ChatGPT

Engineering Blueprint: Multi-Protocol API Gateway & Orchestration System

Overview & Objectives

This blueprint outlines a complete architecture and implementation plan for an **API Gateway and Orchestration System** that serves multiple protocols and domains. The goal is to build a unified gateway that exposes REST, GraphQL, gRPC, SOAP, and event-streaming interfaces over a common backend, aligned with open standards and adaptable to various knowledge domains (medicine, finance, law). The system will enforce high quality and spec compliance (OpenAPI, JSON:API, OData, GraphQL, gRPC, AsyncAPI, OAuth2) while orchestrating complex ingestion and AI processing pipelines. We define clear phases, team role responsibilities, and technical steps to guide an AI programming agent (or engineering team) in implementing this system. Key considerations include robust security, fail-fast GPU job handling, structured error handling, and observability for production readiness.

Key System Capabilities:
- **Multi-Protocol API Façade:** A single gateway serving REST (OpenAPI 3.1 with JSON:API and OData support), GraphQL (with introspection & possible schema stitching), gRPC (for internal AI services), SOAP (for legacy integration), and AsyncAPI-defined event streams.
- **Ingestion & Orchestration:** Endpoints to ingest documents/data (including a two-phase PDF pipeline) and orchestrate GPU-bound jobs (e.g. PDF parsing, embedding, extraction) with fail-fast policies (no CPU fallback).
- **Unified Data Model:** A federated data model with core entities (e.g. Organization, Document, Claim, Entity) extended by domain-specific overlays (medical domain aligning with HL7 FHIR, financial with XBRL, legal with LegalDocML standards).
- **Security & Quality:** OAuth 2.0 based authentication with tenant isolation and fine-grained scopes, adherence to JSON:API formatting rules, OData-compliant filtering, thorough validation, and comprehensive test coverage.
- **DevOps & Documentation:** A monorepo with CI/CD pipelines, contract and performance tests (Schemathesis, GraphQL Inspector, Buf, k6), containerized deployment (Docker Compose and optional K8s), plus a developer docs portal bundling Swagger UI, GraphQL Playground, and AsyncAPI explorer.

Multi-Protocol API Interface (Façade Layer)

Objective: Implement a **unified API gateway** that exposes the system's functionality through multiple standard API protocols, ensuring each adheres to its specification and provides a consistent experience. This layer abstracts the underlying services and provides a **facade** that speaks the client's preferred protocol.

- **OpenAPI 3.1 (RESTful JSON API):** Design a comprehensive REST API documented with an OpenAPI 3.1 spec [1]. OpenAPI (formerly Swagger) defines a language-agnostic interface for HTTP APIs, allowing humans and machines to discover capabilities without guesswork [1]. The REST API will follow the **JSON:API** conventions for request/response format (using media type `application/vnd.api+json` [2] [3]) to ensure consistent resource representation. This means standardized fields (`data`, `attributes`, `relationships`, etc.) and error formatting as per JSON:API spec. We will also support **OData-style filtering** on REST endpoints, using query parameters like `?filter=<field> eq <value>` to enable powerful querying of collections. OData is an open protocol that allows creation of queryable and interoperable REST APIs [4]; by supporting its `$filter` syntax [5], clients can perform rich server-side filtering without custom query params. The OpenAPI spec will enumerate all endpoints, their schemas (via Pydantic models), security requirements, and example requests/responses for clarity and testing.

- **GraphQL API:** Provide a **GraphQL** endpoint (e.g. `/graphql`) that exposes a single unified graph of the system's data and operations. GraphQL is a query language and runtime for APIs that uses a strongly-typed schema to describe data [6]. We will auto-generate the GraphQL schema (SDL) from the same Pydantic models used for REST, ensuring consistency in data definitions across protocols. For example, a Pydantic `Document` model can yield a GraphQL `Document` type with corresponding fields. The GraphQL schema will include Query and Mutation types for core operations (e.g., query documents, create ingestion jobs, retrieve results). It will support **introspection** (default in GraphQL) so clients can query the schema and types. We will implement resolvers that allow **cross-resource joins**, so a single GraphQL query can fetch, for instance, an Organization and all Documents associated with it, even if that data resides in different backend stores. If needed, use schema stitching or Apollo Federation techniques to combine subgraphs from microservices [7]. (For example, if the knowledge graph and document store each expose GraphQL, we could federate them.) GraphQL gives clients the flexibility to request exactly the data they need and nothing more [8], which is important for diverse frontends. We will ensure mutations exist for operations like triggering an ingest or extraction, mapping closely to the REST actions.

- **gRPC Services:** Implement **gRPC** endpoints for internal, high-performance interactions, especially for GPU-bound or streaming tasks. gRPC is a modern, high-performance RPC framework running over HTTP/2 that uses Protocol Buffers for data serialization [9]. It's well-suited for connecting microservices efficiently with built-in support for load balancing, tracing, and auth [9]. We will define `.proto` files for services such as `IngestionService`, `EmbeddingService`, `ExtractionService`, etc., with RPC methods corresponding to heavy operations (e.g. `RunMinerU` for PDF processing, `EmbedChunks`, `ExtractEntities`). For example:

```
// IngestionService in ingestion.proto
service IngestionService {
    rpc StartIngest(IngestRequest) returns (JobStatus) {};
    rpc GetIngestStatus(JobId) returns (JobStatus) {};
}
```

Each RPC call will be documented in the proto and we will use a tool like **Buf** to lint and generate code for multiple languages, ensuring the contract is clear and versioned. **Alternate backends:** By abstracting these operations as gRPC interfaces, we allow swapping implementations (e.g., a local GPU service vs. a cloud API) without changing the gateway. The gateway or orchestrator can choose which gRPC endpoint to call based on configuration, enabling flexibility in model backends (for example, using a local Qwen-3 model vs. calling an external embedding API). This design ensures our system can integrate different ML engines over time under the same interface.

- **SOAP Adapter:** For legacy compatibility, provide a **SOAP** interface (WSDL) that wraps key operations. SOAP (XML-based web services) may be required by certain enterprise clients. We will use an adapter pattern: implement SOAP endpoints that internally invoke our REST or gRPC logic. For instance, a SOAP operation `<IngestPDF>` could map to calling the `/ingest/` REST API. We can utilize a Python library (e.g. **zeep** or **spyne**) or a lightweight Java service to publish a WSDL that describes operations analogous to the REST endpoints. The SOAP messages will carry the same data (using XML schema definitions corresponding to our JSON schemas). While SOAP is not a focus for new functionality, this adapter ensures any legacy systems can integrate without needing to parse JSON or use HTTP/2. The emphasis is on **minimal** SOAP support (only for critical functions), with the adapter kept stateless and thin.

- **AsyncAPI & Event Streams:** Implement asynchronous event-driven communication using **Server-Sent Events (SSE)** and define it in an **AsyncAPI** specification. AsyncAPI is an open standard to describe and document event-driven APIs, analogous to how OpenAPI describes REST [10]. The system will publish events for long-running jobs and significant state changes. For example, when an ingest job is started, updated, or finished, an event like `jobs.ingest.started` or `jobs.ingest.completed` is emitted. We will create an AsyncAPI document describing channels such as `jobs/ingest` or `jobs/{id}/events` along with the event payload schemas. The primary implementation for clients will be **SSE** over HTTP (content-type `text/event-stream` [11]): clients can connect to an endpoint (e.g. `GET /jobs/{id}/events`) to stream events in real-time. Each event will be a JSON message (in an SSE `data:` frame) containing fields like `job_id`, `status`, `progress`, etc. For example, after a `POST /ingest/...` returns a job ID, the client can open `/jobs/123/events` to receive updates (queued -> processing -> completed with result or error) [12]. The AsyncAPI spec will formally document these channels and messages, enabling use of tools like AsyncAPI UI to explore them. In addition, this event stream is useful for UI updates and for loosely coupling components (e.g., a monitoring component could subscribe to all `jobs.*` events). The gateway will ensure events are **server-driven** (SSE or WebSockets if needed later) and reliably delivered to authorized clients.

Each protocol must be fully documented and tested. By offering this multi-protocol façade, we cater to RESTful integration, graph-based querying, high-speed internal calls, legacy enterprise needs, and reactive event-driven patterns all at once. This meets clients “where they are” while maintaining one coherent backend. Below, we detail how to implement each piece in alignment with relevant standards and project requirements.

REST API Endpoints (OpenAPI + JSON:API + OData)

Objective: Scaffold all required REST endpoints (per the Mercor Medical v1 scope) with correct routing, request/response schemas, and behavior. Enforce **JSON:API** formatting and **OData filter** syntax uniformly. Validate inputs and outputs against Pydantic models and the OpenAPI spec to ensure contract fidelity.

Endpoints to Implement: (from Mercor v1 specification)
- `POST /ingest/*` – **Ingestion** of various source data. There are multiple `/ingest` endpoints for

different sources: - `POST /ingest/clinicaltrials` – Pull clinical trial data by NCT ID and produce an IR (intermediate representation) document [13] [14]. Request body includes identifiers (NCT ids), and on success returns an IngestResponse (possibly with status per ID). Partial successes are indicated with HTTP 207 Multi-Status with a batch response [15].
- `POST /ingest/dailymed` – Download a drug label (SPL) by set ID or NDC code, parse it and write to IR [16].
- `POST /ingest/pmc` – Fetch a PubMed Central Open Access article by PMCID, convert to IR [17].
Each ingest endpoint requires an OAuth scope `"ingest:write"` [18] and returns either 200 (completed) or 207 if some sub-requests failed, with errors in the response details. All `/ingest` endpoints will first **validate input IDs** (e.g., valid NCT format) with deterministic checks; any invalid ID yields a 400 error (`VALIDATION_ERROR`) without processing [19]. This fail-fast validation ensures we don't run long jobs on bad input. On success, the ingest operation triggers the pipeline (see Orchestration below) and immediately returns a response containing either results (for quick operations) or a job resource reference.

- `POST /chunk` – Run the semantic chunker on one or more IR documents [20]. This splits documents into logical chunks (paragraphs, tables, etc.) for downstream processing. Requires `"chunk:write"` scope [21]. The request contains IR document IDs or content, and the response is a `ChunkResponse` with references to newly created chunks (each chunk gets an ID and metadata). The chunking is synchronous if fast, but for large documents it might spawn background tasks (in which case the endpoint could return a job ID plus an initial response indicating chunking started, and progress would stream via SSE).

- `POST /embed` – Compute embeddings for chunks/facets/concepts [22]. This triggers the vectorization of content: e.g. generating SPLADE sparse embeddings and Qwen-3 dense embeddings for the given objects. Requires `"embed:write"` scope [23]. The request specifies which items to embed (e.g. chunk IDs or facet IDs) and which embedding models to use. The operation may offload to GPU services via gRPC. The response (`EmbedResponse`) includes status or resulting vector IDs [24]. Because embedding uses heavy models (SPLADE, Qwen), the API should be prepared to handle this asynchronously – likely returning a job reference immediately and streaming the results when ready. We will implement a fail-fast policy: if no GPU is available or the model server is down, respond with an error quickly (do not silently run on CPU). Clients can then retry or handle the failure.

