TenderIQ – Enterprise Tender Intelligence System

6-Week Internship Prototype

---

1. Glossary of Terms

Term	Definition
API	Application Programming Interface; a set of rules allowing software components to communicate
Chunking	The process of breaking down large documents into smaller, manageable pieces for processing
Embeddings	Vector representations of text that capture semantic meaning, used for similarity search
FAISS	Facebook AI Similarity Search; library for efficient similarity search of dense vectors
FastAPI	High-performance Python web framework for building APIs
GPT4All	Open-source local language model that can run on consumer hardware
Gradio	Python library for creating customizable UI components for machine learning models
LangChain	Framework for developing applications powered by language models
LLM	Large Language Model; AI models trained on vast amounts of text data capable of generating human-like text
llama.cpp	C++ implementation of Meta's LLaMA model optimized for CPU inference
PyMuPDF	Python binding for MuPDF, used for PDF document parsing
python-docx	Python library for creating and updating Microsoft Word (.docx) files
RAG	Retrieval-Augmented Generation; technique that enhances LLM responses with retrieved context
RFP/RFQ	Request for Proposal/Quotation; formal documents soliciting supplier information and pricing
SentenceTransformers	Framework for state-of-the-art sentence and text embeddings
SQLite	Self-contained, serverless SQL database engine
Streamlit	Python library for turning data scripts into shareable web apps
Tender	A formal written offer to contract goods or services at a specified cost or rate
Token	The smallest unit of text that an LLM processes (roughly 4 characters in English)
Vector Index	Database structure optimized for similarity search of vector embeddings

2. Project Context and Objectives

Context: United Telecoms Ltd. regularly responds to complex tender documents (RFPs, RFQs) that span eligibility criteria, timelines, technical specifications, compliance rules, and submission formats. Manually reviewing and extracting actionable information is time-consuming and error-prone.

Primary Objectives:

1. **Automate** the ingestion and semantic analysis of large tender documents.
2. **Enable** interactive, natural-language Q&A over uploaded tenders, grounded in source text.
3. **Extract** key tasks or requirements (deadlines, deliverables, mandatory criteria) for easy tracking.
4. **Deliver** a fully offline, open-source prototype runnable on standard laptops within 6 weeks.

---

3. Detailed Requirements

3.1 Use Cases and Application Context

TenderIQ addresses real-world challenges faced by organizations responding to complex tender documents. Below are key use cases and application scenarios:

Why TenderIQ is Needed

Organizations typically face these challenges when responding to tenders:

1. **Information Overload**: Government and enterprise tenders often span 100+ pages with critical requirements buried within dense text.

2. **Tight Deadlines**: Bid teams typically have 2-4 weeks to respond, with limited time to thoroughly analyze documents.

3. **Critical Oversight Risk**: Missing a single requirement or deadline can disqualify an otherwise strong proposal.

4. **Resource Constraints**: Small-to-medium businesses lack dedicated tender specialists to analyze documents.

Primary Use Cases

1. **Initial Tender Assessment**

- **Scenario**: A business development manager receives a 150-page RFP and needs to quickly determine if it's worth pursuing.
- **TenderIQ Usage**: Upload document, ask "What are the key qualification requirements?" and "What is the submission timeline?"
- **Value**: Enables rapid go/no-go decision based on accurate information without reading the entire document.

2. **Compliance Verification**

   - **Scenario**: Proposal writer needs to ensure all mandatory requirements are addressed.
   - **TenderIQ Usage**: Query specific compliance items like "What certifications are required?" or "Are there any financial prerequisites?"
   - **Value**: Creates comprehensive compliance checklist with source references to the original document.

3. **Task Management**

   - **Scenario**: Project manager coordinating multiple team members for bid response.
   - **TenderIQ Usage**: Extract and prioritize all action items, deadlines, and deliverables from the document.
   - **Value**: Converts dense text into structured tasks that can be assigned and tracked.

4. **Specific Information Retrieval**

   - **Scenario**: Technical writer needs details about integration requirements buried in the document.
   - **TenderIQ Usage**: Ask targeted questions like "What APIs need to be supported?" or "What are the performance SLAs?"
   - **Value**: Retrieves precise information with page references in seconds versus hours of manual searching.

Real-World Example

Consider a mid-sized software company responding to a government healthcare tender:

1. Business Development uploads a 200-page RFP to TenderIQ
2. System processes document, creating searchable knowledge base
3. Team asks questions about eligibility, deadlines, budget, and technical requirements
4. TenderIQ extracts 45 distinct tasks with deadlines and responsibilities
5. Project manager assigns tasks to team members through the system
6. Throughout the 3-week response period, team members query specific sections as they work
7. Final proposal addresses all requirements with direct references to the original RFP

This integrated approach helps the company submit a fully compliant bid on time, despite limited resources and a complex tender document.

3.2 Functional Requirements

Each requirement includes specific success criteria to help determine when it has been successfully implemented:

Requirement	Description	Success Criteria
Document Ingestion	Accept PDF and DOCX uploads via a user interface.	• UI includes upload button/form • System accepts files up to 10MB • Confirms successful upload to user • Stores files in appropriate directory
Text Extraction & Chunking	Parse each document into coherent text chunks preserving context.	• Extracts 95%+ of visible text from PDF/DOCX • Creates chunks of 500-1,000 tokens • Preserves paragraph/section context • Processes document in <60 seconds
Embedding & Indexing	Compute semantic embeddings for chunks and create vector index.	• Successfully generates embeddings for all chunks • Creates persistent vector index • Embedding process completes in <2 minutes • Index supports similarity search
Retrieval-Augmented Q&A	Retrieve relevant chunks and generate LLM answers.	• Query returns top-k most relevant chunks • LLM answer generated in <10 seconds • Answers directly address user queries • Response uses information from retrieved chunks
Source Reference	Display source snippet/metadata with answers.	• Each answer includes originating text • Source includes page/section when available • UI presents source in distinguishable format
Task Extraction	Identify key actions or requirements from tenders.	• Extracts dates, deadlines, requirements • Lists tasks in structured format • Captures at least 80% of critical items • Minimal false positives
User Interface	Provide chat interface for uploading, querying, and viewing.	• Clean, responsive UI design • Upload, chat, and task view functions • Works in modern browsers • Intuitive controls with minimal instructions needed

