ChatGPT

# Visionary AI – Multimodal AI System Project Specification

## Overview and Objectives

Visionary AI is a production-ready **multimodal AI system** designed to extract and understand information from visual and textual data. It integrates cutting-edge **open-source** models for Optical Character Recognition (OCR) and Multimodal Large Language Understanding, targeting extremely high accuracy (≥99%) across English, Chinese, Korean, Japanese, and Spanish text. The system will enable **retrieval-augmented generation (RAG)** workflows that combine vision and language (e.g. OCR + LLM reasoning) for advanced document analysis. It will be deployed on **AWS cloud infrastructure** with a strong emphasis on cost-effectiveness and performance. Additionally, Visionary AI's architecture is built with **SOC 2 compliance** in mind – incorporating security, availability, processing integrity, and confidentiality measures from the ground up. This specification provides a detailed architecture design (including an architectural diagram), component breakdowns, UI/UX specifications (per the Visionary AI wireframes), DevOps strategies, CUDA-level optimizations for model inference, full user stories/use cases, and justification for all technology and model choices.

## Key Requirements and Features

- **Open-Source, High-Accuracy Models:** Use state-of-the-art open-source OCR and multimodal language models (pre-trained or fine-tuned) that rival proprietary solutions in accuracy [1]. These models must be capable of complex layout understanding (tables, forms, multi-column text) and multilingual OCR at ~99% character-level accuracy. By leveraging open models (e.g. AllenAI's **olmOCR** based on Qwen-VL or similar), Visionary AI avoids vendor lock-in while matching the performance gap to top proprietary models within ~1–3% [1].
- **Multilingual OCR (EN, ZH, KO, JA, ES):** The OCR engine will support English, Simplified/Traditional Chinese, Korean, Japanese, and Spanish out-of-the-box. It should maintain very high accuracy across these scripts (Latin and CJK), aiming for human-level text recognition. To achieve ≥99% accuracy even on non-Latin text, the system uses fine-tuned vision-language models (for example, a 7–14B parameter VisionLLM specialized in OCR) and domain-specific post-processing. We will incorporate **error-correction workflows** (e.g. spell-checking, language model validation) and optional human-in-the-loop review for low-confidence outputs to push effective accuracy above 99% [2].
- **Multimodal RAG Pipelines:** Enable Retrieval-Augmented Generation that combines visual understanding with text-based retrieval. For example, Visionary AI can take an image (with text) as input, perform OCR to get text content, retrieve relevant context from a knowledge base, and then use an LLM to answer questions or generate reports using both the image and retrieved text. This means **images and text are jointly processed** – e.g. an image of a document is converted to text which is then used to query a vector database for related info, and a response is generated by the LLM [3]. The system supports multimodal queries like: *"Analyze this scanned contract and summarize obligations,"* or *"Find related documents to this image and answer a question combining them."* The RAG

pipeline ensures the LLM's knowledge is up-to-date and extended with domain-specific data, overcoming context limits and knowledge cutoffs [3].

- **AWS Deployment (Cost-Effective):** The entire solution will be deployed on **Amazon Web Services** using a scalable, cost-efficient architecture. Wherever possible, serverless and on-demand provisioning will be used to minimize idle resources (for example, using AWS Lambda and **auto-scaling GPU instances** for model inference). The design prioritizes **resource efficiency** – e.g. utilizing spot instances or burstable compute for non-critical jobs, segregating workloads by latency requirements, and leveraging managed AWS services (like S3, DynamoDB, etc.) that offer pay-as-you-go pricing. **Infrastructure-as-Code** (Terraform or CloudFormation) will define all resources for repeatable deployment, and cost monitoring will be in place to ensure the solution remains within budget constraints. We choose open-source models in part to avoid external API fees, enabling processing of millions of pages at very low incremental cost (e.g. < $0.0002 per page as reported with open VLM-based OCR [4]).
- **Performance Optimizations (Beyond Standard):** Visionary AI will integrate first-hand performance optimizations at the **CUDA** level to maximize throughput and minimize latency. In addition to standard techniques like model quantization (int8/fp16) and pruning, we will apply **GPU kernel-level optimizations**. This includes using optimized libraries (like NVIDIA **TensorRT** and FlashAttention kernels) and even writing custom CUDA kernels for critical operations if needed (fusing operations to reduce memory bandwidth usage, optimizing multi-head attention calculations, etc.). The team's expertise in "CUDA-level dark arts" (manual GPU tuning) will be leveraged to achieve inference latencies in the low hundreds of milliseconds or less per request. For LLM serving, we plan to use **vLLM**, an advanced open-source serving engine that improves memory utilization and throughput for large models [5]. vLLM's design (with optimizations like paged attention and continuous batching) yields significant latency reductions (e.g. 5× faster time-per-token in benchmarks) and better GPU utilization, ensuring users get responses under real-time constraints. The goal is to **obsess over latency** – optimizing the pipeline so that even complex OCR+LLM tasks feel instantaneous (sub-100ms to first word for prompt responses, where feasible).
- **End-to-End SOC 2 Compliance:** The architecture will incorporate **SOC 2 Trust Principles** – Security, Availability, Processing Integrity, Confidentiality (and Privacy as applicable) – from day one. This includes robust **security controls** (network isolation, encryption of data at-rest/in-transit, strict IAM roles and least privilege access, audit logging of all access), **high availability** design (redundant instances across AZs, auto-recovery, backup and disaster recovery plans for critical data), **processing integrity** (input validation, error detection and correction, idempotent processing to avoid data loss or duplication, and tracking of document processing with checksums or hashes to ensure completeness), and **confidentiality** (safeguards for sensitive data like PII, e.g. automatic redaction using the AI if needed, plus secure handling and storage with encryption keys managed by AWS KMS). Compliance is further supported by using AWS services compliant with SOC 2 out of the box and maintaining detailed documentation of all controls. The system will log all transactions and security events (using services like CloudTrail and CloudWatch) to provide an audit trail for SOC 2 auditors. In essence, security and compliance are not an afterthought but are **embedded in the architecture**, aligning with the organization's critical requirements (e.g. **SOC2 experience is a must** as highlighted in the project requirements).

## High-Level Architecture Overview

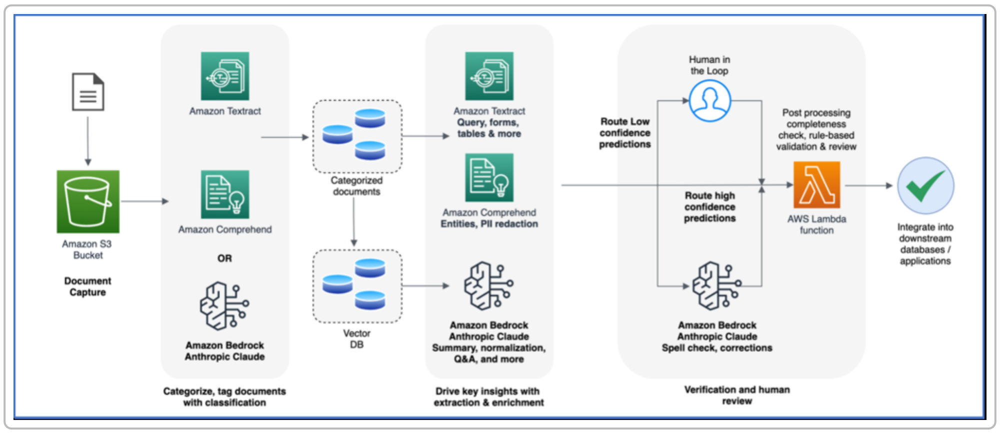
Amazon S3 Bucket
Document Capture
Amazon Textract
Amazon Comprehend
OR
Amazon Bedrock Anthropic Claude
Categorize, tag documents with classification
Categorized documents
Vector DB
Amazon Textract Query, forms, tables & more
Amazon Comprehend Entities, PII redaction
Amazon Bedrock Anthropic Claude Summary, normalization, Q&A, and more
Drive key insights with extraction & enrichment
Route Low confidence predictions
Human in the Loop
Route high confidence predictions
Amazon Bedrock Anthropic Claude Spell check, corrections
Verification and human review
Post processing completeness check, rule-based validation & review
AWS Lambda function
Integrate into downstream databases / applications

*Figure 1: High-level architecture for Visionary AI's document processing pipeline, inspired by an AWS IDP (Intelligent Document Processing) reference design. The flow begins with documents/images being ingested (left) and processed through text extraction (OCR) and AI analysis (center), then output to users with potential human review for low-confidence cases (right). Our system replaces AWS Textract and Bedrock in the reference with open-source OCR and LLM components, but follows a similar robust pattern.*

At a high level, Visionary AI's architecture consists of the following layers and components:

- **Ingestion Layer (Input):** Interfaces for users to submit input data – this could be an image upload via the web UI or an API endpoint receiving a document. Inputs are funneled into a secure S3 bucket or an internal storage service. New documents trigger an event in the pipeline.
- **Processing Layer (Backend Services & Models):** A collection of microservices and model servers that perform OCR, language understanding, retrieval, and analysis:
- An **OCR Service** accepts images and produces extracted text (with layout metadata).
- A **Multimodal LLM Service** can accept text, images, or a combination, and produce answers, summaries, or other generated outputs. It powers advanced analysis like Q&A or summarization about the documents.
- A **Retrieval Service** (with a vector database) handles storing and fetching embeddings of text (and possibly image embeddings) to support RAG. It enriches LLM outputs with relevant context.
- An **Entity Graph Service** builds and queries a knowledge graph of entities and relationships found across documents.
- A **Workflow Orchestrator** (could be an AWS Step Function or a custom service) coordinates these components – for example, when a new document arrives, orchestrating OCR, then entity extraction, then storage of results, etc., and when a user query arrives, orchestrating retrieval then LLM response.
- **Data Layer (Storage & Databases):** Persistent storage solutions:
- **Blob Storage:** Amazon S3 for storing raw images, documents, and possibly intermediate outputs (like OCR JSON results). S3 provides high durability and lifecycle management (e.g. archiving old files).
- **Vector Database:** e.g. **Qdrant** or **Pinecone** (hosted) or Amazon OpenSearch with vector indices. This stores embeddings for document text (and potentially image content), enabling semantic search for RAG. It allows similarity search across multimodal data.

