awesome — here's a **single, synthesized technical architecture** for **HealthIQ-AI v5**, blending:

- the canonical/immutable backbone we designed,
- Claude's structural/service-layer recommendations,
- Loveable AI's UX & realtime/progress ideas (only those that fit our stack now).

I'll show the **full folder architecture** and **code inclusions** devs can drop in.
Then I'll list the **build steps (1→6)** and give you **detailed instructions for Step 1 only** (so we move carefully, as you prefer).

HealthIQ-AI v5 — Synthesized Technical Architecture

Tech stack (decided)

- **Backend**: Python 3.11+, FastAPI, Pydantic v2, Uvicorn, SQLAlchemy (Supabase/Postgres), Celery (background jobs), Redis (queues), pytest + mypy
- **AI**: Gemini (narrative layer; fed with structured JSON), prompt templates in repo
- **Frontend**: Next.js (App Router) + React + TypeScript, Tailwind + Radix UI, TanStack Query (server state), Zustand (UI state)
- **Data viz**: Recharts (upgrade to ECharts if needed), D3 for custom heatmaps
- **Streaming**: SSE for pipeline progress (upgrade to WebSockets only if bi-directional becomes necessary)
- **Infra**: Supabase (auth, DB, storage), Docker, GitHub Actions, Sentry, OpenTelemetry

---

Repository layout (monorepo)

```
healthiq/
├─ backend/
│  ├─ app/                    # FastAPI app (APIs, middleware)
│  │  ├─ main.py
│  │  ├─ middleware.py
│  │  └─ routes/
│  │     ├─ analysis.py       # start analysis, get result (DTO), SSE events
│  │     ├─ ingest.py         # upload preview (canonical mapping)
```

```
│ │   └─ health.py          # liveness, readiness
│ ├─ core/
│ │ ├─ canonical/           # SSOT + normalization (Claude: “boundary is sacred”)
│ │ │ ├─ resolver.py        # alias ↦ canonical (load from ssot yaml)
│ │ │ └─ normalize.py       # names+units normalization → BiomarkerPanel
│ │ ├─ models/              # Pydantic (frozen, extra="forbid")
│ │ │ ├─ biomarker.py       # BiomarkerValue, BiomarkerPanel
│ │ │ ├─ user.py            # UserContext
│ │ │ ├─ context.py         # AnalysisContext
│ │ │ └─ results.py         # ClusterHit, InsightResult, AnalysisDTO
│ │ ├─ pipeline/
│ │ │ ├─ context_factory.py   # ingest → normalize → AnalysisContext
│ │ │ ├─ scorer.py          # biomarker → ScoreResult/flags
│ │ │ ├─ orchestrator.py     # orchestrates: score → clusters → insights → dto
│ │ │ └─ events.py          # SSE event emitter (phase updates)
│ │ ├─ clustering/          # USP engines (range-agnostic, canonical-only)
│ │ │ ├─ engine.py          # BaseClusterEngine
│ │ │ ├─ rules.py           # declarative rules over canonical IDs + flags
│ │ │ ├─ heart.py
│ │ │ ├─ inflammation.py
│ │ │ └─ metabolic_age.py
│ │ ├─ insights/
│ │ │ ├─ base.py            # BaseInsight (reads AnalysisContext)
│ │ │ ├─ registry.py
│ │ │ └─ cardiovascular.py
│ │ ├─ dto/
│ │ │ └─ builders.py        # context+outputs → FE DTOs (no logic here)
```

```
| | ├─ services/              # (Claude) cross-cutting services
| | | ├─ storage.py           # Supabase buckets
| | | ├─ notifications.py      # email/push
| | | └─ reports.py           # PDF/Shareable report generator
| | └─ logging.py
| ├─ ssot/                    # single source of truth (editable by domain team)
| | ├─ biomarkers.yaml         # canonical IDs, synonyms, display names
| | ├─ ranges.yaml             # reference ranges + evidence
| | └─ units.yaml
| ├─ jobs/
| | ├─ worker.py               # Celery worker (analysis pipeline)
| | └─ tasks.py                # task: run_analysis(analysis_id, payload)
| ├─ tests/
| | ├─ enforcement/
| | | └─ test_canonical_only.py # post-normalization maps must be canonical-only
| | ├─ integration/
| | | └─ test_pipeline_e2e.py
| | └─ unit/
| |     ├─ test_resolver.py
| |     ├─ test_normalize.py
| |     ├─ test_scorer.py
| |     └─ test_clusters.py
| └─ tools/
|     └─ seeds.py               # seed SSOT → DB
├─ frontend/
| ├─ app/
| | ├─ upload/page.tsx
```

```
│ │ ├─ analysis/[analysisId]/page.tsx
│ │ └─ layout.tsx
│ ├─ components/
│ │ ├─ Analysis/
│ │ │ ├─ ProgressPipeline.tsx      # (Loveable) phase/sse progress
│ │ │ ├─ RealtimeNormalizer.tsx     # raw→canonical mapping preview
│ │ │ └─ ClusterFormation.tsx       # staged animations (reduced motion aware)
│ │ ├─ Visualization/
│ │ │ ├─ InteractivePanels/
│ │ │ ├─ ClusterRadar/
│ │ │ ├─ TemporalTrends.tsx
│ │ │ └─ CorrelationHeatmap.tsx     # deterministic, backend-driven
│ │ ├─ Narrative/
│ │ │ ├─ InsightCards/
│ │ │ ├─ RecommendationEngine.tsx
│ │ │ └─ ProgressTracking.tsx
│ │ └─ Collaboration/
│ │     ├─ ShareableReports.tsx
│ │     └─ ProviderShare.tsx
│ ├─ lib/
│ │ ├─ api.ts                    # typed client (fetch, SSE)
│ │ ├─ state.ts                  # Zustand store (selection, bookmarks)
│ │ └─ queryClient.ts            # TanStack Query setup
│ └─ styles/ (tailwind.css etc.)
├─ docs/
│ ├─ architecture.md
│ ├─ api_contracts.md
```

```
│  └─ ux_playbook.md
└─ ops/
   ├─ docker/
   ├─ k8s/
   ├─ github-actions/
   └─ observability/
```

