# AI Document System Deployment Plan

[Project Document: AI Document Processing System - Environment Architecture, Hardware Sizing, and Implementation Strategy](#)

[Executive Summary](#)

[1. Introduction](#)

[1.1. Purpose and Scope of the AI Document Processing System](#)

[1.2. Document Purpose and Audience](#)

[2. System Architecture Overview](#)

[2.1. Core Components and Their Roles](#)

[2.2. Foundational Architectural Principles](#)

[3. Environment-Specific Architectures](#)

[3.1. Development Environment Architecture (Single-Node, Docker-focused)](#)

[3.2. User Acceptance Testing (UAT) Environment Architecture (Multi-Node, Scalability Testing)](#)

[3.3. Production Environment Architecture (Distributed, High Availability, Disaster Recovery)](#)

[4. Hardware Sizing and Resource Allocation](#)

[4.1. Sizing Methodology and Key Assumptions](#)

[4.2. Development Environment Hardware Sizing](#)

[4.3. User Acceptance Testing (UAT) Environment Hardware Sizing](#)

[4.4. Production Environment Hardware Sizing](#)

[5. Implementation Strategy](#)

[5.1. Phased Deployment Approach](#)

[5.2. Continuous Integration/Continuous Deployment (CI/CD) Pipeline Integration](#)

[5.3. Data Migration and Seeding Considerations](#)

[6. Operational Considerations](#)

[6.1. Monitoring and Centralized Logging Strategy](#)

[6.2. Security Hardening Across Environments](#)

[6.3. Backup and Disaster Recovery Strategy](#)

[6.4. Performance Optimization Guidelines](#)

[7. Conclusion and Recommendations](#)

[Works cited](#)

# Project Document: AI Document Processing System - Environment Architecture, Hardware Sizing, and Implementation Strategy

### Executive Summary

This report details the project document, implementation strategy, hardware sizing, and architectural design for a complete AI document processing system across Development, User Acceptance Testing (UAT), and Production environments. Leveraging a modular architecture with components like Ollama (LLM), PostgreSQL, Kafka, MinIO, Redis, Tika, FastAPI, and Streamlit, the system is designed for high throughput, scalability, and robust operation. The architecture evolves from a localized development setup to distributed, highly available clusters in production, with tailored hardware specifications and comprehensive operational strategies for security, monitoring, and disaster recovery.

### 1. Introduction

#### 1.1. Purpose and Scope of the AI Document Processing System

The AI Document Processing System is engineered to automate the ingestion, extraction, processing, and analysis of various document formats utilizing advanced Artificial Intelligence and Machine Learning capabilities. Its fundamental purpose is to significantly enhance operational efficiency in handling extensive document volumes by providing functionalities such as precise text extraction, AI-powered summarization, and interactive AI-driven conversational interfaces. The system is designed to deliver a resilient and scalable solution for organizations requiring sophisticated document intelligence.

The system incorporates the following core components, each contributing to its comprehensive functionality:

* **LLM Runtime**: Ollama is utilized for local language model inference, enabling on-premise AI processing.1
* **Database**: PostgreSQL serves as the primary relational database for structured data storage, including document metadata and processing results.1
* **Message Broker**: Apache Kafka facilitates high-throughput event streaming, ensuring asynchronous communication and scalable data pipelines.1
* **Document Processing**: Apache Tika and Unstructured.io are employed for robust document content and metadata extraction from diverse file types.1
* **Object Storage**: MinIO provides S3-compatible object storage for efficient and scalable management of raw and processed document files.1
* **Caching**: Redis is integrated for performance optimization through in-memory data caching, reducing latency and improving responsiveness.1
* **Web Server**: Nginx functions as a reverse proxy, managing incoming web traffic and routing requests to appropriate backend services.1
* **Application Framework**: FastAPI and Streamlit are used for developing the high-performance backend API and interactive web interfaces, respectively.1
* **Containerization**: Docker is leveraged for packaging and orchestrating all services, ensuring consistent deployment across different environments.1

The system's core functionality encompasses document upload, AI-powered summarization and key information extraction, interactive AI chat, and comprehensive system health monitoring.1 This document outlines the architectural design, hardware sizing, and implementation strategy, detailing the evolution from a single-node development setup to an enterprise-grade distributed system.

#### 1.2. Document Purpose and Audience

This document serves as a comprehensive technical guide for technical leads, IT directors, system architects, and implementation teams responsible for the deployment and ongoing management of the AI Document Processing System. It provides detailed architectural blueprints, specific hardware specifications, and a strategic implementation roadmap to ensure a successful, scalable, and secure deployment across various lifecycle environments.

### 2. System Architecture Overview

#### 2.1. Core Components and Their Roles

The AI Document Processing System is composed of several interdependent services, each fulfilling a distinct and critical role within the overall data processing and AI workflow:

* **Ollama (LLM Runtime):** This component is central to the system's AI capabilities, enabling local inference of Large Language Models such as Llama3.1, CodeLlama2, Mistral, and LLaVA. It powers AI-driven summarization, key information extraction, and interactive chat functionalities.1
* **PostgreSQL (Database):** As the primary relational database, PostgreSQL is responsible for the persistent storage of structured data. This includes metadata associated with processed documents, the results of AI analyses, records of chat sessions, and comprehensive system logs.1
* **Apache Kafka (Message Broker):** Kafka provides a high-throughput, fault-tolerant, and scalable platform for real-time event streaming. It is strategically utilized for asynchronous communication between various services, particularly for queuing document processing tasks, thereby decoupling components and enhancing system responsiveness.1
* **Apache Tika (Document Processing):** This content analysis toolkit is crucial for detecting and extracting metadata and textual content from a wide array of document formats, including PDFs, DOCX files, and plain text. It integrates with Tesseract OCR to handle text extraction from image-based documents.1
* **MinIO (Object Storage):** MinIO serves as an S3-compatible object storage server, purpose-built for storing large binary files such as raw uploaded documents and their processed versions. Its design ensures high performance and scalability for efficient file management.1
* **Redis (Caching):** Redis, an in-memory data structure store, is employed for performance optimization. It caches frequently accessed data and transient metadata, significantly reducing load on the primary database and improving overall system response times.1
* **FastAPI (Backend Application Framework):** This modern, high-performance Python web framework forms the backbone of the system's API. It manages document uploads, orchestrates the initiation of processing workflows, and handles all interactions with the AI components.1
* **Streamlit (Frontend Application Framework):** Streamlit is a Python library used to create the interactive web application, serving as the user-facing interface. It provides functionalities for document upload, displaying processing status, and facilitating the AI chat experience.1
* **Nginx (Web Server/Reverse Proxy):** Nginx acts as the system's primary entry point for all web traffic. It functions as a reverse proxy, efficiently routing incoming requests to the appropriate backend services (FastAPI, Streamlit, MinIO, Tika), and is configured for SSL/TLS termination to secure external communications.1
* **Docker (Containerization):** Docker is fundamental to packaging and orchestrating all system services into isolated containers. This approach ensures environmental consistency across different deployment stages and simplifies both initial setup and ongoing management.1

#### 2.2. Foundational Architectural Principles

The system's design adheres to several foundational architectural principles, ensuring its robustness, efficiency, and long-term viability within an enterprise context:

* **Scalability**: The architecture is inherently designed for horizontal scalability. This means that system throughput and capacity can be increased by simply adding more instances of stateless services such as FastAPI, Streamlit, Tika, and Ollama. Additionally, clustered data stores like PostgreSQL, Kafka, MinIO, and Redis are configured to expand their capacity and performance by incorporating additional nodes.3
* **High Availability (HA)**: Critical components are designed with built-in redundancy and automated failover mechanisms. This strategy aims to minimize downtime and ensure continuous operational capability even in the event of individual component failures. This encompasses clustering, data replication, and intelligent load balancing across service instances.3
* **Security**: Security is a multi-layered consideration embedded throughout the system's design. This includes robust network isolation, stringent firewall rules, comprehensive data encryption (both in transit and at rest), strict access control policies, and a commitment to regular security audits.1
* **Maintainability**: The system promotes modularity, a clear separation of concerns, and extensive use of containerization. These practices simplify deployment processes, streamline updates, facilitate troubleshooting, and ease ongoing operational management. Comprehensive logging and monitoring capabilities are integral to providing essential operational visibility.1

The architectural design of this system, particularly the selection of open-source components with inherent distributed capabilities, directly reinforces the principles of scalability and high availability. Components such as Apache Kafka, MinIO, Redis Cluster, and PostgreSQL with replication are naturally suited for horizontal scaling and maintaining continuous operation in the face of failures.3 This contrasts with a simple single-node deployment, requiring a shift towards more complex cluster management and distributed system considerations.

However, a notable architectural consideration arises with Ollama, the Large Language Model (LLM) runtime. While the other components are inherently designed for distributed environments, Ollama's primary design focuses on local language models and running LLMs locally.1 This characteristic presents a unique challenge for achieving production-grade distributed LLM inference. A single Ollama instance, while suitable for development, would become a significant bottleneck and a single point of failure in a production environment demanding high concurrency and availability. To overcome this, specific strategies become necessary. These include deploying multiple Ollama instances to serve concurrent users, which, while effective, requires substantial memory resources as each instance loads its own copy of the model.17 Alternatively, specialized distributed inference frameworks like vLLM can be employed, designed to break down massive models and spread them across multiple GPUs or entire machines for efficient processing.22 Ollama itself has made strides in this direction with enhanced GPU selection algorithms and support for concurrent requests.17 This highlights a critical design decision point where a component's inherent local-first design necessitates tailored architectural patterns to align with the overall system's enterprise-grade requirements.

