Deliverable A: Product Requirements Document (PRD) - CORTEX

1.0 Vision and Strategic Goal

Vision Statement

To provide a centralized, live platform where teams can instantly visualize pre-analyzed customer feedback and collaboratively manage emergent issues in a transparent, globally accessible environment.

Strategic Goal

CORTEX will function as a dedicated collaboration and insights hub. Its primary goal is to act as the final step in the data analysis workflow, where users upload locally processed, sanitized datasets to populate an interactive dashboard. The platform's core value is not in data processing, but in facilitating data-driven conversations and actions. It will achieve this by offering clear visualizations and a live, globally public issue tracking system, enabling team members across an organization to identify, flag, and monitor the resolution of key customer issues derived from the data.

2.0 Target User Persona and Problem Statement

Primary Persona

"Alex," a Business Analyst or Product Manager. Alex's team runs sentiment analysis models on their local machines, producing a set of structured data files (CSVs/XLSX) that contain insights like sentiment scores and topic clusters.

Problem Statement

Alex has access to perfectly analyzed datasets but lacks a shared, interactive tool to visualize the findings and coordinate a response with other teams (e.g., support, development). Emailing spreadsheets is inefficient and leads to version control issues. They need a web-based platform where they can simply upload their final data files to generate a consistent dashboard. Crucially, they need a simple, centralized system to create and track "issue tickets" based on the insights, ensuring that all stakeholders across the company have visibility into what problems are being addressed and their current status.

3.0 Product Goals and Success Metrics

Goal 1: Instantaneous Insight Visualization

Enable users to go from a bundle of pre-analyzed data files to a fully interactive dashboard in seconds.

- **Key Performance Indicator (KPI): Time-to-Dashboard.** The time from a successful file bundle upload to the rendering of all dashboard visualizations.
- **Target Metric:** The system must successfully ingest the data bundle and render the complete "Command Center" dashboard in under 20 seconds.

Goal 2: Facilitate Transparent, Global Issue Tracking

Provide a single, platform-wide system for all users to create, track, and resolve issues derived from customer feedback, fostering cross-team awareness.

- **Key Performance Indicator (KPI): Issue Engagement Rate.** The percentage of created issues that are updated (e.g., moved to 'cleared') within a 7-day period.
- **Target Metric:** Achieve an issue engagement rate of over 60%, indicating active use of the collaborative tracking feature.

4.0 Feature Epics

Epic 1: Multi-File Data Ingestion

Provide a frictionless system for users to upload a specific bundle of three pre-analyzed files (reviews, comment_cluster, eval_sentiment). This includes robust validation of file formats and their relational integrity based on a shared id key.

Epic 2: Multi-Frame Insights Dashboard

Create a central command center with a fixed three-column layout. The left sidebar provides navigation to four distinct frames in the central content area (Command Center, Product Pulse, Service Hub, Integrations). The right sidebar provides a live, platform-wide view of the issue tracker.

Epic 3: Global Issue Management

Implement a collaborative, platform-wide issue tracking system. This system allows any user to create an issue based on insights from the data (e.g., topic clusters) and track its status

('pending' or 'cleared'). All issues are visible to all users of the platform.

Epic 4: User and Project Management

Establish a secure architecture supporting user registration, authentication, and the logical separation of uploaded data into distinct projects.

5.0 Assumptions and Out of Scope

Assumptions

- All data cleaning, preprocessing, sentiment analysis, and clustering are performed by the user on their local machine before upload. The platform performs **no** NLP or ML processing.
- Users will upload a complete and correctly formatted bundle of three data files.
- The id field will serve as a valid primary key to join data across the files.

Out of Scope (for v1.0)

- Any on-platform data processing.
- Slack integration (the UI will contain a placeholder).
- Time-series analysis or trend calculation (the dashboard reflects a snapshot from a single upload).
- Private or team-based issue tracking. All issues are public to all users on the platform.
- Customizable user roles and permissions.

Deliverable B: Software Requirements

Specification (SRS) - CORTEX

1.0 System Overview and Architecture

1.1 Architectural Blueprint

The CORTEX platform will be engineered as a streamlined web application with a decoupled frontend and backend. The architecture is simplified to focus on efficient data ingestion, storage, and real-time communication for the issue tracker, as all heavy computation is handled offline by the user.

The system is composed of three core components:

1. **Frontend (React SPA):** A single-page application responsible for all UI rendering, including the fixed three-column layout, data visualizations, and the issue creation/tracking interface.
2. **API Gateway (FastAPI/NodeJS):** A high-performance API that manages user authentication, validates and ingests the uploaded data bundles, and provides real-time WebSocket and REST endpoints for the global issue tracker.
3. **Database (MongoDB):** A NoSQL database that acts as the persistence layer for user accounts, project data (including the ingested analysis results), and the global issues collection.

1.2 Technology Stack Rationale

The technology stack is optimized for data visualization, real-time interaction, and rapid development.

Component	Technology Choice	Justification & Synergy

Frontend Framework	React	Its ecosystem of data visualization libraries (e.g., Recharts) is ideal for building the "Command Center" dashboard. Component-based architecture simplifies managing the multi-frame layout.
Backend API	FastAPI (or NodeJS)	High-performance framework well-suited for building a robust RESTful API for data ingestion and providing real-time WebSocket data for the global issue tracker.
Database	MongoDB	Its flexible, document-based model is a natural fit for storing the various pre-analyzed datasets and the simple schema required for the global issue tracker.
Deployment Platform	Render	Provides generous free tiers for the necessary components (web service, static site, managed database), making it a cost-effective choice.

2.0 Functional Requirements

2.1 Epic: Multi-File Data Ingestion

- **REQ-FN-101:** The system shall provide an interface for an authenticated user to upload a bundle of three specific files: reviews (CSV/XLSX), comment_cluster (CSV/XLSX), and eval_sentiment (CSV/XLSX).
- **REQ-FN-102:** The system must validate the uploaded bundle to ensure all three files are present and contain the required columns as per the defined schemas.
    - reviews: must contain an id column.
    - comment_cluster: must contain id and cluster (list of strings) columns.
    - eval_sentiment: must contain id, sentiment, confidence_score, and polarity_score columns.
- **REQ-FN-103:** Upon successful upload, the system shall parse the files, join the data on the id key, and persist the structured data into the database, linked to the user's project.