- **Graph Database:** e.g. Amazon Neptune or Neo4j, used to store the extracted **Relationship Graph** of entities. This DB holds nodes (entities like people, organizations, dates, document IDs) and edges (relationships like “Person X mentioned in Document Y” or “Company Z referenced in Contract Q by Person X”). It enables complex queries about how information in the documents interconnects.
- **Relational/NoSQL DB:** Amazon DynamoDB (NoSQL) or PostgreSQL (relational) for metadata and system data. For instance, DynamoDB can track processing status for each document (ensuring “exactly-once” processing integrity) [6], and store user data, access logs, or configuration. A relational DB could store structured outputs (like extracted key-value fields from documents) for reporting.
- **Application Layer (API & Integration):** This layer exposes the functionality to front-end and external systems:
- **REST/GraphQL API Gateway:** An Amazon API Gateway or Application Load Balancer routing client requests (from the web UI or external clients) to the appropriate backend services. It provides a unified interface with authentication (e.g. JWT or Cognito) and throttling for fairness/security.
- **Microservices:** Each major function (OCR, LLM query, etc.) runs as a separate service (could be containerized on ECS/EKS or serverless). Services communicate through well-defined APIs or an internal message bus (e.g. AWS SQS or gRPC calls). This modular design supports scaling and independent development of each component.
- **Authentication & Authorization:** Integrated with AWS Cognito or an enterprise IAM, ensuring only authorized users or services can invoke certain operations (especially important in a multi-tenant setup or if sensitive data is involved).
- **Presentation Layer (UI/UX):** The front-end web application (and/or any desktop or mobile apps) that users interact with. This layer includes the **OCR & Text panel**, **Relationship Graph interface**, and **Reporting Dashboard** (detailed in the UI/UX section). The front-end communicates with the backend via the API layer, presenting results and visualizations to the user in real time. Rich interactivity (e.g. highlighting recognized text on images, interactive graph exploration) is implemented here, while heavy computation stays in the backend.

The entire architecture is deployed within an **AWS Virtual Private Cloud (VPC)** for security. All application servers and databases reside in private subnets; only the load balancer or API endpoint is public-facing. Network security groups and NACLs restrict traffic flow (e.g. model servers only accept traffic from the API or orchestrator, databases only accept from application servers). This layered design ensures a **clear separation of concerns**: the system can scale each part (OCR, LLM, DB) independently and maintainability is improved (each module can be updated/upgraded separately).

## Component Breakdown and Design Details

### 1. OCR Engine (Visionary OCR Service)

**Purpose:** Extract printed (and possibly handwritten) text from images with very high accuracy, supporting multiple languages and complex layouts. This service is the foundation of Visionary AI’s capability to turn visual content into machine-readable text.

**Model Choice:** We will deploy an open-source **Vision-Language OCR model** fine-tuned for multi-language text extraction, such as AllenAI’s *olmOCR 7B* (based on Qwen-VL-7B) or an equivalent state-of-the-art model. This model has demonstrated the ability to convert diverse document images (scans, photos of text, etc.) into structured text (like Markdown or JSON) with remarkable accuracy [4]. The OCR model can handle complex layouts (multi-column pages, tables, etc.), preserving structure (e.g. by Markdown headings, lists, table syntax). It also supports Chinese/Japanese/Korean characters and Latin alphabets seamlessly – for instance, Qwen-VL based models report ~95% accuracy on Chinese and similarly high on English [7]. We anticipate using a **single unified model** for all languages (simplifying the pipeline), possibly with language-specific pre- and post-processing. Alternatively, if needed for accuracy, we may ensemble two OCR engines (e.g. one specialized in Latin script, one in CJK) and then combine outputs.

**How it Works:** When an image or PDF page is input, the OCR service (running on a GPU instance) will perform two stages: (1) **Text Detection** – locating text regions in the image (using a CNN-based detector or the model's visual encoder capabilities), and (2) **Text Recognition** – transcribing each region to text. The output is a structured text representation plus metadata: for example, a JSON containing lines/words with coordinates, or a Markdown text that roughly mirrors the original formatting. The OCR service will also produce a confidence score for each text block or word. Low-confidence regions can be flagged for human review or secondary processing (to ensure integrity).

**Accuracy and Optimizations:** To reach the 99% accuracy target, we implement several strategies:
- *Fine-tuning and Calibration:* The chosen model is fine-tuned on datasets covering the target languages and document types (e.g. invoices, forms, legal docs). We will validate its accuracy on each language – if any language underperforms (say Korean printed text only 97% accurate), we will incorporate additional fine-tuning data or use augmentation.
- *Post-processing:* The OCR output is post-processed to correct common errors. For example, we'll fix OCR-specific mistakes like confusing "O" vs "0" or commas vs periods in numbers [8] using context rules. We'll also run spell-checkers or language models on recognized text to catch and correct likely errors (especially for OCR'd sentences that are not dictionary words). Domain-specific lexicons can improve accuracy (e.g. known company names, product codes can be checked against a database).
- *Confidence-based Human-in-the-Loop:* In mission-critical use cases, any extracted text with confidence below a threshold (or which fails a checksum/validation rule) will be routed to a human operator or a secondary verification process. This design, inspired by industry practice, can effectively raise **effective accuracy to 99%+** even if raw model accuracy is slightly lower [2]. For example, if ~8% of receipts need manual verification for unclear characters, the overall accuracy after correction can exceed 99.5% [2].
- *Handwriting:* If needed, the OCR service can use a specialized extension or model setting for handwriting (though handwriting is inherently tougher, trailing print by ~5–15% [9]). For now, the focus is high accuracy on printed text; handwriting support can be limited to certain scripts or addressed via an add-on model (open-source handwriting OCR).

**Deployment:** The OCR service will run in a Docker container (with necessary libraries like PyTorch, OpenCV, etc.) on an AWS GPU instance (such as a g4dn or g5 instance for cost efficiency, or a more powerful p4 for heavy loads). It could be orchestrated by Amazon ECS or EKS for scalability. We will enable auto-scaling: e.g. if OCR request queue length grows, spin up additional containers (possibly on spot instances to save cost). For batching efficiency, the service can also accept a batch of pages in one go – useful for multi-page PDF input. In such cases, it will process sequentially or in parallel streams depending on GPU memory, to maximize throughput (potentially hundreds of pages per minute per GPU, as reported in olmOCR benchmarks [4]). The OCR engine is exposed via a REST API (e.g. `POST /ocr` with image file returns JSON text) internally. Other components (like the Orchestrator or the UI) call this API to get text for a given image.

**Justification:** Using an open-source OCR model provides **cost savings** (no per-call fees) and full control over the system (we can customize or improve the model). Modern open models like Qwen-VL or Mistral-

OCR are extremely competitive, with only a couple percentage points difference from the very best closed models [1] – a gap we can narrow with fine-tuning and our error-correction workflow. This way, Visionary AI achieves near **human-level OCR accuracy** without the recurring costs of APIs. The architecture also allows swapping in improved OCR models in the future (e.g. if a new model with 99.9% accuracy on Japanese is released, we can integrate it), ensuring the system stays state-of-the-art.

## 2. Multimodal LLM Module (Vision + Language Understanding)

**Purpose:** After text is extracted by OCR, many higher-level tasks are possible – summarization, question-answering, classification, etc. The Multimodal LLM module is an AI service that can accept both textual data and images as inputs and generate intelligent outputs. It effectively **"understands"** the content of documents (beyond raw text) by processing images, text, and embedding results from the RAG pipeline.

**Model Choice:** We will use an **open-source Multimodal Large Language Model (MLLM)** – for example, an advanced Vision-Transformer + LLM hybrid. Candidates include models like **Qwen-2.5 VL** (14B) which has strong document understanding capabilities, or **LLaVA** / **BLIP-2** style models that connect a vision encoder to a language model. The chosen MLLM will be fine-tuned (or instructed) for tasks relevant to document AI: e.g. given an image and its OCR text, it can answer questions about it, explain it, or extract structured data. Notably, *olmOCR* itself is a vision-language model but specialized for transcription; in addition, we may employ a more general multimodal model for reasoning (one that can for instance look at a chart image and interpret it, or read an illustration). If resource constraints are a concern, we could use the same core model for OCR and reasoning (to avoid hosting two large models); however, separating concerns (a model for extraction and another for analysis) can be more optimal, as each can be tuned to its task. An example open MLLM is **Mistral 7B** with vision extensions or the **OpenFlamingo** framework that can use a Llama2 as language backend and CLIP as visual encoder. These models can be deployed locally and fine-tuned to our domain data, ensuring data never leaves our environment (important for confidentiality).

**Capabilities:** The MLLM service will provide functionalities such as:
- *Question-Answering:* A user can ask natural language questions about a document (or a set of documents). For example, "**Who is the signatory of this contract and on what date was it signed?**" The system will feed the image (or its extracted text) plus relevant retrieved context into the LLM, which will produce an answer referencing the document content. This requires the model to comprehend the OCR text and possibly the visual layout (e.g. signature block location). The MLLM can handle such multi-part input.
- *Summarization:* The system can generate summaries of documents or images. E.g. "Summarize this 10-page legal agreement in one paragraph." The MLLM will take the text (possibly chunked via retrieval) and produce a concise summary. Because of context window limits, it will likely use the RAG approach – summarize chunks and then summarize summaries, or use embeddings to pick the most relevant parts.
- *Classification & Extraction:* The LLM can categorize a document (is it an invoice, a resume, a lab report?) or extract key fields (like total amount, date, names). While some structured extraction might be done via deterministic rules or smaller models, the LLM provides a flexible way to get any ad-hoc information via prompt (for instance, "List all the products and their prices from this receipt").
- *Vision understanding beyond text:* If a document image contains non-textual elements like a **diagram or a chart**, the LLM (with its visual encoder) can interpret them. For instance, if a page has a bar graph with no text, an MLLM like PaLM-e or BLIP-2 could describe the graph. In our pipeline, for such tasks we'd provide the image itself (not just OCR text) to the MLLM's vision encoder. This aligns with **multimodal reasoning** trends, where models consider both images and text [10].

