use: simple liveness/readiness check for deployments.

SERVICE 2: DB Service

• Location: microservices/db\_service

• Default port: 8013

• Env vars (in .env or environment):

POSTGRES\_DB, POSTGRES\_USER, POSTGRES\_PASSWORD, POSTGRES\_HOST, POSTGRES\_PORT, DB\_CONNECT\_MAX\_RETRIES, DB\_CONNECT\_RETRY\_INTERVAL

• Endpoints:

– GET /health

• Returns

{ "status": "ok", "service": "db" }

• Use: confirm connectivity to Postgres (actual CRUD lives in code but isn’t exposed yet—you’ll want to add /users, /sessions,

etc.)

SERVICE 3: RAG Service

• Location: microservices/rag\_service

• Default port: 8010

• Env vars: none beyond standard Chroma path (‘chroma\_db’ folder)

• Endpoints:

– GET /health

• Returns

{ "status": "ok", "service": "rag" }

– GET /query/query

• Query parameters:

• prompt (str, required)

• top\_k (int, default=3)

• tags (csv list, optional)

• Returns

{ "result": "<concatenated text from top-K docs>" }

• Use: backwards-compatible RAG search.

– GET /rag/query

• Same signature & behavior as /query/query (just a second mount point).

• (NOTE: there is a “api.py” in this folder that defines a richer chat-orchestration layer—GET /query?… and POST /generate—which is

currently not wired into main.py. We should decide ASAP whether to bring those routes live here or move them to a dedicated chat

service.)

SERVICE 4: Storage (“Session”) Service

• Location: microservices/storage\_service

• Default port: 8011

• Env vars: none (all file-based/Pydantic models)

• Routers included under:

prefix=“/aq” ⇒ AQ scoring

prefix=“/session” ⇒ reflection

prefix=“/profile” ⇒ user-profile CRUD

• Endpoints:

– GET /health

• Returns service status (note: currently reports "service": "rag"—we should fix to "storage")

– POST /aq/api/session/aq

• Body (JSON):

{

user\_id: str,

reflection\_depth: int,

action\_responsiveness: int,

recalibration\_speed: int,

alignment\_literacy: int,

self\_initiated\_behavior: int,

feedback\_integration: int,

emotional\_regulation: int

}

• Returns

{ aq: float, tier: str, message: str }

• Use: record and classify Alignment Quotient.

– POST /session/api/session/reflection

• Body (JSON): { user\_id: str, reflection: str }

• Returns whatever handle\_reflection(...) returns.

• Use: store and process end-of-turn reflections.

– GET /profile/api/user/{user\_id}

• Path param: user\_id (str UUID)

• Returns full UserProfile JSON (Pydantic model).

– PUT /profile/api/user/{user\_id}

• Body (JSON): { name?: str }

• Returns updated UserProfile JSON.

SERVICE 5: Stats Service

• Location: microservices/stats\_service

• Default port: 8012

• Env vars: none

• Endpoints:

– GET /health

• Returns { "status": "ok", "service": "stats" }

– GET /stats/test

• Returns { "message": "Stats route is alive" }

• Use: basic sanity check for analytics ingestion.

SERVICE 6: Shared Library

• Location: microservices/shared

• Not itself a server—packages out common cache, metrics, tag inference, prompt\_engine, agents, etc.

• To consume: add PYTHONPATH=/app or install as a pip editable.

–––

Authentication Notes

• The only routes today that enforce an API key are in the chat-orchestration layer in api.py (X-XAVIGATE-KEY header). All other

services are wide open.

• For MVP you’ll want to:

– Wire auth\_service into each service (e.g. via HTTP call or FastAPI dependency).

– Centralize key storage in env var XAVIGATE\_KEY (or switch to JWTs once you have user login).