- `POST /retrieve` – Perform multi-strategy retrieval (BM25 + SPLADE + dense fusion) to return top-N relevant chunks for a query [25]. Requires `"retrieve:read"` scope (read-only) [26]. The request will contain a query (text and/or filters) and perhaps a flag for using a reranker. The response (`RetrieveResponse`) provides the ranked list of chunks with relevance scores and span highlights [27]. This operation should be fast (sub-second) using precomputed indexes (OpenSearch, FAISS, etc.), so it can be handled synchronously. We will ensure the output conforms to JSON:API (list of resources in `data` array, each with type "chunk" and relevant attributes like content snippet and score). OData filters can be supported here for advanced querying, though search queries likely use their own DSL.

- `POST /map/el` – Entity linking adjudication [28]. Links extracted entities to canonical identifiers or ontologies (e.g., mapping a drug name to a RxNorm ID). Requires `"map:write"` scope [29]. The request (`ELRequest`) might contain mention texts or preliminary mappings, and the service responds with an `ELResponse` mapping to standardized entities [30]. This likely involves calling an ontology service or using local vocabularies. Implementation will ensure determinism (always choose the same ID for the same mention if confidence is high), and may involve rules (e.g., if multiple IDs are acceptable, choose the most specific) as noted in the design. We will incorporate those business rules into this endpoint.

- `POST /extract/{kind}` – Span-grounded information extraction for a given facet kind [31] [32]. This covers endpoints like:

- `/extract/pico`
- `/extract/effects`
- `/extract/ae` (adverse events)
- `/extract/dose`

- `/extract/eligibility`
  All are handled by the same path with a `{kind}` parameter indicating which extraction to perform. The request will reference a chunk (or set of chunks) and possibly a payload of facet data, then the system uses LLMs or rules to extract the specific info (PICO elements, effect measures, etc.) from those chunks. Requires `"extract:write"` scope [33]. The response (`ExtractResponse`) contains extracted structures (like JSON with the identified spans and normalized values) [34]. Each kind has its own schema for the output (e.g., PICO extraction returns participants, interventions, outcomes spans). We will enforce that if an extraction yields no spans or fails validation, the API responds with a 422 or error – aligning with a "fail closed" principle: no partially correct extraction should be saved [35]. Validation rules (like ensuring evidence spans are present in the payload) will be applied before writing results [36].

- `POST /kg/write` – Write nodes/edges to the Knowledge Graph (Neo4j) with provenance [37]. This endpoint takes a set of extracted facts (nodes and relationships, e.g. Evidence or EvidenceVariable from the extraction step) and upserts them into the graph database, ensuring no duplicates. Requires `"kg:write"` scope [38]. The request (`KGWriteRequest`) includes the `doc_id` (source document) and a list of extractions or triples to assert [39]. The service will translate these into Cypher `MERGE` operations with appropriate idempotency keys so that repeated writes don't duplicate information [40]. The response confirms counts of nodes/relationships written [40] or warnings if some were skipped. We must also attach an `ExtractionActivity` node for each write (to log which model and prompt generated it, timestamp, etc.), enabling provenance tracking [41]. The API will reject any write attempts that don't meet validation (e.g. missing required references or failing SHACL shape checks on the graph data). This ensures the KG stays consistent.

In addition to these primary endpoints, the OpenAPI spec will define supporting components (schemas for requests/responses, security schemes, etc.). The **OAuth2 security scheme** is defined globally in OpenAPI with flows for client credentials, listing all required scopes [42]. Each path is annotated with the scopes it needs (as shown above). We will include standard HTTP response codes and error structures for each endpoint: e.g. 400 for validation errors, 401/403 for auth issues, 500 for server errors, and 422 for processing failures with details. Error responses will follow **RFC 7807 Problem+JSON** format for REST: a consistent structure with `type`, `title`, `status`, `detail`, and optionally an `errors` array for field-specific issues. This aligns with JSON:API error object recommendations as well.

JSON:API Compliance: All REST responses (except streams or plain-text) will be formatted per JSON:API v1.1. That means using top-level keys like `data`, `errors`, or `meta`. For example, a successful `GET /documents` might return:

```
{
  "data": [ { "type": "Document", "id": "123", "attributes": { ... } }, ... ],
  "included": [ ... ],
  "meta": { "total": 100 }
}
```

We will use a library or middleware to enforce this (or carefully construct responses). The content type **MUST** be `application/vnd.api+json` for requests and responses [3]. We'll also implement endpoints to support inclusion (`include` parameter) and sparse fieldsets (`fields[Type]=...`) as per JSON:API, allowing clients to optimize payloads.

OData Filtering and Query Options: For any list-fetching endpoint (if we add `GET` endpoints for resources like documents or chunks), we will support OData query options such as `$filter`, `$select`, `$expand` (to fetch related resources), and `$top`/`$skip` for pagination. This will make the API familiar to consumers of OData and allow advanced queries. The `$filter` parameter will accept expressions like `status eq 'active' and organizationId eq '42'` which our API layer will parse and apply to the database query (or reuse an existing OData parsing library in Python). OData's standardized approach to filtering and querying is beneficial for interoperability [43]. By conforming to these conventions, clients can plug our service into OData-aware tools and expect them to work.

Example CLI (OpenAPI) – During development, we will provide a Swagger UI and also allow using tools like `curl` or HTTPie for quick tests. For instance, a CLI usage:

```
# Trigger an ingestion of a ClinicalTrials.gov study by NCT ID
http POST https://api.example.com/ingest/clinicaltrials Authorization:"Bearer
$TOKEN" ids:='["NCT01234567"]'
```

This should return a JSON:API document with either immediate result or a job ID plus status.

By scaffolding these endpoints early (perhaps using FastAPI or Flask with Pydantic models for quick generation of docs), we establish the contract. The AI agent or developers can then implement each endpoint's logic, gradually replacing stub responses with real pipeline integrations. **Schemathesis** (property-based testing for OpenAPI) will be integrated to test these endpoints against the spec, generating random valid inputs to ensure the implementation never violates the schema (e.g., always returns required fields, handles edge cases). This helps maintain spec alignment over time.

GraphQL API Design & Integration

Objective: Provide a GraphQL interface on top of the core data and services, enabling clients to query and mutate data across the system in a single unified graph. The GraphQL implementation should reuse existing data models and support relational traversals (e.g., querying a document and the organization that owns it in one request).

Schema Generation: We will generate the GraphQL Schema Definition Language (SDL) from the Pydantic models and service definitions. Each core entity (Document, Organization, Claim, Entity, etc.) becomes a GraphQL Object Type with fields corresponding to its attributes. For example:

```
type Document {
  id: ID!
  title: String
  content: String
  status: String
  organization: Organization @relation(name: "BELONGS_TO")
  claims: [Claim!] @relation(name: "DERIVED_CLAIMS")
}
```

We'll define custom resolvers for fields that require computation or cross-database joins. For instance, the `organization` field on `Document` may fetch data from a SQL database or an internal service. We can use an **ORM or dataloader** pattern to batch these lookups efficiently. The GraphQL queries can thus navigate associations naturally.

Query and Mutation Examples:
- *Queries:* A client could query the API like:

```
query {
  document(id: "123") {
    title
    status
    organization { name tenantId }
    claims(filter: { type: "Eligibility" }) {
      text
      entities { name type }
    }
  }
}
```

This would retrieve a document and its related organization and filtered claims in one round-trip. GraphQL's ability to traverse related objects and fetch exactly the needed fields improves efficiency for complex UIs [8]. We will support arguments on fields for filtering, pagination, and sorting (e.g., a `filter` input for claims as shown, or `first/offset` for lists). These will map to underlying query parameters or DB queries.

- *Mutations:* We will expose mutations that correspond to key POST operations from the REST API. For example:

```
type Mutation {
  startIngestion(source: String!, ids: [String!]!): JobInfo!
  chunkDocument(documentId: ID!): ChunkResult!
  embedChunks(chunkIds: [ID!]!): JobInfo!
  extractFacet(kind: String!, chunkId: ID!): ExtractionResult!
  writeKnowledgeGraph(docId: ID!, extractions: [ExtractionInput!]!):
WriteResult!
}
```

The `startIngestion` mutation would accept a source type (e.g. "clinicaltrials") and a list of IDs, then internally trigger the same logic as the `POST /ingest/*` endpoints, returning a `JobInfo` type containing at least a job ID or immediate results. Similarly, `embedChunks` would start an embedding job and return a job reference if asynchronous. This gives GraphQL clients first-class support to initiate long-running processes without falling back to REST.

Subscriptions for Events: GraphQL also supports subscriptions for real-time updates. If our stack allows (e.g., using WebSockets), we might implement GraphQL subscriptions mirroring the SSE events (like `subscription { jobUpdates(jobId: "XYZ") { status progress } }`). However, this can be an enhancement if needed; initially, SSE might suffice. The AsyncAPI and SSE strategy described earlier covers eventing outside of GraphQL.

Schema Stitching & Federation: If the system's data is partitioned across microservices (for instance, if the Knowledge Graph has a separate GraphQL endpoint via Neo4j GraphQL), we will integrate them. We can use **Apollo Federation** to compose sub-schemas or **GraphQL Mesh** to wrap REST/gRPC as GraphQL and merge them [7] [44]. For example, we could have an `AnalysisService` that provides a GraphQL API for analytics; our gateway could federate that so clients see a single schema. Using GraphQL Mesh, one could even incorporate external APIs (REST/SOAP) into the GraphQL schema at runtime [7]. This "universal connector" approach means our GraphQL can answer queries that involve, say, our internal data plus an external ontology API, seamlessly. This is a stretch goal; primarily, we ensure our own system is covered by the GraphQL schema.

Introspection & Documentation: By default, GraphQL introspection is enabled, meaning clients (and tools like GraphiQL or Apollo Studio) can query the schema. We will maintain an SDL file (e.g., `schema.graphql`) in the repo, and use **GraphQL Inspector** in CI to detect breaking changes whenever the schema is updated. This helps maintain backward compatibility for GraphQL clients. We'll also include the GraphQL endpoint in the docs portal: embedding a **GraphQL Playground** or GraphiQL IDE so developers can explore and test queries against the live API.

Authorization in GraphQL: Since our underlying operations already enforce OAuth2 scopes, we will apply the same in GraphQL resolvers. For example, the `startIngestion` mutation will check that the requesting context's token has `ingest:write` scope, or return an unauthorized error. We can propagate the user identity and scopes from the HTTP layer into the GraphQL resolvers (most GraphQL server frameworks allow access to the request context). In multi-tenant scenarios, the context would also carry the tenant ID so resolvers filter data accordingly (e.g., only return documents for the tenant making the request).