3.3 Multi-Document Management

TenderIQ implements a "tender project" concept to effectively manage multiple related documents and their relationships. This approach addresses two critical real-world scenarios in tender management:

Tender Project Concept

A tender project represents a logical grouping of all documents related to a specific tender opportunity. This includes:

- The main tender document (RFP/RFQ)
- Amendments and addendums
- Clarification documents
- Q&A documents published by the issuing authority
- Any other supporting materials

This grouping ensures that all related documents are treated as a cohesive unit, with proper versioning and relationship tracking between them.

Detailed Examples

Example 1: City Metro Rail Expansion Tender

Consider a tender project for a city's metro rail expansion:

1. **Main Document (March 1, 2025):**
    - "Metro Rail Expansion RFP 2025-003"
    - Contains original submission deadline: April 15, 2025
    - Specifies steel requirements: "Grade A steel meeting ISO 9001:2015 standards"
    - States payment terms: "30% advance, 50% upon milestone completion, 20% after final inspection"
2. **Amendment 1 (March 20, 2025):**
    - "Amendment to RFP 2025-003"
    - Extends submission deadline to May 1, 2025
    - No changes to other requirements
3. **Clarification Document (March 25, 2025):**
    - "Responses to Vendor Queries for RFP 2025-003"
    - Clarifies inspection procedures but doesn't modify requirements
4. **Amendment 2 (April 10, 2025):**
    - "Second Amendment to RFP 2025-003"
    - Updates steel requirements: "Grade A+ steel meeting ISO 9001:2018 standards"
    - Modifies payment terms: "20% advance, 60% upon milestone completion, 20% after final inspection"

When a user asks: "What is the submission deadline for the Metro Rail project?", TenderIQ:

1. Identifies this as a tender-specific query for the "Metro Rail Expansion" project
2. Retrieves the original deadline (April 15) from the main document
3. Detects the updated deadline (May 1) from Amendment 1
4. Prioritizes the newer information
5. Returns: "The submission deadline for the Metro Rail Expansion project is May 1, 2025. Note: This was extended from the original April 15 deadline per Amendment 1 dated March 20, 2025."

Example 2: Hospital Equipment Supply Tender

A tender project for hospital equipment acquisition:

1. **Main Document (January 15, 2025):**
    - "General Hospital Equipment Supply Tender 2025-H01"
    - Lists 50 equipment items with specifications
    - Requires delivery within 90 days of contract award
2. **Q&A Document (February 5, 2025):**
    - Contains answers to vendor questions
    - Clarifies warranty requirements
3. **Amendment (February 20, 2025):**
    - Removes 5 equipment items from the original list
    - Adds 3 new equipment items with specifications
    - Extends delivery timeline to 120 days

When the user asks: "What equipment is required for the Hospital tender?", the system:

1. Combines the original list of 50 items minus the 5 removed items plus the 3 new items
2. References both the original document and amendment
3. Returns the current consolidated list of 48 items with their specifications

Query Types and Examples

Tender-Specific Queries:

These queries target information within a single tender project and require context from all its related documents:

- **Example Query:** "What are the current payment terms for the Metro Rail project?"
    - **System Process:**
        1. User selects "Metro Rail Expansion" project before querying
        2. System retrieves original payment terms (30/50/20)
        3. Identifies updated terms in Amendment 2 (20/60/20)
        4. Returns: "Current payment terms are 20% advance, 60% upon milestone completion, and 20% after final inspection, as specified in Amendment 2 (April 10, 2025). This supersedes the original terms of 30/50/20 split."
- **Example Query:** "What safety standards must be met for the Hospital Equipment tender?"
    - **System Process:**
        1. Searches across all documents in the Hospital Equipment project
        2. Consolidates safety requirements from main document and any amendments
        3. Returns comprehensive list with source references

General Cross-Tender Queries:

These queries analyze information across multiple tender projects to provide broader insights:

- **Example Query:** "What is the average submission timeline for tenders published in 2025?"
    - **System Process:**
        1. User does not select a specific project, indicating a general query
        2. System calculates days between publication and submission dates across all 2025 tenders
        3. Returns: "The average submission timeline is 42 days, based on 7 tenders published in 2025. Individual timelines range from 30 to 60 days."
- **Example Query:** "Which tenders have had their deadlines extended this quarter?"
    - **System Process:**
        1. Identifies all tender projects with amendments modifying submission dates
        2. Filters to current quarter
        3. Returns list of affected tenders with original and extended dates

Rationale

This approach is essential because:

1. **Document Relationships:** Tender amendments often modify or supersede information in the original document, as shown in the Metro Rail example where amendment 2 significantly changed steel requirements and payment terms
2. **Versioning:** Latest information must take precedence when answering queries, ensuring users get the current submission deadline (May 1) rather than the outdated one (April 15)
3. **Context Preservation:** Questions about a specific tender require context from all its related documents to provide accurate answers about current requirements
4. **Cross-Tender Analytics:** Organizations need insights across multiple tenders to identify patterns like average submission timelines and amendment frequency

Implementation Details

Structural Requirements:

- Each document is associated with exactly one tender project (e.g., "Metro Rail Expansion" or "Hospital Equipment")
- Documents have a type designation (main, amendment, clarification, Q&A) with clear relationships
- Documents include metadata like version number and publication date to establish precedence
- The system maintains relationship links between related documents to trace requirement changes

User Interface Workflow:

1. User first selects an existing tender project or creates a new one
2. When uploading documents, user specifies the document type and its relationship to existing documents
3. Query interface allows selecting a specific tender project or querying across all tenders
4. Query results clearly indicate source documents and whether information has been superseded

System Logic:

1. **Context-Aware Retrieval:** When querying a specific tender, the system retrieves context from all related documents
2. **Chronological Prioritization:** Information from newer documents (amendments) takes precedence over older versions
3. **Conflict Resolution:** When contradictions exist between documents, the system notes the discrepancy and prioritizes the most recent authoritative source
4. **Cross-Project Aggregation:** For general queries across all tenders, the system indicates which tender each piece of information comes from

3.4 Non-Functional Requirements

- **Offline Operation:** No external APIs or cloud services; all components run locally.
- **Hardware Constraints:** Must function on standard intern laptops (4–8 CPU cores, 8–16 GB RAM).
- **Rapid Development:** Leverage high-level frameworks and prebuilt components to meet the 6-week deadline.
- **Simplicity & Maintainability:** Single service architecture; minimal dependencies; clear code organization.

---

4. System Architecture

The TenderIQ system employs a layered architecture designed to process tender documents efficiently while maintaining clear separation of concerns and data flow. At its core, the architecture follows a pipeline model where documents move through sequential processing stages—from initial upload through parsing, embedding, and storage—to create a searchable knowledge base. This design prioritizes offline operation, minimizes dependencies, and maximizes performance on constrained hardware, aligning with the project's non-functional requirements.

The system is composed of six interconnected layers. The User Interaction Layer provides intuitive interfaces for document upload, question asking, and results viewing. The Backend Service acts as an orchestrator, routing requests, coordinating processing pipelines, and managing data operations. Three specialized pipelines—Document Processing, Query Processing, and Task Extraction—handle the core intelligence functions. These use local models to transform raw documents into searchable knowledge and extract insights. The Data Storage Layer maintains persistent records across five stores, ensuring efficient data access patterns while minimizing redundancy. This architecture balances rapid development needs with system robustness through a single-service design that combines dependency isolation with comprehensive logging and error handling, enabling efficient troubleshooting and maintenance.

Below is the high-level data flow for TenderIQ:

flowchart LR subgraph "1. User Interaction Layer" U[User] -->|1a. Create/Select Project| UI["1a-d. UI: Streamlit/Gradio"] U -->|1b. Upload Document| UI U -->|1c. Ask Question| UI UI -->|1d. Display Results| U end subgraph "2. Backend Service" UI -->|API Request| Backend["2. Backend: FastAPI/Flask"] ProjManager["2b. Project Manager"] <--> Backend Backend -->|3a. Parse & Chunk| Parse["3a. PyMuPDF/python-docx + LangChain Splitter"] Backend -->|3c. Embed| Embed["3c. SentenceTransformers"] Backend -->|3d. Index| Index["3d. FAISS/Chroma"] Backend -->|4a. Query Embedding| Embed Backend -->|4b. Vector Retrieval| Index Index -->|Return Chunks| Backend subgraph "4. Query Processing Pipeline" Backend -->|4c. Context Assembly| Context["4c. Context Assembly"] Context -->|4d. Prompt Construction| Prompt["4d. Prompt Construction"] Prompt -->|4e. Process| LLM["4e. Local LLM: llama.cpp or GPT4All"] LLM -->|4f. Format Response| Backend VersionResolver["4g. Version Conflict Resolver"] <--> Context end Backend -->|5a. Task Identification| Tasks["5. Hybrid: LLM + Rule-based"] Tasks -->|5b. Format Tasks| Backend Monitor["2a. Logging & Monitoring"] <--> Backend Cache["Cache System"] <-->|Store/Retrieve| Backend end subgraph "6. Data Storage Layer" ProjectStore[("6a. ProjectStore\n(Tender Projects)")] DocStore[("6b. DocStore\n(Uploaded Files)")] ChunkStore[("6c. ChunkStore\n(Text Chunks + Metadata)")] VectorDB[("6d. VectorDB\n(Vector Embeddings)")] TaskStore[("6e. TaskStore\n(Extracted Tasks)")] CacheDB[("6f. CacheDB\n(Query Cache)")] end UI -->|Project Operations| ProjectStore ProjectStore -->|Document Relationships| DocStore ProjManager <-->|Manage| ProjectStore UI -->|Upload with Context| DocStore Parse --> ChunkStore Parse --> DocStore Embed --> VectorDB Tasks --> TaskStore Backend -->|Response| UI Cache --> CacheDB

4.1 Data Flow Detailed Explanation

The architecture follows a layered approach with numbered components that align with the architecture diagram shown above:

1. **User Interaction Layer:**
   - **1a. Project Selection/Creation:** User selects an existing tender project or creates a new one
   - **1b. Document Upload:** User uploads tender document (PDF/DOCX) with project context through the Streamlit/Gradio UI
   - **1c. Question Asking:** User submits natural language questions about tender content with project scope
   - **1d. Results Viewing:** UI displays answers, source references, and extracted tasks
2. **Backend Service:**
   - Acts as the central API gateway (FastAPI/Flask) handling all requests
   - Coordinates between processing pipelines and data stores
   - **2a. Logging & Monitoring:** Captures application events, errors, and performance metrics
   - **2b. Project Manager:** Handles tender project lifecycle, document relationships, and version tracking
3. **Document Processing Pipeline:**
   - **3a. Parse & Chunk:** PyMuPDF/python-docx extracts text; LangChain splitter divides into chunks
   - **3b. Text Chunks:** Structured text segments with metadata (source location, page numbers, project context)
   - **3c. Embedding:** SentenceTransformers model converts text chunks to 384-dim vectors
   - **3d. Index Creation:** Organizes embeddings for efficient similarity search with project metadata
4. **Query Processing Pipeline:**