Furthermore, while all components contribute to the overall hardware footprint, the Large Language Model (LLM) component, powered by Ollama, emerges as the primary driver of hardware cost and complexity in a production environment. This is particularly true concerning its demanding GPU memory (VRAM) and processing power requirements.23 The selection of the LLM model size (e.g., 7B, 13B, 70B parameters) and the chosen quantization level (e.g., INT4, FP8, FP16, FP32) directly dictates the necessary GPU memory.24 For instance, running a Llama 3.1 model with 70 billion parameters at 16-bit quantization (FP16) would require an estimated 168 GB of VRAM, plus additional memory for the KV cache, which can be even larger than the model itself.24 Such requirements quickly escalate hardware needs and associated costs, demanding high-end GPUs like NVIDIA A100s or RTX 4090s, often in multi-GPU server configurations.23 This makes careful model selection and optimization, including exploring different quantization levels, crucial for managing the overall infrastructure investment and ensuring the feasibility of the deployment.

### 3. Environment-Specific Architectures

#### 3.1. Development Environment Architecture (Single-Node, Docker-focused)

* **Objective:** The Development environment is meticulously designed for rapid iteration, ease of setup, and isolated development workflows. Its primary objective is to maximize developer productivity by minimizing environmental complexities and accelerating feedback loops.
* **Architecture Overview:** This environment is typically a single-node setup, where all system components are configured to run on a single developer workstation or a dedicated virtual machine. The recommended and most efficient approach is to extensively leverage Docker and Docker Compose for containerization.1 This strategy ensures that each service operates within its own isolated container, simplifying dependency management and guaranteeing environmental consistency across different development machines.
  + **Containerization:** PostgreSQL, Redis, Kafka (with Zookeeper), Apache Tika, and MinIO are all deployed as individual Docker containers. Their orchestration is managed through a docker-compose.yml file, which facilitates quick spin-up, graceful shutdown, and isolated operation of each service.1
  + **Local Bindings:** All services are configured to bind to localhost. For example, the FastAPI backend listens on localhost:8000, the Streamlit frontend on localhost:8501, and Ollama for LLM inference on localhost:11434.1 This simplifies local access and testing.
  + **Component Configuration:** Components utilize default or minimal configurations as detailed in the initial installation guide.1 This approach prioritizes functional readiness over performance tuning for development purposes.
  + **LLM Integration:** Ollama, the local LLM runtime, can run either as a Docker container or natively on the host system. It pulls and manages LLM models locally, utilizing available CPU resources or a single GPU if the developer's machine is equipped with one.1
  + **Nginx (Optional):** While Nginx can be containerized for development 1, it is often optional in this environment, as developers can directly access individual services on their respective ports.
* **Key Characteristics:**
  + **Simplicity:** Features minimal setup complexity, allowing developers to quickly onboard and begin work.
  + **Isolation:** Docker containers prevent conflicts between different services and avoid interference with the host operating system.
  + **Reproducibility:** The docker-compose.yml file acts as a blueprint, ensuring that the development environment is consistent and reproducible across all developer machines, eliminating "it works on my machine" issues.
  + **Resource Efficiency:** The architecture is designed to operate efficiently on standard developer workstation hardware.

The "optional" Docker Compose setup described in the installation guide 1 is, in practice, a critical best practice for any team-based development. While the primary installation guide focuses on native systemd services, which can lead to variations in system dependencies, library versions, and configuration paths across different developer machines, Docker Compose directly addresses these challenges. By encapsulating all service dependencies and their configurations within isolated containers, Docker Compose guarantees that every developer operates within the exact same environment. This standardization significantly reduces setup time, minimizes debugging efforts related to environmental inconsistencies, and ultimately accelerates developer onboarding. The ability to quickly spin up and tear down a consistent environment with a single command (

docker-compose up/down) directly contributes to developer productivity and reduces integration issues later in the development cycle. Therefore, for effective team collaboration and streamlined development, the Docker Compose setup should be considered essential rather than merely optional.

#### 3.2. User Acceptance Testing (UAT) Environment Architecture (Multi-Node, Scalability Testing)

* **Objective:** The UAT environment serves as a crucial pre-production staging area. Its objective is to enable end-users and stakeholders to rigorously validate the system's functionality, performance, and user experience under conditions that closely mimic the production environment. This phase aims to identify and rectify issues related to integration, performance bottlenecks, and complex user workflows before the system is deployed to live production.
* **Architecture Overview:** The UAT environment transitions to a multi-node setup, typically comprising a small cluster of virtual machines or dedicated servers. This scaled-down version of the production architecture is designed to test distributed component interactions, basic load balancing, and preliminary performance characteristics.
  + **Distributed Components:** Key services such as PostgreSQL, Apache Kafka, MinIO, and Redis are deployed across multiple nodes. While not necessarily configured with the full high-availability features of production, this multi-node deployment allows for testing data consistency and failover capabilities. For instance, PostgreSQL might be set up with a primary-standby replication 19, Kafka as a multi-broker cluster (e.g., 3 brokers) 3, MinIO as a multi-node distributed deployment 6, and Redis either with Sentinel for basic HA or a small cluster.8
  + **Network Configuration:** Services communicate over a dedicated internal network, utilizing hostnames or internal IP addresses, thereby moving away from the localhost bindings prevalent in the development environment.
  + **LLM Integration:** Multiple Ollama instances (e.g., 2-4 instances) are deployed on separate GPU-enabled nodes. These instances are fronted by a load balancer to test LLM inference concurrency and performance under a simulated load.2
  + **Document Processing:** Multiple Apache Tika server instances (e.g., 2-4 instances), potentially containerized, are deployed to handle concurrent document extraction requests effectively.4
  + **Application Deployment:** Multiple instances of both the FastAPI backend and Streamlit frontend applications are deployed behind Nginx, which is configured for basic load balancing (e.g., round-robin or least-connected methods).10
  + **Reverse Proxy:** Nginx is configured as a central reverse proxy and load balancer for the FastAPI and Streamlit applications, incorporating basic health checks to ensure service availability.1
  + **Monitoring & Logging:** Initial monitoring scripts 1 and basic centralized logging solutions 26 are implemented to gather preliminary performance metrics and identify any emerging issues.
* **Key Characteristics:**
  + **Production Mimicry:** The architecture closely resembles the production environment in terms of its distributed nature and component interactions, albeit at a reduced scale.
  + **Performance Baseline:** This environment is critical for establishing initial performance benchmarks and identifying early bottlenecks within the distributed system.
  + **User Validation:** It provides a stable and representative environment for end-users to validate the system's functionality, usability, and adherence to business requirements.
  + **Integration Testing:** Ensures that all distributed components interact seamlessly and correctly as a cohesive system.

The role of UAT as a "scaled-down production" testbed is paramount for de-risking the final production deployment. While the provided guide does not explicitly differentiate UAT security or monitoring strategies from production 1, a crucial aspect for UAT is the implementation of realistic yet non-invasive security testing.21 If UAT were to remain a single-node environment, it would fail to expose critical issues related to network latency, inter-node communication, distributed consensus, or failover mechanisms that are inherent to a multi-node production setup. Bugs related to concurrency, distributed transactions, or race conditions might only surface in the live production environment, leading to costly outages. Therefore, to effectively test the "production-readiness" of the system, the UAT environment must architecturally mirror the distributed nature of production. This involves deploying PostgreSQL in a primary-standby configuration, Kafka with multiple brokers, MinIO in a distributed setup, and potentially multiple Ollama instances, even if the overall capacity is lower than production. This architectural mirroring in UAT allows for early detection of distributed system issues, performance bottlenecks under realistic (though not peak) distributed loads, and validation of deployment scripts and CI/CD pipelines 28 in a more complex environment. Furthermore, UAT is the ideal environment for comprehensive security testing, including potentially invasive scans, because it uses non-production data, thereby minimizing risk to live systems.21 This also allows for establishing performance baselines, which are vital for comparison with production performance and identifying regressions or anomalies. This strategic approach ensures that UAT is not merely a larger development environment, but a critical bridge for validating the system's architecture and integration, ultimately saving significant time and cost in the long run.

#### 3.3. Production Environment Architecture (Distributed, High Availability, Disaster Recovery)

* **Objective:** The Production environment represents the live system, meticulously designed for maximum uptime, stringent data integrity, robust security, and optimal performance under peak operational loads. It incorporates comprehensive high availability and disaster recovery mechanisms to ensure business continuity.
* **Architecture Overview:** This environment features a fully distributed, multi-node cluster architecture with redundancy implemented at every layer. Components are strategically deployed across multiple availability zones or geographically distinct data centers to ensure resilience against localized failures. Advanced load balancers are utilized to efficiently distribute traffic and manage automated failover processes.
  + **PostgreSQL:** To achieve high availability, a PostgreSQL cluster is deployed with multiple primary and standby instances. Streaming replication is used for continuous data synchronization, complemented by a failover manager such as Patroni or Pgpool-II for automated failover capabilities.19 For disaster recovery, Point-in-Time Recovery (PITR) is enabled through continuous Write-Ahead Log (WAL) archiving to a separate, secure, and offsite location. Regular physical (  
    pg\_basebackup) and logical (pg\_dump) backups are also performed.30
  + **Apache Kafka:** A robust multi-broker Kafka cluster (e.g., 3-5 brokers) is implemented. Critical topics are configured with a high replication factor (e.g., 3) to ensure fault tolerance and data durability across brokers.3 For simplified cluster management without an external Zookeeper, KRaft mode is recommended for newer versions. Cross-datacenter replication, typically using Kafka MirrorMaker 2, is established for geographic redundancy and disaster recovery.33
  + **Ollama (Local LLM):** For high throughput and the ability to serve larger models, a cluster of Ollama inference servers is deployed. For very large models or extreme concurrency, specialized distributed inference frameworks like vLLM are considered, which offer tensor parallelism and pipeline parallelism across multiple GPUs and nodes.22 Each server dedicated to Ollama should be equipped with high-performance GPUs. A dedicated load balancer (e.g., Nginx or an API Gateway) distributes incoming requests across the Ollama instances, incorporating health checks and failover capabilities.17
  + **Apache Tika:** Multiple Tika server instances are deployed as a scalable service, ideally within a container orchestration platform like Kubernetes, and positioned behind a load balancer. This setup enables handling high volumes of concurrent document parsing requests efficiently.4
  + **MinIO (Object Storage):** A multi-node, multi-drive distributed MinIO deployment is established, comprising a minimum of 4 hosts, ideally 8 or more, to ensure robust data protection and high availability through erasure coding.6 It is critical to use locally-attached NVMe/SSD storage configured in a Just a Bunch Of Disks (JBOD) setup. For disaster recovery across different regions, MinIO site replication is configured to synchronize data between distinct deployments.35
  + **Redis (Caching):** A Redis Cluster is deployed for automatic data sharding and high availability, supporting horizontal scaling of the caching layer. Alternatively, for smaller deployments requiring HA without sharding, a Redis Sentinel setup with multiple master-replica sets is used. A minimum of three Sentinel instances on different physical servers is recommended for robustness.8
  + **FastAPI Backend & Streamlit Frontend:** Multiple instances of both the FastAPI backend and Streamlit frontend applications are deployed across various nodes. These applications are managed by a container orchestration platform (e.g., Kubernetes) and fronted by a robust load balancer (e.g., Nginx or a cloud-native load balancer) with advanced health checks and session persistence capabilities.10
  + **Nginx (Web Server/Reverse Proxy/Load Balancer):** Nginx is configured for advanced load balancing methods, including least-connected and IP-hash (for session persistence), to efficiently distribute traffic. It performs SSL/TLS termination for all external traffic and can be augmented with Web Application Firewall (WAF) capabilities for enhanced security.1
* **Key Characteristics:**
  + **Maximum Uptime:** Achieved through comprehensive redundancy and automated failover mechanisms for all critical components.
  + **Data Durability:** Ensured via robust data replication, continuous WAL archiving, and resilient backup strategies.
  + **Security:** Enforced through comprehensive firewall rules, end-to-end SSL/TLS encryption, stringent access control, regular security audits, and potential WAF/DDoS protection.1
  + **Performance:** Optimized for high throughput and low latency, supported by tuned configurations and horizontal scaling capabilities to handle peak loads.
  + **Disaster Recovery:** Facilitated by cross-region replication and well-defined recovery procedures (Recovery Point Objective/Recovery Time Objective).30