Minimal code inclusions (drop-in stubs)

Core: resolver.py (SSOT loader + alias resolution)

```
# backend/core/canonical/resolver.py
from typing import Dict, Set
import yaml, pathlib


class BiomarkerAliasResolver:
    def __init__(self, canonical: Set[str], alias_map: Dict[str, str]):
        self.canonical = set(canonical)
        self.alias_to_canonical = {k.lower(): v for k, v in alias_map.items()}

    def to_canonical(self, key: str) -> str:
        k = key.strip().lower().replace(" ", "_")
        return self.alias_to_canonical.get(k, k)

    def is_canonical(self, key: str) -> bool:
        return key in self.canonical

def load_from_ssot(base: pathlib.Path) -> BiomarkerAliasResolver:
```

```
    bm = yaml.safe_load((base / "biomarkers.yaml").read_text())
    canonical = set(bm["canonical"])
    alias_map = bm.get("aliases", {})
    return BiomarkerAliasResolver(canonical, alias_map)
```

Core: normalize.py (names+units → BiomarkerPanel)

```
# backend/core/canonical/normalize.py
from typing import Dict, Any
from pydantic import ValidationError
from ..models.biomarker import BiomarkerPanel, BiomarkerValue
from .resolver import load_from_ssot
import pathlib

SSOT_BASE = pathlib.Path(__file__).parents[2] / "ssot"

def normalize_panel(raw: Dict[str, Any]) -> BiomarkerPanel:
    r = load_from_ssot(SSOT_BASE)
    canon = {}
    for raw_key, entry in (raw or {}).items():
        key = r.to_canonical(raw_key)
        if isinstance(entry, dict):
            val = entry.get("value", entry.get("Value"))
            unit = entry.get("unit", entry.get("Unit", ""))
            status = (entry.get("status") or "unknown").lower()
            display = entry.get("displayName") or entry.get("name")
        else:
            val, unit, status, display = entry, "", "unknown", None
        try:
```

```
            canon[key] = BiomarkerValue(
                value=float(val), unit=unit, status=status,
                display_name=display, provenance={"source":"upload","original_key":raw_key}
            )
        except (TypeError, ValueError, ValidationError):
            continue
    return BiomarkerPanel(biomarkers=canon)
```