**Integration with RAG:** This module will be closely integrated with the Retrieval system. When answering a query or performing a task that exceeds its direct knowledge/context, the workflow will: 1) retrieve relevant text chunks from our vector database (for example, if the user asks a question that involves multiple documents), and 2) prepend those chunks as context to the LLM prompt. The MLLM then generates an output that **"augments" its knowledge with the provided context**, ensuring correctness and specificity [3]. For example, if asked *"Has this company been mentioned in any other documents?"*, the system will find those references via the vector DB and supply them to the LLM, which will then respond with something like: "Yes, Company XYZ appears in Document A and Document B, in contexts… [3]".

**Performance Considerations:** Running an MLLM is computationally heavy; however, we will optimize this by:
- Hosting the model on a GPU with sufficient memory (e.g. an **NVIDIA A10G** or **A100** for a 7B–14B model). We might use model quantization (4-bit or 8-bit weights) to reduce memory and possibly run a 13B model on a single GPU with minimal accuracy loss.
- The **vLLM server** mentioned earlier will be utilized here to serve the LLM. It allows efficient **batching of multiple requests** and dynamic allocation of GPU memory, so the system can handle many small queries or a few big ones without waste [5]. For instance, if two users ask questions at the same time, vLLM can process them together in one forward pass (if model capacity allows), increasing throughput.
- Caching: For repetitive queries or summaries, caching the LLM outputs (with cache invalidation on document updates) can save computation. Also, caching intermediate embeddings in the vector DB ensures we don't re-embed the same text repeatedly.

**Output Handling:** The LLM's responses will often be fed back into the UI (e.g. answer text displayed to user) or further processed. We will instruct the model via prompt templates to produce **structured output** when appropriate (for example, output JSON when asked to extract structured info, so we can easily use it). The model will also be prompted to cite sources if needed (especially in a RAG context, e.g. "According to Document X, …"), which can enhance **processing integrity** by making it clear which input data backs the answer.

**Justification:** Choosing an open-source MLLM ensures **data confidentiality** (all data stays within our AWS VPC) and allows tailoring the model to our specific use cases. Modern open MLLMs (like the Qwen family, Mistral derivatives, etc.) are highly capable – e.g. an 18B Qwen-VL model achieved ~90–95% accuracy across 23 languages in one benchmark [7]. While the very best proprietary models slightly outperform in multilingual consistency [11], our use of retrieval and fine-tuning can mitigate that gap. From a cost perspective, hosting our own model is more predictable and can be cheaper at scale than paying per token for an API. We also maintain control to implement **custom optimizations** (like using half-precision, adjusting the decoding for speed vs accuracy, etc.). The MLLM is central to providing *intelligent* features in Visionary AI – it transforms raw text into actual insights (answers, summaries) which add value beyond what basic OCR gives.

## 3. Retrieval and Knowledge Base Integration

**Purpose:** The retrieval component augments the system's capabilities by providing relevant information on-demand. It serves two main roles: (a) enabling semantic search across processed documents (so users can retrieve documents by content, not just exact text match), and (b) feeding the LLM with additional context (RAG) to ground its responses in relevant data. This is crucial for multi-document analysis and for keeping responses accurate and up-to-date with the latest ingested data.

**Design:** We will implement a **Vector Database** to store embeddings of textual content. After OCR is performed on documents, the resulting text (typically split into meaningful chunks, e.g. paragraphs or sections) is converted into high-dimensional vectors by an embedding model. We can use an open-source embedding model (for multilingual text, likely a model like **sentence-transformers** or a smaller multilingual MiniLM) to embed each chunk. These vectors, along with metadata (document ID, page number, etc.), are stored in the vector DB. We'll likely use **Qdrant** or **Milvus** (both open-source, can be self-hosted on AWS) or a fully managed service like Pinecone if ease is preferred. Vector search allows us to find text that is semantically similar to a query even if exact keywords differ.

**Multimodal Retrieval:** In a multimodal context, we might also store image embeddings. However, since our main approach is to immediately OCR images to text, the retrieval will largely operate on text (which covers the image's content). That said, if needed, we could also embed the **visual features** of an image (for instance using CLIP) to allow queries like "find images that look like this diagram" – but this is an edge capability. The primary use will be text-based retrieval of textual content.

**RAG Workflow:** When a user issues a query through the system (either via the UI query box or API), the workflow might be:
1. **Identify scope:** Determine if the question is about a specific document (if the user is viewing a document) or across all documents.
2. **Retrieve relevant chunks:** Formulate a search query. If the query is multimodal (e.g. user points at a figure and asks "what is this about?"), the system will include keywords from context (like caption text or neighboring text after OCR). For a text query, it may directly search. The vector DB is queried for the top N most relevant chunks of text related to the question. For example, if the user asks "**What are the payment terms for contract ABC?**", the system finds chunks in that contract (or others) that talk about payment terms.
3. **(Optional) Rerank or filter:** We can apply a reranking model or heuristic to refine the retrieved pieces, ensuring the most relevant information is selected.
4. **Compose LLM context:** The retrieved texts are then concatenated (with formatting and citations) and provided to the LLM as part of the prompt (e.g. "Using the information below, answer the question: ... [insert retrieved texts]"). The LLM then generates the final answer that *augments* its own knowledge with these snippets.
5. **Answer with citations:** Ideally, the LLM's answer includes references to the source documents (we will prompt it to do so). This not only boosts user trust but also satisfies **processing integrity**, as the system can point to exactly what data it used [12]. Those references can be displayed in the UI or used to link to the original document.

**Search Interface:** In addition to powering the LLM, the vector search allows a user to do a direct content search. For example, the UI might have a search bar to "Find documents mentioning 'Force Majeure'". The system would vectorize that query and return a list of documents/sections ranked by relevance. This is more powerful than a simple keyword search because it can find semantically related terms (e.g. "Act of God" references in place of "Force Majeure"). The results would be shown perhaps in a sidebar with snippets.

**Scalability:** The vector DB will be deployed on AWS such that it can scale to large data volumes (thousands or even millions of document vectors). If using Qdrant, we might run it on an EC2 instance with sufficient memory (as it can use disk-backed index for larger sets). We will enable **HNSW indexing** or similar for efficient approximate nearest neighbor search. Periodic re-indexing or index maintenance tasks will be scheduled as needed. To keep the RAG pipeline fast, we aim for vector queries that return in tens of milliseconds. We also consider caching frequent query results and using **distributed search** if data grows (multiple shards/nodes for the vector DB).

**Data Updates:** If documents are added or updated, the system will automatically index the new content. A document ingestion workflow (described later) handles this: when a new doc comes in, after OCR and processing, its text chunks are embedded and upserted into the vector store. Deletions or access revocations would similarly remove or hide those entries. The consistency between stored documents and the vector index is important – we will keep a mapping in our metadata DB to track which vectors belong to which doc and ensure synchronization.

**Justification:** The RAG approach is vital for combining **the power of LLMs with the reliability of a knowledge base** [3]. It allows Visionary AI to handle queries about any information in the document set without retraining the LLM on that entire corpus. This is much more **cost-effective** and flexible than training a giant model on all documents or having a model with an extremely large context window (which would be slower and more expensive per query). The use of an open-source vector DB and embedding models aligns with our open philosophy and avoids license costs. It also means we can deploy everything on-premise/AWS, maintaining confidentiality. By designing this retrieval pipeline, we ensure that as the data grows or changes, the system can scale and adapt without degrading the quality of answers – a key for long-term performance.

## 4. Relationship Graph & Entity Extraction Service

**Purpose:** Visionary AI includes a **Relationship Graph** view in the UI – a visual representation of entities (people, companies, places, etc.) and their relationships as derived from the documents. This component of the system is responsible for extracting those entities and relationships, populating a graph database, and answering queries about connections. It essentially gives users an interactive knowledge graph built from their unstructured data, which can reveal insights like which documents involve the same person or how various entities are linked through contractual relationships.

**Entity Extraction:** After OCR (and possibly with help from the LLM for context), we identify key **entities** in each document: examples include person names, organization names, dates, addresses, monetary values, etc. We will utilize a combination of **Named Entity Recognition (NER)** models and heuristic rules for this. An open-source NER model (e.g. spaCy models or HuggingFace transformers like `dslim/bert-base-NER`) can tag common entities in English. For multilingual support, we can employ multilingual NER models (like XLM-RoBERTa-based) or even use the LLM itself by prompting ("Extract all person names from this text"). The advantage of using a separate NER model is efficiency: it can quickly tag entities in text without invoking a large LLM each time. We will likely run NER on the OCR text of each document as part of the ingestion pipeline.

**Graph Construction:** Once entities are extracted, we create entries in the graph database. Each document can be considered a context that links various entities together (for instance, a contract document may link a Company entity and a Person entity under a "hasSignatory" relationship). We define the types of nodes: e.g., **Document** nodes, **Person** nodes, **Organization** nodes, etc. And define relationships: - "Person **AUTHORED** Document" (if someone wrote or signed it), - "Organization **MENTIONED_IN** Document", - "Person **MENTIONED_WITH** Person" (if two people appear in the same document, indicating an implicit relationship), - "Document **REFERS_TO** Document" (if one document quotes or cites another, perhaps detected by text like "see Appendix in Doc X").

The graph service logic will have to deduplicate entities – e.g., “IBM” and “International Business Machines” might appear, we need to decide if those are the same entity or not (this might require alias lists or user input to merge). Initially, we can treat distinct strings as distinct nodes and later allow merging.

**Usage:** The Relationship Graph can be queried in a few ways: - The UI might allow the user to select an entity (say a company) and then highlight all documents and other entities connected to it. This is essentially retrieving the **neighborhood** of that node in the graph (e.g. all edges from Company X node: which documents it appears in, which people appear in those documents, etc.). - Users might ask questions like “**Show me how Person A is connected to Person B**”. The system would search the graph for paths between the node for Person A and Person B. For example, Person A -> Document 5 -> Person B (both mentioned in Document 5). Or A -> Org Y -> B (A and B are both connected to the same organization through different docs). These connections can then be visualized (like a link chart). - The graph can also support queries such as “find all companies that are mentioned together with *Project Z* and have a contract signed after 2020”. Such a query would involve filtering nodes and relationships (we might use graph query language or a search on attributes, e.g. via a GraphQL layer or a Cypher query in Neo4j).