2.2 Epic: Multi-Frame Insights Dashboard

- **REQ-FN-201:** The main dashboard shall feature a fixed three-column layout. The left and right sidebars are fixed, while the center column is the primary content area.
- **REQ-FN-202:** The left sidebar shall contain navigation links that, when clicked, render a corresponding view in the center content area:
    - **Command Center:** An empty frame intended for future dashboard visualizations based on the uploaded data.
    - **Product Pulse:** An empty frame for future product-centric insights.
    - **Service Hub:** A view containing the interface for creating new issues.
    - **Integrations:** An empty frame for future integration settings.
- **REQ-FN-203:** The right sidebar shall contain a real-time clock at the top, followed by two distinct, independently scrollable flexbox containers: "Review Explorer" and "Slack Alerts". The "Slack Alerts" container will be an empty placeholder.

2.3 Epic: Global Issue Management

- **REQ-FN-301:** The "Review Explorer" module in the right sidebar shall function as a global issue tracker, visible to all authenticated users on the platform.
- **REQ-FN-302:** The "Service Hub" view shall provide a form for any user to create a new issue. The form will require the user to input:
    - A cluster or issue category (e.g., "Gameplay").

    - An issue keyword (e.g., "Visuals").
    - A description of the issue (e.g., "The game loading visual lags a lot...").
- **REQ-FN-303:** Issues listed in the "Review Explorer" must display a status tag: 'p' for pending (red) and 'c' for cleared (green).
- **REQ-FN-304:** Any user shall be able to change the status of any issue from 'pending' to 'cleared'.
- **REQ-FN-305:** The creator of an issue shall be able to delete it.

3.0 User Interface and Experience (UI/UX) Specification

3.1 Key Screen Wireframe Descriptions

- **Login/Signup Page (3-Frame Flow):** The three-stage flow (Landing -> Form -> Verification) remains as previously specified.
- **Insights Dashboard (3-Column Layout):**
    - **Left Sidebar (Fixed, 250px width):** Contains navigation links: "Command Center," "Product Pulse," "Service Hub," and "Integrations."
    - **Main Content Area (Fluid width):** This area's content changes based on the left sidebar selection. Initially, all frames are empty containers, except for the "Service Hub," which contains the issue creation form.
    - **Right Sidebar (Fixed, 350px width):** This sidebar is fixed and does not scroll.
        - **Clock:** A large digital clock at the top.
        - **Review Explorer (Independently Scrollable):** A container listing all global issues, each with a status tag ('p' or 'c'), description, and creator info. This container has its own vertical scrollbar.
        - **Slack Alerts (Placeholder):** An empty container with the title "SLACK ALERTS" for future implementation.

4.0 Backend and API Specification

4.1 RESTful API Endpoints

Endpoint	Method	Description	Auth	Request Body (JSON)	Success Response	Error Responses
/api/v1/auth/register	POST	Initiates user signup.	None	{"username": "...", "email": "...", "password": "..."}	202 Accepted	400, 409
/api/v1/auth/verify	POST	Verifies the 6-digit code to activate an account.	None	{"email": "...", "code": "..."}	200 OK	400, 401
/api/v1/auth/login	POST	Authenticates a user and returns a JWT.	None	{"username": "...", "password": "..."}	200 OK	401
/api/v1/projects/{id}/upload-bundle	POST	Uploads the bundle of pre-analyzed data files. (Multipart/form-data)	JWT	3 Files	200 OK: {"message": "Data ingested."}	400, 413

/api/v1/issues	GET	Retrieves the list of all global issues for the Review Explorer.	JWT	N/A	200 OK: [{"issue_id": "...", "status": "pending",...}]	N/A
/api/v1/issues	POST	Creates a new global issue.	JWT	{"cluster": "...", "keyword": "...", "description": "..."}	201 Created: {"issue_id": "...",...}	400
/api/v1/issues/{issue_id}	PUT	Updates an issue's status (e.g., to 'cleared').	JWT	{"status": "cleared"}	200 OK: {"message": "Issue updated."}	400, 404
/api/v1/issues/{issue_id}	DELETE	Deletes a global issue.	JWT	N/A	204 No Content	403, 404

5.0 Data Model and Persistence Layer

users Collection

Stores user account information. (Schema remains as previously defined).

projects Collection

Stores metadata for each data upload.

reviews_data Collection

Stores the ingested and joined data from the user-uploaded files, linked to a project_id.

Field Name	Data Type	Description
_id	ObjectId	Unique identifier for the document.
project_id	ObjectId	Foreign key to the projects collection.
review_id	String	The original primary key from the uploaded reviews file.
cluster_info	Array of Strings	Data from the comment_cluster file for this review_id.
sentiment_info	Object	Contains sentiment, confidence_score, and polarity_score.

issues Collection

A single, global collection to store all public issues for the Review Explorer.

Field Name	Data Type	Description

_id	ObjectId	Unique identifier for the issue.
cluster	String	The user-defined issue category (e.g., "Gameplay").
keyword	String	The user-defined issue sub-category (e.g., "Visuals").
description	String	The detailed description of the issue.
status	String	The current status of the issue ('pending' or 'cleared').
created_by	ObjectId	Foreign key to the users collection.
created_at	ISODate	Timestamp of when the issue was created.

6.0 Deployment and Operations

6.1 Cloud Deployment

The application will be deployed to Render.com. The render.yaml is simplified as the Celery worker is no longer needed.