MVP-Critical Endpoints (suggested priority)

1. Chat /generate and/or GET /query?… with auth header

2. RAG search (GET /query/query)

3. Profile load and update (GET/PUT /profile/api/user/{id})

4. Session logging (POST /session/api/session/reflection)

5. AQ scoring (POST /aq/api/session/aq)

6. Health checks & stats (for deployment pipelines)

Next Steps

• Please confirm that these match what you see in your running containers, and whether you’d like to fold in the richer

chat-orchestration endpoints from api.py now or postpone that until after the six-service split.

• Let me know if any endpoints are missing, or if you’d like example curl invocations, OpenAPI snippets, or a Postman/Insomnia

collection generated.

\*\*\*\*\*\*\*\*\*\*\*UPDATED REFACTOR\*\*\*\*\*\*\*\*\*\*\*\*

■■■ Phase 1: Scaffold a New Chat Service ■■■

• Create a new folder microservices/chat\_service/ (or a new repo)

• Add:

– main.py with a FastAPI app, cors, health-check on /health

– Dockerfile (base python:3.13-slim, install requirements, expose 8015)

– requirements.txt (FastAPI, httpx, pydantic, python-dotenv, OpenAI SDK)

– client.py (thin wrappers around httpx calls to auth, rag, storage, stats services)

• Wire up a FastAPI dependency that:

1. Reads `X-XAVIGATE-KEY` header

2. Calls `auth\_service/verify` (or our future `/auth/verify-key`) to validate

■■■ Phase 2: Lift & Refactor Orchestration Endpoints ■■■

• Copy over POST /generate (and/or GET /query) from rag\_service/api.py into chat\_service/main.py.

• Rename endpoint to /chat (or /generate) and update import paths:

– Replace direct from storage\_service.\* imports with await client.get\_profile(user\_id) etc.

– Replace run\_query(...) with await client.rag\_query(prompt, …)

– Replace direct OpenAI client with a singleton openai = OpenAI(api\_key) in this service

– Replace direct session-upsert with await client.append\_session(uuid, conversation\_log)

– Remove background purge thread (belongs in rag or storage services)

■■■ Phase 3: Wire in HTTP Service-to-Service Calls ■■■

• In chat\_service/client.py use httpx.AsyncClient to call:

1. \*\*Auth\*\* → validate key

2. \*\*Storage\*\* → GET `/profile/api/user/{id}`, GET `/session/api/session/history/{id}`

3. \*\*RAG\*\* → GET `/rag/query?prompt=…&top\_k=…&tags=…`

4. \*\*Stats\*\* → fire any analytics events (e.g. POST `/stats/event`)

5. \*\*Storage\*\* → POST `/session/api/session/reflection` or a new `/chat/log` if you prefer

• Configuration via env:

– `AUTH\_URL`, `STORAGE\_URL`, `RAG\_URL`, `STATS\_URL`, `XAVIGATE\_KEY`, `OPENAI\_API\_KEY`

■■■ Phase 4: Remove Orchestration from RAG Service ■■■

• In microservices/rag\_service/main.py drop the old import api + router include

• Keep only pure RAG endpoints (/health, /query)

• Delete or archive rag\_service/api.py once chat-service is stable

■■■ Phase 5: CI/CD & Containerization ■■■

• Add Dockerfile in chat\_service/, map port 8015

• Add a simple pytest suite:

– Unit-test client.py with respx mocks for downstream calls

– Integration test: spin up chat\_service + lightweight mocks for auth, rag, storage (via httpx.MockTransport)

• Add GitHub Actions (or your CI):

– Lint (flake8/mypy), pytest, build image

■■■ Phase 6: End-to-End Smoke & Canary ■■■

• Deploy chat-service behind your API Gateway on /chat

• Point a test frontend at /chat and confirm:

– Invalid key → 401

– Valid key + simple prompt → hits rag, storage, returns answer + plan + critique + follow-up

• Monitor logs & metrics (via stats-service) for errors and latencies

■■■ Phase 7: Cleanup & Optimize ■■■

• Remove any leftover direct imports or dead code in other services

• Tune timeouts, batch some downstream calls to reduce tail latency

• Flesh out OpenAPI spec (auto-generate from FastAPI) and share with frontend team

• (Optional) Introduce gRPC or AWS Lambda packaging once the API shape is stable

I spotted that the auth\_service is crashing because it can’t find the requests package. I’ve just added requests to

microservices/auth\_service/requirements.txt.