The transition from a single-node development setup to a multi-node distributed architecture across UAT and Production environments introduces a significant increase in complexity concerning networking, configuration management, and overall operational overhead. The initial installation guide primarily configures services to run on localhost 1, which is fundamentally insufficient for a distributed production system where services reside on different machines. If services remain bound to

localhost, they cannot communicate across the network, leading to system failure. Therefore, localhost references must be replaced with actual IP addresses, hostnames, or service discovery mechanisms (e.g., Kubernetes service names, internal DNS entries). This necessitates updating configuration files (e.g., Kafka's server.properties, PostgreSQL's postgresql.conf, application environment variables) and carefully configuring network firewalls 1 to allow inter-node traffic on specific ports, moving from

allow from localhost to allow from <internal\_network\_CIDR> or specific node IPs. This also impacts docker-compose setups where service names become hostnames.1 This inherent complexity necessitates the adoption of sophisticated orchestration tools, such as Kubernetes (though not explicitly detailed in the base guide, its implication is strong given Docker's use), which are designed to automate the deployment, scaling, and management of containerized applications. Furthermore, to maintain consistency, reliability, and efficiency across these complex, multi-environment deployments, robust Continuous Integration/Continuous Deployment (CI/CD) pipelines become essential.28 These pipelines streamline the software delivery process, ensuring that deployments are consistent, repeatable, and traceable, thereby accelerating development agility and reducing operational burden.

A critical factor for the performance and reliability of this distributed system is the underlying network infrastructure. Components like Apache Kafka, MinIO, and inter-LLM communication (if using distributed inference frameworks like vLLM) are highly dependent on low-latency, high-bandwidth networking between nodes.22 For instance, Kafka's fault tolerance relies on followers replicating data from leaders; slow networks can significantly delay this process, impacting recovery time objectives.3 Similarly, MinIO's erasure coding distributes data shards across nodes, and network performance directly influences read/write speeds.6 If the network infrastructure suffers from high latency or insufficient bandwidth, it will directly impede data replication, object storage operations, and the overall efficiency of distributed LLM inference, potentially leading to slower processing, increased message lag, and even cluster instability. Therefore, investing in robust, high-speed, low-latency network infrastructure (e.g., 10GbE or 25GbE/100GbE for high-demand components) is as crucial as selecting powerful CPUs and GPUs for a performant and reliable production environment. Network design should be a primary consideration, not an afterthought, especially when planning for multi-availability zone deployments.

**Table: Component Deployment Matrix by Environment**

| Component | Development Environment | UAT Environment | Production Environment |
| --- | --- | --- | --- |
| **Ollama (LLM)** | Single instance (CPU/single GPU) | Single/Multiple instances (GPU-enabled nodes) for basic concurrency | Multiple instances with load balancing; potentially vLLM for distributed inference (tensor/pipeline parallelism) on dedicated GPU clusters |
| **PostgreSQL** | Single instance (Docker) | Primary-Standby (2 nodes) | Primary-Standby cluster (2+ nodes) with replication, PgBouncer, PITR |
| **Apache Kafka** | Single broker (Docker) | Multi-broker cluster (3 nodes) with replication factor 1-2 | Multi-broker cluster (3-5+ nodes) with replication factor 3+, MirrorMaker for DR |
| **Apache Tika** | Single instance (Docker) | Single/Multiple instances (Docker) | Multiple instances (Docker) behind load balancer |
| **MinIO** | Single node, single drive (Docker) | Multi-node, multi-drive (4 nodes) with erasure coding | Multi-node, multi-drive (4+ nodes) with erasure coding, site replication for DR |
| **Redis** | Single instance (Docker) | Single instance (dedicated VM) / Sentinel (3 nodes) | Redis Cluster (3+ master nodes with replicas) or Redis Sentinel (3+ nodes) |
| **FastAPI Backend** | Single instance (Docker/local) | Multiple instances (containerized) behind Nginx | Multiple instances (containerized) in a scalable cluster (e.g., Kubernetes) behind Nginx |
| **Streamlit Frontend** | Single instance (Docker/local) | Single/Multiple instances (containerized) behind Nginx | Multiple instances (containerized) in a scalable cluster (e.g., Kubernetes) behind Nginx |
| **Nginx** | Single instance (local reverse proxy) | Single instance (reverse proxy, basic load balancer) | Multiple instances (HA pair/cluster) as reverse proxy and advanced load balancer |

### 4. Hardware Sizing and Resource Allocation

#### 4.1. Sizing Methodology and Key Assumptions

Hardware sizing is a critical exercise driven by expected workload, performance requirements (latency, throughput), data volume, and budget constraints. The methodology employed involves a phased approach:

1. **Baseline from Development**: Starting with minimum requirements to ensure a functional system for individual developers.
2. **Scaling for UAT**: Incrementally increasing resources to support realistic testing scenarios and establish preliminary performance baselines.
3. **Production Sizing**: Extrapolating from UAT baselines and incorporating considerations for peak load, high concurrency, long-term data retention, and stringent high availability needs.

Key assumptions underpinning these sizing recommendations include:

* **Document Volume**: Average documents processed per day, average document size, and peak ingestion rates.
* **LLM Usage**: The number of concurrent AI chat users, the frequency of summarization requests, and the specific LLM model size chosen (e.g., 7B, 13B, 70B parameters).24
* **Data Growth**: The anticipated growth rate for object storage (MinIO) and structured data (PostgreSQL) over a defined period (e.g., 1-3 years).
* **Concurrency**: The maximum number of simultaneous document uploads and processing tasks the system is expected to handle.
* **Uptime SLA**: The target Service Level Agreement (SLA) for system uptime (e.g., 99.9% or 99.99%).20

#### 4.2. Development Environment Hardware Sizing

The Development environment is typically hosted on a single developer workstation or a dedicated virtual machine. The sizing prioritizes cost-effectiveness and sufficient resources for individual developers to build, test, and debug application features locally.

* **CPU:** A minimum of 4 CPU cores is recommended (e.g., Intel Core i5/i7 or AMD Ryzen 5/7). While some components might function with fewer, 4 cores provide a better development experience, especially when running multiple services concurrently.25 Ollama specifically recommends 4 cores, with 8 cores being ideal for running 13B LLM models.25
* **RAM:** A minimum of 16 GB of RAM is recommended, with 32 GB being highly advisable. This capacity supports the operating system, Python virtual environments, and the various Docker containers for all system services.25 For instance, Ollama requires at least 8 GB of RAM for 7B models and 16 GB for 13B models 25, while PostgreSQL can function with a minimum of 2 GB.40
* **GPU (Optional but Recommended for LLM):** A GPU is not strictly required for Ollama, but its inclusion significantly enhances performance, particularly for larger LLM models.25 A minimum of 8 GB VRAM is suggested, with 16 GB VRAM being recommended for general AI computer vision tasks, which translates well to LLM inference.39 An NVIDIA GeForce RTX 3050/3060 or equivalent would be suitable.
* **Storage:** A minimum of 256 GB SSD is recommended, with 500 GB NVMe being preferable for faster read/write speeds. This capacity accommodates the operating system, application code, Docker images, and initial data. Ollama requires 12 GB for installation and base models, plus additional space for model data 25, and PostgreSQL needs at least 10 GB of high-speed disk space (SSD/NVMe).40 NVMe drives are generally recommended for their fast read/write speeds, which significantly reduce the time required to load models and datasets.23
* **Network:** Standard workstation connectivity (1 GbE Ethernet) is sufficient for development purposes.

The distinction between "minimum" and "recommended" hardware for local LLM inference 25 highlights a critical trade-off in the development environment: developer experience versus initial cost. While minimum specifications might allow the system to function, they can lead to significantly slower LLM inference times during local testing and iteration. Without a dedicated GPU, LLM inference, especially for larger models or complex queries, will be notably slow on a CPU. This directly translates to longer feedback loops for developers, increasing development time and potential frustration. Investing in the "recommended" hardware, particularly a dedicated GPU with sufficient VRAM, can yield a substantial positive return on investment by accelerating development cycles, enabling more realistic local testing, and improving overall developer morale. This approach ultimately impacts project timelines and the quality of the developed features.