YAML

```yaml
# render.yaml: Declarative infrastructure for CORTEX on Render.com
services:
  # 1. The Backend API
  - type: web
    name: cortex-api
    env: python # or node
    runtime: docker
    dockerfilePath:./backend.Dockerfile
    plan: free
    envVars:
      - key: DATABASE_URL
        fromService:
          type: pserv
          name: cortex-db
          property: connectionString
      - key: JWT_SECRET_KEY
        generateValue: true

  # 2. The React Frontend
  - type: static
    name: cortex-frontend
    runtime: docker
    dockerfilePath:./frontend.Dockerfile
    plan: free
    routes:
      - type: rewrite
        source: /*
        destination: /index.html

databases:
  # 3. The MongoDB Database
  - name: cortex-db
    databaseName: cortex
    plan: free
```

7.0 Architecture

7.1 Overview

0 — TL;DR (for people who skim and cause production fires)

- Ingest: **format-free** uploads (CSV/JSON/XLSX/NDJSON/Parquet/ZIP). Only required column: id. Main file (with text) is always present. Classification & clustering files optional.
- Processing: **Node.js (Fastify) API** → create job → **BullMQ + Redis** queue → **Node workers** do streaming parse, merge by id, validate, upsert to **Postgres** (Prisma ORM).
- Auth: **Firebase Auth** (Google OAuth + email/password). Backend validates tokens.
- Dashboard: **React + Vite + TypeScript + Tailwind**. Widgets are **dynamic** — only render when dataset supports them (label, cluster, timestamp). Top stats show Best/Worst/Highest; polarity used for ranking, not worshipped.
- Dev infra: **Docker Compose** dev stack; migrate to k8s / Cloud Run later.
- API: RESTful, OpenAPI-first patterns (cursor pagination, filters, idempotent jobs) — industry-safe and scale-aware without forcing GraphQL.

---

1 — System components (concise)

- **Frontend**: React (Vite) + TypeScript + Tailwind. React Query for server state. Recharts for visuals.
- **Auth**: Firebase Auth (client SDK) + Firebase Admin validation on server.
- **API**: Node.js + Fastify (TypeScript). OpenAPI spec. X-Request-ID tracing.
- **Queue**: BullMQ with Redis broker.
- **Workers**: Node worker processes (BullMQ) for processDataset, computeAggregates, export.
- **Storage**: Firebase Storage (or S3) for uploaded files.
- **DB**: PostgreSQL primary; Prisma ORM recommended.
- **Monitoring**: Bull Board + Prometheus metrics + Grafana (or managed stack) + Sentry.
- **Local orchestration**: Docker Compose (api, worker, redis, postgres, frontend, bullboard).

---

2 — Ingestion & processing (format-free, id-driven)

2.1 Ingestion assumptions

- **Only required invariant**: every file contains id.
- **Guaranteed**: user uploads **main** dataset containing text (comments).
- **Optional**: label, confidence, cluster_id, cluster_label, timestamp.
- Accepted file formats: CSV/TSV, Excel (xlsx), JSON, NDJSON, Parquet, ZIP containing any of the above. Autodetect encoding (UTF-8 primary, ISO fallback).

2.2 Upload flow (user-facing)

1. Client requests upload token/URLs from POST /api/v1/datasets.
2. Client uploads files (direct to storage signed URLs or via multipart).
3. Client calls POST /api/v1/datasets/{datasetId}/process or API auto-triggers job on successful upload. Job enqueued in BullMQ.

2.3 Worker processing pipeline (job: **processDataset**)

- **Step A — Schema sniff**: stream first N rows from each file. Detect candidate columns (id, text, label, confidence, cluster_id, cluster_label, timestamp).
- **Step B — User mapping (if ambiguous)**: UI shows inferred mapping for confirmation; user can remap columns (one-click).
- **Step C — Validate**: ensure id in main file and text exists. If missing, fail early and produce processingReport.
- **Step D — Stream-merge**: left-join auxiliary files to main by id. Streaming/chunked processing (batch size e.g., 500–2000) using Node streams. Avoid full memory loads.
- **Step E — Normalize**: timestamps → UTC ISO8601; label standardization if possible; scrub PII optionally.
- **Step F — Upsert**: write to records table using upsert (primary key: (dataset_id, id)), in batches for performance. Record meta JSONB for unknown fields.

- **Step G — Aggregation**: compute label distribution, cluster counts, top samples per cluster, timeline buckets (if timestamp exists) — store aggregates in dataset_metrics cache table for fast reads.

- **Step H — Finish**: update jobs status, attach processingReport (errors/warnings), notify front-end or let it poll.

2.4 Failure handling & idempotency

- Jobs are idempotent: use jobId + upsert semantics so re-processing is safe.

- Persistent processingReport records parse errors, missing IDs, duplicates, rejected rows.

- Retries: exponential backoff; max retry count configurable (e.g., 5). Fatal errors set job to failed and surface report.

---

3 — Data model (Postgres + Prisma sketch)

Tables (fields abbreviated)

- orgs (id, name, settings JSONB, created_at)

- users (id, firebase_uid, name, email, role, org_id)

- projects (id, org_id, name, created_at)

- datasets (id, project_id, name, status, uploaded_at, uploaded_by, meta JSONB)

- records (dataset_id, id, text, timestamp TIMESTAMP NULL, label TEXT NULL, confidence FLOAT NULL, cluster_id TEXT NULL, cluster_label TEXT NULL, meta JSONB, created_at)

    - Primary Key: (dataset_id, id)

    - Indexes: dataset_id, label, cluster_id, timestamp

- dataset_metrics (dataset_id, metrics JSONB, computed_at)

- jobs (id, dataset_id, type, status, progress JSONB, processing_report JSONB, created_at, finished_at)

- tickets (id, dataset_id, title, description, record_refs JSONB, assignee_id, status, category, created_by, created_at, updated_at)

Notes: extra unknown columns saved as meta JSONB on records for traceability.

---

4 — API design (industry-standard, fail-safe patterns)

You said "implement only what's needed" but "fail-safe like REST/GraphQL." So: **OpenAPI-first REST** with predictable, cacheable resources, cursor-based pagination, and clear job semantics. That pattern is scalable and generator-friendly.

Key design rules

- Versioned endpoints: /api/v1/....
- Auth header: Authorization: Bearer <Firebase-ID-Token>.
- Use X-Request-ID for tracing/logging.
- Cursor-based pagination (opaque cursors), not offset.
- Idempotent endpoints for resource-creation where necessary (Idempotency-Key header optional).
- Uploads via signed URLs (minimize backend memory usage).
- Jobs for long-running tasks; job endpoints return jobId and status.
- Strict HTTP status code usage and structured error payloads.