  - **4a. Query Embedding:** Transforms user question into vector representation
  - **4b. Vector Retrieval:** Searches vector index for semantically similar chunks within project scope
  - **4c. Context Assembly:** Combines retrieved chunks with metadata from related documents
  - **4d. Prompt Construction:** Formats question and context into LLM-optimized prompt
  - **4e. LLM Processing:** Local model (llama.cpp/GPT4All) generates answer from prompt
  - **4f. Response Formatting:** Structures answer with source references for display
  - **4g. Version Conflict Resolver:** Identifies and resolves contradictions between documents, prioritizing newer information

5. **Task Extraction Pipeline:**

  - **5a. Task Identification:** Uses hybrid approach combining LLM and rule-based techniques to extract action items
  - **5b. Task Formatting:** Structures tasks with metadata (deadlines, requirements, priority, document source)

6. **Data Storage Layer:**

  - **6a. ProjectStore:** Repository of tender projects and their metadata
  - **6b. DocStore:** Original uploaded documents (PDF/DOCX) with project associations
  - **6c. ChunkStore:** Processed text chunks with metadata including project context
  - **6d. VectorDB:** FAISS/Chroma index of document embeddings with project filtering capability
  - **6e. TaskStore:** Structured repository of extracted tasks and requirements
  - **6f. CacheDB:** Cache of frequent queries and responses for performance optimization

Additional System Features:

- **Error Handling:** Comprehensive error logging and graceful failure recovery
- **Performance Optimization:** Response caching to improve frequently asked questions
- **Data Persistence:** All processed data persisted to disk for session continuity

Complete Flow: Document Upload → Processing Pipeline → Embedding & Indexing → Question Input → Retrieval → Context Assembly → LLM Answer Generation → Response Formatting → Display → Task Extraction (Parallel)

---

5. Design and Technical Stack

5.1 Technology Components

Component	Selected Technologies	Description
Language & Framework	Python 3.10+	Core programming language for all components
Backend	FastAPI (or Flask) with Uvicorn	Serving REST endpoints for document processing and Q&A
Document Parsing & Chunking	PyMuPDF (fitz) python-docx LangChain's RecursiveCharacterTextSplitter	PDF text extraction DOCX extraction Breaking text into ~500-1,000-token chunks
Embeddings & Semantic Search	SentenceTransformers ("all-MiniLM-L6-v2") FAISS-CPU or ChromaDB	384-dim offline embeddings Nearest-neighbor lookup
Retrieval-Augmented Q&A	LangChain's RetrievalQA chain	Wiring: query embedding → vector search → retrieve top-k chunks → local LLM prompt
Local LLM	llama.cpp with 7B 4-bit quantized LLaMA-2 model (Alternative) GPT4All	Primary local model Easier "drop-in" alternative
Frontend UI	Streamlit or Gradio	Single-page, chat-style interface with file upload and answer display
Data Storage	Local filesystem SQLite or simple JSON	For uploaded docs & extracted chunks For persisting metadata and extracted task lists
Dev & Ops	Git (GitHub/GitLab) Virtual environments (venv/conda) or Docker Python's logging	Source control Environment isolation Basic error handling

5.2 Installation & Dependencies

```
# Core dependencies
pip install fastapi uvicorn pymupdf python-docx langchain

# Embedding & Vector Search
pip install sentence-transformers faiss-cpu
# OR: pip install sentence-transformers chromadb

# LLM Runtime
pip install llama-cpp-python
# OR: pip install gpt4all

# Frontend
pip install streamlit
# OR: pip install gradio

# Optional: Environment Management
pip install virtualenv
# OR: conda create -n tenderiq python=3.10
```

5.3 Component Integration

The technology stack enables a complete end-to-end workflow:

1. **Document Intake**: User uploads PDF/DOCX through Streamlit/Gradio → FastAPI backend receives files
2. **Processing Pipeline**: PyMuPDF/python-docx extracts text → LangChain splitter creates chunks
3. **Knowledge Base Creation**: SentenceTransformers embeds chunks → FAISS/Chroma indexes vectors
4. **Query Processing**: User asks question → query embedded → top chunks retrieved → llama.cpp/GPT4All generates answer
5. **Result Presentation**: Answer with source references and extracted tasks displayed in UI

All components run locally without external dependencies, ensuring data privacy and offline operation.

5.4 Hardware Requirements

The following hardware specifications are recommended for smooth operation of the TenderIQ prototype:

Component	Minimum Requirements	Recommended Specifications	Notes
CPU	4 cores, 2.5 GHz	8+ cores, 3.5+ GHz	LLM inference is CPU-intensive; more cores significantly improve performance
RAM	8 GB	16 GB	4-5 GB needed for quantized 7B model alone; additional RAM for embeddings and application
Storage	10 GB free	20+ GB free	~4 GB for LLM model, plus space for document storage and vector indices
GPU	Not required	Integrated or basic CUDA-compatible	Optional acceleration for embedding generation
OS	Windows 10+, macOS 12+, Ubuntu 20.04+	Same	Cross-platform compatibility with Python 3.10+

Note that performance will vary significantly based on hardware. On minimum-spec machines, LLM inference may take 10-30 seconds per response, while recommended hardware can achieve 2-10 second response times depending on prompt length and complexity.