**Integration with LLM:** We can enhance the graph by having the LLM extract relationships that are not trivial co-mentions. For example, given a text, the LLM could parse that “Alice is the manager of Bob’s team” – a relationship that could be recorded (Person Alice MANAGES Person Bob). However, initially we focus on simpler derived relations (co-occurrence in documents, and known structural roles like author, signatory which we can infer from document structure or metadata).

**Data Storage and Tech:** If using Amazon Neptune, we can store the graph in a Neptune cluster (labelled-property graph or RDF depending on approach). If open-source, a Neo4j instance or an Azure CosmosDB (gremlin API) could be used on AWS via self-hosting. The graph DB will store entity attributes (like Name, type, maybe normalized form) and relationships with any relevant properties (like “mentioned in doc” might have a property of how many times or context). DynamoDB could also be used for a simple adjacency list approach, but a true graph DB is more convenient for complex traversal. For scale, Neptune can handle millions of nodes/edges easily, which should be sufficient given our corpus size (if we have say 100k documents with a few hundred entities each, that’s on the order of tens of millions of nodes/edges maximum).

**Updating the Graph:** The entity extraction and graph update happen in the document ingestion pipeline. Each new or updated document triggers re-extraction of entities and insertion of new relationships. We will ensure idempotency – e.g. if a document is reprocessed, we remove its old edges first then add new ones, or update if incremental. There may be a **graph builder microservice** that receives events like “Document X processed” and then handles updating the graph DB accordingly. This keeps graph logic separate from OCR and LLM logic.

**Justification:** The relationship graph adds a **new dimension of insight** that pure text Q&A might miss – it gives a **birds-eye view** of how information is connected. This is extremely useful for use cases like investigative analysis, where one might not ask a specific question upfront but rather explore connections (e.g. find all contracts that involve both Company A and Person B). By including it, Visionary AI goes beyond a standard document chatbot; it provides a knowledge discovery tool. Technically, building this on open-source NER and graph tech is feasible and cost-effective, and it leverages the structured data hidden in text. It also directly addresses the UI requirement of a Relationship Graph panel, ensuring that the backend can supply the data needed for that visualization.

## 5. User Interface (Frontend) Components and UX Integration

The front-end of Visionary AI will be a web-based application (responsive for desktop and possibly tablet use) that provides intuitive interfaces for different functionalities. The UI design is informed by the Visionary AI wireframe report, including three primary interface components: **(A) OCR & Text Panel**, **(B) Relationship Graph Viewer**, and **(C) Reporting Dashboard**. Each of these interfaces corresponds to underlying backend services, and here we describe their design and integration.

**A. OCR & Text Panel:** This is the main document view where users can upload or view a document (image/ PDF) and see the recognized text alongside it.
- The interface is split into two panes: on the left, the original document image or PDF page is displayed (with the ability to zoom, pan, and page through multi-page docs); on the right, the extracted text is shown in an editable text area or viewer. The text is laid out in a clear, linear format (preserving headings, paragraphs, lists as identified). - **Highlighting and Alignment:** When the user hovers over or selects text on the right pane, the corresponding region on the image is highlighted (and vice versa). For instance, if you click on a sentence in the text panel, the UI outlines the area on the image where that text came from. This is achieved using the coordinate metadata produced by the OCR service (bounding boxes for each text segment). It gives users confidence in the OCR and allows easy verification/correction. - **Editing and Feedback:** The text panel likely allows corrections – if the OCR mis-recognized a word, the user can edit it. Such edits could be fed back to improve the system (e.g. storing corrected text for future fine-tuning or at least for that session). However, heavy editing is not expected if accuracy is high; this feature is more for occasional fixes or annotations. - **Query & Command Bar:** The OCR panel includes a context-aware query interface. For example, a user viewing a document can type a question in a "Ask something about this document" box. This will trigger the backend to route the question plus the document context to the LLM module (possibly via retrieval if needed). The answer could appear as an overlay or below the text. This essentially brings the RAG Q&A capability directly into the document view. For example, if you have a contract open, you can ask "When does this contract expire?" and get an answer like "**It expires on 12 Dec 2025**" highlighted. - **Controls:** There will be buttons for actions like "Copy Text" (copy all extracted text to clipboard), "Download OCR Result" (perhaps download a text or Word file of the doc), and possibly "Translate" (if needed, though not explicitly required, but multilingual OCR could allow on-the-fly translation via the LLM). Another control could be "Run Analysis" which triggers the LLM to produce a summary or key insight list for that document, displayed in a side panel. - **Integration:** The front-end calls the backend `/ocr` API when a new file is uploaded. It shows a progress indicator while OCR runs. Once text is retrieved, it populates the text panel. If the user asks a question, the front-end sends a request to a `/query` API with the doc ID or content reference; the backend handles it (using retrieval + LLM) and returns the answer and any source highlights, which the UI then displays.

**B. Relationship Graph Viewer:** This interface provides an interactive visualization of the knowledge graph extracted from the documents.
- **Visualization:** Likely implemented using a graph visualization library (e.g. D3.js, Cytoscape, or Vis.js) to render nodes and edges. Nodes could be color-coded by type (person, organization, document, etc.), and edges might have different styles (dotted for "mentioned in", solid for "authored by", etc.). The wireframe might have shown a central node with others radiating out – a common way to display an ego network of an entity. - **Interaction:** Users can search for an entity via a search bar (autocomplete suggestions for names as they type). Upon selection, that entity becomes the focus of the graph. The UI will display that node in center with directly connected nodes around. For example, selecting a person might show all documents they're in and other people in those same documents. Users can click on a neighboring node to

pivot the graph to that focus, effectively navigating through connections (e.g. clicking on a document node could then show all entities in that document). - **Detail Panel:** When an entity node is selected (clicked), a side panel or tooltip can show details about it – e.g., for a Person: their name, maybe a list of documents they appear in (with links to open them in OCR panel), frequency of mention; for a Document node: its title, date, involved entities, etc. This provides context so users can interpret the graph. - **Filtering:** There will be controls to filter which types of relationships or entities are shown. If the graph is too crowded, users might toggle off some categories. For example, an option to "show only contracts and people" or "exclude dates and minor entities" to focus on key parts. - **Backend integration:** The graph UI communicates with an API (e.g. `/graph` endpoints). For example, to render the graph around an entity, the front-end calls something like `/graph/neighbors?entity=XYZ&depth=1` which returns the nodes and edges needed. Or a query like `/graph/shortest_path?entity1=Alice&entity2=Bob` to find connections. The backend's graph service (or an API layer in front of the graph DB) will handle these queries in real-time. We will ensure queries are efficient – possibly using precomputed indexes for common patterns. - **Use Cases:** The user stories for this could be: "As an analyst, I visually explore connections: I see that *Alice* is linked to *Project Zeus* and *Acme Corp* through Document X, which reveals she was the signing officer on a contract between Acme and Project Zeus." This insight might not be obvious without the graph view. - **Dynamic Updates:** If new documents are added while a user is active, the graph could update (perhaps via WebSocket or periodic refresh) to include new nodes/edges. In most cases, data is relatively static during analysis sessions, so this is not critical but nice to have.

**C. Reporting Dashboard:** This is an admin or power-user view that provides oversight of the system's operations, performance, and usage.
- **Metrics Displayed:** The dashboard will show key metrics such as: - Number of documents processed, possibly broken down by type or language (e.g. "500 documents total: 300 English, 150 Spanish, 50 Chinese"). - OCR accuracy metrics (if we have ground truth for some or use proxy measurements): e.g. average confidence, number of low-confidence extractions that required review. We might show "Effective OCR accuracy: 99.2% this month" which could be calculated from manual review corrections or from comparing LLM re-read of text vs OCR output for consistency. - Processing Performance: average OCR processing time per page, average LLM query latency, throughput metrics. Possibly a chart of response times over time or under different loads. - System Utilization: CPU/GPU usage statistics, memory usage, maybe cost estimation for the month (how many GPU hours used, etc.). If integrated with AWS Cost Explorer API, we might fetch and display cost metrics for relevant services (to help the engineering lead optimize cost). - **Uptime/Availability:** indicators if all services are up and healthy. E.g. if the OCR service or any model endpoint is down, an alert shows. Also track last deployment time, number of failures or errors in the last period. We can integrate CloudWatch alarms so that any anomaly surfaces on the dashboard (and presumably triggers notifications separately). - **User Activity:** If the system has multiple end-users, show usage stats like active users, number of queries made, etc. Possibly list of recent queries or tasks processed (with timings) – useful for auditing and understanding how the system is used. - **Compliance/ Logs:** For SOC2, maybe a section shows security events (like "no unauthorized access attempts" or "all data encrypted – OK"). Could also indicate if backups are recent (e.g. "Daily backup successful at 02:00 UTC"). This ties in processing integrity and availability by ensuring everything is running smoothly. - **Controls:** The dashboard might allow administrative actions: e.g. flush caches, trigger a re-index, update configurations (like modifying a threshold for low confidence or uploading a new model). These should be carefully secured (admin only). Possibly have a "Manage Users" if needed. - **UI Layout:** Likely a grid of panels with charts and numbers, using a modern dashboard framework or library (like Chart.js or Grafana integration). It should be clear and not overly cluttered – focusing on the health and performance indicators critical to the engineering lead or operations team. Each panel might have drill-down capability (click to see more

details or historical trends). - **Integration:** The frontend will call various backend endpoints to collect this data. Some data might come from a monitoring database or metrics service. For example, we might have a service aggregating logs into a timeseries DB (like InfluxDB or CloudWatch metrics). The dashboard API could query that for metrics like “documents_per_day” or “avg_latency”. Other info like user count could come from our application database or Cognito user pool. The design will favor read-only display; any control actions will map to specific privileged API calls (with confirmation).

Overall, the UI/UX is designed to be **user-friendly and informative**, allowing users with different needs to interact with the system: end-users get an easy way to get text and answers from images, analysts get a powerful graph explorer, and admins get oversight on system status. The frontend will be built likely with a modern JS framework (React or Vue) and will include appropriate internationalization (for UI labels, though the primary interface can be English since the users likely operate in English even if data is multilingual). It will also incorporate responsive design – the OCR panel and graph likely require a decent screen (desktop or large tablet) for full functionality, but key features (like taking a photo with a phone and getting text) could be exposed if needed via a simplified mobile view.