Core endpoints (final)

- POST /api/v1/datasets → create dataset meta & return signed upload URLs OR accept multipart small files. (Returns datasetId and uploadUrls).
- POST /api/v1/datasets/{datasetId}/process → enqueue processDataset. Accept options: {timestampFallback: boolean}.
- GET /api/v1/jobs/{jobId} → job status + progress + processingReport URL if failed.
- GET /api/v1/datasets/{datasetId}/summary → returns available flags (hasLabel, hasCluster, hasTimestamp) and cached aggregates (labelDistribution, clusterDistribution, timeline) — this is the dashboard payload.
- GET /api/v1/datasets/{datasetId}/records?limit=&cursor=&filters... → cursor pagination, filters (label, cluster_id, timestamp range, search q).

- POST /api/v1/tickets → create ticket (accepts record_refs array).
- GET /api/v1/tickets?datasetId=&status=&cursor= → tickets listing.
- GET /health → health checks (postgres, redis, storage connectivity).

Response envelope

```
{
  "data": {...},
  "meta": {...},
  "errors": [...]
}
```

Errors include code, message, details.

---

5 — Dashboard rules (dynamic, minimal, useful)

You insisted minimal and dynamic. This is exactly how the UI should behave.

Top-line principle

Render **only** visuals that are supported by the dataset. No empty placeholders. Polarity is used for ranking items in top stats but **not** displayed as a giant KPI.

Guaranteed UI components (if present)

- **Top compact stats row** (always show): Best performing category, Worst performing category, Highest average-confidence group, Total rows. These values use label/cluster info when present.
- **Donut (Classification)**: Render only if hasLabel=true. Segments = label counts.
- **Cluster stacked bar chart**: Render only if hasCluster=true. Each bar = cluster; stacks = label counts. Tooltip shows top sample texts.
- **Timeline (Sentiment over time)**: Render only if hasTimestamp=true.
- **Records table / Browser**: Always present; supports filters and allows creating tickets on row(s).
- **Cluster sample cards**: If clusters exist, show top-N clusters with sample texts and label split.

Interaction behavior

- Click on a chart element applies filter to records table.
- "Create ticket" can be invoked from any table row; if classification absent, category input is free-text or select.
- Provide an inline CTA: Run inference (optional) if hasLabel=false — enqueues a worker job to call external inference microservice or user-provided model endpoint.

Backend contract for summary

GET /datasets/{id}/summary returns available flags and keys only for present metrics; frontend renders per keys present. Example in architecture earlier.

---

6 — Worker & queue patterns (BullMQ specifics)

- Use separate named queues (e.g., dataset-processing, aggregates, exports) for isolation and monitoring.
- Concurrency: worker instances configured per queue with concurrency limits per CPU and memory profile.
- Job payload includes datasetId, fileRefs, jobId, options, userId.
- Handle graceful shutdown: ack in-progress jobs properly or requeue if interrupted.
- Use Bull Board for job inspection and admin requeue.
- Setup rate-limits and max job concurrency to avoid PostgreSQL connection overload. Monitor queue depth and scale workers horizontally.

---

7 — Observability & runbook

- **Logs**: structured JSON logs with requestId, datasetId, jobId. Send to centralized logging (ELK/Cloud Logging).
- **Metrics**: Prometheus metrics (api_latency_seconds, job_processing_time_seconds, rows_processed_total, parse_failures_total) → Grafana dashboards.

- **Tracing**: correlate X-Request-ID across frontend → api → worker → db.
- **Alerts**:
    - Job failure rate > 5% over 10 minutes.
    - Queue depth > threshold.
    - DB error spike or connection saturation.
- **Runbook actions**: job retry, manual requeue, rollback partial writes (staging table cleanup) — documented and accessible via admin endpoints.

---

8 — Security & compliance

- Validate Firebase tokens on each request (no trust of client).
- Signed upload URLs; storage objects not public.
- Access control by org/project: scope queries and job triggers by org_id.
- PII scrub configurable: remove emails/phone/SSNs via regex or optional NER scrub step.
- Rate limit uploads and job creation per org to protect costs.
- TLS everywhere. Secure cookies or Authorization header per your preference.

---

9 — Deployment & scaling guidance

- **Dev**: Docker Compose (api, worker, redis, postgres, frontend, bullboard).
- **Prod**: Split services into containers in Cloud Run / ECS / k8s; use managed Postgres and Redis; autoscale workers based on queue depth.
- **Scaling rules**:
    - API: scale horizontally; keep stateless.
    - Workers: scale to handle backlog; monitor DB write throughput.
    - DB: scale vertically initially, move to read replicas for heavy read loads

(dashboard), or use materialized view pattern for complex aggregations.

- **Migration path**: start Postgres; later add a cold storage layer for old datasets (S3 + archive table refs).

---

10 — Why this design (short rationales)

- **Format-free ingestion** maximizes compatibility with arbitrary model outputs — your product promise.
- **Node + BullMQ** gives a single-language stack (TS) across API and workers, easier dev flow and local orchestration.
- **Postgres** as canonical store for fast, flexible filters, joins, and aggregates — critical for dashboards and ticket relationships.
- **OpenAPI-first REST** provides stability, caching options, and generator compatibility for scaffolding without forcing GraphQL overhead.