6. Implementation Plan (6 Sprints)

Team Roles

- **Intern 1 – Backend/NLP Engineer:** Document parsing, chunking, embedding workflow, vector index.
- **Intern 2 – Integration Engineer:** REST API, orchestrate modules, vector search, LLM integration.
- **Intern 3 – Frontend/UI Developer:** Streamlit/Gradio interface, file upload, chat display, task checklist.

Sprint Breakdown

Sprint	Timeline	Focus & Goals	Time Allocation	Deliverables	Acceptance Criteria

Sprint	Timeline	Focus & Goals	Time Allocation	Deliverables	Acceptance Criteria
1: Project Setup & Planning	Week 1	• Define scope & roles • Repo & environment setup • Basic API + UI scaffolding • Empty FastAPI/Flask and Streamlit/Gradio app running	• Planning: 1 day • Environment setup: 1 day • API scaffold: 1.5 days • UI scaffold: 1.5 days	• Git repo with README • Environment setup docs • Project structure	• Both UI and backend "Hello World" run locally • All devs able to run code from fresh clone • README documents tech stack and setup
2: Document Ingestion & Parsing	Week 2	• File upload endpoint • PDF/DOCX text extraction • Text chunking implementation • Storage of extracted chunks	• Upload endpoint: 1 day • PDF/DOCX parsing: 2 days • Chunking: 1 day • Storage: 1 day	• Upload form in UI • Backend file reception • Text extraction & chunking	• System accepts PDF & DOCX files • Extracts and chunks text correctly • UI confirms successful upload • Documents stored with metadata
3: Embeddings & Vector Indexing	Week 3	• Integrate SentenceTransformers • Set up FAISS/Chroma index • Index parsed chunks • Query interface	• Embedding setup: 1 day • Vector DB setup: 2 days • Integration: 2 days	• Auto embedding process • Populated Vector DB • Query API endpoint	• Chunks automatically embedded • Vector search works correctly • Search returns relevant results • Performance meets requirements
4: Retrieval & LLM Q&A	Week 4	• Implement retrieval function • Wire LLM inference • Basic QA API • Prompt engineering	• Retrieval: 1 day • LLM setup: 1.5 days • API endpoint: 1.5 days • Prompt design: 1 day	• /ask endpoint • Query→answer demo • Basic prompt templates	• LLM answers questions accurately • Response time acceptable • Answers reference source content • System handles edge cases
5: Frontend Chat & Source Reference	Week 5	• Chat interface • Display answers with sources • Task extraction • Error handling	• Chat UI: 1.5 days • Source display: 1 day • Task extraction: 2 days • Error handling: 0.5 days	• Interactive UI • Source references • Task checklist • Robust user experience	• UI is intuitive and responsive • Sources correctly displayed • Tasks extracted with high accuracy • Graceful failure handling
6: Testing, Polish & Handover	Week 6	• End-to-end testing • Bug fixes & optimization • Documentation & handoff • Final demo preparation	• Testing: 2 days • Bug fixes: 1 day • Documentation: 1 day • Demo prep: 1 day	• Working prototype • User documentation • Developer docs • Demo script	• System passes all test scenarios • No critical bugs remaining • Documentation is comprehensive • Demo shows core capabilities

Project Management

- **Daily Coordination:** 15-minute standup meetings to discuss progress and blockers
- **Sprint Planning:** Half-day session at start of each sprint to assign tasks and set priorities
- **Sprint Review:** Demo session at the end of each sprint to showcase completed work
- **Task Tracking:** Kanban board (Trello or GitHub Projects) to visualize workflow
- **Code Reviews:** Required for all pull requests before merging to main branch
- **Technical Lead:** One team member (rotating by sprint) will serve as the technical point of contact for decision-making and issue resolution

7. Diagrams & Descriptions

7.1 Architecture Overview

```
graph LR %% User Interaction Layer subgraph "1. User Interaction" U[User] -->|1a. Upload Document| UI["UI: Streamlit/Gradio"] U -->|1b. Ask Question| UI UI -->|1c. View Results| U
end %% Backend Services subgraph "2. Backend Service" %% Main API Gateway BE["API: FastAPI/Flask"] -->|2a| Logger["Logging & Monitoring"] %% Document Processing
Pipeline subgraph "3. Document Processing" BE -->|3a| Parse["Parse & Chunk\n(PyMuPDF/python-docx)"] Parse -->|3b| Chunks["Text Chunks + Metadata"] Chunks -->|3c|
Embed["Embedding\n(SentenceTransformers)"] Embed -->|3d| VecIdx["Index Creation"] end %% Query Processing Pipeline subgraph "4. Query Processing" BE -->|4a|
QEmbed["Query Embedding"] QEmbed -->|4b| Retrieve["Vector Retrieval"] Retrieve -->|4c| Context["Context Assembly"] Context -->|4d| Prompt["Prompt Construction"] Prompt --
>|4e| LLM["Local LLM\n(llama.cpp/GPT4All)"] LLM -->|4f| Response["Response Formatting"] end %% Task Extraction Pipeline subgraph "5. Task Extraction" BE -->|5a| TaskExt["Task
Identification\n(LLM/Keyword)"] TaskExt -->|5b| TaskFmt["Task Formatting"] end end %% Data Storage Layer subgraph "6. Data Stores" DocStore[("Uploaded Files")]
ChunkStore[("Text Chunks")] VectorDB[("Vector Index\n(FAISS/Chroma)")] TaskDB[("Tasks & Metadata")] CacheDB[("Response Cache")] end %% Inter-component Connections UI --
>|"API Request"| BE BE -->|"API Response"| UI %% Data Persistence Connections Parse -.->|"Save"| DocStore Parse -.->|"Store"| ChunkStore VecIdx -.->|"Persist"| VectorDB
TaskFmt -.->|"Save"| TaskDB Response -.->|"Optional Cache"| CacheDB %% Query Flow Connections Retrieve -->|"Search"| VectorDB Response -->|"Answer + Sources"| BE
TaskFmt -->|"Structured Tasks"| BE TaskDB -.->|"Load"| UI %% Error Handling BE -->|"Errors"| Logger %% Performance Optimizations BE -->|"Cache Check"| CacheDB CacheDB --
>|"Cache Hit"| BE
```

Figure: End-to-end prototype flow from document ingestion through Q&A and task extraction.