Example – GraphQL SDL snippet:
To illustrate the shared model approach, consider the Pydantic model for a Document:

```
class Document(BaseModel):
    id: str
    title: str
    content: str
    status: Optional[str]
    org_id: str  # foreign key to Organization
```

From this, we derive GraphQL types:

```
type Document {
  id: ID!
  title: String!
  content: String
  status: String
  organization: Organization @resolver(name: "resolve_document_org")
}
```

The resolver `resolve_document_org` will use `org_id` to fetch the Organization (perhaps via a cached lookup or DB query). We also add `Organization.documents` field to go the other way:

```
type Organization {
  id: ID!
  name: String!
  documents(filter: DocumentFilter, first: Int, offset: Int): [Document!]!
}
```

Now the GraphQL server can handle queries that navigate both directions. We define input types like `DocumentFilter` to support filtering (these inputs map to constructing an OData-like filter behind the scenes or a database query).

By carefully designing the GraphQL schema in parallel with REST, we ensure **no duplicate business logic**. Resolvers can call internal Python functions or services that are also used by the REST endpoints. This avoids divergence in behavior between GraphQL and REST.

Finally, we'll ensure to test GraphQL thoroughly: using **GraphQL query tests** for common queries and edge cases, and using GraphQL's own introspection queries to verify the schema matches expected types (which is further enforced by GraphQL Inspector in CI). This dual-interface approach (REST + GraphQL) offers great flexibility to clients, and by sharing one data model and auth system underneath, we minimize maintenance overhead.

gRPC Microservices for GPU-bound Tasks

Objective: Offload heavy computation (like PDF parsing, embedding generation, ML inference) to dedicated microservices using gRPC, and integrate those services into the gateway orchestration. The design should allow plug-and-play of different service implementations (to accommodate different model backends or future improvements).

gRPC Service Design: We will create proto definitions for each major AI task. Based on the requirements, likely services include: - **MinerUService** (for PDF ingestion using MinerU OCR and layout analysis). - **EmbeddingService** (for generating SPLADE and Qwen embeddings). - **ExtractionService** (for running LLM-based extraction pipelines). - **MappingService** (for any ontology mappings if needed via gRPC, though that might be simpler logic not requiring separate service).

Each `.proto` file will define the RPC methods and message types. For example, `mineru.proto`:

```
syntax = "proto3";
package ai.mercor.mineru;
service MinerUService {
  // Convert a PDF (passed by URL or uploaded) into IR (intermediate
representation)
  rpc ProcessPDF(ProcessPDFRequest) returns (ProcessPDFResponse);
  // Health check
  rpc Ping(Empty) returns (Pong);
}
message ProcessPDFRequest { string document_id = 1; bytes pdf_data = 2; }
message ProcessPDFResponse { string document_id = 1; string status = 2; string
message = 3; }
```

And `embedding.proto`:

```
package ai.mercor.embed;
service EmbeddingService {
  rpc EmbedChunks(EmbedChunksRequest) returns (EmbedChunksResponse);
}
message EmbedChunksRequest { repeated string chunk_ids = 1; bool use_splade =
2; bool use_qwen = 3; }
message EmbedChunksResponse { map<string, EmbedResult> results = 1; }
message EmbedResult { string chunk_id = 1; bytes splade_vector = 2; bytes
qwen_vector = 3; string status = 4; }
```

The **proto packages** are versioned (e.g. `ai.mercor.embed.v1`) so changes can be managed. We will use Buf's breaking change detection on these protos in CI to avoid accidental incompatible changes. Buf can also generate documentation for the protos and even an OpenAPI stub for gRPC-transcoding if we wanted to expose gRPC via HTTP/JSON.

Service Implementation: The actual services can be implemented in Python (using libraries like gRPC Python), or more likely in performant languages like C++/Rust if needed. However, since Python with GPU libraries (PyTorch, etc.) is common, we might implement prototypes in Python. These services are **internal** – not exposed to end-users, only the API gateway calls them. They may run on separate servers or containers, potentially with GPU access. For instance, the **MinerUService** would wrap the MinerU container/CLI: when `ProcessPDF` is called, it runs the MinerU pipeline on the given PDF and returns when done (or streams progress via gRPC streaming if needed). The **EmbeddingService** might call a vLLM serving instances for Qwen embeddings or run local inference if models are loaded. Because these are internal, we can enforce stricter fail-fast logic: e.g., if the EmbeddingService receives a request but no GPU memory is available, it could immediately return an error status, which the gateway translates to a 503 or a job error event, thereby **failing fast** rather than queuing indefinitely.

Alternate Backends & Flexibility: We design the gateway to not assume a single implementation. For example, if in the future an alternate embedding model is used, one could deploy a different EmbeddingService (with the same proto interface). The gateway or orchestrator can be configured via YAML/ENV to point to different host:port or to use a mock implementation. Similarly, if MinerU is replaced by a different PDF processing tool, as long as we implement the `MinerUService.ProcessPDF` RPC accordingly, the rest of the system is unaffected. This decoupling follows the Open-Closed principle: support new models by adding new services without modifying gateway core logic.

Integration into Orchestration: The API gateway (or a separate orchestration component it calls) will use gRPC clients to invoke these services. For example: - When a user calls `POST /ingest/pmc` for a PDF, the gateway might see the media type is PDF and instead of auto-processing, it will place a record in the ledger (`pdf_downloaded`) and then call MinerUService via gRPC to process that PDF. - When `POST /embed` is called, the gateway collects the target chunks and then dispatches gRPC calls to the EmbeddingService (potentially in parallel or batch) to get vectors. - For extraction (`/extract/*`), if these are using an LLM, we might treat the LLM as a service too. Possibly those could also be gRPC calls to a hosted model server or to a local microservice running something like vLLM or OpenAI API wrapper.

Error Handling and Timeout Policies: We will configure **fail-fast** behaviors for these RPC calls. That includes setting aggressive timeouts (e.g., if embedding doesn't respond in X seconds, abort) and not retrying endlessly at the gateway level. Instead, rely on the orchestrator's retry policies or manual re-submit. This ties into the "fail-fast GPU policies" – for example, if a GPU job fails (exception or out-of-memory), the service returns an error and the orchestrator marks the job as failed (`embed_failed` state) immediately [45]. There is no automatic CPU fallback (since the spec explicitly forbids it [46]), meaning the error is propagated and logged, not silently handled in a slower way. This ensures that we maintain performance guarantees and avoid unpredictable slow operations.

Security for gRPC: Internal gRPC calls will be secured likely by network isolation or mTLS. We can issue service certificates and use mutual TLS between the gateway and these microservices for authentication, as noted in the security requirements [47]. Alternatively, if on Kubernetes, use the service mesh's mTLS. Each gRPC call could also carry metadata such as a request ID for tracing and possibly user context if needed (though heavy auth is not needed internally beyond trust between services).

Testing gRPC Services: We will include integration tests for these services. For example, using **grpcurl** or a small test client to call the RPC with known inputs and validating outputs. Also, using Buf's conformance tests if available. By keeping protos in the repo, front-end teams could generate clients in their languages

(Java, Go, etc.) easily, but since we also have REST/GraphQL, that may be redundant. The main thing is the gateway has robust client stubs (likely using Python's gRPC stub code) and error handling around them.

In summary, gRPC acts as the **efficient glue** for our AI/ML components, ensuring the API gateway remains responsive and not blocked on heavy compute. This microservice approach supports scalability (we can scale out, e.g. multiple embedding service instances behind a load balancer) and modularity (update one service without touching others). It aligns with modern cloud-native practices, using an RPC framework that is widely adopted for microservices (CNCF incubating project) [9] .

Two-Phase PDF Ingestion & GPU Job Orchestration

Objective: Implement the ingestion pipeline for sources, with special handling for PDFs (two-phase processing) and a centralized orchestration mechanism that coordinates all steps and state transitions. This includes queueing, status tracking, and enforcing the **GPU-only execution policy** for certain steps.

Pipeline Overview: According to the Mercor project spec, two execution modes are supported [48] [49] : - **Auto-Pipeline (Non-PDF):** For structured sources (APIs, JSON, XML, HTML, etc.), the pipeline runs fully automated in one go. E.g., ClinicalTrials JSON or JATS XML can go through: *download -> parse -> IR -> chunk -> facet -> embed -> index -> (optional extract/map) -> KG*. These steps occur sequentially without manual intervention. - **Two-Phase Pipeline (PDFs):** For PDF documents (e.g., research papers, scanned guidelines), the pipeline is split: 1. **Phase 1 (Preprocessing):** `download -> (STOP) -> mineru-run (GPU OCR/layout) -> IR`. After downloading the PDF and storing it, the process **stops** until a GPU job is explicitly run. The `mineru-run` step (MinerU OCR) must be manually or separately triggered to produce the IR (intermediate representation, e.g., JSON or XML of the document structure) [49] . Once MinerU completes successfully, the system marks a state and waits. 2. **Phase 2 (Post-processing):** `... -> (STOP) -> postpdf-start -> chunk -> facet -> embed (GPU) -> index -> (optional extract/map) -> KG`. After OCR, an explicit trigger (`postpdf-start`) resumes the pipeline: chunking the IR, generating facets, embedding, etc., then indexing and optionally extraction and graph writing [49] .

The orchestration must track state through **ledger flags** that act as a single source of truth [45] : - `pdf_downloaded` – PDF is saved, awaiting OCR. - `pdf_ir_ready` – OCR (MinerU) done, IR is ready, awaiting post-processing. - `auto_inflight` / `auto_done` – markers for auto-pipeline sources (e.g. an ingest that is currently running or completed). - Failure states like `mineru_failed`, `embed_failed`, etc., with error details logged [50] .

Implementation Approach: We will implement an **Orchestrator component** (which could be part of the API service or a separate service like an Airflow/Dagster workflow or just an async background manager within the gateway). The orchestrator's responsibilities: 1. On an ingest request, determine the pipeline mode (based on media type or source type). For non-PDF, start the pipeline immediately (potentially in a background task or queue). For PDF, perform only Phase 1 and then halt. 2. Interact with the gRPC services for any GPU-bound steps: - In auto-mode, when hitting the "embed" step, call EmbeddingService (gRPC). If it fails, mark `embed_failed`. - In two-phase, when a PDF is downloaded, call MinerUService to process it. This might be done via a queued job system if multiple PDFs line up (to avoid GPU contention). 3. Maintain a **job queue and state**: Possibly use a message queue or short-term database table for jobs. Each job has an ID and a state from the ledger above. The orchestrator updates states as steps complete. 4. Enforce **fail-fast GPU policy**: If a GPU-required step is invoked when GPU is not available or returns an error, the orchestrator immediately marks the job failed. For example, if MinerUService returns an error (no GPU, or OCR failed), we mark `mineru_failed` and do not proceed to chunking. We will **not** implement any CPU fallback path [46] – this is by design, as the spec requires GPU mandatory for those steps. This means the system's capacity must be managed (e.g., queue jobs if GPU busy, but also possibly reject new jobs if queue is full to avoid indefinite delays). 5. Provide **commands to control pipeline**: We will likely expose internal endpoints or CLI commands for the orchestrator, such as: - `POST /ingest/pdf` could trigger Phase 1 and return a job ID with state `pdf_downloaded`. - `POST /jobs/{id}/mineru-run` (or reuse `/ingest/pmc` endpoint with a flag) triggers the MinerU GPU job for that PDF, transitioning to `pdf_ir_ready` or `mineru_failed`. - `POST /jobs/{id}/postpdf-start` triggers Phase 2 (downstream steps) once IR is ready. However, instead of requiring the client to call these, we might automate via a **GPU job queue**: e.g., upon PDF ingest, automatically enqueue a MinerU task. The spec explicitly says MinerU is never chained automatically and requires explicit run [51], so perhaps the intent is a human or separate process triggers it. For a fully autonomous agent-run system, we might implement an **automatic queue** despite the wording, but to stay true: the orchestrator could have a configuration flag: in an "agent-managed" mode it auto-runs MinerU when possible, or in "strict" mode it waits for admin input. Given our AI agent context, we likely want autonomy: so we could allow the agent to call an internal API to simulate the manual trigger.