7.2 Artifacts

7.2.1 OpenAPI YAML

```
openapi: 3.0.3
info:
  title: CORTEX API
  version: "1.0.0"
  description: |
    CORTEX - API spec (OpenAPI 3). Minimal, scalable, and generator-friendly.
    Auth: Firebase ID token (Bearer).
servers:
  - url: https://api.example.com/api/v1
    description: Production
  - url: http://localhost:4000/api/v1
    description: Local dev

components:
  securitySchemes:
    FirebaseBearer:
```

```
    type: http
    scheme: bearer
    bearerFormat: JWT
    description: "Firebase ID token (client obtains via Firebase SDK)."

  schemas:
    ErrorDetail:
      type: object
      properties:
        code:
          type: string
        message:
          type: string
        details:
          type: object
          additionalProperties: true

    ErrorResponse:
      type: object
      properties:
        errors:
          type: array
          items:
            $ref: '#/components/schemas/ErrorDetail'

    UploadFileRef:
      type: object
      properties:
        filename:
          type: string
        url:
          type: string
        role:
          type: string
          description: "one of: main, classification, clustering, other"
      required:
        - filename
        - url

    DatasetCreateRequest:
      type: object
      required:
        - projectId
        - name
        - files
      properties:
        projectId:
          type: string
```

```
    name:
      type: string
    description:
      type: string
    files:
      type: array
      items:
        type: object
        properties:
          filename:
            type: string
          contentType:
            type: string
          role:
            type: string
            description: "main|classification|clustering|other"
        required:
          - filename
          - role

DatasetCreateResponse:
  type: object
  properties:
    datasetId:
      type: string
    uploadUrls:
      type: array
      items:
        $ref: '#/components/schemas/UploadFileRef'
    jobId:
      type: string
  required:
    - datasetId
    - uploadUrls

ProcessOptions:
  type: object
  properties:
    timestampFallback:
      type: boolean
      description: "If true, use ingest time fallback for missing timestamps (opt-in)."
    dedupeStrategy:
      type: string
      enum: ["first", "last", "merge"]
      description: "How to handle duplicate ids across files."

Job:
  type: object
```

```
  properties:
    jobId:
      type: string
    datasetId:
      type: string
    type:
      type: string
      description: "processDataset|computeAggregates|export"
    status:
      type: string
      enum: ["queued","running","succeeded","failed","cancelled"]
    progress:
      type: object
      additionalProperties: true
    processingReportUrl:
      type: string
    createdAt:
      type: string
      format: date-time
    finishedAt:
      type: string
      format: date-time

  DatasetSummary:
    type: object
    properties:
      datasetId:
        type: string
      meta:
        type: object
        properties:
          rows:
            type: integer
          sources:
            type: array
            items:
              type: string
          timestampRange:
            type: object
            properties:
              from:
                type: string
                format: date-time
              to:
                type: string
      topStats:
        type: object
        properties:
```

```
    bestCategory:
      type: string
    worstCategory:
      type: string
    highestConfidenceCount:
      type: integer
    totalRows:
      type: integer
available:
  type: object
  properties:
    hasLabel:
      type: boolean
    hasCluster:
      type: boolean
    hasTimestamp:
      type: boolean
labelDistribution:
  type: array
  items:
    type: object
    properties:
      label:
        type: string
      count:
        type: integer
clusterDistribution:
  type: array
  items:
    type: object
    properties:
      clusterId:
        type: string
      labelSummary:
        type: array
        items:
          type: object
          properties:
            label:
              type: string
            count:
              type: integer
      sample:
        type: array
        items:
          type: string
timeline:
  type: array
```

```
      items:
        type: object
        properties:
          date:
            type: string
            format: date
          pos:
            type: integer
          neu:
            type: integer
          neg:
            type: integer

Record:
  type: object
  properties:
    datasetId:
      type: string
    id:
      type: string
    text:
      type: string
    timestamp:
      type: string
      nullable: true
    label:
      type: string
      nullable: true
    confidence:
      type: number
      format: float
      nullable: true
    clusterId:
      type: string
      nullable: true
    clusterLabel:
      type: string
      nullable: true
    meta:
      type: object
      additionalProperties: true
    createdAt:
      type: string
      format: date-time

RecordsList:
  type: object
  properties:
```

```
    items:
      type: array
      items:
        $ref: '#/components/schemas/Record'
    nextCursor:
      type: string
      nullable: true

  TicketCreateRequest:
    type: object
    required:
      - datasetId
      - title
    properties:
      datasetId:
        type: string
      title:
        type: string
      description:
        type: string
      recordRefs:
        type: array
        items:
          type: object
          properties:
            datasetId:
              type: string
            id:
              type: string
      assigneeId:
        type: string
      category:
        type: string

  Ticket:
    type: object
    properties:
      id:
        type: string
      datasetId:
        type: string
      title:
        type: string
      description:
        type: string
      recordRefs:
        type: array
        items:
```

```
          type: object
          properties:
            datasetId:
              type: string
            id:
              type: string
        assigneeId:
          type: string
          nullable: true
        status:
          type: string
          enum: ["open","in_progress","closed"]
        category:
          type: string
        createdBy:
          type: string
        createdAt:
          type: string
          format: date-time
        updatedAt:
          type: string
          format: date-time

security:
  - FirebaseBearer: []

paths:
  /health:
    get:
      summary: Health check
      tags: [System]
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  status:
                    type: string
                  services:
                    type: object
                    additionalProperties:
                      type: string
        '503':
          description: Unhealthy
          content:
```

```
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'

/datasets:
  post:
    summary: Create dataset record and return upload URLs
    tags: [Datasets]
    security:
      - FirebaseBearer: []
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/DatasetCreateRequest'
    responses:
      '201':
        description: Created. Returns signed upload URLs and datasetId.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/DatasetCreateResponse'
      '400':
        description: Validation error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ErrorResponse'
      '401':
        description: Unauthorized

/datasets/{datasetId}/process:
  post:
    summary: Trigger processing job for uploaded dataset
    tags: [Datasets]
    security:
      - FirebaseBearer: []
    parameters:
      - name: datasetId
        in: path
        required: true
        schema:
          type: string
    requestBody:
      required: false
      content:
        application/json:
```

```
        schema:
          $ref: '#/components/schemas/ProcessOptions'
    responses:
      '202':
        description: Job accepted
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Job'
      '400':
        description: Bad request
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ErrorResponse'
      '401':
        description: Unauthorized
      '404':
        description: Dataset not found

  /jobs/{jobId}:
    get:
      summary: Get job status and progress
      tags: [Jobs]
      security:
        - FirebaseBearer: []
      parameters:
        - name: jobId
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Job status
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Job'
        '404':
          description: Not found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ErrorResponse'

  /datasets/{datasetId}/summary:
    get:
```

```
    summary: Get dataset summary (dashboard payload). Only returns available metrics.
    tags: [Datasets]
    security:
      - FirebaseBearer: []
    parameters:
      - name: datasetId
        in: path
        required: true
        schema:
          type: string
    responses:
      '200':
        description: Dataset summary
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/DatasetSummary'
      '404':
        description: Not found
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ErrorResponse'

  /datasets/{datasetId}/records:
    get:
      summary: List records (cursor-based pagination) with filters
      tags: [Records]
      security:
        - FirebaseBearer: []
      parameters:
        - name: datasetId
          in: path
          required: true
          schema:
            type: string
        - name: limit
          in: query
          schema:
            type: integer
            default: 50
            maximum: 1000
        - name: cursor
          in: query
          schema:
            type: string
        - name: label
          in: query
```

```
        schema:
          type: string
      - name: cluster_id
        in: query
        schema:
          type: string
      - name: q
        in: query
        schema:
          type: string
          description: Full-text search query
      - name: timestamp_from
        in: query
        schema:
          type: string
          format: date-time
      - name: timestamp_to
        in: query
        schema:
          type: string
          format: date-time
    responses:
      '200':
        description: Record list
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/RecordsList'
      '400':
        description: Invalid query
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ErrorResponse'

  /tickets:
    post:
      summary: Create a ticket (link to record refs optional)
      tags: [Tickets]
      security:
        - FirebaseBearer: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TicketCreateRequest'
      responses:
```

```
    '201':
      description: Ticket created
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Ticket'
    '400':
      description: Validation error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'
    '401':
      description: Unauthorized

  get:
    summary: List tickets (filterable)
    tags: [Tickets]
    security:
      - FirebaseBearer: []
    parameters:
      - name: datasetId
        in: query
        schema:
          type: string
      - name: status
        in: query
        schema:
          type: string
          enum: ["open","in_progress","closed"]
      - name: assigneeId
        in: query
        schema:
          type: string
      - name: limit
        in: query
        schema:
          type: integer
          default: 50
      - name: cursor
        in: query
        schema:
          type: string
    responses:
      '200':
        description: List of tickets
        content:
          application/json:
```

```
          schema:
            type: object
            properties:
              items:
                type: array
                items:
                  $ref: '#/components/schemas/Ticket'
              nextCursor:
                type: string
    '401':
      description: Unauthorized

/tickets/{ticketId}:
  get:
    summary: Get ticket
    tags: [Tickets]
    security:
      - FirebaseBearer: []
    parameters:
      - name: ticketId
        in: path
        required: true
        schema:
          type: string
    responses:
      '200':
        description: Ticket details
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Ticket'
      '404':
        description: Not found
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ErrorResponse'

  put:
    summary: Update ticket (status/assignee/category/description)
    tags: [Tickets]
    security:
      - FirebaseBearer: []
    parameters:
      - name: ticketId
        in: path
        required: true
        schema:
```

```
          type: string
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              status:
                type: string
                enum: ["open","in_progress","closed"]
              assigneeId:
                type: string
              category:
                type: string
              description:
                type: string
    responses:
      '200':
        description: Updated ticket
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Ticket'
      '400':
        description: Validation error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ErrorResponse'

security:
  - FirebaseBearer: []
```