7.2 Component Interaction

This section details the interactions between components in the architecture diagram (section 6.1):

1. User Interaction Layer

- **Document Upload Interface:**

```
# Streamlit example
uploaded_file = st.file_uploader("Upload Tender Document", type=["pdf", "docx"])
if uploaded_file:
    file_path = save_uploaded_file(uploaded_file)
    process_document(file_path)
```

- **Query Interface:** Chat-style input with history tracking and source display
- **Result Visualization:** Formatted answers with highlighted source text and confidence scores

2. Backend Service

- **API Gateway:** RESTful endpoints handling document and query processing

```
# FastAPI example
@app.post("/upload")
async def upload_document(file: UploadFile):
    # Process document upload

@app.post("/ask")
async def process_question(question: Question):
    # Process query and return answer
```

- **Logging & Monitoring:** Structured logging with error tracking

```
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("tenderiq")
```

3. Document Processing Pipeline

- **Parse & Chunk:** Extracts text and splits into semantic units

```
# Document processing flow
if file.endswith('.pdf'):
    text = extract_pdf_text(file_path, pymupdf)
elif file.endswith('.docx'):
    text = extract_docx_text(file_path, python_docx)

chunks = text_splitter.split_text(
    text,
    chunk_size=800,
    chunk_overlap=100
)
```

- **Metadata Extraction:** Preserves source context (page numbers, sections, formatting)
- **Embedding Generation:** Converts text to vector representations

```
# Embedding generation
embeddings = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")
embedded_chunks = embeddings.embed_documents([chunk.text for chunk in chunks])
```

- **Index Creation:** Organizes vectors for efficient retrieval

```
# Vector index creation
vectorDB = FAISS.from_embeddings(
    text_embeddings=embedded_chunks,
    embedding=embeddings,
    metadatas=[chunk.metadata for chunk in chunks]
)
vectorDB.save_local("vector_index")
```

4. Query Processing Pipeline

- **Query Embedding:** Transforms question into vector space
- **Vector Retrieval:** Performs similarity search to find relevant chunks

```
# Retrieval process
question_embedding = embeddings.embed_query(question)
similar_chunks = vector_db.similarity_search_by_vector(
    question_embedding,
    k=5  # Top-k chunks
)
```

- **Context Assembly:** Combines retrieved chunks with metadata

```
# Context assembly
context = "\n\n".join([f"Source: {chunk.metadata['source']}, Page: {chunk.metadata['page']}\n{chunk.text}" for chunk in similar
```

- **Prompt Construction:** Formats query and context for optimal LLM response

```
# Prompt template
prompt = f"""Answer the question based only on the provided context.

Context:
{context}

Question: {question}

Answer:"""
```

- **LLM Processing:** Generates answer using local language model

```
# LLM integration
from llama_cpp import Llama

model = Llama(model_path="models/llama-2-7b-chat.Q4_K_M.gguf")
response = model(prompt, max_tokens=512)
```

- **Response Formatting:** Structures LLM output with source references

5. Task Extraction Pipeline

- **Task Identification:** Detects requirements, deadlines, and deliverables using a hybrid approach

```
# Task extraction using hybrid approach (both methods work together)

# Component 1: LLM-based extraction (comprehensive but may miss specific formats)
task_prompt = f"Extract all tasks, requirements, and deadlines from:\n{document_text}"
tasks_text = model(task_prompt, max_tokens=1024)
llm_tasks = parse_llm_tasks(tasks_text)

# Component 2: Rule-based extraction (targeted for specific patterns)
deadline_pattern = r"due.{0,15}(\d{1,2}[/-]\d{1,2}[/-]\d{2,4})"
requirement_pattern = r"(must|shall|required to)\s+([^.!?;]+)"

# Combine results from both approaches for maximum coverage
all_tasks = merge_task_results(llm_tasks, rule_based_tasks)
```

- **Task Formatting:** Structures tasks with metadata for tracking

```
# Task structuring
tasks = [
    {
        "description": task_description,
        "deadline": deadline if deadline else None,
        "source": {"document": doc_name, "page": page_num},
        "status": "pending"
    }
    for task_description, deadline, doc_name, page_num in extracted_task_data
]
```

6. Data Storage Layer

- **Document Store:** Local filesystem storage for uploaded files
- **Chunk Store:** Structured storage for text segments and metadata
- **Vector Database:** FAISS/Chroma index for similarity search

```
# Vector DB persistence
import os

storage_path = "./data/vector_stores"
os.makedirs(storage_path, exist_ok=True)
vector_db.save_local(f"{storage_path}/{document_id}")
```

- **Task Database:** JSON or SQLite storage for extracted tasks
- **Response Cache:** Optional in-memory or persistent cache for frequent queries

```
# Simple response caching
response_cache = {}

def get_answer(question):
    if question in response_cache:
        return response_cache[question]

    answer = generate_answer(question)  # Process normally
    response_cache[question] = answer   # Cache for future
    return answer
```