#### 4.3. User Acceptance Testing (UAT) Environment Hardware Sizing

The UAT environment's sizing represents a strategic balance between replicating production conditions and managing costs. It must be robust enough to uncover performance issues, validate scalability, and ensure the system's readiness for production.

* **Objective:** Sufficient resources to simulate realistic user loads and data volumes, validate performance, and test distributed component interactions. Balance between cost and production fidelity.
* **Recommendations (Cluster of 3-5 Virtual Machines/Servers):**
  + **General Node (for Nginx, FastAPI, Streamlit, Tika, Redis):**
    - **CPU:** 4-8 vCPU per node (e.g., Intel Xeon E3/E5 equivalent). While Tika and Redis have relatively low individual requirements 41, these nodes will host multiple application components and require sufficient processing power for concurrency.
    - **RAM:** 32-64 GB per node. This provides ample memory for application processes, caching (Redis), and general OS operations.23
    - **Storage:** 200-500 GB NVMe SSD per node for OS, application binaries, and temporary files.
  + **PostgreSQL Node (2 nodes: Primary + Standby):**
    - **CPU:** 8-16 vCPU per node. Multi-core processors are essential for PostgreSQL in a production-like environment to handle concurrent connections and background processes.40
    - **RAM:** 64 GB per node. This allows for generous allocation to shared\_buffers and work\_mem, critical for database performance.44
    - **Storage:** 500 GB - 1 TB NVMe SSD per node, dedicated for data and Write-Ahead Logs (WAL). Fast disks are crucial for database performance.40
  + **Apache Kafka Node (3 nodes):**
    - **CPU:** 8-16 vCPU per node. While Kafka is often light on CPU, more cores are beneficial for parallelism and if TLS is enabled.32
    - **RAM:** 32-64 GB per node. Kafka heavily utilizes the OS page cache for message buffering, so ample RAM is critical for performance.32
    - **Storage:** 1-2 TB SSD/NVMe per node for Kafka logs. Separation of OS disks from Kafka storage is recommended.32
  + **MinIO Node (4 nodes):**
    - **CPU:** 4-8 vCPU per node. Server-grade CPUs are recommended for production-like deployments.38
    - **RAM:** 32-64 GB per node. MinIO benefits from sufficient memory for its operations.38
    - **Storage:** 2-5 TB NVMe/SSD per node, configured in a Just a Bunch Of Disks (JBOD) setup for optimal MinIO performance.6
  + **LLM Node (1-2 GPU-enabled nodes):**
    - **CPU:** 8-16 vCPU per node. High clock speeds and multiple cores are beneficial for data preprocessing and feeding the GPU.23
    - **RAM:** 64-128 GB per node. Sufficient RAM is necessary to load LLM models and handle intermediate data.23
    - **GPU:** 1-2 NVIDIA GPUs with 16-24 GB VRAM each (e.g., RTX 3060/3070/3080/4070/4080). These GPUs provide a balance of performance and cost for UAT-level LLM inference.23
    - **Storage:** 500 GB - 1 TB NVMe SSD for storing LLM models and scratch space.
* **Networking:** A 1 GbE or 10 GbE internal network is recommended for inter-node communication, ensuring sufficient bandwidth for data transfer between distributed services.32

UAT hardware sizing is not about achieving peak production capacity, but rather about establishing performance baselines and validating the system's scalability under *representative* loads. If UAT is undersized or maintains a single-node architecture, it cannot accurately predict production behavior. This includes critical aspects such as network latency effects, inter-node communication overhead, distributed consensus mechanisms, and failover behaviors. Consequently, bugs related to concurrency, distributed transactions, or race conditions might only manifest in the live production environment, leading to costly outages and significant operational disruptions. By scaling UAT to a multi-node setup with moderate resources, even if not at full production scale, it enables:

* **Accurate Performance Baselines:** Measuring response times, throughput, and resource utilization under controlled, realistic distributed loads.
* **Scalability Validation:** Identifying bottlenecks that emerge when components communicate across a network, or when multiple instances of an application run concurrently.
* **Configuration Validation:** Testing how configuration changes (e.g., Kafka replication factors, PostgreSQL tuning parameters) impact performance and stability in a distributed context.

This approach transforms UAT into a strategic investment for de-risking the production deployment. It allows for the identification of architectural flaws or performance limits *before* they impact live users, thereby saving significant time, resources, and potential reputational damage in the long run.

#### 4.4. Production Environment Hardware Sizing

The Production environment demands dedicated, high-performance hardware configured for maximum uptime, data durability, and optimal performance under peak load conditions.

* **Objective:** Meet high-throughput, low-latency demands with maximum uptime, data durability, and disaster recovery capabilities.
* **Recommendations (Enterprise-Grade Cluster):**
  + **PostgreSQL Cluster (3+ nodes: Primary + 2+ Standbys):**
    - **CPU:** Dual 12-core+ processors (e.g., Intel Xeon E5-2697v2/v4 or AMD EPYC equivalent). High core counts are crucial for handling concurrent connections and background database operations efficiently.23
    - **RAM:** 128 GB - 256 GB per node. This substantial RAM allocation is critical for PostgreSQL's shared\_buffers, operating system cache, and work\_mem to minimize disk I/O and improve query performance.23
    - **Storage:** 2-5 TB NVMe SSD per node, dedicated for data and Write-Ahead Logs (WAL). Fast, reliable disks are paramount for database performance and integrity.40
  + **Apache Kafka Cluster (3-5+ Broker nodes + 3 KRaft Controller nodes):**
    - **Broker CPU:** Dual 12-core+ sockets (e.g., Intel Xeon, AMD EPYC). More cores are generally preferred over faster clock speeds for Kafka, providing better concurrency.32
    - **Broker RAM:** 64 GB - 128 GB per node. Kafka heavily relies on the OS page cache for efficient message buffering, with heap size typically configured up to 6GB.32
    - **Broker Storage:** 12 x 1 TB (or more) NVMe SSDs per node, configured with RAID 10 for optimal read/write performance and data protection. It is crucial to separate OS disks from Kafka storage disks.32
    - **KRaft Controller:** Each KRaft controller node requires 4 cores CPU, 4 GB RAM, and a 64 GB SSD.32
    - **Network:** 10 GbE or 25 GbE network interfaces are essential for high-throughput data streaming between brokers.32
  + **Ollama/LLM Inference Cluster (Dedicated GPU Servers):**
    - **CPU:** Dual 18-Core E5-2697v4 or similar high-core count CPUs (e.g., AMD Ryzen Threadripper, Intel Xeon Scalable) are necessary to efficiently feed data to the GPUs and handle pre/post-processing tasks.23
    - **RAM:** 256 GB - 512 GB DDR4/DDR5 per server. This large memory capacity is required for loading large LLM models, managing intermediate data, and supporting concurrent inference requests.23
    - **GPU:** Multiple NVIDIA A100 (40GB/80GB HBM2) or RTX 4090 (24GB GDDR6X) GPUs per server. The exact number and type of GPUs depend critically on the chosen LLM model size, quantization factor, and desired concurrent inference throughput.23 For instance, a Llama 3.1 70B model at FP16 precision requires approximately 168 GB of VRAM, necessitating multiple high-end GPUs to accommodate.24
    - **Storage:** 2-4 TB NVMe SSD for rapid loading and storage of LLM models, plus additional SATA/SSD for OS and logs.23
    - **Cooling & Power:** Liquid cooling systems are highly recommended for high-performance builds running LLMs for extended periods. A power supply unit (PSU) of 1000W or more is typically required to handle the combined wattage of high-end GPUs and CPUs.23
    - **Network:** 10 GbE or 25 GbE network interfaces are crucial for inter-server communication, especially if utilizing distributed inference frameworks like vLLM that involve data transfer between GPUs across different machines.22
  + **Apache Tika Servers (Multiple instances):**
    - **CPU:** 4-8 vCPU per instance.
    - **RAM:** 8-16 GB per instance. While Tika is memory-efficient 42, multiple instances will require sufficient RAM to handle concurrent document parsing requests.
    - **Storage:** 100-200 GB SSD for OS, Tika JAR files, and temporary processing files.
  + **MinIO Cluster (4+ nodes):**
    - **CPU:** Server-grade CPUs with support for modern SIMD instructions (e.g., AVX-512), such as Intel Xeon Scalable or AMD EPYC, with 8+ vCPU per pod.37
    - **RAM:** 128 GB - 256 GB RAM or higher per node. MinIO benefits from ample memory for caching and internal operations.37
    - **Drives:** NVMe Flash drives (PCIe Gen 4.0, 30 TB or higher per drive) are recommended, configured in a JBOD (Just a Bunch Of Disks) setup and formatted with XFS for optimal performance.6
    - **Network:** Single/Dual 100 GbE or higher network interfaces are critical to support the maximum possible throughput of the attached storage.37
  + **Redis Cluster (3+ Master nodes with N-1 replicas):**
    - **CPU:** A minimum of 4 CPU cores per node, with 6-10 cores recommended for large datasets and high ingestion rates.41
    - **RAM:** 8 GB+ per node. For production, 250MB RAM with one primary and one replica is a minimum, but more is needed for larger datasets and higher throughput.41
    - **Disk:** 20GB in the /var folder and 1GB in the /opt folder for logs, in addition to OS footprint.41
    - **Network:** 10 GbE or more network interfaces are recommended for high-performance caching and replication.41
  + **FastAPI & Streamlit Application Servers:**
    - **CPU:** 8-16 vCPU per server to support multiple Uvicorn worker processes for FastAPI and Streamlit instances.
    - **RAM:** 64-128 GB per server to accommodate application memory usage and potential caching.
    - **Storage:** 200-500 GB NVMe SSD for OS and application binaries.
  + **Nginx Load Balancers (HA Pair):**
    - **CPU:** 4-8 vCPU per instance.
    - **RAM:** 16-32 GB per instance.
    - **Storage:** 100-200 GB SSD for OS and Nginx configurations/logs.
    - **Network:** High-speed network interfaces (e.g., 10 GbE) to handle high volumes of incoming traffic.