GPU Orchestration Considerations:
- **Dedicated GPU Queues:** To handle contention, separate queues for different GPU tasks might be used [52]. For example, one queue for MinerU (which might be long-running OCR tasks) and one for embeddings (many short tasks). This prevents a large PDF OCR from blocking all embeddings. We can implement this via thread pools or Celery queues designated for GPU tasks, using labels. The orchestrator will place tasks accordingly. - **Capacity Planning:** We will monitor GPU memory and utilization. Possibly integrate with NVIDIA's tools or simply keep a count of concurrent tasks. In config, define max concurrent MinerU jobs = N (perhaps 1 per GPU) and max concurrent embedding jobs = M (we can often run multiple embedding jobs on one GPU if memory allows, but careful with the 8B model). - If the limits are exceeded, new tasks should **wait or reject fast**. A fail-fast policy might mean if the queue is full, respond with a 503 telling the client to retry later (so that the system doesn't accept more work than it can handle). Alternatively, queue but notify via SSE that it's queued. The design in the spec hints at returning `{status:"queued", job_id}` and then streaming progress via SSE [12]. We will follow that: the initial response of a job-submitting call can indicate queued status and a job ID, and then asynchronous events update when the job actually starts and completes.

State Management: We will use a durable store for job states, e.g., a database table or a simple key-value store (Redis) keyed by job_id. Each job entry will have: - `id`, `type` (e.g. ingest_pdf, embed, etc.), `status` (one of the ledger states or in-progress markers), `created_at`, `updated_at`, `result` (if any or path to artifact), `error` (if failed). This allows the gateway to report status on GET requests or SSE, and also recovers state if the system restarts. We can also store partial outputs (like path to IR files) to pass from phase 1 to phase 2.

Orchestration Example Flow (PDF Ingest):
1. Client calls `POST /ingest/pmc` with a PDF file (or reference). The gateway saves the PDF, creates a new job record `J123` with status `pdf_downloaded` [45], and returns a response: `{ "data": { "id": "J123", "type": "job", "attributes": { "status": "pdf_downloaded" } } }` along with an SSE URL `/jobs/J123/events` for updates [12].
2. The orchestrator enqueues a MinerUService call for job J123. When a GPU worker is free, it calls MinerU. If

MinerU fails (exception), orchestrator updates J123 status to `mineru_failed`, logs the error payload, and emits an SSE event `event: error` with details. If MinerU succeeds, it produces an IR artifact (say stored in S3 or DB) and orchestrator updates status to `pdf_ir_ready` [45]. An SSE event `event: update` with status `pdf_ir_ready` is sent.
3. Now, either automatically or via an API call, Phase 2 starts. In an automated scenario, the orchestrator would immediately continue to chunking; in a manual scenario, the client calls e.g. `POST /chunk` with doc_id referencing the new IR. Let's assume automation for agent's sake: orchestrator sets status `auto_inflight` for post-processing. It runs chunking (likely a local function, not GPU heavy), updates job state and emits events, then runs facet generation, then calls EmbeddingService gRPC for embeddings. The embeddings step uses GPU; if it fails -> `embed_failed` with error emitted. If it succeeds -> job status moves to `auto_done` or `completed`. It then optionally triggers extraction if configured (the problem statement says "(optional) extract/map -> KG" are part of pipeline [53] [54], possibly auto-run or could be separate calls by an agent). We might not fully auto-run extraction unless specified by client, to avoid unwanted writes. Perhaps the orchestrator stops after indexing, and the client (or agent) explicitly calls extraction and KG writing endpoints to complete the loop. This matches the "agent calls APIs" model indicated by the separate `/extract` and `/kg/write` endpoints.
4. Throughout the process, every state change is pushed to the SSE channel. The final event would likely be `event: done` with a summary or the final outputs (e.g., IDs of chunks created, count of extractions, etc.). After that, the SSE stream can close or stay open if more sub-tasks might happen.

Fail-Fast Policy Enforcement: Key to note: *GPU is mandatory for certain steps (MinerU OCR, SPLADE expansion, Qwen embeddings) and there is no CPU fallback* [46]. This means in code, before calling such a service, optionally check an "GPU available" flag. If false, we immediately mark the job failed. In practice, if our services are deployed correctly, they won't accept jobs if no GPU; but an extra layer of check might be a health endpoint that indicates if GPU memory is sufficient or service alive. Also, if a GPU job returns partial output (like ran out of memory half-way), we fail the entire job because partial results are not acceptable without review.

Logging and Audit: Every step's outcome (including errors) will be logged. We include unique **run IDs** for MinerU runs, etc. as part of provenance (the spec mentions storing MinerU run IDs and content hashes for audit [55] [56]). If a step fails, we log the error payload, perhaps store the input that caused it (for debugging) and expose a simplified message to the user via the SSE/error response. All pipeline actions affecting data (like writing to KG) will also create audit entries (ExtractionActivity nodes in KG, etc.). This ensures we can trace what happened for each job – crucial in a medical compliance context.

Integration with Workflows or Schedulers: Although a custom orchestrator is described, an alternative is using a workflow engine (like Apache Airflow or Prefect). The snippet shows something about DAG tasks (maybe in Airflow) [57]. We might incorporate a lightweight workflow engine if it helps manage retries, scheduling and monitoring. However, given the complexity, a simpler asyncio or queue-based orchestrator might be easier for the AI agent to implement.

Testing Orchestration: We will simulate various scenarios: - Successful PDF ingest end-to-end. - GPU failure mid-way (simulate MinerU throwing error). - Non-PDF ingest auto mode. - Multiple simultaneous jobs to test queuing and SSE differentiation. We will write unit tests for state transitions (given an initial state and an event, do we get expected next state), and integration tests that stub out the gRPC services (return dummy responses quickly) to run the entire pipeline fast. We also enforce that no illegal state transitions occur (e.g., cannot go from `pdf_downloaded` to `auto_done` without intermediate steps).

In conclusion, this orchestration mechanism ensures **reliability and traceability** in the pipeline. It is designed to accommodate real-world issues (GPU contention, partial failures, manual interventions) while maximizing automation for efficiency. By separating the PDF pipeline into two phases, we respect the need for manual control or special scheduling of expensive OCR tasks. The fail-fast approach is aligned with project risk mitigation (no slow CPU fallback, rather address the resource issue or fail clearly) [46]. The result is a controlled yet flexible pipeline that an AI agent can manage programmatically via the defined APIs and events.

Real-Time Events & AsyncAPI Streams

Objective: Provide a mechanism for clients (and internal components) to receive real-time updates on long-running processes and other events, using standard protocols and documented via AsyncAPI. This improves user experience (no constant polling) and decouples components through event-driven architecture.

Server-Sent Events (SSE) Implementation: The primary delivery mechanism for events will be **Server-Sent Events**, which allow the server to push text events over an HTTP connection. SSE was chosen because it's simple, works over HTTP/1.1, and is easy to consume in web clients (EventSource API) without complex libraries. We will implement an endpoint such as `GET /jobs/{id}/events` that upgrades the connection to an event stream (content-type `text/event-stream`) [58]. When a client connects, the server will start sending events for that specific job (filtering by job ID to isolate stream). Optionally, we might allow wildcard streams (e.g. an admin could connect to `/jobs/events?stream=all` to get all jobs updates, but careful with security). Each event will be formatted per SSE: beginning with `event: <type>` (like "progress" or "complete") and `data: <json>` containing event details, ending with a double newline. For example:

```
event: progress
data: {"job_id":"J123", "status":"chunking", "progress":50}
```

We will use the SSE comment `: keep-alive` technique or ping events to keep the connection alive over long jobs. If the server restarts, clients should reconnect (EventSource can do that automatically with Last-Event-ID if we implement it).

Types of Events:
- **Job Lifecycle Events:** For ingestion, embedding, etc., as described, including statuses like queued, started, completed, failed, with timestamps and possibly intermediate progress percent.
- **Domain-Specific Events:** We might also emit events for things like “new document ingested” or “KG updated” if needed by downstream systems. But those might be more internal. - **Heartbeat/Keepalive:** Periodic events or comments just to ensure clients know connection is alive.

The SSE will handle multi-tenant concerns: only users from the tenant that owns a job can stream that job's events. (We'll check auth on connect and reject if not authorized). SSE is over HTTP, so it uses the same OAuth token (we'll require the GET request to include Authorization header).

AsyncAPI Documentation: We will produce an **AsyncAPI** specification that describes the event streams. AsyncAPI is an open-source spec for event-driven APIs, analogous to OpenAPI but for pub/sub or streaming services [10]. In our AsyncAPI document (likely YAML), we'll define: - **Channels:** e.g.

`jobs/{jobId}/events` as a channel (parameterized by jobId). If there are other streams like `ingest/updates` for all ingests, those too. Each channel will have a `publish` operation (server publishes events) with message schemas. - **Messages:** Define the JSON schema of event payloads, e.g. a `JobStatusChanged` event with fields: `job_id`, `status` (enum of states), `message` (optional human-readable), `progress` (number or percentage), etc. We may have separate message schemas for different event types or use one with an additional `event_type` field. - **Protocol info:** We specify that the protocol is SSE (which can be treated as a kind of asynchronous protocol, though AsyncAPI typically covers MQTT, Kafka, etc., it can document HTTP streaming). We can denote bindings for HTTP. If we foresee using **WebSockets** as well (maybe an upgrade for SSE), we could document that as an alternative.

This AsyncAPI file will allow us to use documentation tools like the AsyncAPI web component or generator to create a docs page showing available events, and it serves as formal documentation for how clients can subscribe. It also sets the stage if we later use a message broker (Kafka, etc.) behind the scenes – we could easily map these SSE events to Kafka topics described in AsyncAPI.