Models: biomarker.py, user.py, context.py, results.py (frozen)

```
# backend/core/models/biomarker.py
from pydantic import BaseModel, Field, ConfigDict
from typing import Optional, Dict, Literal


class BiomarkerValue(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    value: float
    unit: str = ""
    status: Optional[Literal["low","normal","high","unknown"]] = "unknown"
    display_name: Optional[str] = None
    provenance: Optional[dict] = None


class BiomarkerPanel(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    biomarkers: Dict[str, BiomarkerValue] = Field(default_factory=dict)


    def get(self, k: str) -> Optional[BiomarkerValue]:
        return self.biomarkers.get(k)
# backend/core/models/user.py
```

```
from pydantic import BaseModel, ConfigDict
from typing import Optional, Literal

class UserContext(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    age: int
    sex: Literal["male","female","other"]
    bmi: Optional[float] = None
    waist_cm: Optional[float] = None
# backend/core/models/context.py
from pydantic import BaseModel, ConfigDict
from .biomarker import BiomarkerPanel
from .user import UserContext

class AnalysisContext(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    panel: BiomarkerPanel
    user: UserContext
# backend/core/models/results.py
from pydantic import BaseModel, ConfigDict
from typing import Dict, List, Optional

class ClusterHit(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    id: str
    score: float
    confidence: float
```

```
    reasons: List[str]
    evidence: Dict[str, float]  # canonical biomarker → numeric value

class InsightResult(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    id: str
    headline: str
    summary: str
    cluster_id: Optional[str] = None
    confidence: Optional[float] = None
    provenance: Optional[List[dict]] = None

class AnalysisDTO(BaseModel):
    model_config = ConfigDict(frozen=True, extra="forbid")
    analysis_id: str
    user_context: dict
    panel_overview: dict
    clusters: List[ClusterHit]
    insights: List[InsightResult]
    provenance: dict
```

SSE events (progress) + contracts (Loveable's progress UI)

```
# backend/core/pipeline/events.py
from typing import Optional, Dict
from fastapi import Request
from fastapi.responses import StreamingResponse
import asyncio, json, time
```

```
def sse_event(payload: Dict) -> bytes:
    return f"event: analysis_status\ndata: {json.dumps(payload)}\n\n".encode()


async def stream_status(analysis_id: str):
    phases = ["ingest","normalize","scoring","clustering","insights","complete"]
    started = time.time()
    for i, phase in enumerate(phases, 1):
        yield sse_event({
            "analysis_id": analysis_id,
            "phase": phase,
            "progress": i/len(phases),
            "updated_at": int(time.time())
        })
        await asyncio.sleep(0.2)


def sse_endpoint(analysis_id: str):
    return StreamingResponse(stream_status(analysis_id), media_type="text/event-stream")
```