The LLM component (Ollama) will impose a significant burden on the production hardware, particularly due to its GPU VRAM requirements. This disproportionately affects the overall infrastructure cost and complexity compared to other components. The explicit formulas for LLM memory calculation 24 demonstrate that model size and quantization directly multiply to determine the required GPU memory. For instance, a 70B parameter model at FP16 precision demands approximately 168GB of VRAM 24, a massive amount that typically requires multiple high-end GPUs (e.g., A100s or RTX 4090s) within a single server.25 These specialized GPUs are expensive, and the servers designed to house them (with adequate power, cooling, and PCIe lanes) represent a substantial capital expenditure. This contrasts sharply with the more general-purpose server hardware suitable for PostgreSQL, Kafka, MinIO, or Redis, which primarily scale with CPU, RAM, and I/O. This creates a "LLM tax" where a significant portion of the budget and infrastructure complexity is driven by this single component. Therefore, organizations must carefully evaluate the trade-off between desired AI capability (larger models often imply better performance but higher cost) and budget/infrastructure constraints. Strategic choices regarding LLM model size and quantization (e.g., using INT4 or FP8 for reduced VRAM footprint without drastic quality loss) become paramount to control costs and ensure the feasibility of the deployment.

**Table: Hardware Sizing Summary by Environment**

| Metric | Development Environment (Single Node) | UAT Environment (Small Cluster) | Production Environment (Enterprise Cluster) |
| --- | --- | --- | --- |
| **Total CPU Cores** | 4-8 cores | 32-64+ vCPU (across nodes) | 96-256+ vCPU (across nodes, including dedicated LLM CPUs) |
| **Total RAM** | 16-32 GB | 256-512 GB (across nodes) | 1 TB - 2 TB+ (across nodes, including dedicated LLM RAM) |
| **Total GPU VRAM** | 8-16 GB (optional) | 32-48 GB (1-2 GPUs, recommended) | 100 GB - 500 GB+ (multiple high-end GPUs) |
| **Total Storage (SSD/NVMe)** | 256-500 GB SSD/NVMe | 5-10 TB NVMe/SSD (across nodes) | 20-50 TB+ NVMe/SSD (across nodes, high-performance) |
| **Network Interface** | 1 GbE | 1 GbE / 10 GbE internal | 10 GbE / 25 GbE / 100 GbE internal |

**Table: Detailed Production Environment Hardware Requirements (Example per Node/Component)**

| Component | Node Type / Role | CPU (Cores) | RAM (GB) | GPU (VRAM GB) | Storage (Type/Capacity) | Network (GbE) |
| --- | --- | --- | --- | --- | --- | --- |
| **PostgreSQL** | Primary/Standby | Dual 12-core+ | 128-256 | N/A | 2-5 TB NVMe SSD (dedicated) | 10 |
| **Kafka Broker** | Broker Node | Dual 12-core+ | 64-128 | N/A | 12x1 TB NVMe SSD (RAID 10) | 10-25 |
| **Kafka Controller** | KRaft Controller | 4 | 4 | N/A | 64 GB SSD | 10 |
| **Ollama/LLM** | Inference Server | Dual 18-core+ | 256-512 | 24-48+ (e.g., RTX 4090, A100) | 2-4 TB NVMe SSD | 10-25 |
| **Apache Tika** | Processing Worker | 8-16 | 16-32 | N/A | 200-500 GB SSD | 1-10 |
| **MinIO** | Storage Node | 8-16 | 128-256 | N/A | 30 TB+ NVMe Flash (JBOD) | 100 |
| **Redis** | Master/Replica | 4-8 | 8-16 | N/A | 20-50 GB SSD (for logs/OS) | 10 |
| **FastAPI Backend** | Application Server | 8-16 | 64-128 | N/A | 200-500 GB NVMe SSD | 10 |
| **Streamlit Frontend** | Application Server | 8-16 | 64-128 | N/A | 200-500 GB NVMe SSD | 10 |
| **Nginx** | Load Balancer | 4-8 | 16-32 | N/A | 100-200 GB SSD | 10 |

### 5. Implementation Strategy

#### 5.1. Phased Deployment Approach

The implementation of the AI Document Processing System will follow a structured, phased deployment approach to manage complexity, mitigate risks, and ensure a stable and reliable rollout. This strategy builds upon the foundational steps outlined in the installation guide.1

* **Phase 1: Foundation and Core Services (Development & UAT)**
  + **Objective:** To establish the fundamental infrastructure and core shared services required for the system's operation. This phase focuses on setting up the underlying operating environment and foundational data/messaging layers.
  + **Steps:**
    - **System Preparation:** This initial step involves updating the operating system and installing all essential core dependencies, including Python, Docker, and a compatible Java Development Kit (JDK 17 or higher).1 This forms the stable base for all subsequent installations.
    - **Virtual Environment Setup:** A dedicated Python virtual environment is created and activated. This isolates Python dependencies for the project, preventing conflicts and ensuring a clean development workspace. Core Python libraries necessary for the applications are then installed.1
    - **Core Services Deployment (Development):** In the development environment, PostgreSQL, Apache Kafka (with Zookeeper), MinIO, and Redis are deployed using Docker Compose.1 This provides a quick, isolated, and reproducible setup for individual developers.
    - **Core Services Deployment (UAT):** For the UAT environment, multi-node deployments of PostgreSQL, Kafka, MinIO, and Redis are established on dedicated virtual machines or servers. This can utilize native installation methods or be automated via infrastructure-as-code tools.
    - **Nginx Configuration:** Nginx is set up as a reverse proxy to provide initial access to the application services.1
  + **Verification:** After completing this phase, comprehensive checks are performed to confirm that all foundational services are running correctly, are accessible, and are communicating as expected.1
* **Phase 2: Application Development and Integration (Development & UAT)**
  + **Objective:** To develop the core AI document processing logic and user interfaces, and to ensure their seamless integration with the foundational services established in Phase 1.
  + **Steps:**
    - **Database Schema Initialization:** The PostgreSQL database schema, including tables for processed documents, chat sessions, and system logs, is initialized by executing the provided SQL script.1
    - **FastAPI Backend Development:** The backend API is developed using FastAPI. This involves implementing endpoints for health checks, secure document uploads (integrating with MinIO and Kafka), triggering document processing workflows (integrating with Tika and Ollama, and storing results in PostgreSQL), and managing AI chat interactions.1
    - **Streamlit Frontend Development:** The user-facing web interface is developed using Streamlit. This includes creating the UI for document upload forms, displaying processing status, and providing an interactive AI chat interface, all designed to communicate seamlessly with the FastAPI backend.1
    - **Application Startup Scripts & Systemd Services:** Shell scripts are created to activate the Python virtual environment and launch the FastAPI backend and Streamlit frontend applications. Systemd service files are then configured to ensure these applications run as background services, start automatically on boot, and can be easily managed.1
  + **Verification:** This phase concludes with thorough end-to-end testing, validating document upload, processing, and AI chat functionality through both the user interface and direct API calls.1
* **Phase 3: AI Capabilities and Optimization (UAT & Production)**
  + **Objective:** To integrate the core AI capabilities (LLM and document extraction) and perform system-level optimizations for enhanced performance and reliability.
  + **Steps:**
    - **Ollama Installation & Model Loading:** Ollama is installed to enable local Large Language Model (LLM) inference. This involves pulling recommended LLM models such as llama3.1:8b, codellama2:7b, mistral:7b, and llava:7b to support various AI functionalities.1
    - **Apache Tika Installation:** Apache Tika is deployed to facilitate robust document content extraction. The Docker-based installation method is recommended for its simplicity and consistency.1
    - **Performance Tuning:** System-level performance optimizations are implemented, including increasing file descriptor limits and configuring kernel parameters for high performance. Additionally, PostgreSQL-specific tuning parameters are adjusted for optimal database performance.1
  + **Verification:** This phase involves validating that LLM models are loaded and accessible, Apache Tika can accurately process documents, and the system performs efficiently under simulated loads. Logs are reviewed for any errors or warnings.1
* **Phase 4: Security Hardening & Operationalization (Production)**
  + **Objective:** To secure the deployed system against unauthorized access and potential vulnerabilities, and to establish robust operational practices for ongoing management.
  + **Steps:**
    - **Firewall Configuration:** Uncomplicated Firewall (UFW) rules are meticulously configured for the production environment, allowing only strictly necessary inbound and outbound network traffic.1
    - **SSL/TLS Configuration:** SSL/TLS encryption is implemented for all external-facing services via Nginx, using certificates obtained from Certbot (for Let's Encrypt) or an enterprise Certificate Authority.1
    - **Monitoring & Logging Setup:** A comprehensive centralized logging solution (e.g., ELK Stack, Better Stack, SolarWinds Log Analyzer) is deployed to aggregate logs from all system components.26 Advanced monitoring tools (e.g., Prometheus with Grafana) are set up to collect real-time metrics and provide operational insights.1
    - **Backup & DR Implementation:** Component-specific backup strategies and comprehensive disaster recovery plans are designed and implemented to ensure data durability and system resilience.
  + **Verification:** This final phase includes conducting thorough security audits, penetration testing 21, and simulated disaster recovery drills to validate the effectiveness of the implemented measures.