Example AsyncAPI snippet: (YAML format)

```
channels:
  jobs/{jobId}/events:
    parameters:
      jobId:
        description: Job identifier
        schema: { type: string }
    subscribe:
      summary: Receive events for a specific job
      message:
        $ref: '#/components/messages/JobUpdate'
components:
  messages:
    JobUpdate:
      payload:
        type: object
        properties:
          job_id: { type: string }
          status: { type: string, enum:
["pdf_downloaded","pdf_ir_ready","chunking","embedding","completed","failed"] }
          progress: { type: integer, minimum: 0, maximum: 100 }
          detail: { type: string }
```

The above defines a channel and a message schema. We'd also specify server info (e.g., `protocol: http` with `x-sse: true` or similar to note SSE usage).

Event Publishing Implementation: Within the orchestrator or the specific service handling an operation, whenever a state change happens, we will send an event. This can be implemented by maintaining a list of open SSE connections per job. For example, when `/jobs/J123/events` is hit, we store the response stream object keyed by J123. The orchestrator, when updating J123's state, looks up any open streams for

J123 and writes the event to them. We must handle if no one is listening (that's fine, we just don't send or the data goes nowhere). We also ensure to close streams when jobs are done (or after some timeout) to free resources.

For scalability, if many clients subscribe, we need to be mindful of resource usage. SSE is lightweight but each is a HTTP connection. We can use an async framework like FastAPI or Starlette which can handle many connections without threads (via async I/O). We might also restrict SSE to mainly internal or a few external consumers to avoid having thousands of open streams (which might require a more robust pub/sub infra). Given typical usage, number of SSE clients should be manageable (e.g., an analyst watching a handful of jobs, or an agent tracking its tasks).

Alternative/Eventual Enhancements: In a more complex setup, we could push events via **webhooks** (client provides a callback URL in the request, and we POST updates to it) or integrate with a message queue where clients consume from. The OpenAPI 3.1 spec actually supports describing webhooks. However, SSE fits our immediate needs and the spec explicitly lists SSE as a supported format (along with NDJSON for bulk) [59] . The AsyncAPI spec could also describe an **MQTT or Kafka channel** for these events if we plan to broadcast them internally. For now, we scope to SSE for simplicity.

Testing Events: We will test SSE by simulating a client. For example, using `curl`:

```
curl -N -H "Accept: text/event-stream" -H "Authorization: Bearer $TOKEN"
https://api.example.com/jobs/J123/events
```

The `-N` keeps curl running to stream. We expect to see events come through as the job progresses. We will also intentionally test event ordering and content (ensuring first event maybe is queued or started, final is completed). If the SSE endpoint is implemented in FastAPI, we might use fastapi's `EventSourceResponse` or similar to manage the streaming properly.

By implementing robust event streams, we significantly improve the **reactivity** of the system. Users (or autonomous agents) do not have to poll `/status` repeatedly; they can simply listen for updates, which reduces load and provides real-time feedback. This event-driven approach also means in the future we can have other services react to events (for example, auto-trigger a report generation when an ingest + extraction job completed). It thus supports a more decoupled architecture where components communicate via events in addition to direct API calls.

Security & Access Control (OAuth2, Tenancy, Policies)

Objective: Secure the API and services using OAuth 2.0 standards, implement tenant-based access restrictions, and enforce fine-grained scope controls on every operation. Also incorporate other security best practices (mTLS for internal calls, secrets management, etc.) as needed for a production system handling sensitive data.

OAuth 2.0 Authorization: The system will use **OAuth 2.0** as the main auth mechanism, specifically the Client Credentials flow for service-to-service calls and possibly Authorization Code flow (or JWT Bearer) for user context if needed. OAuth 2.0 is the industry-standard protocol for authorization [60] , designed to allow third-party clients to secure API access without sharing passwords. We will integrate with an OAuth2 provider (could be a dedicated service like Keycloak, Auth0, or a simple in-house server since client-credentials is straightforward). The OpenAPI security scheme defines a token URL (e.g., `https://auth.myorg.com/oauth2/token`) and relevant scopes [42]. Each client (which could be an AI agent or a microservice) will obtain a token with certain scopes.

Scope-Based Access Control: As seen in the OpenAPI paths above, each endpoint requires a specific scope (or set of scopes). For example, ingest endpoints need `ingest:write`, retrieval needs `retrieve:read`, KG writing needs `kg:write` [18] [61]. These scopes are defined in the OAuth server and issued per client. We will implement middleware in the gateway that: - Validates the OAuth 2.0 access token (likely JWT) on each request (using public keys/JWKS to verify signature if JWT, or introspection if opaque tokens). - Checks that the token contains the required scope for the requested endpoint. If not, respond with 403 Forbidden. - Optionally checks tenant ID in the token vs. resource access (for multi-tenant data isolation).

Tenant Isolation: The system is multi-tenant, meaning data of one tenant (organization/customer) must not be accessible by another. We will include a **tenant identifier** in the OAuth token claims (e.g., a claim `tenant_id` or use realm concept). All data in the system will be tagged/partitioned by tenant. For example, Document records have an `org_id`, KG nodes have a `tenant` property or are in separate graph DB instances per tenant. When a request comes in, after auth, we resolve the tenant from token and use it to filter all queries. This can be done via: - A global request context that holds `current_tenant`. - Database queries that automatically include a filter on tenant field. - In GraphQL resolvers, only returning objects belonging to that tenant.

Additionally, when writing data, the backend will stamp the tenant ID on created records. The **policy engine** could be as simple as this tenant matching, or more complex (some roles can access multiple tenants if needed, but likely not here).

Role and Scope Design: We might have composite roles that map to sets of scopes. For instance, a role "ingest_service" might have `ingest:write`, `chunk:write`, `embed:write`, `map:write`, `extract:write`, `kg:write`, basically all write scopes for pipeline execution. Another role "analyst_readonly" might only have `retrieve:read` to search and view results, but not modify data. The OAuth server can issue tokens with appropriate scopes based on the client's role or permissions.

We will document the scopes and their usage. For example: - `ingest:write` – permission to ingest new sources (and implicitly to create documents in the system). - `chunk:write` – permission to chunk documents (likely not separated in practice, but we have it). - `embed:write` – permission to generate embeddings. - `retrieve:read` – permission to retrieve/search (read-only). - `map:write` – permission to do mapping (entity linking). - `extract:write` – permission for extractions. - `kg:write` – permission to write to knowledge graph. And possibly `admin:*` scopes for administrative endpoints or metrics.

The security scheme in OpenAPI has these scopes listed with descriptions [62], which we will include in our API documentation for developers.

OAuth2 Token Handling: The clients will present tokens in the `Authorization: Bearer <token>` header. We'll use JWTs so that the gateway can verify them quickly without an extra network call. The token will include scopes and tenant and possibly an expiry (we will honor expirations). If a token is expired or invalid, respond 401. If valid but missing scopes, 403.

Refresh Tokens or Long-Running Jobs: With client credentials, the token is usually short-lived (e.g., 1 hour). For long jobs, a token might expire mid-way. Since SSE is a continuous connection, if the token expires during an SSE, we might consider that acceptable for the duration of the connection (since the request was authorized at start). Alternatively, we could implement token refresh by closing stream and requiring reconnect with a new token when expired. This is a minor detail; likely not needed if tokens are sufficiently long or if the SSE connections are not extremely long-lived.

Additional Security Layers:
- **API Keys for Internal Use:** The spec mentions an API key fallback for internal jobs [58]. We can support an alternative auth mechanism (e.g., an `x-api-key` header) for internal components that cannot easily do OAuth client credentials. This key would be a static secret with limited privileges (like triggering certain tasks). However, OAuth2 will be primary for external integrations. - **mTLS and Network Security:** All internal service-to-service calls (gRPC between gateway and GPU services, database connections, etc.) will be secured. We will use **mutual TLS** for gRPC calls, meaning each service has a certificate. Additionally, if on Kubernetes, we restrict services to cluster network. For databases like Neo4j, we use network policies so only the API can reach it. Vault or AWS Secrets Manager will manage credentials (no plain text passwords in code). - **Audit Logging:** We log authentication and authorization events: token of client X used on endpoint Y (maybe just in debug logs or access logs). Also, important changes (like writing to KG) might trigger an audit log entry including who (which client id) did it. This might be required for compliance if sensitive data.
- **Rate Limiting:** Implement global or per-tenant rate limits to prevent abuse or runaway agents. For example, limit ingest calls to, say, 10 per second per client, retrieval calls maybe higher. We can use a token-bucket in memory or integrate with a gateway like KrakenD or Envoy for ratelimiting. The design might include a lightweight **middleware** using an in-memory counter or Redis for distributed rate limiting. If limits exceeded, return HTTP 429 Too Many Requests with a `Retry-After` header. - **CORS and Network config:** If the API will be called from browsers (maybe if a web UI is built), we'll enable proper CORS headers. If not, we might restrict to certain IPs.

Example OAuth2 Flow (Client Credentials): 1. The AI agent (client) requests a token: `POST https://auth.myorg.com/oauth2/token` with basic auth (client_id & secret) and body `grant_type=client_credentials&scope=ingest:write%20kg:write...`. The auth server responds with `{ "access_token": "...", "expires_in": 3600, "scope": "...", "token_type": "Bearer" }`. 2. The agent calls the API with `Authorization: Bearer ...`. The API gateway's auth middleware decodes the JWT, checks signature and scope. 3. If ok, the request proceeds. Otherwise, gets a 401 or 403 with a JSON:API error object indicating insufficient_scope perhaps.

Testing Security: We will write tests to ensure: - Endpoints reject missing or bad tokens. - Endpoints reject tokens without required scope (e.g., use a token with `retrieve:read` to attempt a `POST /ingest`, expect 403). - Multi-tenant isolation: create data under tenant A, then use a token for tenant B to try to access it, expect 404 or no results. Possibly set up separate test tenants in the database and run cross-checks.

By following OAuth 2.0 best practices and robust scope checks, we ensure only authorized actions are allowed. OAuth 2.0 is widely understood and supported by libraries, so implementing it will align us with industry standard security [60] . Combined with tenant constraints and careful error reporting (no leaking of data in error messages), this provides a secure foundation for the platform.

Data Model & Domain Adaptability

Objective: Define a **federated data model** that captures core concepts while allowing specialization for different knowledge domains. This ensures our system can handle medical data (current focus) and be extended to financial or legal data in the future without a complete redesign.

Core Entities (Federated Model): We identify common high-level entities that transcend domains: - **Organization:** Represents a tenant or source organization (e.g., a sponsor company, data provider, hospital, etc.). Could include fields like `id`, `name`, `type`, maybe domain-specific attributes (for a hospital vs. a bank). - **Document:** A generic document or record ingested. In medicine, this could be a clinical trial registry entry, a research paper, a guideline PDF, a drug label, etc. In finance, it might be an SEC filing or XBRL report; in law, a legal case or contract. The Document entity will have fields like `id`, `title`, `content (maybe separate storage)`, `source_type` (to distinguish CTG vs PMC vs others), timestamps, etc. Also references to Organization (owner or publisher). Document is a central entity for retrieval. - **Chunk (or Section):** A piece of a Document (especially for unstructured long docs). This might be domain-agnostic in shape (just text content, maybe type like paragraph or table). - **Claim/Extraction:** A structured piece of information extracted from documents. In medical domain, these are things like PICO elements, outcomes, adverse events, etc. In finance, these could be financial metrics or risk factors extracted; in law, maybe legal assertions or clauses. - **Entity:** A real-world entity mentioned in documents – e.g., a Drug, a Disease/ Condition, a Company, a Legal Entity, etc. Entities are usually linked to canonical databases (ontology or registry IDs). For example, a drug entity might map to a DrugBank or RxNorm entry; a company to a stock ticker or legal registry number. - **Knowledge Graph Node/Edge:** While not exposed as top-level API resources (except via `/kg/write`), internally we have a graph representation linking Documents, Entities, and Claims. For example, an Evidence node linking an Intervention (entity) to an Outcome measure with a relationship like `SHOWED_EFFECT`.