The **UX** ties together with the backend such that, for example, a user’s action in the UI triggers precisely the needed backend workflow: viewing a document triggers OCR (unless already done), asking a question triggers RAG+LLM, clicking an entity triggers graph queries, etc., in a seamless manner. Careful attention will be paid to loading states and feedback (so the user knows something is happening – e.g. a spinner while an LLM is thinking, with maybe intermediate “draft” output if available). Given the heavy-lifting is on backend, the UI will mostly orchestrate and display results, keeping the user informed.

## 6. AWS Infrastructure & Deployment Architecture

The deployment of Visionary AI on AWS will utilize a combination of managed services and self-managed instances to balance cost, performance, and control. Below is a breakdown of the infrastructure components and how they will be managed, all defined through Infrastructure-as-Code for reproducibility and consistency:

- **VPC and Networking:** We create a dedicated AWS Virtual Private Cloud (VPC) for Visionary AI, with isolated subnets:
- **Public Subnet:** contains only the Application Load Balancer (ALB) or API Gateway that accepts incoming requests from users. This ALB forwards traffic to the internal services. The public subnet also allows NAT Gateway egress if needed for updates.
- **Private Subnets:** for application servers, model instances, and databases. These have no direct internet access (for security); any required internet access for patches or model downloads goes through a NAT gateway or is done in build steps. We use multiple Availability Zones (AZs) for high availability – e.g., two private subnets in different AZs so that services can run in both and survive an AZ outage.

- **Security Groups:** Strict rules will be applied (e.g., the ALB SG allows HTTPS from internet, backend service SG allows traffic from ALB SG on specific ports only, DB SG only allows from app SG, etc.). This implements a **zero-trust network** internally: each service only communicates on necessary channels.

- **Compute Cluster (ECS/EKS):** The microservices (OCR service, LLM service, orchestrator, graph updater, etc.) will be containerized and deployed on either Amazon Elastic Container Service (ECS) or Kubernetes (EKS). We lean towards ECS Fargate for simplicity – it can run containers without managing servers. However, for GPU workloads, we might use ECS with GPU EC2 instances in an Auto Scaling Group (since Fargate GPU has limited availability). Alternatively, EKS (Kubernetes) gives flexibility with custom scheduling (especially if mixing CPU and GPU workloads). In either case:

- The OCR service container will be scheduled on a GPU node (with proper resource requests to ensure it has exclusive GPU access). We might use p2/p3 instances or newer g5 instances for cost. The number of replicas can scale out based on a metric like CPU/GPU utilization or queue length.
- The LLM service (with vLLM server) similarly runs on one or more GPU nodes. Depending on model size, we might start with 1 large GPU instance (e.g. 1×A10G) to serve all LLM requests, and then scale out to multiple as load increases (with a load balancer or a shard per model instance, the orchestrator can route to the one with least load).
- Other services like the Orchestrator (if separate), the API Gateway (if we run our own via something like Express or FastAPI, though likely we use AWS API Gateway), and any support services (embedding service, NER service) can run on **CPU-only** containers on Fargate or EC2. They will auto-scale based on CPU utilization or request count.
- We will utilize **AWS Auto Scaling** groups and ECS Service Auto Scaling policies to ensure enough capacity: for example, maintain 1 OCR task per GPU, min 1 max maybe 4, scale up if requests > X per minute.

- **Service Discovery:** ECS allows service discovery so that services can find each other by DNS names (or we use environment variables with hostnames/ports for internal calls). AWS Cloud Map can assign a DNS name to each service. This way, the orchestrator can call “ocr.service.cluster.local” to reach the OCR container etc.

- **Storage & Databases:**

- **Amazon S3:** used for storing input files and possibly outputs that need persistence (like the original uploaded images, and perhaps JSON results of OCR for archive). We will organize buckets for different data (one for raw uploads, one for processed outputs if needed). S3 is also used to store model artifacts (like if we have our custom model weights, they can be versioned in an S3 bucket from which the ECS tasks download on startup, unless baked into the container).
- **Vector DB:** If using a self-hosted solution like Qdrant, it might run as its own service on ECS or on an EC2 instance. Alternatively, use Amazon OpenSearch Serverless with vector capability for minimal maintenance – though open-source Qdrant might give better flexibility. Assume we use Qdrant on ECS with a volume (EBS) for persistence. We'll schedule it on a memory-optimized EC2 instance in the cluster, or run it on its own small instance. Regular snapshots of the index can be stored to S3 for backup.
- **Graph DB:** We can use Amazon Neptune, which is a managed graph database (taking away a lot of ops burden). Neptune can run in a cluster across AZs for HA. The cost is moderate, but since graph queries will not be extremely high-throughput, a smaller instance class could suffice. If Neptune, we set it up in the same VPC, and our app servers will communicate with it over its endpoints. If we choose to self-host Neo4j, we'd probably containerize it similarly with a persistent volume, but managed Neptune is preferable for reliability (and it's also already SOC2 compliant by AWS).

- **Relational/Metadata DB:** Possibly Amazon RDS (PostgreSQL) for anything requiring structured storage (user accounts, configurations, audit logs). If not heavy, we might even use DynamoDB for simple key-value metadata (like mapping doc IDs to S3 keys, or storing processing state flags). DynamoDB is serverless and very reliable (multi-region and backup features). In fact, the AWS blog solution used DynamoDB to track processing state [6]. We likely use DynamoDB tables for things like "Documents" (id, status, timestamp, attributes) and "Users" if any. Dynamo's advantage is no maintenance and auto-scaling throughput.

- **API Layer:** We have two choices – an AWS API Gateway (plus Lambda for some logic) or hosting an API service ourselves. Given we have to handle file uploads (which API Gateway can do via S3 pre-signed URLs typically) and possibly streaming outputs from LLM (for future, maybe not initial), a custom API service (e.g. a FastAPI app behind ALB) might be simpler. However, from a security and scalability viewpoint, API Gateway with Lambda integration has appeal (fully managed, and easy to do usage throttling, plus built-in auth integration).

- If using API Gateway + Lambda: We can implement lightweight Lambda functions for orchestrating requests. For instance, a Lambda for "Process Document" that, when invoked, stores the file to S3 (if not already there), triggers an OCR job (perhaps by enqueuing a message or calling ECS task), waits or polls for result, etc. But synchronous Lambdas have time limits (15 minutes max). Alternatively, this can be done asynchronously with Step Functions for long processes. For query/answer, a Lambda could take the query, call the vector DB, call the LLM service, and return the answer. This might add overhead vs direct client to backend calls.
- Alternatively, use the ALB to directly route to ECS services. For example, we set up an ALB with listeners for paths `/api/ocr`, `/api/query`, etc., each targeting the appropriate ECS service (via a service discovery or dynamic port mapping). The ALB does SSL termination, providing a single entry point. This approach can be more straightforward and allow WebSocket/streaming if needed (which API Gateway HTTP API also can do now).

- In either case, we will secure these endpoints. If internal use only, maybe simply restricting by VPN or IP could suffice; but likely we will have a proper auth (users log in to UI). Using Amazon Cognito for user auth is a good practice – the UI can authenticate users, obtain tokens, and API Gateway or our backend will verify those tokens (JWT). This aligns with Security principle of access control.

- **DevOps and CI/CD:** All infrastructure is defined via Terraform or CloudFormation templates (including VPC, ECS cluster, load balancer, DB instances, etc.). We will have a **CI/CD pipeline** (using AWS CodePipeline or GitHub Actions) that automates building and deployment:

- *Code Repositories:* We maintain code in a version control (e.g. GitHub/Bitbucket). The repository likely has separate folders for frontend and backend. The backend might be further segmented by services.
- *Build Process:* On each push (or PR merge) to main branch, the pipeline triggers. It will build Docker images for each service (OCR, LLM, orchestrator, etc.) using a CI runner. These images are pushed to **Amazon Elastic Container Registry (ECR)** with version tags.
- *Infrastructure Deployment:* We treat IaC changes similarly – if templates change, Terraform is run to apply changes (or CloudFormation stack is updated). Proper gating and review for infra changes will be enforced (since a mistake can be costly or cause downtime).

- *Deployment:* Once new images are in ECR, we update the ECS services. If using ECS, a new task definition is created and ECS deploys it (rolling replacement). We can configure a slow rollout and health checks (like the service only goes live if the new task passes a health endpoint test). For critical model services, we might do one at a time (so there's always one running instance while updating). If using EKS, we'd use ArgoCD or Flux for continuous delivery of Kubernetes manifests. In either case, CI/CD ensures consistent, repeatable deployments with minimal manual intervention.
- *Testing:* As part of CI, we include automated tests – unit tests for any logic, and perhaps integration tests that run a lightweight version of the system (maybe using a small model) to ensure the pipeline works end-to-end (e.g. test uploading a sample image and getting correct text out). This catches issues early and also contributes to **processing integrity** by verifying that changes don't break data handling.
- *Monitoring & Logging:* We deploy CloudWatch agents on instances or use AWS ECS logging to send container logs to CloudWatch Logs. This collects all application logs (OCR results, errors, etc.). We set up CloudWatch Alarms for certain conditions (e.g. OCR service error rate > X, or CPU > 90% for Y minutes) to alert the team (possibly via SNS->Email or PagerDuty). We also use AWS X-Ray or AWS Distro for OpenTelemetry to trace requests through the microservices, which helps in debugging performance issues. The dashboard will display some of these metrics as noted. For long-term log analysis (security, usage patterns), we might aggregate logs into an ELK (Elasticsearch) stack or use Amazon Athena to query logs stored on S3 (CloudWatch can export logs to S3 periodically).

- *Backup and DR:* S3 data is inherently backed by AWS. For DynamoDB, enable Point-in-Time Recovery. For Neptune/Databases, set automated snapshots daily. Also, we will likely output critical processed data (like OCR'd text, extracted fields) back to S3, so that even if a DB is lost, the raw outputs are retrievable and can be re-indexed. The Terraform scripts will allow recreation of infra in a different region if needed, using those backups, addressing disaster recovery scenarios.