#### 5.2. Continuous Integration/Continuous Deployment (CI/CD) Pipeline Integration

* **Objective:** To automate the build, test, and deployment processes across Development, UAT, and Production environments. This ensures consistency, reduces manual errors, and enables rapid, reliable software releases.
* **Strategy:**
  + **Centralized CI/CD:** A centralized CI/CD platform (e.g., GitHub Actions, GitLab CI/CD, Jenkins) will be utilized to manage pipelines for all microservices within the system. This approach promotes consistency across different services and reduces duplication of pipeline logic, serving as a single source of truth for build, test, and deployment processes.28
  + **Containerization:** Dockerfiles for each application component (FastAPI, Streamlit) will be leveraged to create portable and consistent build artifacts. This ensures that the application behaves identically across all environments, from development to production.1
  + **Automated Testing:** Unit, integration, and end-to-end tests will be integrated directly into the CI pipeline. This ensures that code changes are continuously validated, maintaining a high standard of code quality and preventing regressions.11
  + **Environment-Specific Deployments:** The CI/CD pipelines will be configured for environment-specific deployments:
    - **Development:** Automated deployment upon every code commit for immediate feedback.
    - **UAT:** Deployment upon successful completion of integration tests in the CI pipeline.
    - **Production:** Deployment after UAT sign-off and typically requiring a manual approval gate, ensuring a controlled release. 29
  + **Secrets Management:** A robust system for centralized secrets and environment variables management will be implemented to securely handle sensitive credentials for different environments.28
  + **Deployment Strategies:** For production deployments, advanced strategies such as blue-green deployments or canary releases will be considered to minimize downtime and risk during updates.29

The transition from a single-node setup, where manual systemctl commands are feasible for starting services 1, to multiple environments (Dev, UAT, Prod) with distributed components makes manual deployment unsustainable and highly error-prone. In such complex, multi-node environments, relying on manual processes for updates and configurations can lead to inconsistencies, extended downtime, and significant operational burden. The adoption of CI/CD pipelines becomes a non-negotiable requirement for efficient and reliable operations. By automating the entire software delivery lifecycle, from code integration and testing to deployment, CI/CD ensures that every deployment is consistent, repeatable, and traceable. This directly translates to faster release cycles, a drastic reduction in human error, and simplified rollback procedures, which are critical for managing the complexity of a distributed AI system and maintaining high availability. The investment in setting up a robust CI/CD pipeline is therefore essential for managing the complexity of a distributed, multi-environment AI system, directly impacting the agility of the development team and the reliability of the production system, thereby reducing operational overhead in the long run.

#### 5.3. Data Migration and Seeding Considerations

* **Initial Data Loading:** A clear plan will be established for initial data loading into the UAT and Production PostgreSQL databases and MinIO object storage. For UAT, this may involve generating synthetic data or using a carefully selected, anonymized subset of real data. For production, a comprehensive data migration strategy will be developed.
* **Schema Evolution:** Database migration tools, such as Alembic for PostgreSQL (as suggested by best practices for FastAPI production deployments 11), will be utilized to manage and apply schema changes systematically across all environments, ensuring consistency and preventing data integrity issues during updates.
* **Data Synchronization (UAT to Prod):** A strategy will be defined for synchronizing relevant non-sensitive data from UAT to Production (e.g., configuration data, specific test cases). Additionally, a process for periodically refreshing or resetting UAT data will be implemented to ensure test validity.

### 6. Operational Considerations

#### 6.1. Monitoring and Centralized Logging Strategy

* **Objective:** To achieve comprehensive visibility into the system's health, performance, and potential operational issues across all Development, UAT, and Production environments. This enables proactive problem identification and efficient troubleshooting.
* **Centralized Logging:**
  + **Tools:** A robust centralized logging solution will be implemented to aggregate logs from all system components. Popular choices include the ELK Stack (Elasticsearch, Logstash, Kibana) or commercial offerings such as Better Stack or SolarWinds Log Analyzer.26 This aggregation is critical for correlating events across distributed services.
  + **Log Rotation:** logrotate will be configured for application-specific logs to manage disk space consumption and ensure logs are archived and compressed regularly.1
  + **Log Locations:** Standardized log locations will be enforced across all environments for consistent collection by logging agents. These include system logs (/var/log/syslog), service-specific logs (journalctl -u service-name), application logs (~/ai-document-system/logs/), Nginx logs (/var/log/nginx/), and PostgreSQL logs (/var/log/postgresql/).1
* **Monitoring:**
  + **Tools:** A comprehensive monitoring system, such as Prometheus for metric collection combined with Grafana for visualization and alerting, will be deployed. This system will collect a wide array of metrics from all services (e.g., CPU utilization, RAM usage, disk I/O, network throughput, application-specific metrics like request latency, error rates, and queue lengths).
  + **System Monitoring Script:** The provided monitor.sh script 1 will be utilized and enhanced to perform basic health checks and resource utilization monitoring. This script will be scheduled to run regularly via  
    cron for continuous oversight.
  + **Alerting:** Proactive alerting mechanisms will be configured for critical thresholds. This includes alerts for high CPU/memory usage, service downtime, elevated error rates, Kafka consumer lag, and LLM inference latency. These alerts will notify operations teams promptly of potential issues, enabling rapid response.

While the initial installation guide provides foundational logging and monitoring capabilities through local scripts and log rotation configurations 1, enterprise-grade environments demand a more sophisticated approach. In a distributed system with numerous interconnected components, relying solely on local logs and basic monitoring makes it exceedingly difficult to diagnose issues, identify root causes, or understand system-wide performance. The shift to centralized logging solutions (e.g., Better Stack, SolarWinds Log Analyzer 26) and advanced monitoring platforms (like Prometheus/Grafana) is crucial. These tools aggregate data from all components into a single pane of glass, allowing for comprehensive correlation of events, real-time performance analysis, and proactive detection of anomalies. This moves beyond reactive troubleshooting, enabling operations teams to anticipate and address potential problems before they impact end-users, thereby ensuring higher system stability and performance.

**Table: Key Monitoring Metrics and Tools**

| Component | Key Metrics to Monitor | Recommended Tools/Methods |
| --- | --- | --- |
| **Overall System** | CPU Utilization, Memory Usage, Disk I/O, Network Throughput, System Uptime | Prometheus, Grafana, monitor.sh script, htop, df -h, free -h |
| **FastAPI Backend** | Request Latency, Error Rate (HTTP 5xx), Throughput (Requests/sec), Active Connections, Queue Lengths | Prometheus/Grafana (FastAPI Exporter), Application Logs (Centralized Logging) |
| **Streamlit Frontend** | Page Load Time, User Sessions, Frontend Errors | Browser Dev Tools, Application Logs (Centralized Logging) |
| **Ollama (LLM)** | GPU Utilization, GPU Memory Usage, Inference Latency, Model Load Times, Model Cache Hits/Misses | NVIDIA-SMI, Prometheus/Grafana (Node Exporter, custom Ollama exporter) |
| **PostgreSQL** | Active Connections, Query Latency, Disk I/O, Cache Hit Ratio, WAL Activity, Replication Lag, Autovacuum Activity | Prometheus/Grafana (pg\_exporter), pg\_stat\_activity, PostgreSQL logs |
| **Apache Kafka** | Broker Throughput (In/Out), Producer/Consumer Latency, Consumer Lag, Partition Count, Replication Status, Zookeeper Health | Prometheus/Grafana (JMX Exporter, Kafka Exporter), Kafka Manager/Control Center, Zookeeper logs |
| **Apache Tika** | Document Processing Rate, Error Rate, Response Time, Memory/CPU Usage | Prometheus/Grafana, Tika Server Logs |
| **MinIO** | Bucket Size, Object Count, Read/Write Throughput, Latency, Erasure Code Health, Disk Usage, Replication Status | Prometheus/Grafana (MinIO Exporter), MinIO Console, MinIO Logs |
| **Redis** | Cache Hit Ratio, Memory Usage, Connections, Latency, Replication Status, Persistence (RDB/AOF) Status | Prometheus/Grafana (Redis Exporter), redis-cli info, Redis Logs |
| **Nginx** | Request Rate, Error Rate, Connection Count, Upstream Health, Cache Hit/Miss Ratio | Prometheus/Grafana (Nginx Exporter), Nginx Access/Error Logs |

#### 6.2. Security Hardening Across Environments

* **Objective:** To rigorously protect the system from unauthorized access, data breaches, and a wide array of cyber threats across all environments.
* **Firewall Configuration (UFW):**
  + **Default Policy:** The Uncomplicated Firewall (UFW) will be configured with a strict default policy: deny all incoming connections and allow all outgoing connections.1
  + **Allowed Ports (External):** Only essential external-facing ports will be explicitly opened: SSH (22) for secure administration, HTTP (80) for initial web access, and HTTPS (443) for secure web traffic.1
  + **Allowed Ports (Internal/Inter-service):** For distributed UAT and Production environments, inter-service communication ports will be precisely configured. This includes PostgreSQL (5432), Kafka (9092), Redis (6379), Ollama (11434), Tika (9998), MinIO (9000, 9001), FastAPI (8000), and Streamlit (8501). These ports will be restricted to specific internal network ranges (CIDR blocks) or authorized node IP addresses, moving away from broad localhost allowances.1
* **SSL/TLS Configuration (UAT & Production):**
  + **External Traffic:** All external-facing services exposed via Nginx will enforce SSL/TLS encryption. This will be achieved using certificates obtained from Certbot (for Let's Encrypt) or enterprise-issued Certificate Authorities.1
  + **Internal Traffic:** For highly sensitive inter-service communication within the production environment, Mutual TLS (mTLS) will be considered, particularly for Kafka and database connections, to ensure authenticated and encrypted communication between services.
* **Access Control:**
  + **Least Privilege:** The principle of least privilege will be strictly enforced for all system users, service accounts, and application components, granting only the minimum necessary permissions required for their function.
  + **Database:** Strong, complex passwords will be set for PostgreSQL users 1, and a policy for regular password rotation will be implemented.
  + **MinIO:** Strong credentials will be configured for the MINIO\_ROOT\_USER 1, and fine-grained access will be managed using MinIO's Identity and Access Management (IAM) policies.
* **Data Encryption:**
  + **At Rest:** Data stored in MinIO and PostgreSQL will be encrypted at rest, utilizing underlying disk encryption or database-level encryption features.
  + **In Transit:** All network communication, especially for sensitive data, will be encrypted using SSL/TLS.
* **Regular Audits & Testing:**
  + Regular security audits, vulnerability scans, and penetration testing will be conducted to identify and address potential weaknesses.21
  + Dedicated test user accounts will be utilized for security scans to minimize any risk to live production data.21
* **Patch Management:** A robust patch management process will be established for the operating system, all installed libraries, and every deployed software component to address known vulnerabilities promptly.