We will model these in code (Pydantic/ORM models) and ensure each has a clear primary key and relationships. The **federation** aspect means we can join these across contexts: e.g., a Claim links a Document and some Entities; an Organization has many Documents, etc.

Domain Overlays: On top of the core model, we define domain-specific extensions or mappers: - **Medical (FHIR Alignment):** The medical domain overlay will align with HL7 FHIR resources for evidence. For instance, our `EvidenceVariable` concept (which might represent a population or outcome of a study) maps to FHIR's `EvidenceVariable` resource [63] . Similarly, an extracted effect measure maps to FHIR's `Evidence` resource [64] . We ensure our data fields (like effect size, confidence interval, etc.) correspond to elements in FHIR so that conversion is possible [65] [64] . We might not expose raw FHIR through our API (though we could have an endpoint to output a FHIR Bundle of results for interoperability). But internally, this alignment guides what we capture (e.g., using standard coding for units via UCUM, SNOMED codes for conditions, etc., as referenced in the spec [66]). We also incorporate clinical ontologies: SNOMED CT, LOINC, RxNorm, etc. to normalize fields. The `Entity` model in medical context might have a subtype or field `coding` that contains these standard codes. Our KG schema (CDKO-Med as per spec) is explicitly aligned to FHIR profiles for evidence [67], which ensures completeness for clinical use cases. - **Finance (XBRL):** For financial documents, XBRL provides a standard way to represent financial statements. If we ingest, say, an SEC 10-K report, our Document can be mapped to an XBRL instance document's concepts. A `Claim` in finance might be something like "Revenue for 2021 = $X", which in XBRL is a fact with a specific concept (like us-gaap:Revenue). The overlay here means our `Extraction` model would have fields to store XBRL concept names or identifiers, and values with proper units/currencies. If we were to output to an XBRL or JSON format, we'd have those mappings. Essentially, the data model in core might just call it `Metric` entity or such, but overlay tells us that in finance domain, use the XBRL taxonomy for definitions. - **Legal (LegalDocML):** LegalDocML (also known as Akoma Ntoso in some contexts) is an XML standard for legal documents. A Document of type legal case or legislation could be structured according to LegalDocML (sections, articles, references). Our model might have `Clause` or `Section` entity types for legal documents. The overlay would map a `Claim` or `Entity` to legal notions. For instance, an Entity might be "Defendant" or "ClauseReference", which in LegalDocML have specific tags. Our extraction for law might identify things like judgments or ratios, which we'd map to a LegalDocML schema if needed. This likely will be a future extension; we design the model to not be narrowly clinical. For example, instead of assuming all extractions have a PICO type, we generalize to a type + key-value data, where the type can be PICO for medical or "LegalArgument" for law, etc.

Implementing Overlays: We can approach this by **inheritance or composition** in code. For example, define base classes:

```python
class BaseDocument(BaseModel):
    id: str
    title: str
    content: str
    org_id: str
    # ... common fields

class MedicalDocument(BaseDocument):
    # fields specific to medical, or methods to export to FHIR etc.
    trial_ids: List[str] = []

class LegalDocument(BaseDocument):
    # maybe references to court, case number, etc.
    case_number: str
```

Alternatively, keep a single Document model but have a `domain` field and store extra domain-specific data in a JSONB column or related table. E.g., Document has `domain = "medical"` and then another table `DocumentMedicalOverlay` with Document.id as FK containing medical-specific fields. This avoids sparse columns in one table and keeps domain data optional.

The knowledge graph likely will accept all domains, but maybe we maintain separate graphs per domain to avoid mixing vocabularies. Or use a shared graph but with labels indicating domain. (The spec's KG is very medical-specific right now, but if extended, we might have distinct subgraphs.)

Schema Governance: The model definitions (Pydantic, GraphQL types, DB schema) should clearly separate what's core vs extension. Documentation should list which fields are always present and which appear only for certain domains or source types. This helps frontends know what to expect. Also, by modularizing overlays, if we onboard a new domain, we add a new module rather than alter core logic. For example, adding an overlay for "Geospatial data" could be done by adding new entity types and extraction logic, without touching ingestion of clinical trials.

Example – Medical Domain Entities:
- *Study/Trial:* In our model, a Clinical Trial from CT.gov could be a Document with source_type "clinical_trial". But perhaps we decide to have a distinct entity for Study since it has structured fields (start date, phase, etc.). The spec's pseudo-ontology mentions nodes for Studies, Arms, Interventions, etc. [63]. Those might not all surface in our API, but they exist in the KG. Possibly we expose them via GraphQL – e.g., `Document` of type trial has related `interventions` (drugs) and `outcomes`. - *Outcome/Evidence:* These are essentially the extracted claims (effect sizes, etc.), represented as Evidence in KG with links to EvidenceVariable (population, outcome definitions) [68] [64]. Our `Claim` model could be abstract but in medical overlay, we know a Claim of kind "endpoint" corresponds to an effect measure and we attach fields for value, CI, p-value, etc.
- The mapping to FHIR means we ensure fields like `value` and `ci` align to FHIR's `Statistic` sub-component [64]. We likely will create an **export function** that can transform our internal objects to a FHIR Bundle (as shown by an exporter interface in the spec [69]). This could even be an endpoint `/export/fhir` for a given doc_id which uses the overlay mapping.

Quality and Validation: Each domain overlay can enforce additional validation rules. For instance, in medical, if we have a numeric result that should have a unit, we ensure UCUM units are present and correct (the spec mentions UCUM normalization and SHACL for units/codes [70]). In finance, ensure the currency or period is included for monetary values, etc. These can be done via Pydantic validators or separate validation layers (like the SHACL shapes for graph or just Python logic pre-write).

Multi-domain Operation: It's important that adding domains does not break existing ones. This is where a **configuration-driven** approach helps. The system could have a config file listing enabled domains and their parameters. For example, `domains: [ "medical", "finance" ]` and each domain module provides certain adapter implementations or model extensions. If domain = medical, ingest sources A, B, C are available; if finance, ingest sources X, Y, etc., and certain extraction pipelines differ.

The **Adapter SDK** (discussed next) ties in here: adding a new domain might largely involve adding new adapter definitions (for new data sources) and maybe new extraction patterns. The core gateway remains the same.

To summarize, our data model is **federated** in the sense that it forms a common graph of nodes (docs, entities, claims) that can represent various forms of knowledge, and we have layers that align those to specific standards (FHIR, XBRL, etc.). This ensures adaptability: the investment in building this for medical can be leveraged for other fields by just extending, not rewriting, the platform.

Adapter SDK for Ingestion (Plug-in Sources)

Objective: Provide a plug-in mechanism for integrating new data sources into the ingestion pipeline with minimal code changes, inspired by the Singer and Airbyte connector models. This allows “live API ingestion” from varied sources (APIs, databases, etc.) via declarative YAML configurations.

Concept: The ingestion system should be extensible: today we have sources like ClinicalTrials.gov, PubMed, FDA APIs, etc., but tomorrow we might need to ingest a new data feed (e.g., a new clinical registry or a financial datasource). Following the model of **Singer** taps/targets and **Airbyte** connectors, we will create an **Adapter SDK** where each source adapter is defined by: - A **YAML contract** describing how to fetch and transform data from the source. - Possibly custom code if needed, but the goal is to allow simple sources to be added just by configuration.

Adapter Definition: Each adapter YAML might include: - **Source type and endpoint:** e.g., REST API base URL, or file format. - **Authentication details:** e.g., API keys or OAuth for that source (referenced securely, not stored in plain YAML). - **Extraction logic:** This could be a JSON schema or JMESPath queries to extract fields, or a small script. - **Mapping to IR:** How to map the source data into our intermediate representation (Document/Block/Section objects). For example, the YAML might say which fields map to Document title, what constitutes a Block, etc. - **Batching/pagination strategy:** e.g., how to iterate over pages of an API, or to handle rate limits. - **Commands:** Possibly CLI commands or docker image references if the adapter runs externally.

Singer uses the concept of **tap** (source) and **target** (destination) with a **catalog** of streams. Airbyte has a low-code YAML where you can specify the HTTP requests and field mappings [71]. We will aim for something similar: a YAML that our ingest code can read and then perform ingestion accordingly. For instance:

```yaml
source: "ClinicalTrialsAPI"
auth:
  type: "api_key"
  key_env: "CTG_API_KEY"
requests:
  - url: "https://clinicaltrials.gov/api/v2/studies/{id}"
    method: GET
    params: { format: "JSON" }
    response:
      type: "json"
      extract: "$.studies[0]"   # JSONPath to the study object
mapping:
  document:
    id: "$.id"
    title: "$.title"
    status: "$.status"
    org_id: "$.sponsor"
  blocks:
    - iterate: "$.sections[*]"
      mapping:
         id: "$.section_id"
         text: "$.text"
         type: "$.type"
```

The above is pseudo-spec, but it outlines that for each study ID, do a GET, then map fields to our Document model and sub-sections to Block model.

Execution of Adapters: The ingestion endpoints (`/ingest/clinicaltrials`, etc.) will use these adapter definitions. Possibly they call a generic ingestion engine passing the source name and IDs. For example, `POST /ingest/clinicaltrials` could internally load the "ClinicalTrialsAPI" adapter YAML, then for each provided NCT id, execute the defined steps. This is somewhat like Airbyte's Connector Builder which can run purely from config for many REST APIs [71]. If an adapter is more complex (like requires special parsing of PDF or an FTP download), we allow hooking in a Python class that implements a certain interface (the SDK part).

Our Adapter SDK will define a base class like:

```
class BaseAdapter:
    def fetch(self, id: str) -> Any: ...
    def parse(self, raw: Any) -> Document: ...
    def post_process(self, doc: Document): ...
```

And we can either auto-generate an adapter class from YAML (by creating a subclass that uses the YAML to implement fetch/parse) or manually code it. We likely will auto-generate for simple HTTP/JSON sources. For PDFs, the adapter's fetch might download the PDF, and parse might simply mark it as needing MinerU (so parse just outputs a Document with minimal metadata and content as binary or path).