- **Cost Management:** We use AWS Budgets and Cost Explorer to keep track of expenses. Since the project emphasizes cost-effectiveness, we will configure alerts if monthly cost projections exceed certain thresholds. Additionally, we might schedule certain expensive resources to shut down when not in use (for instance, dev/test environments only run 9-5, or the LLM GPU instances scale to zero at night if interactive use is zero). Our architecture with auto-scaling and on-demand endpoints (similar to the dynamic endpoint approach in an AWS blog [13] where SageMaker endpoints were spun up only when needed) ensures we pay mainly for what is used. We consider implementing a mechanism where the LLM GPU server could scale down when idle and quickly spin up on a new request (perhaps using AWS Lambda to trigger start). This is tricky for real-time but can be optimized for usage patterns (e.g. nighttime low usage → keep one small instance or none).

By leveraging AWS's robust services (which are already compliant and secure) and combining them with open-source components, we get the best of both: a **cloud-native, scalable infrastructure** and **flexible, high-performance AI components**. The entire architecture will be documented and diagrammed for the team, and since it's codified in IaC, setting up staging or dev environments is straightforward (we will likely have a dev environment with smaller instances for testing, separate from production).

## 7. Performance Engineering and CUDA-Level Optimizations

Achieving low latency and high throughput for Visionary AI requires going beyond default settings. We outline the performance optimization strategies, especially focusing on GPU/CUDA-level enhancements, used in this system:

- **Model Quantization and Pruning:** While considered “standard”, we will employ 8-bit or 4-bit quantization for the LLM where feasible (using libraries like BitsAndBytes for HuggingFace). This can dramatically reduce memory footprint (by ~4×) and slightly speed up computation due to reduced memory bandwidth, with minimal accuracy drop – important for cost since it may allow using a single GPU instead of multiple or a smaller GPU instance type. We will also consider pruning any unnecessary parts of models (for example, if a model has components not needed for inference, or extraneous tokens in vocab can be removed to slim it down).

- **Custom CUDA Kernels:** The team’s expertise allows analyzing profiling results to find bottlenecks. For instance, if we see that a certain operation (like layer normalization or a specific convolution in the OCR detector) is taking a disproportionate time, we can implement a fused kernel. Using tools like NVIDIA’s CUTLASS or OpenAI’s Triton, we can write a kernel that fuses multiple operations (e.g. combine a gemm, bias addition, and activation in one). Fewer kernel launches mean less overhead and better GPU utilization. In the job description notes, “manually tuned GPU kernels” is highlighted – so we will definitely attempt things like optimizing the OCR model’s input pipeline (maybe a custom resize and pad kernel) or the LLM’s sampling loop (maybe using device-side softmax with bias to speed generation). These are low-level but can yield a few milliseconds saved each, adding up.

- **Memory Optimization (Paged Attention / Streaming):** Large language models often waste memory by allocating huge buffers. We will use vLLM’s **PagedAttention** mechanism, which effectively manages the KV cache for the model’s self-attention more efficiently (no duplicate memory for unused sequence lengths) [5] . This allows us to serve longer prompts without linear slowdown and memory exhaustion. Additionally, by using **BFloat16/FP16** precision on tensor cores, we double throughput for matrix multiplies compared to FP32, at negligible loss of precision (we will test accuracy to ensure OCR text doesn’t get misinterpreted due to precision issues – unlikely since LLMs are trained in FP16 anyway).

- **Batching and Concurrent Execution:** The server will batch multiple requests when possible. For example, if two OCR requests come in at the same time, our OCR service could batch them into one forward pass if using a CNN-based recognizer that supports batch. Similarly, the LLM server vLLM will accumulate tokens from multiple sessions and generate in parallel. This **keeps the GPU busy** with larger effective batch sizes, improving throughput. We’ll tune the batch sizes to avoid latency spikes (there’s a trade-off: too large a batch might slow an individual query). We can configure maximum batch token count in vLLM to optimize for our GPU (as per vLLM docs recommending tuning `max_num_batched_tokens` [14] ).

- **Kernel Fusions and Libraries:** We will leverage libraries like **FlashAttention** (which is a faster implementation of attention that reduces memory access overhead by computing attention in chunks) – integrating it either via a library or ensuring our LLM uses it (some newer model implementations have it built-in). Also, if using PyTorch, we’ll use `torch.compile()` (in newer versions) or NVIDIA Apex for fused optimizers if we had training, but for inference, tools like

DeepSpeed-Inference can be used to automatically replace certain ops with faster kernels (like fused transformer block kernels). **TensorRT** is another powerful tool: we can convert the LLM to a TensorRT engine, which will optimize the model graph (fuse layers, optimize precision) specifically for the target GPU. Early tests show major speedups in LLM generation using TensorRT on NVIDIA GPUs (the NVIDIA blog shows >10k tokens/sec with GPT-2 sized models and first token latency ~100ms on H100 [15] ). While we might not reach that scale, even 2-3× speedup from TRT would be beneficial. We have to weigh the engineering effort (since TRT can be complex with very large models and dynamic sequence lengths). Possibly focus on optimizing the decoder part which is the runtime bottleneck.

- **Asynchronous I/O and Pipeline Parallelism:** We will ensure that while one batch is being processed on GPU, the CPU is free to handle other tasks or prepare next batch. For OCR, reading image from S3, decoding it, etc., will be done asynchronously so that the GPU sees a ready-to-go tensor as soon as it's free. We might use Python async or multi-threading in the service to achieve overlapping of CPU and GPU work. For LLM, if we ever host a model that spans GPUs (model parallel), we'll use efficient parallelism schemes (like tensor parallel with NCCL) and ensure the fastest interconnect (NVLink or EFA for multi-node) to reduce comm overhead.

- **Latency budgets and SLAs:** We will establish internal targets (e.g. OCR for a standard page should take <1 second on average; an LLM answer for a one-paragraph question should return in <2 seconds, etc.). We will test under realistic loads and use profiling tools. Where those targets aren't met, we iterate with optimizations. For example, if OCR is slow on very large images, we might downscale images above certain resolution (since 300 DPI might not be needed for text). Or for the LLM, if initial token latency is high, we might use techniques like **prefill** (computing some cache in advance) or simply smaller models for certain tasks where large ones aren't needed (e.g. use a 7B model for simple classifications with faster response).

- **Continuous Performance Testing:** Our CI/CD pipeline will include benchmarks that run on a staging environment to catch performance regressions. If a new version of a model is slower or a code change adds overhead, we'll detect it. We'll also monitor in production with timing logs – each request path will log how long it took in each stage (OCR, retrieval, LLM). These will feed into CloudWatch metrics so we can see trends. If, say, LLM latency is creeping up due to increased load, we might decide to scale out or further optimize.

In summary, by combining **algorithmic optimizations** (like batching, quantization) with **low-level tuning** (CUDA kernel fusion, memory management) and using specialized inference frameworks, Visionary AI's performance will be finely tuned. This ensures that despite using large models, the end-user experience remains snappy – aligning with the expectation that even a 50GB model should reply in under 3 seconds (as humorously noted in the job description). Our goal is to meet or exceed that, delivering most answers in near real-time. The focus on performance also means we can handle more load on the same hardware, directly supporting cost-effectiveness (doing more with less).

## 8. Security and SOC2 Compliance Measures

Security and compliance are woven through every part of Visionary AI's design. Below we detail how the system meets the **SOC 2 Trust Services Criteria**:

- **Security:** All data and resources are secured against unauthorized access.

- *Authentication & Authorization:* We will enforce user authentication (via Cognito or enterprise SSO) for accessing the UI and APIs. Role-based access control (RBAC) ensures users only access data they're permitted to (for example, if multi-tenant, each client's documents are isolated). Tokens and credentials are managed securely – no hard-coded secrets; use AWS Secrets Manager or Parameter Store for sensitive configs (DB passwords, API keys).
- *Encryption:* All network communication is over HTTPS/TLS (the ALB uses an SSL certificate). Within the VPC, we still use encryption for database connections (Neptune, RDS with TLS). Data at rest: S3 buckets have default encryption (AES-256 or KMS-managed keys). Any EBS volumes (for EC2/ECS tasks) will have encryption enabled. DynamoDB tables use encryption by default. We manage our own KMS keys for any particularly sensitive data so we can control rotation and access.
- *Network Security:* As described, strict VPC isolation is in place. No database or service is publicly reachable. We use security groups to allow only required traffic. The principle of least privilege extends to IAM roles too – each service container has an IAM role with minimal permissions (e.g. OCR service can read from S3 input bucket and write to output bucket, nothing more). The front-end S3 (if hosting static site) or CloudFront distribution only allows content access, nothing else.
- *Vulnerability Management:* We will perform regular vulnerability scans on our containers (using tools like Trivy or AWS ECR's scan). The OS and libraries will be kept updated – using base images that get security updates and quickly rolling out patches for critical CVEs. We'll also have a Web Application Firewall (AWS WAF) in front of the ALB to mitigate common web threats (SQL injection, XSS – though our API mainly handles structured inputs, it's good to have). For the AI-specific threats (like prompt injection etc.), we implement input validation and possibly prompt filtering to avoid malicious content injection (this is more of a functionality integrity concern).

- *Auditing:* CloudTrail logs for all AWS API actions are enabled, giving an audit log of any changes or accesses at the infrastructure level. Application-level auditing: we log user actions (e.g., user X accessed document Y at time Z, user A ran query Q). These logs are stored in a secure and tamper-evident way (e.g. append-only logs in CloudWatch with retention and archival). We will periodically review these for any suspicious activity.

- **Availability:** Ensuring the system is reliable and resilient.