The firewall strategy must evolve significantly for distributed UAT and Production environments compared to the localhost rules provided in the initial guide.1 The guide's allowance of traffic "from localhost to any port" 1 is fundamentally insecure and non-functional in a multi-node setup. In such an environment, services like PostgreSQL, Kafka, MinIO, and Redis run on distinct machines. If they are only permitted to connect from

localhost, they cannot communicate across the network. Conversely, changing the rule to allow any would expose critical services to the entire network, creating a severe security vulnerability. For distributed systems, firewall rules must explicitly permit traffic *between* specific nodes or from defined internal network segments (CIDR blocks) on the required ports. This ensures that only authorized internal services can communicate, while external access remains highly restricted. This necessitates a more sophisticated network design and firewall management strategy for UAT and Production, moving beyond simple localhost rules to a segmented network approach with granular ingress/egress rules for each service and environment. This is a critical security and operational shift that directly impacts the system's overall security posture and resilience against external and internal threats.

#### 6.3. Backup and Disaster Recovery Strategy

* **Objective:** To ensure business continuity and minimize data loss (Recovery Point Objective - RPO) and downtime (Recovery Time Objective - RTO) in the event of system failures, data corruption, or catastrophic disasters.30
* **PostgreSQL Backup & Recovery:**
  + **Strategy:** A robust strategy combines frequent physical backups (pg\_basebackup) with continuous Write-Ahead Log (WAL) archiving. This enables Point-in-Time Recovery (PITR), allowing the database to be restored to any specific moment in time.19
  + **Logical Backups:** pg\_dump will be used for logical backups, suitable for smaller databases, schema-only backups, or specific table restorations.30
  + **Storage:** All backups, including WAL archives, will be stored in a separate, secure, and geographically distant (offsite/multi-region) location to protect against localized disasters.31
  + **Testing:** Regular testing of backup restorability to a separate, non-production environment is crucial to validate backup integrity and practice recovery procedures.31
* **Apache Kafka Backup & Disaster Recovery:**
  + **Replication:** Kafka's built-in replication factor (minimum 3 for critical production topics) provides real-time data redundancy and high availability within the cluster. Each partition has leader and follower replicas, ensuring data availability even if a broker fails.3
  + **Cross-Datacenter Replication:** For disaster recovery across geographic regions, Apache Kafka MirrorMaker 2 (MM2) or similar tools will be implemented to replicate data to a parallel Kafka cluster in a different data center. This ensures high RPO/RTO for critical data.33
  + **Filesystem Backups:** As a more economical option for scenarios tolerating longer RTOs (e.g., days), regular filesystem backups of Kafka log directories will be performed.34
  + **Consumer Offsets:** A strategy for backing up and restoring consumer offsets will be in place to ensure consumers can resume processing from the correct position after a recovery event.33
* **MinIO Backup & Disaster Recovery:**
  + **Erasure Coding:** MinIO's distributed deployment inherently uses erasure coding for data protection and self-healing within a single storage pool. This provides high availability and data durability against drive or node failures.6
  + **Site Replication:** MinIO site replication will be configured to synchronize data (objects, buckets, IAM policies, configurations) across two or more distinct MinIO deployments located in different racks, data centers, or geographic regions. This is the primary mechanism for business continuity and disaster recovery.6
  + **Bucket Replication:** Specific bucket replication rules can be used to copy objects and their versions to a target bucket, which is useful for restoring specific datasets or for compliance purposes.35
  + **Verification:** Regular validation of replication status and periodic execution of recovery drills are essential to ensure the effectiveness of the MinIO DR strategy.35
* **Redis Backup & Recovery:**
  + **Persistence:** Redis persistence mechanisms, including RDB snapshots (point-in-time backups) and AOF (Append-Only File) logs (transactional logs), will be configured for data durability.8
  + **High Availability:** Redis Cluster provides automatic data sharding and high availability through its master-replica model. Alternatively, Redis Sentinel can be used to manage master-replica setups with automated failover for smaller deployments not requiring sharding. A minimum of three Sentinel instances on different physical servers is recommended for robustness.8
  + **Backup:** RDB snapshots and AOF files will be regularly backed up to a separate, secure storage location.
* **Application Code & Configuration:**
  + **Version Control:** All application code will be stored in a version control system (e.g., Git) with appropriate branching and merging strategies.
  + **Infrastructure-as-Code (IaC):** Infrastructure and application configurations will be managed using IaC principles (e.g., Ansible, Terraform) and stored securely within the version control system.

The initial installation guide, while comprehensive for setup, lacks detailed disaster recovery strategies for the distributed components.1 Implementing robust disaster recovery (DR) strategies for each distributed component (PostgreSQL, Kafka, MinIO, Redis) is critical for ensuring business continuity. These components, by their nature, store different types of data and have distinct replication and recovery mechanisms. For instance, PostgreSQL relies on WAL archiving for Point-in-Time Recovery 30, while Kafka uses its replication factor and tools like MirrorMaker for cross-datacenter replication.33 MinIO employs erasure coding and site replication for data protection and recovery.35 Without these specific, component-tailored DR plans, the system remains vulnerable to data loss and extended downtime in the event of major failures. Therefore, a comprehensive DR strategy must involve understanding the unique characteristics of each component's data, implementing appropriate replication and backup mechanisms, and defining clear Recovery Point Objectives (RPO) and Recovery Time Objectives (RTO) to align technical capabilities with business requirements for data loss tolerance and recovery speed. This proactive planning and regular testing of DR procedures are essential for safeguarding against unforeseen events and maintaining operational resilience.

**Table: Backup and Recovery Strategy Summary**

| Component | Backup Method(s) | Recovery Mechanism(s) | RPO/RTO Considerations | Storage Location |
| --- | --- | --- | --- | --- |
| **PostgreSQL** | pg\_basebackup (physical), WAL archiving, pg\_dump (logical) | PITR, pg\_restore from backups | Low RPO (minutes) with WAL archiving; RTO depends on data size for full restore | Offsite/Multi-region storage |
| **Apache Kafka** | Replication Factor (intra-cluster), MirrorMaker 2 (inter-cluster), Filesystem snapshot | Leader election (HA), Cluster failover, Restore from replicated cluster/snapshots | Very low RPO (seconds) with MM2; higher RTO (hours-days) with filesystem backups | Separate cluster, Geographic region, S3/Object Storage |
| **MinIO** | Erasure Coding (intra-cluster), Site Replication, Bucket Replication | Automatic self-healing, Site failover, Bucket restore | Low RPO (near-realtime) with site replication; higher RTO with bucket replication | Separate MinIO deployment, Geographic region |
| **Redis** | RDB snapshots, AOF logs, Redis Cluster/Sentinel HA | Automatic failover (Cluster/Sentinel), Restore from RDB/AOF | Low RPO (seconds-minutes) with AOF/HA; higher RTO with RDB restore | Separate storage, different availability zone |
| **Application Code** | Version Control (Git) | CI/CD pipeline for automated deployment | RTO depends on CI/CD speed | Git Repository (remote) |
| **Configuration** | Version Control (Git), Infrastructure-as-Code | Automated deployment via CI/CD | RTO depends on CI/CD speed | Git Repository (remote) |

#### 6.4. Performance Optimization Guidelines

* **System Limits:** Critical system limits will be increased to support high-performance operations. This includes increasing the maximum number of open file descriptors (e.g., to 65536) and tuning kernel parameters such as vm.max\_map\_count and net.core.somaxconn to optimize memory mapping and network connections.1
* **PostgreSQL Tuning:** PostgreSQL configuration will be meticulously tuned for optimal performance. Key parameters to adjust include shared\_buffers, effective\_cache\_size, work\_mem, maintenance\_work\_mem, checkpoint\_completion\_target, wal\_level, max\_wal\_size, and random\_page\_cost (especially for SSDs).1
* **Kafka Tuning:** Performance optimization for Kafka will involve allocating sufficient RAM for the OS page cache, optimizing disk I/O through the use of RAID 10 and NVMe drives, and carefully considering CPU allocation if TLS encryption is enabled, as it can significantly increase CPU requirements.32
* **Ollama/LLM Optimization:**
  + **GPU Acceleration:** Leveraging dedicated GPU hardware is paramount for efficient LLM inference.
  + **Model Quantization:** Optimizing LLM model quantization (e.g., using Q4\_K or Q5\_1 formats) can significantly reduce memory footprint while maintaining acceptable accuracy.2
  + **Concurrency Settings:** Tuning Ollama's environment variables such as OLLAMA\_NUM\_PARALLEL (for concurrent requests) and OLLAMA\_MAX\_LOADED\_MODELS (for loading multiple models simultaneously) is crucial for maximizing throughput.17
  + **Distributed Inference:** For very large models or extremely high concurrency, investigating and potentially migrating to advanced distributed inference frameworks like vLLM is recommended.22
* **Redis Tuning:** Redis configuration will be optimized for efficient caching. This includes setting maxmemory to define the maximum memory usage and configuring maxmemory-policy (e.g., allkeys-lru) to manage key eviction strategies.1
* **Nginx Tuning:** Nginx performance will be optimized by configuring worker\_processes to match the number of CPU cores, and by tuning proxy buffers and timeouts to handle high volumes of concurrent connections efficiently.15
* **Application Optimization:** FastAPI and Streamlit applications will undergo profiling to identify and eliminate performance bottlenecks. This includes optimizing database queries, streamlining I/O operations, and implementing asynchronous processing for any CPU-bound tasks to ensure the application remains responsive.11

### 7. Conclusion and Recommendations

This report has provided a comprehensive blueprint for designing, implementing, and operating a scalable AI document processing system across development, UAT, and production environments. The proposed architecture leverages robust, distributed open-source technologies to ensure high availability, performance, and data integrity throughout the system's lifecycle.