Singer Integration: We could even directly integrate Singer taps for some sources. Singer taps output data in a standard JSON format (stitched via STDOUT). If there's a Singer tap for a given source (e.g., a tap-clinicaltrials might not exist, but hypothetically), we could run it and capture output to feed our pipeline. However, Singer is more ETL to databases, not exactly our use-case of on-demand ingestion. Still, by acknowledging Singer's approach, we ensure our design isn't reinventing unnecessarily. The key is to **not hardcode** how each source works, but configure it.

Example - Adding a New Source: Suppose we want to add a new medical literature source, say arXiv papers. We would create `adapters/arxiv.yaml` describing its API (or maybe it's just RSS feed). Then add a new endpoint `/ingest/arxiv` which simply calls the adapter engine with that config. The agent doesn't need a full code release, just drop the YAML in a known location and deploy. This makes the system **declarative and extensible**. As reference, frameworks like Apache Camel or Singer aim for exactly this kind of plug-in connectivity [72]. They allow connecting to a variety of endpoints without starting from scratch [72]. Our system will mimic that concept in a domain-specific way.

Adapter Library & Testing: We will maintain a library of adapters for all official sources (CT.gov, PubMed, etc.). These will be under version control. We will write tests for each adapter (maybe with sample data or hitting a sandbox API) to ensure they work. Also, the system might have a small CLI to run an adapter standalone for testing:

```
$ python ingest_entry.py --source clinicaltrials --id NCT01234567
```

This would output either the created Document ID or the content. This is hinted by some lines in the spec about an ingest_entry script [57]. Indeed, that looks like an Airflow DAG task calling an ingest entrypoint with a payload. We might reuse similar architecture: tasks in a scheduler that call into adapters.

Live API Ingestion: This phrase suggests that ingestion might not just be one-off jobs but possibly continuous syncs (like a scheduled fetch of new data from sources). In the future, we might integrate something like scheduled adapter runs (like Airbyte does incremental syncs). For now, our focus is on on-demand ingestion, but we keep the door open for continuous ingestion (which could be a separate service polling and pushing to /ingest endpoints).

By providing this adapter SDK, the system is not limited to currently known sources. An AI agent could even be tasked in future to create a new adapter given a spec – just by writing a YAML to define how to get data from a new API. This boosts adaptability across *multiple knowledge domains*, as each domain's data sources can be plugged in using the same framework. Finance domain could have adapters for EDGAR (SEC API), stock data APIs, etc., and law domain for legal databases or EDGAR (legal case repos). All co-exist under the unified ingestion interface.

To emphasize, this approach follows the strategy of "configuration-driven connectivity" found in many integration platforms [72]. We'll cite that as an inspiration: *connector SDKs/frameworks like Apache Camel or Singer let developers plug into various endpoints without starting from scratch* [72]. We strive for a similar plug-in model here.

Edge Services & Operational Concerns

Objective: Incorporate features that improve operability and reliability of the system at the edges, including monitoring, tracing, error handling, caching, and rate limiting. These are essential for a production-quality system that is observable and user-friendly.

- **Monitoring & Tracing (Observability):** We will integrate **Prometheus metrics** collection and **OpenTelemetry tracing** into the gateway and microservices. Each request (REST, GraphQL, etc.) will record metrics like request count, latency, response codes, etc., tagged by endpoint and outcome. For example, a counter `api_requests_total{endpoint="/ingest/clinicaltrials",status="200"}` increments on successful calls, and similar for errors. Latency histograms help identify slow endpoints. We'll expose a `/metrics` endpoint for Prometheus to scrape. For tracing, we instrument the code with OpenTelemetry SDK: each incoming request starts a trace span, and we propagate context to any gRPC calls or database calls. This way, a single trace can show the entire journey of an ingest job: from API call through orchestrator steps and gRPC calls, maybe even down to database writes. This distributed tracing is valuable for debugging performance issues or failures across components. We could use Jaeger or Zipkin to collect traces. The code might include something like:

```
with tracer.start_as_current_span("embed_service_call") as span:
    span.set_attribute("chunk_count", len(chunks))
    response = embed_stub.EmbedChunks(request)
```

which attaches trace info to the gRPC call. We also ensure to log trace IDs in application logs so we can correlate.

- **Structured Error Responses:** When errors occur, we will return structured error envelopes consistently.

- For REST APIs: use **RFC 7807** Problem Details in JSON [59]. This means an error response might look like:

```
{
  "errors": [{
    "status": "400",
    "title": "Validation Error",
    "detail": "Invalid NCT ID format",
    "code": "VALIDATION_ERROR"
  }]
}
```

This conforms to both RFC7807 and JSON:API error object structure (they are compatible with slight differences). We include a machine-readable `code` and human-readable message. For certain domain errors, we might include specific fields (e.g., which parameter was wrong). But we avoid leaking internal info. We will define a central error handling module that catches exceptions and formulates these responses.
- For GraphQL: GraphQL errors will be in the `errors` array of the response, but we can attach an `extensions` object with additional info (GraphQL spec allows that). For instance:

```
{
  "errors": [{
     "message": "Forbidden",
     "extensions": { "code": "FORBIDDEN", "scopeRequired": "kg:write" }
  }]
}
```

This aligns with GraphQL best practices (e.g., Apollo's error codes in extensions). We will map HTTP status codes to GraphQL error codes appropriately.
- We ensure that error responses do not vary unpredictably. They should always either be a Problem+JSON (for REST) or GraphQL error format. This helps clients programmatically handle errors. In the OpenAPI spec, we document the possible error responses for each operation (with a schema reference to an `Error` model, as seen in the snippet [73]).

- We adopt a “fail closed” principle: better to error out than return partial/misleading data. Many of our endpoints (like extract) will reject if something is off (no spans, validation fail) [35] , using a 4XX code to indicate client-side issue if input missing parts, or 500 if something unexpected happened.

- **Caching (ETags & Conditional Requests):** For GET endpoints (like future addition: `GET /document/{id}` or `GET /retrieve` results for same query), we will implement HTTP caching support. Specifically:

- Include **ETag headers** on responses, which are hashes of the content. For example, an ETag could be the document content hash or a version number from the database.
- Support **If-None-Match** on requests: if the client sends an ETag and the resource hasn’t changed, we return 304 Not Modified with no body. This reduces bandwidth and speeds up clients re-fetching data.
- Use **Cache-Control** headers appropriately. For dynamic data we might set `Cache-Control: private, no-store` if data shouldn’t be cached by intermediaries. But for relatively static or public metadata, we could allow caching. In a multi-tenant system, likely private caching only (client-side).
- Also consider caching at the application layer: e.g., results of certain expensive queries (embedding retrieval or external API calls) could be cached in Redis to avoid recomputation if the same request comes again soon. We must be careful with cache invalidation (especially if underlying data changes). But e.g., for retrieval results for identical queries, a short TTL cache could be useful.

- The document mentions use of ETag explicitly (which implies these practices).

- **Rate Limiting:** As mentioned in security, we implement rate limiting to protect against abuse. We might integrate a library in the FastAPI app that uses an in-memory counter per IP or token. Or better, use a distributed approach with Redis if we have multiple instances. Basic strategy:

- Define rates for different categories (maybe higher for read operations, lower for heavy writes).
- Use a token bucket or leaky bucket algorithm. For example, allow burst of 5 and sustained 1 per second for ingest for each client.
- When limit exceeded, respond 429 Too Many Requests. Possibly include in the response headers or body the time to reset.
- Also log these events (could feed into monitoring to see if certain clients need higher limits or if an attack is occurring).

- In a deployed scenario behind a reverse proxy, the proxy could also do rate limiting, but implementing at app level ensures even if directly hit, we protect it.

- **Request/Response Logging & Debugging:** In non-production modes, we can allow a trace of input/output (with sensitive data scrubbed) for debugging. Also, correlation IDs: We can generate a unique ID per request (if not present) and include it in responses (`X-Request-ID` header). This is useful for support: a user can report “Request ID abc failed” and we find it in logs/traces.

- **Prometheus & Alerting:** The metrics we collect will feed into alerts. For example, alert if error rate > X% in last 5 minutes, or if GPU job queue length > N (indicating backlog), or if extraction accuracy metrics dip (harder to auto-alert on that). Also track resource usage: we can instrument GPU usage via DCGM exporter or custom scripts and scrape them. So ops can know if we're nearing capacity.

- **Deployment Topology Considerations (Edge):** The "edge" of the system might also involve an API gateway or CDN. In a production environment, we might put Cloudflare or an API management layer in front for additional caching and WAF (Web Application Firewall). Not in scope to implement, but our design should not hinder it. Using standard headers and codes means those components work easily with our service.

By addressing these edge concerns, we ensure the system is robust in real-world use: it can be debugged (tracing), tuned (metrics), it behaves nicely with web infrastructure (caching, proper error codes), and it self-protects (rate limits). All these are critical for quality in an enterprise environment, complementing the functional requirements we built.

Project Structure, CI/CD, and Deployment

Objective: Establish a clear monorepo structure for all components, along with CI/CD pipelines that enforce code quality, contract compliance, and performance benchmarks. Provide infrastructure-as-code for local and production deployment, including Docker Compose for simplicity and Kubernetes manifests for scaling.

Monorepo Layout: All code (gateway, microservices, adapters, etc.) will live in a single repository for easy coordination. A proposed structure:

```
/api-gateway-monorepo/
├── gateway/                # API Gateway (REST/GraphQL server)
│   ├── main.py
│   ├── routes/            # route handlers for REST
│   ├── graphql/           # GraphQL schema and resolvers
│   ├── security/          # auth middleware
│   ├── events/            # SSE implementation
│   └── models/            # Pydantic models (core entities, requests,
responses)
├── services/
│   ├── mineru_service/    # gRPC service for MinerU
│   ├── embed_service/     # gRPC service for embeddings
│   ├── extract_service/   # maybe for extraction tasks (or this could be in
gateway)
│   └── ... (other microservices)
├── adapters/
│   ├── base.py            # Adapter SDK base classes
│   ├── clinicaltrials.yaml
│   ├── dailymed.yaml
│   ├── pmc.yaml
│   └── ... (other adapters)
├── proto/
```

```
│   ├── mineru.proto
│   ├── embed.proto
│   └── ... (proto files)
├── docs/
│   ├── openapi.yaml       # OpenAPI 3.1 spec
│   ├── asyncapi.yaml      # AsyncAPI spec
│   ├── schema.graphql     # GraphQL SDL
│   └── developer_guide.md # Additional docs for developers
├── tests/
│   ├── unit/
│   ├── integration/
│   ├── contract/
│   ├── performance/
│   └── test_data/
├── ops/
│   ├── Dockerfile
│   ├── docker-compose.yml
│   ├── k8s/
│   │   ├── gateway-deployment.yaml
│   │   ├── gateway-service.yaml
│   │   ├── mineru-deployment.yaml
│   │   ├── ...
│   └── monitoring/
│       ├── prometheus.yml
│       └── grafana_dashboards/
└── ci/
    ├── github_workflows/   # if using GitHub Actions
    ├── Dockerfile.ci       # image with tools like schemathesis, k6, etc.
    └── scripts/            # CI bash or Python scripts
```

This layout groups components logically. The `gateway` is a Python FastAPI (or Flask) app for REST & SSE plus possibly Starlette for GraphQL (or we use a unified FastAPI for both). The `services` could be separate Python processes for gRPC services, each possibly with their own Docker image if needed. Adapters are mostly config, plus maybe code if needed (could have subfolders if some adapters need custom parser code). The `proto` folder is for .proto definitions – we will use Buf to generate code into the respective service directories (or a common pb2 library).