7. Integration Points

- **Error Handling:** Comprehensive system for catching and logging failures

```
try:
    process_document(file_path)
except DocumentParsingError as e:
    logger.error(f"Failed to parse document: {e}")
    return {"status": "error", "message": str(e)}
```

- **Performance Monitoring:** Tracking processing times and resource usage
- **UI/Backend Communication:** RESTful API conventions with structured responses

7.3 Sequence Diagram

This sequence diagram maps the chronological flow of interactions between system components during critical user operations. The visualization traces the complete request/response cycle, showing precisely how data moves through the system, which components handle specific processing steps, and how asynchronous operations are coordinated.

sequenceDiagram participant User participant UI participant Backend participant DocProcessor participant Embedder participant VectorDB participant LLM participant TaskExtractor %% Document Upload and Processing flow User->>UI: Upload Document UI->>Backend: POST /upload (multipart/form-data) Backend->>Backend: Log request Backend->>DocProcessor: Process document DocProcessor->>DocProcessor: Parse & chunk text DocProcessor->>Embedder: Generate embeddings Embedder->>VectorDB: Store vectors & metadata Backend->>TaskExtractor: Extract tasks (async) TaskExtractor->>Backend: Return tasks Backend->>UI: Return upload success UI->>User: Display confirmation %% Query and Answer flow User->>UI: Ask Question UI->>Backend: POST /ask (question) Backend->>Backend: Check response cache Note over Backend: Cache Miss Backend->>Embedder: Embed question Embedder->>VectorDB: Search similar chunks VectorDB->>Backend: Return relevant chunks Backend->>Backend: Assemble context Backend-

>>Backend: Construct prompt Backend->>LLM: Generate answer LLM->>Backend: Return response Backend->>Backend: Format response Backend->>Backend: Update cache Backend->>UI: Return answer + sources UI->>User: Display answer & references %% Task View flow User->>UI: View Tasks UI->>Backend: GET /tasks Backend->>UI: Return task list UI->>User: Display task checklist

Key Sequence Flows

1. **Document Upload and Processing**
    - The flow begins with user uploading a tender document
    - Documents are processed asynchronously to allow for immediate user feedback
    - Task extraction happens in parallel with indexing to optimize performance
2. **Question Answering**
    - Shows the complete question-to-answer flow, including caching check
    - Illustrates the context assembly and prompt construction steps
    - Demonstrates how source references flow back to the UI
3. **Task Retrieval**
    - Simple flow for accessing previously extracted tasks
    - Direct database access without heavy processing

This sequence diagram helps developers understand the timing relationships between components, especially for asynchronous operations and parallel processing paths.

7.4 Data Model Diagram

This entity-relationship diagram maps the key data structures in the TenderIQ system and their relationships. The diagram illustrates how data flows through the system and how different entities are persisted and connected.

erDiagram TenderProject ||--|{ Document : contains Document ||--|{ Chunk : contains Document ||--|{ Task : generates TenderProject { string id PK string title string description timestamp creation_date string status } Document { string id PK string tender_project_id FK string filename string filepath string mime_type enum doc_type int version timestamp publication_date int page_count timestamp upload_date string status } Chunk ||--|{ Embedding : has Chunk { string id PK string document_id FK int page_number int chunk_index string text_content string section_header int token_count } Embedding ||--|| VectorIndex : indexed_in Embedding { string id PK string chunk_id FK array vector int dimension string model_name } VectorIndex { string id PK string name string index_type int vector_count int dimension timestamp created_at timestamp updated_at } Task { string id PK string document_id FK string description date deadline string priority string status string source_page string source_text } Query ||--|| Response : generates Query ||--|{ Chunk : retrieves Query { string id PK string text timestamp timestamp array relevant_chunk_ids } Response { string id PK string query_id FK string answer_text float confidence_score array source_chunks timestamp generated_at }

Key Data Entities

1. **Document**
    - Represents an uploaded tender document
    - Contains metadata about the file itself
    - Links to derived chunks and extracted tasks
2. **Chunk**
    - Represents a segment of text from a document
    - Contains source information (page number, section)
    - Links to its embedding vector
3. **Embedding**
    - Vector representation of a text chunk
    - Includes metadata about the embedding model
    - Referenced in vector indices for similarity search
4. **VectorIndex**
    - Collection of embeddings organized for efficient search
    - Metadata about the index itself (type, dimensions)
5. **Task**
    - Extracted requirement or action item
    - Contains deadline and priority information
    - Links back to source document and location
6. **Query & Response**
    - Represents user questions and system answers
    - Links to relevant chunks used for generating responses

- Stores confidence scores and source references

This data model provides a clear blueprint for database design and shows how different pieces of information relate to each other in the system. It's particularly useful for understanding data persistence requirements and implementing the storage layer components.

7.5 UI Wireframes

These wireframes present the key screens of the TenderIQ user interface. The mockups display the layout, components, and interaction paradigms of each primary interface in the system.