**Key Recommendations:**

1. **Adopt a Phased Implementation:** Adhering strictly to the outlined phased deployment strategy is crucial. This approach allows for building a stable foundational infrastructure before progressively introducing complex functionalities and scaling the system, thereby mitigating risks and ensuring a controlled rollout.
2. **Invest in Centralized CI/CD:** Prioritizing the establishment of a robust, centralized Continuous Integration/Continuous Deployment (CI/CD) pipeline is essential. This automation is critical for managing deployments, ensuring consistency across diverse environments, and enabling rapid, reliable software releases, significantly reducing manual effort and potential errors.
3. **Strategic LLM Hardware Planning:** It is imperative to recognize that the Large Language Model (LLM) component will be the primary driver of specialized hardware costs and architectural complexity. Organizations must carefully evaluate the chosen LLM model sizes and their performance requirements. For high-scale production, exploring and potentially adopting advanced LLM serving frameworks like vLLM is recommended to optimize resource utilization and manage escalating hardware investments.
4. **Prioritize Robust Networking:** Ensuring a high-speed, low-latency network infrastructure is paramount, especially for inter-node communication within distributed clusters. Network performance is a critical factor for the overall system's throughput and reliability, particularly for data-intensive components like Kafka and MinIO, and for distributed LLM inference.
5. **Implement Comprehensive Observability:** From the UAT phase onwards, deploy centralized logging and advanced monitoring solutions. This will provide deep, real-time insights into system health, performance metrics, and potential issues across all distributed components, enabling proactive problem identification and efficient root cause analysis.
6. **Enforce Layered Security:** A multi-layered security approach must be consistently applied across all environments. This includes adapting firewall rules for distributed setups, enforcing end-to-end SSL/TLS encryption, implementing stringent access controls, and conducting regular security audits to maintain a strong security posture.
7. **Develop Proactive DR Plans:** Detailed, component-specific backup and disaster recovery plans with clearly defined Recovery Point Objectives (RPO) and Recovery Time Objectives (RTO) are fundamental. Regular testing of these plans is crucial to safeguard against data loss, ensure business continuity, and validate the system's resilience in the face of unforeseen events.

By diligently following this detailed project plan, organizations can successfully deploy a powerful and resilient AI document processing system capable of meeting the demanding requirements of an enterprise environment.

#### Works cited

1. Step-by-Step Installation and Configuration Guide\_.docx
2. Step-by-Step Guide to Using Ollama: Local LLM Inference Made Easy | by Aman Shekhar, accessed on August 3, 2025, <https://shekhar14.medium.com/step-by-step-guide-to-using-ollama-local-llm-inference-made-easy-afba037f7a94>
3. Kafka Architecture - GeeksforGeeks, accessed on August 3, 2025, <https://www.geeksforgeeks.org/apache-kafka/kafka-architecture/>
4. Apache Tika – Apache Tika, accessed on August 3, 2025, <https://tika.apache.org/>
5. Elestio managed service for Apache Tika | Elest.io, accessed on August 3, 2025, <https://elest.io/open-source/tika>
6. MinIO High Performance Object Storage — MinIO Object Storage ..., accessed on August 3, 2025, <https://min.io/docs/minio/kubernetes/upstream/operations/concepts/architecture.html>
7. Core Operational Concepts — MinIO Object Storage for Linux, accessed on August 3, 2025, <https://min.io/docs/minio/linux/operations/concepts.html>
8. High Availability HSTS Cluster With Redis Database - IBM, accessed on August 3, 2025, <https://www.ibm.com/docs/en/ahts/4.3?topic=linux-high-availability-hsts-cluster-redis-database>
9. Scale with Redis Cluster | Docs, accessed on August 3, 2025, <https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/>
10. Building Interactive Data Apps with FastAPI, Pydantic, and Streamlit: A Real-World Example, accessed on August 3, 2025, <https://jeftaylo.medium.com/building-interactive-data-apps-with-fastapi-pydantic-and-streamlit-a-real-world-example-b27c68646a5f>
11. FastAPI in Production: Build, Scale & Deploy – Series A: Codebase ..., accessed on August 3, 2025, <https://dev.to/mrchike/fastapi-in-production-build-scale-deploy-series-a-codebase-design-ao3>
12. My Experience Building A FastAPI + Streamlit App - Pybites, accessed on August 3, 2025, <https://pybit.es/articles/my-experience-building-a-fastapi-streamlit-app/>
13. Streamlit • A faster way to build and share data apps, accessed on August 3, 2025, <https://streamlit.io/>
14. Using nginx as HTTP load balancer, accessed on August 3, 2025, <http://nginx.org/en/docs/http/load_balancing.html>
15. Understanding NGINX: Architecture, Configuration & Alternatives | Solo.io, accessed on August 3, 2025, <https://www.solo.io/topics/nginx>
16. Apache Kafka cluster and components - IBM, accessed on August 3, 2025, <https://www.ibm.com/docs/en/oala/1.3.7?topic=components-apache-kafka-cluster>
17. Is Ollama ready for Production? - Collabnix, accessed on August 3, 2025, <https://collabnix.com/is-ollama-ready-for-production/>
18. Using Ollama in Production: A Developer's Practical Guide - Collabnix, accessed on August 3, 2025, <https://collabnix.com/using-ollama-in-production-a-developers-practical-guide/>
19. PostgreSQL High Availability Basics: Understanding Architecture ..., accessed on August 3, 2025, <https://www.enterprisedb.com/blog/postgresql-high-availability-basics-understanding-architecture-and-3-common-patterns>
20. PostgreSQL High Availability Options: A Guide - Yugabyte, accessed on August 3, 2025, <https://www.yugabyte.com/postgresql/postgresql-high-availability/>
21. Best practices for running a security scan in a production environment, accessed on August 3, 2025, <https://help.hcl-software.com/appscan/Enterprise/10.4.0/topics/c_best_practices_production_scan.html>
22. Distributed Inferencing across multiple machines | GoPenAI, accessed on August 3, 2025, <https://blog.gopenai.com/unlocking-the-power-of-distributed-inferencing-with-vllm-ae8a41f42fef>
23. Recommended Hardware for Running LLMs Locally - GeeksforGeeks, accessed on August 3, 2025, <https://www.geeksforgeeks.org/deep-learning/recommended-hardware-for-running-llms-locally/>
24. Lenovo LLM Sizing Guide, accessed on August 3, 2025, <https://lenovopress.lenovo.com/lp2130-lenovo-llm-sizing-guide>
25. How to Run LLMs Locally with Ollama AI - GPU Mart, accessed on August 3, 2025, <https://www.gpu-mart.com/blog/run-llms-with-ollama>
26. 10 Best Log Monitoring Tools in 2025 | Better Stack Community, accessed on August 3, 2025, <https://betterstack.com/community/comparisons/log-monitoring-tools/>
27. Enterprise Server Log Management System - SolarWinds, accessed on August 3, 2025, <https://www.solarwinds.com/log-analyzer/use-cases/enterprise-log-management>
28. Centralizing CI/CD Pipeline Logic for Microservices Architecture - evoila GmbH, accessed on August 3, 2025, <https://evoila.com/blog/centralizing-ci-cd-pipeline-logic-for-microservices-architecture/>
29. CI/CD for microservices - Azure Architecture Center | Microsoft Learn, accessed on August 3, 2025, <https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd>
30. Database Backups and Disaster Recovery in PostgreSQL - TigerData, accessed on August 3, 2025, <https://www.tigerdata.com/blog/database-backups-and-disaster-recovery-in-postgresql-your-questions-answered>
31. PostgreSQL Backup Strategies for Enterprise-Grade Environments - Percona, accessed on August 3, 2025, <https://www.percona.com/blog/postgresql-backup-strategy-enterprise-grade-environment/>
32. Running Kafka in Production with Confluent Platform | Confluent ..., accessed on August 3, 2025, <https://docs.confluent.io/platform/current/kafka/deployment.html>
33. Apache Kafka Backup: Everything You Need To Know - Confluent, accessed on August 3, 2025, <https://www.confluent.io/learn/kafka-backup/>
34. DR for Kafka Cluster : r/apachekafka - Reddit, accessed on August 3, 2025, <https://www.reddit.com/r/apachekafka/comments/1i93cmi/dr_for_kafka_cluster/>
35. Site Failure Recovery — MinIO Object Storage for Container, accessed on August 3, 2025, <https://min.io/docs/minio/container/operations/data-recovery/recover-after-site-failure.html>
36. Kafka Hardware Requirements - Medium, accessed on August 3, 2025, <https://medium.com/@akash.d.goel/kafka-hardware-requirements-9328886fe88f>
37. Recommended Hardware & Configuration | MinIO, accessed on August 3, 2025, <https://www.min.io/product/reference-hardware>
38. Hardware Checklist — MinIO Object Storage for Kubernetes, accessed on August 3, 2025, <https://min.io/docs/minio/kubernetes/upstream/operations/checklists/hardware.html>
39. Hardware requirements - AI Computer Vision, accessed on August 3, 2025, <https://docs.uipath.com/ai-computer-vision/standalone/2023.4/user-guide/hardware-requirements>
40. Minimum System Requirements for PostgreSQL | Galaxy, accessed on August 3, 2025, <https://www.getgalaxy.io/learn/glossary/how-to-meet-minimum-system-requirements-in-postgresql>
41. Requirements summary | Docs - Redis, accessed on August 3, 2025, <https://redis.io/docs/latest/integrate/redis-data-integration/installation/reqsummary/>
42. Apache Tika Environment Setup - Tutorialspoint, accessed on August 3, 2025, <https://www.tutorialspoint.com/tika/tika_environment.htm>
43. Apache Tika Quick Guide - Tutorialspoint, accessed on August 3, 2025, <https://www.tutorialspoint.com/tika/tika_quick_guide.htm>
44. PostgreSQL configuration recommendations - Radiant Logic Documentation, accessed on August 3, 2025, <https://developer.radiantlogic.com/ia/descartes/best-practice/02-databases/01-postgres-recommendations/>
45. Configuring disaster recovery - BMC Documentation, accessed on August 3, 2025, <https://docs.bmc.com/xwiki/bin/view/IT-Operations-Management/On-Premises-Deployment/BMC-Helix-IT-Operations-Management-Deployment/itomdeploy242/Administering/Configuring-disaster-recovery/>