7.2.2 Schema and Postgres DDL

7.2.2.1 schema.prisma (Postgres + Prisma)

```
// schema.prisma
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
```

```
  url      = env("DATABASE_URL")
}

enum JobStatus {
  queued
  running
  succeeded
  failed
  cancelled
}

enum TicketStatus {
  open
  in_progress
  closed
}

model Org {
  id       String    @id @default(uuid())
  name     String
  settings  Json?    @db.JsonB
  createdAt DateTime  @default(now())

  users    User[]
  projects  Project[]
}

model User {
  id        String   @id @default(uuid())
  firebaseUid String   @unique
  name       String?
  email      String   @unique
  role       String
  orgId      String
  createdAt   DateTime @default(now())

  org        Org     @relation(fields: [orgId], references: [id])
  projects   Project[]
  tickets    Ticket[]  @relation("TicketAssignee")
  createdTickets Ticket[] @relation("TicketCreator")
}

model Project {
  id        String    @id @default(uuid())
  orgId     String
  name      String
  createdAt  DateTime  @default(now())
```

```
  org       Org      @relation(fields: [orgId], references: [id])
  datasets  Dataset[]
}

model Dataset {
  id        String   @id @default(uuid())
  projectId String
  name      String
  status    String
  uploadedBy String?
  uploadedAt DateTime @default(now())
  meta      Json?    @db.JsonB

  project   Project  @relation(fields: [projectId], references: [id])
  records   Record[]
  metrics   DatasetMetric[]
  jobs      Job[]

  @@index([projectId])
  @@index([uploadedAt])
}

model Record {
  // 'rowId' maps to external 'id' in uploaded files
  datasetId  String
  rowId      String
  text       String?
  timestamp  DateTime?   @db.Timestamp(6)
  label      String?
  confidence Float?
  clusterId  String?
  clusterLabel String?
  meta       Json?       @db.JsonB
  createdAt  DateTime    @default(now())

  dataset    Dataset     @relation(fields: [datasetId], references: [id])

  @@id([datasetId, rowId])
  @@index([datasetId, label])
  @@index([datasetId, clusterId])
  @@index([datasetId, timestamp])
}

model DatasetMetric {
  id        String  @id @default(uuid())
  datasetId String
  metrics   Json    @db.JsonB
  computedAt DateTime @default(now())
```

```
  dataset    Dataset  @relation(fields: [datasetId], references: [id])

  @@index([datasetId])
}

model Job {
  id             String     @id @default(uuid())
  datasetId      String
  type           String
  status         JobStatus  @default(queued)
  progress       Json?      @db.JsonB
  processingReport Json?    @db.JsonB
  createdAt      DateTime   @default(now())
  finishedAt     DateTime?

  dataset        Dataset    @relation(fields: [datasetId], references: [id])

  @@index([datasetId])
  @@index([status])
}

model Ticket {
  id        String     @id @default(uuid())
  datasetId  String
  title     String
  description String?
  recordRefs Json?     @db.JsonB
  assigneeId String?
  status     TicketStatus @default(open)
  category   String?
  createdBy  String?
  createdAt  DateTime   @default(now())
  updatedAt  DateTime?

  dataset    Dataset   @relation(fields: [datasetId], references: [id])
  assignee   User?    @relation("TicketAssignee", fields: [assigneeId], references: [id])
  creator    User?    @relation("TicketCreator", fields: [createdBy], references: [id])

  @@index([datasetId])
  @@index([assigneeId])
  @@index([status])
}
```

Save that as schema.prisma. Run:

```
npx prisma generate
```

```
npx prisma migrate dev --name init
```

(if you're using CI/migrations differently, generate SQL with `prisma migrate dev --create-only` and use the SQL file directly).