API routes (analysis start, events, result)

```
# backend/app/routes/analysis.py
from fastapi import APIRouter, BackgroundTasks
from pydantic import BaseModel
from ...core.pipeline.events import sse_endpoint
from ...core.canonical.normalize import normalize_panel
from ...core.pipeline.context_factory import build_analysis_context
from ...core.pipeline.orchestrator import run_pipeline_sync # first pass sync; move to Celery


router = APIRouter()
```

```
class StartPayload(BaseModel):
    biomarkers: dict
    user: dict

@router.post("/analysis/start")
def start_analysis(payload: StartPayload):
    # Create analysis_id, persist raw
    analysis_id = "an_"  # TODO: generate ULID
    # For MVP: run inline (later move to Celery and return immediately)
    run_pipeline_sync(analysis_id, payload.model_dump())
    return {"analysis_id": analysis_id}

@router.get("/analysis/events")
def analysis_events(analysis_id: str):
    return sse_endpoint(analysis_id)

@router.get("/analysis/result")
def analysis_result(analysis_id: str):
    # TODO: read from DB; for MVP return stub
    return {"analysis_id": analysis_id, "clusters": [], "insights": []}
```

Frontend: core pieces for Loveable's UX (minimal stubs)

```
// frontend/lib/api.ts
export async function startAnalysis(payload: any) {
  const res = await fetch("/api/analysis/start", { method: "POST", body: JSON.stringify(payload),
headers: { "Content-Type": "application/json" }});
  return res.json();
```

```
}

export function openSSE(analysisId: string, onEvent: (e: any)=>void) {
  const es = new EventSource(`/api/analysis/events?analysis_id=${analysisId}`);
  es.addEventListener("analysis_status", (evt: MessageEvent) => onEvent(JSON.parse(evt.data)));
  return () => es.close();
}
// frontend/components/Analysis/ProgressPipeline.tsx
"use client";
import { useEffect, useState } from "react";
export default function ProgressPipeline({ analysisId }: { analysisId: string }) {
  const [phase, setPhase] = useState("idle"); const [progress,setProgress]=useState(0);
  useEffect(() => {
    const es = new EventSource(`/api/analysis/events?analysis_id=${analysisId}`);
    const handler = (e: MessageEvent) => { const d = JSON.parse(e.data); setPhase(d.phase);
setProgress(d.progress); };
    es.addEventListener("analysis_status", handler);
    return () => es.close();
  }, [analysisId]);
  return <div className="p-4 rounded-xl border">
    <div className="text-sm font-medium">Analysis progress</div>
    <div className="mt-2 h-2 bg-gray-200 rounded"><div className="h-2 bg-indigo-600 rounded"
style={{width:`${Math.round(progress*100)}%`}}/></div>
    <div className="mt-2 text-xs text-gray-600">Phase: {phase}</div>
  </div>;
}
```

Build plan (high level), then detailed Step 1

Steps 1–6 overview

1. **Canonical boundary + models** (resolver, normalize, Biomarker* models, enforcement test)
2. **ContextFactory & Orchestrator** (immutable AnalysisContext; pass canonical-only downstream; plug SSE)
3. **Scorer + flags** (per-biomarker scoring; produce inputs for clusters; unit tests)
4. **Cluster engines (USP)** (heart, inflammation, metabolic age) + rules + evidence/confidence fields
5. **DTOs + FE progress shell** (ProgressPipeline + RealtimeNormalizer; basic radar & cards)
6. **Narratives & services** (Gemini prompt layer; notifications, storage, shareable reports)

And here are the detailed instructions for Step 1.

Step 1 — Canonical boundary + models (do this now)

Goal: From messy upload keys → strict **canonical-only** BiomarkerPanel, guaranteed by a test.

1. Create:
    - backend/core/canonical/resolver.py (SSOT loader + alias resolution)
    - backend/core/canonical/normalize.py (maps names/units → BiomarkerPanel)
    - backend/core/models/{biomarker.py,user.py,context.py,results.py}
    - backend/tests/enforcement/test_canonical_only.py
2. Paste the snippets above for these files unchanged.
3. Run the guard test:

```
$env:PYTHONPATH = "backend"

pytest backend\tests\enforcement\test_canonical_only.py -q
```

4. ✅ When green, we have a hard boundary: **aliases exist only at ingest**, everything else is canonical & immutable.

If you want, I'll guide **Step 2** next (ContextFactory + Orchestrator + SSE status), with small, safe increments so the team can start wiring Loveable's progress UI immediately.