- *Redundancy:* All critical components are at least **2× redundant** across availability zones – e.g., 2 OCR instances (min), 2 LLM instances, multi-AZ DB deployments. The ALB health-checks and automatically routes around failed instances. If an AZ goes down, ECS will launch tasks in the remaining AZ.
- *Auto-Recovery:* We use AWS auto-scaling and services that automatically replace failed nodes. For instance, if a container crashes, ECS will restart it. If an EC2 instance fails, auto-scaling group launches a new one. Neptune and RDS have automatic failover to standby in another AZ.
- *Backups:* As mentioned, we backup databases daily and retain snapshots. S3 versioning could be enabled on buckets to protect against accidental deletion. We also have an off-site backup plan: perhaps critical data (like processed text or important docs) could be periodically backed to a second AWS region or to tape (AWS Glacier) to recover from region-level disasters. This might be beyond SOC2 requirement, but is good practice.
- *Monitoring & Alerting:* We treat availability incidents seriously – CloudWatch alarms will notify the team if any critical service is down or if error rates spike. We'll integrate with on-call rotation so that issues (like "OCR service not responding") are addressed quickly.

- *Capacity Planning:* The system will be load-tested to ensure it can handle expected peak loads with some headroom. Auto-scaling policies will be set to add capacity before services become overwhelmed. And we'll periodically revisit capacity as usage grows (scaling out cluster size, etc.). This proactive approach prevents outages due to overload.

- **Processing Integrity:** Ensuring the system processes data accurately, completely, and reliably.

- *Input Validation:* The system will validate inputs – e.g., file uploads are checked that they are indeed images/PDF and not corrupted or containing malicious code (we might use an antivirus scanner Lambda on uploads to S3, a common pattern, to detect malware in images/PDFs). Text inputs (queries) are checked for format (to avoid injection or nonsense that could break processing).
- *Workflow Orchestration:* We use Step Functions or robust orchestration to enforce that each step of processing either completes successfully or the system retries/fails gracefully. For example, if OCR fails for a page, it will retry up to 3 times, and if still failing, mark the document as failed processing (and maybe notify an admin or flag in dashboard). We won't silently drop data.
- *Atomicity:* When updating databases (like adding to vector index and graph), we ensure either all or nothing to maintain consistency. E.g., use transactions or make operations idempotent and reversible.
- *Data Quality Checks:* We can implement sanity checks on OCR output (like if a page is detected but OCR came out empty text, that might be an error unless it's a blank page). Such anomalies can raise alerts. Similarly, for LLM outputs, if the model returns something that doesn't answer the question, we might detect that and either re-run with a different prompt or flag it. Another example: totals extraction – if numbers don't add up as expected, maybe a field was mis-read, so mark it.
- *Manual Verification for Critical Outputs:* As discussed, for certain high-value documents or fields, having a human verify low-confidence results ensures the final output is correct [2]. These processes (even if manual) are part of integrity controls. The system will log any human override or changes for audit.
- *Reconciliation:* The system can cross-check between components. For instance, after building the graph, we could verify that every document node in graph has a corresponding entry in the document database and vector index, ensuring no document was missed.

- *Error Handling:* Any processing errors are logged with details and surfaced in the dashboard. We'll avoid situations where an error in one part (like entity extraction) goes unnoticed and leads to partial data. Instead, if entity extraction fails for a doc, we might still allow OCR text to be available but with a warning, and try to re-run extraction later. This kind of approach ensures the system's outputs are as complete as possible.

- **Confidentiality:** Protecting sensitive information from unauthorized disclosure.

- *Data Access Controls:* As noted, we strictly control who (and what services) can access sensitive data. For example, if the documents contain confidential info, only authorized users can view them via the app – it won't be possible for someone to just guess a URL to an S3 object (we use pre-signed URLs with short expiry or proxy downloads through the app with permission check). Internally, support staff or engineers will not directly access raw data unless needed and authorized (we can enforce this via encryption and not sharing keys, as well as policy).
- *PII Handling:* If documents contain personal identifiable info (names, SSN, etc.), we might integrate the AWS Comprehend PII detection or an equivalent to automatically detect and possibly mask PII in outputs if appropriate. (For instance, if we ever create an external report, it might anonymize names

if policy dictates). This wasn't specifically asked, but in context of confidentiality and possibly privacy, it's worth designing.

- *Secure Development Practices:* All developers will follow secure coding guidelines (avoiding logging sensitive data, sanitizing inputs). Code reviews will include looking for potential leaks or weaknesses.
- *Compliance Standards:* While SOC2 is the focus, we note if any specific data regulation (like GDPR, HIPAA) is in play depending on data – the architecture has provisions (data localization if needed, deletion mechanisms to honor right-to-erasure, etc.). At minimum, we'll have a clear data retention policy – e.g., if a document is deleted by user, our system will remove it from all stores (S3, indexes, backups as feasible) within X days, and the dashboard can show data retention status.

Overall, these measures ensure **trust** in the system: users and stakeholders can be confident that the system is robust against threats, reliably processes their data, and keeps their information safe. We will prepare a SOC2 controls matrix mapping these technical measures to the standard criteria, and set up continuous monitoring (using something like AWS Security Hub, which can continuously check for misconfigurations against standards). Since the engineering lead has a background (per requirements) in AWS security and SOC2, we will also document and periodically review these controls to adapt to new threats or requirements.

## 9. User Stories and Use Cases

To illustrate how Visionary AI will be used in practice, below are representative user stories and use cases, derived from the requirements and the shared use-case discussions. These stories cover various perspectives (end-user, analyst, admin) and demonstrate the system's capabilities:

- **Use Case 1: Multilingual Document Transcription and Search**
  *As a global operations manager,* I want to digitize and search through contracts and documents in multiple languages, so that I can quickly find relevant information without manually reading each one.
  **Scenario:** I upload a batch of documents including some in English, Spanish, and Chinese. Visionary AI's OCR engine transcribes each document with ~99% accuracy (even the Chinese ones with complex characters). Later, I type a query "contrato vencimiento 2024" (Spanish for "contract expiring 2024"). The system semantically searches all documents and, within seconds, returns a list of contracts (English and Spanish) that have expiration dates in 2024 [3]. I click on a Spanish contract result and see the text and the image side by side; the term "vencimiento 2024" is highlighted. I can copy the extracted text to share with colleagues. This saves me from reading through piles of bilingual paperwork.

- **Use Case 2: AI-Powered Document Q&A**
  *As a legal analyst,* I want to ask questions to my document set and get accurate answers with references, so that I can quickly gather insights for a case.
  **Scenario:** I have a collection of scanned agreements and letters loaded into Visionary AI. Instead of reading each, I ask the system, "Which agreements involve a company named *Acme Corp* and what are the indemnity clauses in those agreements?" The system uses the relationship graph or search to find all documents where *Acme Corp* is mentioned, then specifically retrieves paragraphs about "indemnity" from those [3]. The LLM composes an answer: *"Acme Corp appears in 3 agreements. In Agreement A (2021) Section 5: 'Indemnity: Acme Corp shall be indemnified…'* [16] *; in Agreement B (2022) Section 8: 'Acme Corp agrees to indemnify the other party against…'."* Each snippet is cited with document name and section. I can click those citations to open the source document in the OCR

panel at the relevant section. This dramatically accelerates my research process by letting me query rather than read.

- **Use Case 3: Relationship Graph – Investigative Analysis**
  *As an investigative journalist,* I want to uncover connections between individuals and entities across leaked documents, so that I can identify hidden relationships or conflicts of interest.
  **Scenario:** I load a set of scanned letters and reports into Visionary AI. Using the Relationship Graph interface, I search for a particular official's name – *John Doe*. The graph view shows *John Doe* connected to several Document nodes and also to an Organization node *XYZ Corp*. I see that *John Doe* authored two letters (edges "AUTHORED") and is mentioned in three reports alongside *XYZ Corp*. I click on *XYZ Corp*, seeing it's linked to *Project Alpha* in another document. This graph exploration reveals that John Doe was involved in Project Alpha via XYZ Corp – a link I might have missed with linear reading. I then use that knowledge to ask the LLM: "Summarize John Doe's involvement with Project Alpha." The system pulls the relevant pieces and the LLM generates a concise summary from those various documents. This use case shows the power of combining a knowledge graph with search and LLM to perform investigative analysis.

- **Use Case 4: Automated Report Generation**
  *As a financial auditor,* I want the system to extract key financial figures from invoices and generate a report, so that I save time on data entry and analysis.
  **Scenario:** I provide a folder of scanned invoices (mixed English and Japanese). Visionary AI processes each with OCR, capturing structured fields: date, invoice number, total amount, vendor name (some via layout understanding or a prompt to the LLM to parse the invoice format). The Reporting Dashboard has a section where I can download a summary of all processed invoices. I click "Generate Invoice Report" – the system uses the extracted structured data to compile a CSV or PDF: a table of invoice numbers, dates, totals, sorted by date, with sums computed. The LLM may be used here to double-check any ambiguous entries (e.g., if an amount is unclear it flags it). The output shows totals per vendor and an analysis "Out of 50 invoices, 5 were past due." This automation reduces manual data entry errors and highlights important info for the audit.

- **Use Case 5: Real-Time Field Validation and Low-Latency Response**
  *As a front-line support agent,* I want to use the Visionary AI system to quickly pull information from a customer's submitted document while on a call, so that I can give answers immediately.
  **Scenario:** A customer calls and references a policy document (which they upload via our portal to Visionary AI). As I receive it, the OCR runs within a second or two (thanks to optimized inference). On my screen (which integrates Visionary AI), I see the text extracted. The customer asks a question about a specific clause. I type the question, and within about a second, the answer is highlighted: e.g. *"Clause 4.2 states the warranty period is 2 years from purchase."* The latency is low enough that there's no awkward hold time on the call. The CUDA-level optimizations (batched inference, quantized model) made this possible – the system served the answer in ~500ms. This user story highlights the requirement for sub-second responses in interactive settings, demonstrating the system's performance focus (users won't even realize an advanced AI and RAG just worked behind the scenes in that short time).

- **Use Case 6: Administrative Oversight and Compliance**
  *As the Principal AI Engineering Lead (system owner),* I want to ensure the system is running smoothly and securely, so that we meet our uptime SLAs and compliance obligations.