---

2) Postgres DDL (raw SQL migration)

If you prefer raw SQL to apply directly (e.g., via Flyway / Terraform / psql), here's a migration that matches the Prisma models. It assumes `pgcrypto` extension is available for `gen_random_uuid()`.

```
-- 0001_init.sql
CREATE EXTENSION IF NOT EXISTS "pgcrypto";

CREATE TYPE job_status AS ENUM ('queued', 'running', 'succeeded', 'failed', 'cancelled');
CREATE TYPE ticket_status AS ENUM ('open', 'in_progress', 'closed');

CREATE TABLE orgs (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  name text NOT NULL,
  settings jsonb,
  created_at timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE users (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  firebase_uid text UNIQUE NOT NULL,
  name text,
  email text UNIQUE NOT NULL,
  role text NOT NULL,
  org_id uuid NOT NULL REFERENCES orgs(id) ON DELETE CASCADE,
  created_at timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE projects (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  org_id uuid NOT NULL REFERENCES orgs(id) ON DELETE CASCADE,
  name text NOT NULL,
  created_at timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE datasets (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
```

```
  name text NOT NULL,
  status text NOT NULL,
  uploaded_by text,
  uploaded_at timestamptz NOT NULL DEFAULT now(),
  meta jsonb
);

CREATE INDEX idx_datasets_project_id ON datasets (project_id);
CREATE INDEX idx_datasets_uploaded_at ON datasets (uploaded_at);

CREATE TABLE records (
  dataset_id uuid NOT NULL REFERENCES datasets(id) ON DELETE CASCADE,
  row_id text NOT NULL,
  text text,
  timestamp timestamptz,
  label text,
  confidence double precision,
  cluster_id text,
  cluster_label text,
  meta jsonb,
  created_at timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (dataset_id, row_id)
);

CREATE INDEX idx_records_dataset_label ON records (dataset_id, label);
CREATE INDEX idx_records_dataset_cluster ON records (dataset_id, cluster_id);
CREATE INDEX idx_records_dataset_timestamp ON records (dataset_id, timestamp);

CREATE TABLE dataset_metrics (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  dataset_id uuid NOT NULL REFERENCES datasets(id) ON DELETE CASCADE,
  metrics jsonb NOT NULL,
  computed_at timestamptz NOT NULL DEFAULT now()
);

CREATE INDEX idx_dataset_metrics_dataset_id ON dataset_metrics (dataset_id);

CREATE TABLE jobs (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  dataset_id uuid NOT NULL REFERENCES datasets(id) ON DELETE CASCADE,
  type text NOT NULL,
  status job_status NOT NULL DEFAULT 'queued',
  progress jsonb,
  processing_report jsonb,
  created_at timestamptz NOT NULL DEFAULT now(),
  finished_at timestamptz
);
```

```
CREATE INDEX idx_jobs_dataset_id ON jobs (dataset_id);
CREATE INDEX idx_jobs_status ON jobs (status);

CREATE TABLE tickets (
 id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
 dataset_id uuid NOT NULL REFERENCES datasets(id) ON DELETE CASCADE,
 title text NOT NULL,
 description text,
 record_refs jsonb,
 assignee_id uuid REFERENCES users(id),
 status ticket_status NOT NULL DEFAULT 'open',
 category text,
 created_by uuid REFERENCES users(id),
 created_at timestamptz NOT NULL DEFAULT now(),
 updated_at timestamptz
);

CREATE INDEX idx_tickets_dataset_id ON tickets (dataset_id);
CREATE INDEX idx_tickets_assignee_id ON tickets (assignee_id);
CREATE INDEX idx_tickets_status ON tickets (status);
```

Run it with plsql or your migration tool.

---

3) Optional: seed data (tiny helpful snippet)

If you want a tiny seed SQL to test quickly:

```
-- seed.sql
INSERT INTO orgs (id, name) VALUES (gen_random_uuid(), 'DemoOrg');

-- create a user and project
INSERT INTO users (firebase_uid, name, email, role, org_id)
VALUES ('fake-fb-uid', 'Demo User', 'demo@example.com', 'admin', (SELECT id FROM
orgs LIMIT 1));

INSERT INTO projects (org_id, name)
VALUES ((SELECT id FROM orgs LIMIT 1), 'Demo Project');
```

7.3 Docker, env and file layout recommendations

docker-compose.yml

```
version: "3.9"

x-common-environment: &common_env
  NODE_ENV: ${NODE_ENV:-development}
  # Firebase admin JSON should be mounted into the api and worker containers
  FIREBASE_ADMIN_SDK_PATH: /run/secrets/firebase_admin_sdk.json
  FIREBASE_STORAGE_BUCKET:
${FIREBASE_STORAGE_BUCKET:-cortex-storage-bucket}
  # Optional: storage provider credentials envs if using S3 (set in .env if used)
  AWS_ACCESS_KEY_ID: ${AWS_ACCESS_KEY_ID:-}
  AWS_SECRET_ACCESS_KEY: ${AWS_SECRET_ACCESS_KEY:-}
  AWS_REGION: ${AWS_REGION:-}

services:
  postgres:
    image: postgres:16
    restart: unless-stopped
    environment:
      POSTGRES_DB: ${POSTGRES_DB:-cortex}
      POSTGRES_USER: ${POSTGRES_USER:-cortex}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-cortexpass}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "${POSTGRES_PORT:-5432}:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7
    restart: unless-stopped
    ports:
      - "${REDIS_PORT:-6379}:6379"
    volumes:
      - redis_data:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
```

```
    timeout: 5s
    retries: 5

  api:
    build:
      context: ./backend
      dockerfile: Dockerfile
    command: sh -c "npx prisma migrate deploy && node dist/server.js"
    restart: on-failure
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_started
    env_file:
      - .env
    environment:
      <<: *common_env
      DATABASE_URL: ${DATABASE_URL}
      REDIS_URL: ${REDIS_URL:-redis://redis:6379}
      PORT: ${API_PORT:-4000}
      # Optional: useful for local dev debugging
      NODE_OPTIONS: --max_old_space_size=2048
    volumes:
      - ./backend:/usr/src/app
      - ./secrets/firebase_admin_sdk.json:/run/secrets/firebase_admin_sdk.json:ro
    ports:
      - "${API_PORT:-4000}:4000"
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:${API_PORT:-4000}/api/v1/health || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 5

  worker:
    build:
      context: ./worker
      dockerfile: Dockerfile
    command: sh -c "node dist/worker.js"
    restart: on-failure
    depends_on:
      - redis
      - postgres
    env_file:
      - .env
    environment:
      <<: *common_env
      DATABASE_URL: ${DATABASE_URL}
```

```
    REDIS_URL: ${REDIS_URL:-redis://redis:6379}
    QUEUE_PREFIX: ${QUEUE_PREFIX:-cortex}
  volumes:
    - ./worker:/usr/src/worker
    - ./secrets/firebase_admin_sdk.json:/run/secrets/firebase_admin_sdk.json:ro
  healthcheck:
    test: ["CMD-SHELL", "node -e \"process.exit(0)\""]
    interval: 60s
    timeout: 10s
    retries: 3

 bullboard:
  image: ghcr.io/vcapretz/bull-board:latest
  restart: unless-stopped
  environment:
    REDIS_URL: ${REDIS_URL:-redis://redis:6379}
    PORT: ${BULLBOARD_PORT:-3001}
  ports:
    - "${BULLBOARD_PORT:-3001}:3001"
  depends_on:
    - redis

 frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  command: sh -c "npm run dev -- --host 0.0.0.0 --port ${FRONTEND_PORT:-5173}"
  env_file:
    - .env
  environment:
    VITE_API_BASE_URL:
${VITE_API_BASE_URL:-http://localhost:${API_PORT:-4000}/api/v1}
    NODE_ENV: ${NODE_ENV:-development}
  volumes:
    - ./frontend:/usr/src/frontend
  ports:
    - "${FRONTEND_PORT:-5173}:5173"

volumes:
 postgres_data:
 redis_data:
```