The `docs` folder holds our API specs and documentation content. We might auto-generate `openapi.yaml` from the FastAPI using scripts, but maintaining it explicitly might be better for planning. GraphQL SDL can be dumped from code or written and kept in sync with code. AsyncAPI we'll write manually.

CI/CD Pipeline: We will configure a pipeline that on each commit or pull request runs: - **Linting & Formatting:** Run flake8/black for Python, eslint for any JS (if docs site has it), buf lint for proto, etc. - **Unit Tests:** Fast tests for functions, models. - **Integration Tests:** Spin up necessary services (maybe using docker-compose in CI) and run tests that exercise multiple components (e.g., call the REST API which triggers a dummy gRPC service). - **Contract Tests:** Use **Schemathesis** to test our running API against the OpenAPI spec. Schemathesis will generate various inputs to ensure our service doesn't 500 on edge cases and matches schema. For GraphQL, use a tool like **GraphQL Inspector** to compare current schema vs last schema for breaking changes. Also maybe use Apollo's `graphql-schema-linter` for schema conventions. - **gRPC Contract Tests:** Buf's `buf breaking` can compare proto changes to avoid breaking clients. Also, we could compile protos and run a small test to ensure the service responds to a simple RPC call as expected. - **Performance Tests:** Perhaps not on every commit, but periodically or on release, run **k6** or Locust tests to ensure performance goals (like P95 latency < X) are being met. For example, k6 script could simulate 5 concurrent ingestion jobs and 20 concurrent retrievals, and we verify the latency and error rates. We integrate these results – if k6 metrics exceed thresholds, the pipeline can fail or at least warn. - **Security Tests:** Run dependency vulnerability scan, and maybe a ZAP scan on the running API (to check for common web vulnerabilities). - **Build & Deploy:** If tests pass and on main branch, build Docker images for each component. Use consistent tags (maybe git SHA or version number). Push to registry. Then either deploy to a staging environment automatically or, for production, perhaps require manual approval.

This likely would be implemented with a CI service (GitHub Actions, GitLab CI, Jenkins, etc.). For example, a GitHub Actions workflow YAML with jobs for test, build, etc. Or if using Jenkins, a Jenkinsfile in `ci/` directory.

Docker Compose (Development & Self-hosted Deploy): We provide a `docker-compose.yml` that stands up the whole stack easily, suitable for local dev or a small-scale deployment. For example:

```
services:
  api-gateway:
    build: ./gateway
    ports: ["8080:8080"]
    env_file: .env
    depends_on: ["neo4j","opensearch","mineru_service","embed_service"]
  neo4j:
    image: neo4j:5
    env_file: neo4j.env
    ports: ["7474:7474", "7687:7687"]
  opensearch:
    image: opensearch:latest
    # config...
  mineru_service:
    build: ./services/mineru_service
    runtime: nvidia   # if using GPU in docker
    environment:
      - CUDA_VISIBLE_DEVICES=0
  embed_service:
    build: ./services/embed_service
    runtime: nvidia
  # ... maybe other services like a schedule worker
```

Additionally, a volume or path for storing ingested PDFs, etc., can be included (or use S3 if available). Compose allows quick startup for a developer or an on-prem test deployment. We might have to provide dummy credentials or instruct to fill env files (for OAuth server URL, etc.).

Kubernetes Manifests: For more robust deployment, we include lightweight K8s manifests. These could be in `ops/k8s/`. They might not be a full Helm chart (unless needed), but enough to deploy to a cluster: - Deployment for gateway, with liveness/readiness probes and resource limits (and possibly nodeSelector if it needs GPU? Actually the gateway itself not, but the embed/mineru services do). - Deployments for each gRPC service. For GPU ones, use `nvidia.com/gpu: 1` resource requests and tolerations so they schedule on GPU nodes. - Services (ClusterIP) for each component. - If needed, an Ingress resource for exposing the gateway with TLS. - ConfigMaps for config files (like the adapter YAMLs could be mounted via ConfigMap, or included in image). - A Secret for OAuth public keys, DB passwords, etc. - (Optional) HorizontalPodAutoscaler for scaling gateway based on CPU or queue length. - (Optional) Argo Workflow template if orchestrator was external, but we have it in app, so no.

These manifests can be applied directly (`kubectl apply -f ops/k8s/`). We might not cover a full production-ready infra (like load balancer, etc.), but enough to be a starting point.

CI Integration for Deployment: We can have a CD step that on tagging a release, deploys to a staging cluster using those manifests (or a Helm chart if we abstract it). Possibly using GitOps (ArgoCD) could be mentioned but not necessary detail here.

Continuous Testing & Monitoring in CI: We may set up nightly or weekly jobs that run extensive tests: - Retrain or refresh any ML models? (Maybe out of scope here). - Rerun performance tests and capture trending metrics (feed into a Grafana maybe). - Check external data sources connectivity (ping an adapter to ensure source API hasn't changed). This ensures the system remains healthy and up-to-date, especially if external APIs updated their responses (the adapter should catch if JSON structure changed, etc., we'd get a test failure).

Static Documentation Site: We will generate a static docs portal (could use a tool like **Docusaurus** or **MkDocs** in `docs/` folder). It will include: - **OpenAPI**: an interactive Swagger UI or Redoc documentation. Possibly host the `openapi.yaml` and include a Swagger UI bundle pointing to it. Swagger UI allows trying out endpoints (if CORS is allowed). - **GraphQL**: embed GraphQL Playground or GraphiQL. This might require the docs site to proxy to the API or we just instruct the user to use the GraphQL Playground at runtime. Alternatively, a static schema documentation (GraphQL Docs) could be generated. - **AsyncAPI**: use AsyncAPI's React component to render the `asyncapi.yaml`. AsyncAPI has a generator for HTML documentation as well. - **Guides and How-tos**: e.g., authentication guide (how to obtain token), example workflows (like how to ingest a new study and retrieve results step by step). - **References**: links to the standards (OpenAPI spec, GraphQL spec, etc.) as provided, so developers can learn background. We include references such as: - OpenAPI Specification [1] - GraphQL documentation [6] - JSON:API format [2] - OData intro [4] - gRPC framework [9] - AsyncAPI initiative [10] - OAuth2 protocol [60] (These references are also included throughout this blueprint as footnotes for convenience.)

We can host this docs site via GitHub Pages or as a static container. Additionally, the gateway itself might serve docs: e.g., at `/docs/openapi/` we host Swagger UI (Swagger UI can be served by the app), at `/docs/graphql/` host GraphQL Playground, etc., so that developers can explore live.

Role Assignments & Team Workflow: While the AI agent might do much automatically, in a real project team we'd see roles like: - *Lead Architect:* defines the overall structure (as we've done), ensures all pieces fit together. - *Backend API Developer:* implements the gateway (REST/GraphQL), ensures security and correctness. - *ML Engineer:* builds the gRPC services for MinerU, embedding, etc., deals with GPU code and model integration. - *Data Engineer:* focuses on the Adapter SDK and ingestion logic for sources, writing new adapters as needed. - *DevOps Engineer:* sets up CI/CD pipelines, Docker/K8s deployment, monitoring stack (Prometheus/Grafana), etc. - *QA Engineer:* writes test plans, uses tools like Schemathesis, k6, and monitors quality metrics, ensures spec compliance. - *Domain Expert (per domain):* validates that domain overlays (FHIR alignment, etc.) are correctly implemented and that the outputs make sense for end users.

These responsibilities might overlap, but clear ownership helps. In our plan, the tasks are segmented in a way an AI agent or human team can implement each in parallel: one could start on the OpenAPI endpoints while another on the GraphQL schema, etc., with the common models defined early to avoid conflicts.

CLI Tools for Developers: We might add helpful scripts: - `make setup` to install dependencies, pre-commit hooks (for linting). - `make test` to run all tests, `make format` to auto-format code. - `make run-local` to start services (or just `docker-compose up`). - CLI to generate code from proto: e.g., `buf generate` (with a buf.yaml configured to output Python stubs to appropriate places). - Possibly a CLI to update the OpenAPI spec from code annotations or vice versa.

By structuring the project well and automating the CI/CD, we ensure **quality and consistency**. Every commit is validated against the contract (OpenAPI, GraphQL, Proto), so we maintain alignment with specs. The performance tests ensure we catch regressions early (like a change that slows down retrieval significantly). Deployment scripts ensure that what we test is what goes live.

Finally, we maintain **documentation** alongside code, and treat it as a living part of the project (with updates in PRs when endpoints change). This emphasis on docs, tests, and CI embodies the "focus on quality and spec alignment" directive of the project.

By following this plan, the AI programming agent (or development team) can systematically build a sophisticated API gateway and orchestration system that is robust, extensible, and aligned with multiple industry standards, all while being adaptable to various data domains.

References (Standards and Specifications)

- OpenAPI Specification – Standard interface description for REST APIs [1]
- GraphQL Query Language – Flexible query API with a typed schema [6]
- JSON:API Format – Convention for JSON-based REST API responses [2]
- OData Protocol – Standard for queryable REST APIs (filtering, etc.) [4]
- gRPC Framework – High-performance RPC for microservices (uses Protocol Buffers) [9]
- AsyncAPI Specification – Documentation for event-driven (async) APIs [10]
- OAuth 2.0 Authorization – Industry-standard protocol for secure API access [60]

---

[1] OpenAPI Specification v3.1.1
https://spec.openapis.org/oas/v3.1.1.html

[2] [3] JSON:API — Latest Specification (v1.1)
https://jsonapi.org/format/

[4] Open Data Protocol - Wikipedia
https://en.wikipedia.org/wiki/Open_Data_Protocol

[5] [43] Basic Tutorial · OData - the Best Way to REST
https://www.odata.org/getting-started/basic-tutorial/

[6] [8] Introduction to GraphQL | GraphQL
https://graphql.org/learn/

[7] [44] [72] Tools and Standards for Universal API Connectivity.pdf
file://file-K9UdZiQWnX8ey6u5pso6n9

[9] gRPC
https://grpc.io/

[10] Getting started with AsyncAPI: how to describe your Kafka cluster | Kafka Summit Europe 2021 - Confluent | DE
https://www.confluent.io/de-de/events/kafka-summit-europe-2021/getting-started-with-asyncapi-how-to-describe-your-kafka-cluster/

[11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [73]
Overall_Project_Scope.md
file://file-LioogMCUPGC4LYv3GMEcc5

[60] OAuth 2.0
https://oauth.net/2/

[71] Connector Development | Airbyte Docs
https://docs.airbyte.com/platform/connector-development