Please rebuild and restart your auth container:

# From your repo root:

docker compose build auth\_service

docker compose up -d auth\_service

docker compose ps # you should now see xavigate\_auth\_service in the list

Then verify it’s healthy:

# Bypass the gateway and talk directly to the container:

curl http://localhost:8014/health

# Expected: {"status":"ok","service":"auth"}

Once that succeeds, test again through NGINX:

curl http://localhost:8080/api/auth/health

# Expected: {"status":"ok","service":"auth"}

Also, note that your Swagger UI and OpenAPI JSON live under the /api/chat prefix now. For example:

• Swagger UI: http://localhost:8080/api/chat/docs

• OpenAPI spec: http://localhost:8080/api/chat/openapi.json

If you try to fetch /openapi.json at port 8080 you’ll still get a 404 because NGINX only proxies /api/chat/… to the chat

service.

After you’ve confirmed auth\_service is back up, everything should flow end-to-end:

1. `POST /api/mntest/submit` → stores MNTEST scores

2. `GET /api/mntest/result` → retrieves them

3. `GET /api/rag/query?...` → fetch glossary

4. `POST /api/storage/session-memory` & `GET /api/storage/session-memory/{id}` → session memory

5. `POST /api/chat/query` → full chat orchestration

Let me know once auth is healthy again and we can proceed with wiring the memory and prompt-builder logic.

Smoke test the gateway endpoints

# Auth health

curl http://localhost:8080/api/auth/health

# Chat OpenAPI spec

curl http://localhost:8080/api/chat/openapi.json

# Chat Swagger UI

open http://localhost:8080/api/chat/docs # on macOS this opens the browser

# Submit MNTEST (quick check)

curl -X POST http://localhost:8080/api/mntest/submit \

-H "Authorization: Bearer test" \

-H "Content-Type: application/json" \

-d '{"userId":"u1","traitScores":{"creative":8.0}}'

curl http://localhost:8080/api/mntest/result?userId=u1 \

-H "Authorization: Bearer test"

# RAG search

curl http://localhost:8080/api/rag/query?prompt=creative&top\_k=3

# Chat

curl http://localhost:8080/api/chat/query \

-H "Authorization: Bearer test" \

-H "Content-Type: application/json" \

-d '{"userId":"u1","username":"u1","fullName":"U

One","traitScores":{"creative":8.0},"message":"Hello","sessionId":"s1"}'

If Docker Desktop is already running, try a quick docker info—if that also errors, you’ll need to restart the daemon:

# On macOS you can restart from the whale icon → Troubleshoot → Restart

Once the daemon is healthy, all containers should come up and the NGINX gateway will route /api/auth/health, /api/chat/docs,

/api/chat/openapi.json, etc., without 502s or 404s. Let me know when you’re green across the board!

Now you can re-run:

docker compose down

docker compose up -d

And test again:

# MNTEST submit

curl -X POST http://localhost:8080/api/mntest/submit \

-H "Authorization: Bearer test" \

-H "Content-Type: application/json" \

-d '{"userId":"u1","traitScores":{"creative":8.0}}'

# MNTEST result

curl "http://localhost:8080/api/mntest/result?userId=u1" \

-H "Authorization: Bearer test"

Also remember to quote URLs when using &:

curl "http://localhost:8080/api/rag/query?prompt=creative&top\_k=3"

And ensure your JSON body is on one line or properly escaped when calling /api/chat/query:

curl http://localhost:8080/api/chat/query \-H "Authorization: Bearer test" \-H "Content-Type: application/json" \-d '{"userId":"u1","username":"u1","fullName":"User One","traitScores":{"creative":8.0},"message":"Hello","sessionId":"s1"}'

With those adjustments, everything should route correctly through NGINX to each service. Let me know if you see the expected

JSON responses now!