.env.example (copy to .env and edit)

```
# .env.example - copy to .env and update with secrets

# General
NODE_ENV=development

# Ports (map to host)
API_PORT=4000
FRONTEND_PORT=5173
BULLBOARD_PORT=3001
POSTGRES_PORT=5432
REDIS_PORT=6379

# Postgres (used by Prisma / backend / worker)
POSTGRES_DB=cortex
POSTGRES_USER=cortex
POSTGRES_PASSWORD=cortexpass
DATABASE_URL=postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@postgres:
${POSTGRES_PORT}/${POSTGRES_DB}?schema=public

# Redis (BullMQ)
REDIS_URL=redis://redis:6379

# Firebase admin SDK (mounted as secret at ./secrets/firebase_admin_sdk.json)
# See README: create a service account JSON and place at
./secrets/firebase_admin_sdk.json
FIREBASE_ADMIN_SDK_PATH=/run/secrets/firebase_admin_sdk.json
FIREBASE_STORAGE_BUCKET=cortex-storage-bucket

# AWS S3 (optional) - set if using S3 instead of Firebase Storage
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_REGION=

# Queue prefix for BullMQ
QUEUE_PREFIX=cortex

# Prisma
# If you're using prisma migrations locally, set this value to the same as DATABASE_URL
# DATABASE_URL is used by prisma migrate deploy in API container
# (already defined above)
```

Minimal repo folder layout (put this into your repository root)

```
/cortex-root
├─ backend/
│  ├─ Dockerfile
│  ├─ package.json
│  ├─ src/
│  │  ├─ server.ts       # Fastify server entry (compiled to dist/server.js)
│  │  ├─ routes/
│  │  └─ services/
│  ├─ prisma/
│  │  └─ schema.prisma
│  └─ prisma/migrations/  # prisma migrate files (after running migrate)
├─ worker/
│  ├─ Dockerfile
│  ├─ package.json
│  ├─ src/
│  │  └─ worker.ts       # BullMQ worker entry (compiled to dist/worker.js)
│  └─ dist/
├─ frontend/
│  ├─ Dockerfile
│  ├─ package.json
│  ├─ src/
│  │  ├─ main.tsx
│  │  └─ App.tsx
│  └─ public/
├─ secrets/
│  └─ firebase_admin_sdk.json  # mount here locally (DO NOT git commit)
├─ docker-compose.yml
├─ .env.example
└─ README.md
```

Dockerfile suggestions (very short—put these in each service folder)

backend/Dockerfile (Node + TS build sample)

```
FROM node:20-alpine AS builder
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build   # compile TS to dist
RUN npx prisma generate

FROM node:20-alpine AS runner
WORKDIR /usr/src/app
ENV NODE_ENV=production
COPY --from=builder /usr/src/app/dist ./dist
COPY --from=builder /usr/src/app/node_modules ./node_modules
COPY --from=builder /usr/src/app/prisma ./prisma
CMD ["node", "dist/server.js"]
```

worker/Dockerfile

```
FROM node:20-alpine AS builder
WORKDIR /usr/src/worker
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:20-alpine AS runner
WORKDIR /usr/src/worker
COPY --from=builder /usr/src/worker/dist ./dist
COPY --from=builder /usr/src/worker/node_modules ./node_modules
CMD ["node", "dist/worker.js"]
```

frontend/Dockerfile (dev-friendly)

```
FROM node:20-alpine
WORKDIR /usr/src/frontend
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
CMD ["npm", "run", "dev", "--", "--host", "0.0.0.0"]
```

Quick start checklist

1. Copy .env.example → .env and fill secrets.
2. Place Firebase service account JSON at ./secrets/firebase_admin_sdk.json (do not commit).
3. docker compose build (or docker compose up --build).
4. API container does npx prisma migrate deploy at startup (it will execute migrations you created); adjust command if you prefer manual migrate.
5. Visit:
    - Frontend: http://localhost:5173
    - API: http://localhost:4000/api/v1/health
    - Bull Board: http://localhost:3001

Notes & small gotchas I already handled

- **Firebase admin JSON** is mounted as a file; don't commit it. Use secrets in production (Kubernetes secrets / cloud secret manager).
- **Prisma migrate** runs at container startup in api via npx prisma migrate deploy. If you prefer manual migrations, remove that step from the command and run migrations locally.
- **Volumes for Postgres/Redis** persist data between restarts.
- **Ports** are configurable via .env.
- **Local dev**: the files are bind-mounted so you can npm run dev inside containers or locally and iterate quickly.
- **Production**: swap Docker Compose with your k8s / cloud deployment; use managed Postgres & Redis and secrets manager.