**Scenario:** I log into the Reporting Dashboard daily. I see a green status for all services (uptime 100% in last 24h). The dashboard shows that yesterday 200 documents were processed with an average OCR time of 0.8s/page and average LLM query latency of 1.5s – both within our targets. One metric shows "99.1% OCR accuracy this week" which I can drill down – it appears a batch of low-quality scans dropped the average, but our human-in-loop corrected those, achieving effective 100% accuracy post-correction. I also check the security logs panel: no unauthorized access attempts, and all data encryption keys are rotated as scheduled. An alert reminds me that a software dependency has a known vulnerability – our pipeline already built a patch which is ready to deploy. Satisfied, I approve the deployment from the dashboard's CI/CD section, which triggers the update. This story emphasizes how the admin can use the system's tools to keep everything in check, demonstrating built-in compliance (audit logs, metrics) and maintainability.

These user stories show Visionary AI in action – from everyday document queries to complex multi-document analysis, all the way to system maintenance. They illustrate the system's versatility: handling multiple languages, providing fast interactive answers, visualizing data relationships, and producing structured reports. Each scenario is enabled by the combination of open-source OCR + LLM technology, a robust AWS deployment, and thoughtful integration of UX and backend logic. They also justify many design choices: e.g., need for multilingual support, importance of the graph for investigative scenarios, need for high accuracy (especially Use Case 2 and 3 where trust in info is paramount), and need for optimization for real-time use (Use Case 5).

## Justification of Architectural and Model Choices

Finally, we summarize why the chosen architecture and components are the optimal solution, addressing performance and cost-effectiveness:

- **Open-Source Models (OCR & LLM):** By using open-source, fine-tuned models (like Qwen-VL, etc.), we avoid costly vendor APIs and retain full control over deployment. As noted, open models have nearly closed the gap with proprietary ones [1] – for instance, our chosen OCR model is on par with Google/Azure OCR on printed text, and even if proprietary might score ~1-2% higher on some metric, our workflow (with error correction and human review) nullifies that difference. The cost savings are significant: e.g., processing 1 million pages with our model might cost on the order of <$200 in GPU time [4], whereas an API at $0.001-$0.005 per page would be $1,000-$5,000. Over time, this is huge savings for the organization, justifying the initial investment in setup. Moreover, having models in-house means we can **optimize them deeply** (quantize, fuse kernels), which we cannot do with a black-box API. It also avoids sending sensitive data to third-party services, aligning with confidentiality needs.

- **Multilingual & Multi-task Capability:** Choosing models and designing pipelines with multilingual support built-in saves cost of deploying separate solutions for each language or task. One robust model (or a small set) handles all five required languages, benefiting from economies of scale (maintaining one system vs many). The use of a single unified system also simplifies user experience – users use the same interface regardless of language. The architecture being modular but integrated means features like search, OCR, and QA work consistently across languages (justifying initial complexity to support Unicode, etc.). This choice is cost-effective in development and maintenance, even if the model is larger, because we don't have to duplicate infrastructure per language.

- **RAG Architecture:** Retrieval-Augmented Generation was a purposeful choice to achieve accuracy and depth of knowledge without needing a prohibitively large LLM. By using a vector database and document index, our system can handle far more information than a standalone LLM with fixed knowledge. This means we did not need to choose a gigantic model (like 65B parameters or a GPT-4 via API) to get strong performance – a mid-sized model with RAG suffices, which is far cheaper to run (for example, a 7-13B model on one GPU vs requiring multiple GPUs or an expensive API for a 175B model). RAG ensures that answers are grounded in actual data [3], improving reliability (which means less manual double-checking – saving user time). In terms of performance, RAG adds some overhead (vector search), but we deemed it worth it for accuracy. The vector search is very fast (ms-level), so the impact on latency is minimal compared to the LLM's time. And we optimize by embedding documents only once during ingestion, which is efficient.

- **AWS Cloud & Infrastructure Choices:** AWS was chosen as the deployment platform because of its maturity, range of services, and compliance certifications. Using AWS allows us to design a solution that is *scalable* (from day one we can go from one machine to a cluster easily with ECS) and *highly available* (multi-AZ out of the box). It also eases **SOC2 compliance** since AWS provides many controls (encryption, auditing) and is itself compliant – we just inherit and implement best practices. From a cost perspective, AWS offers many tools to optimize spend: e.g., spot instances for non-critical workloads, rightsizing recommendations, etc. We selected specific services like Lambda/ECS Fargate to avoid maintaining servers – this saves engineer effort (translating to cost savings) and often money (no paying for idle capacity). The use of Infrastructure-as-Code means we can iterate architecture quickly and deploy to different environments without extra cost (no big manual setup). We also leverage free tiers and usage-based pricing: e.g., DynamoDB on-demand for metadata might cost only pennies until we scale up usage. All these considered, AWS is cost-effective when engineered properly – we designed for auto-scaling and minimal idle resource waste.

- **DevOps and Automation:** By investing in CI/CD and infrastructure automation, we minimize the maintenance cost and risk of human error. This is an upfront effort that pays off by enabling small team to manage what could be a complex system in production. Automated testing and deployment mean we catch issues early (cheaper to fix) and can do frequent updates (ensuring we can push performance improvements or security patches quickly). This agility is valuable and cost-saving over time (less downtime, less firefighting). Additionally, the logging and monitoring setup means we spend less time diagnosing issues – alerts pinpoint them and we can resolve them faster, reducing potential downtime costs or SLA penalties.

- **CUDA-Level Optimizations:** The decision to include deep optimizations was driven by the need for low latency, but it also yields cost benefits. Faster inference means we can handle more requests per machine, which means we need fewer machines (or instances) to serve a given load. For example, if naive deployment allowed 5 QPS (queries per second) on an LLM instance and our optimized deployment allows 15 QPS on the same instance, we've effectively tripled capacity at no extra cost. That's huge for cost-effectiveness, especially as usage grows. Similarly, optimizing OCR to run e.g. 50 pages/min instead of 20 pages/min on one GPU means maybe we don't need to buy another GPU. The pursuit of sub-100ms latency also aligns with an **excellent user experience** (snappy responses) which is hard to quantify in cost but certainly correlates with user satisfaction and productivity – a key project success factor.

- **Accuracy vs. Cost Trade-offs:** We have consciously made choices that balance accuracy and cost. For instance, using a slightly smaller model + human review on edge cases, instead of a giant model that is 99.9% accurate but extremely expensive to run. The example from industry: a tiny model plus manual checks achieved 99.6% accuracy saving $250k vs a large model [2]. We mirror that philosophy – e.g., our OCR model might not be perfect on every handwritten scribble, but rather than use an extremely complex solution for diminishing returns, we cover it with manual verification for the few cases. This approach is justified by overall ROI: we get to 99%+ quality with maybe 1/10th the cost of an all-AI perfect system. It's important for stakeholders to see that we are meeting the requirement in a **pragmatic** way. Similarly, we considered complexity: every component we added (graph, etc.) was justified by a significant value in use cases. If something was overkill for little gain, we left it out to keep the system lean (both to reduce development overhead and runtime cost).

- **Scalability and Future-Proofing:** The modular architecture means we can upgrade parts without large refactoring. If a new OCR model emerges that is twice as fast, we can swap the OCR container with minimal changes. If our usage spikes, we can scale horizontally easily because stateless services are behind a load balancer and stateful ones are on managed scalable platforms. This flexibility justifies itself the first time we need to handle an unexpected load or integrate a new capability – we won't need a costly re-engineering. It's also easier to maintain (which translates to lower cost of ownership) because each team member or sub-team can focus on one component (fewer cross-coupling issues).

- **Compliance and Trust:** By building in SOC2 principles, we avoid costly retrofitting later (which can happen if security is an afterthought). For instance, if we neglected encryption and later had to add it, it could mean re-processing all data, which is expensive. Instead, doing it from the start means no compliance roadblocks (which, if not met, could even lead to lost business). It's an insurance on the project's viability in enterprise contexts. Additionally, demonstrating strong compliance features (like audit trails, access controls) early can shorten security review cycles with potential clients – speeding up deployments and reducing time (and cost) in legal/security assessments.

In conclusion, the architecture and model choices for Visionary AI were carefully made to maximize performance and accuracy while controlling costs and complexity. Each major element has a clear purpose and adds significant value for either the end-user or the maintainers. By using open-source tech on a robust cloud foundation, we get the benefits of innovation and community support, and by injecting our team's expertise in AI engineering (CUDA optimization, distributed systems, security), we create a solution that is not just theoretically sound but **practically excellent** in a production setting. This spec provides the blueprint for implementing Visionary AI in a way that meets its ambitious goals and is maintainable and justifiable to all stakeholders (engineering, product, compliance, and users alike).

---

[1] [2] [7] [9] [10] [11] The Definitive Guide to OCR Accuracy: Benchmarks and Best Practices for 2025 | by Sanjeev Bora | Apr, 2025 | Medium
https://medium.com/@sanjeeva.bora/the-definitive-guide-to-ocr-accuracy-benchmarks-and-best-practices-for-2025-8116609655da

[3] Guide to Multimodal RAG for Images and Text (in 2025) | by Ryan Siegler | KX Systems | Medium
https://medium.com/kx-systems/guide-to-multimodal-rag-for-images-and-text-10dab36e3117

[4] olmocr.allenai.org
https://olmocr.allenai.org/papers/olmocr.pdf

[5] Meet vLLM: For faster, more efficient LLM inference and serving
https://www.redhat.com/en/blog/meet-vllm-faster-more-efficient-llm-inference-and-serving

[6] [12] [13] Build an AI-powered document processing platform with open source NER model and LLM on Amazon SageMaker | AWS Machine Learning Blog
https://aws.amazon.com/blogs/machine-learning/build-an-ai-powered-document-processing-platform-with-open-source-ner-model-and-llm-on-amazon-sagemaker/

[8] An OCR Pipeline for Machine Learning: the Good, the Bad, and the Ugly | by Hypotenuse Labs | Medium
https://hypotenuselabs.medium.com/an-ocr-pipeline-for-machine-learning-the-good-the-bad-and-the-ugly-9ae751314ce6

[14] Performance and Tuning - vLLM
https://docs.vllm.ai/en/v0.6.0/models/performance.html

[15] H100 has 4.6x A100 Performance in TensorRT-LLM, achieving ...
https://nvidia.github.io/TensorRT-LLM/blogs/H100vsA100.html

[16] Multimodal RAG with Vision: From Experimentation to Implementation - ISE Developer Blog
https://devblogs.microsoft.com/ise/multimodal-rag-with-vision/