7.5.1 Tender Project Management

```
+----------------------------------------------------+
|  TenderIQ - Tender Intelligence System             |
+----------------------------------------------------+
|                                                    |
|  [Projects] [Upload] [Chat] [Tasks]                |
|                                                    |
|  Tender Projects                        [+ New]    |
|  +------------------------------------------------+|
|  | Name           | Status    | Last Updated      ||
|  |----------------|-----------|-------------------|
|  | Network Upgrade| Active    | June 10, 2025     ||
|  | Data Center    | Completed | May 27, 2025      ||
|  | Cloud Services | Active    | June 12, 2025     ||
|  +------------------------------------------------+|
|                                                    |
|  [View Selected]  [Archive]                        |
|                                                    |
+----------------------------------------------------+
```

7.5.2 Project Detail View

```
+----------------------------------------------------+
|  TenderIQ - Tender Intelligence System             |
+----------------------------------------------------+
|                                                    |
|  [Projects] [Upload] [Chat] [Tasks]                |
|                                                    |
|  Project: Network Upgrade RFP                [Edit]|
|  Status: Active                                    |
|  Created: June 3, 2025                            |
|  Description: Network infrastructure upgrade tender|
|                                                    |
|  Documents:                          [+ Add Doc]   |
|  +------------------------------------------------+|
|  | File               | Type      | Date       |Ver||
|  |-------------------|-----------|-----------|---|
|  | RFP-NET-2025.pdf   | Main      | Jun 3      | 1 ||
|  | Clarification-1.pdf| Clarif.   | Jun 8      | - ||
|  | Amendment-A.pdf    | Amendment | Jun 10     | 2 ||
|  +------------------------------------------------+|
|                                                    |
|  [Chat with Project]  [View Tasks]                 |
|                                                    |
+----------------------------------------------------+
```

7.5.3 Document Upload Screen

```
+------------------------------------------------------+
|  TenderIQ - Tender Intelligence System               |
+------------------------------------------------------+
|                                                      |
|  [Projects] [Upload] [Chat] [Tasks]                  |
|                                                      |
|  Upload Document to: Network Upgrade RFP      [▼]   |
|                                                      |
|  Document Type: [Main Document ▼]                   |
|  Version: [2 ▼] (Only for Main/Amendment types)     |
|                                                      |
|   +---------------------------------------------+   |
|   |                                             |   |
|   |         Drop Tender Document Here          |   |
|   |                                             |   |
|   |                    or                       |   |
|   |                                             |   |
|   |          [Select File to Upload]            |   |
|   |                                             |   |
|   +---------------------------------------------+   |
|                                                      |
|   Supported formats: PDF, DOCX                       |
|                                                      |
|   [ Upload Document ]                                |
|                                                      |
+------------------------------------------------------+
```

7.5.2 Q&A Interface

```
+----------------------------------------------------+
|  TenderIQ - Tender Intelligence System           [↺]|
+----------------------------------------------------+
|                                                    |
| [Documents ▼] RFP-2025-01.pdf                      |
|                                                    |
| +--------------------------------------------------+
| |                                                  |
| | > What is the submission deadline?               |
| |                                                  |
| | The submission deadline for this RFP is July     |
| | 15, 2025 at 5:00 PM EST. Late submissions will   |
| | not be accepted.                                 |
| |                                                  |
| | Sources:                                         |
| | - Page 3, Section 2.1: "All responses must be    |
| |   submitted by 5:00 PM EST on July 15, 2025."    |
| | - Page 12, Section 8.4: "The deadline will not   |
| |   be extended under any circumstances."          |
| |                                                  |
| | > What are the key evaluation criteria?          |
| |                                                  |
| | The key evaluation criteria for this tender are: |
| |  - Technical approach (40%)                      |
| |  - Prior experience (25%)                        |
| |  - Cost proposal (20%)                           |
| |  - Team qualifications (15%)                     |
| |                                                  |
| | Sources:                                         |
| | - Page 8, Section 5.2: "Evaluation Criteria"     |
| |                                                  |
| +--------------------------------------------------+
|                                                    |
| [Type your question here...]       [ Ask Question ]|
|                                                    |
+----------------------------------------------------+
```

7.5.3 Tasks View

```
+--------------------------------------------------+
|  TenderIQ - Tender Intelligence System         [↺]|
+--------------------------------------------------+
|                                                  |
| [Documents ▼] RFP-2025-01.pdf       [Chat] [Tasks] |
|                                                  |
| +---------------+  +-------------------------------+
| | TASKS (15)    |  |                               |
| +---------------+  | Task: Submit company profile   |
| |                  |                               |
| | [ ] Submit co.. | Deadline: June 30, 2025         |
| | [x] Register .. | Priority: High                 |
| | [ ] Prepare te. | Status: Pending                |
| | [ ] Financial.. |                                |
| | [x] Sign NDA    | Source: Page 5, Section 3.2    |
| | [ ] Attend br.. | "All bidders must submit a      |
| | [ ] References  | comprehensive company profile  |
| |                 | with detailed experience in     |
| | + Filter       | similar projects."              |
| | + Sort         |                                |
| |                 | [ Complete ] [ Assign ] [ Edit ]|
| | By Priority ▼   |                                |
| | All statuses ▼  +-------------------------------+
| |                                                |
| |                                                |
| +------------------------------------------------+
|                                                  |
| [ Add Manual Task ]     [ Export Tasks to CSV ]    |
|                                                  |
+--------------------------------------------------+
```

These wireframes serve as a visual guide for UI implementation, showing the three primary interfaces:

1. **Document Upload Screen**
   - Simple drag-and-drop or file selection interface
   - List of previously uploaded documents for quick access
   - Clear indication of supported file formats
2. **Q&A Interface**
   - Conversation-style layout with question/answer pairs
   - Source references with page numbers and quoted text
   - Document selector for navigating between uploaded files
3. **Tasks View**
   - Split-panel interface with task list and details view
   - Filtering and sorting options for task management
   - Task metadata including deadline, priority, and source
   - Action buttons for task status management

Frontend developers should implement these screens using Streamlit or Gradio components, maintaining the core functionality while adapting the design to the capabilities of the chosen framework.

7.6 Deployment Architecture

This deployment architecture diagram maps the TenderIQ system components and their deployment configuration in a production environment. The diagram shows infrastructure requirements, component relationships, and operational flow across different deployment layers.

flowchart TB subgraph "Developer Environment" Dev["Developer Workstation"] -->|Push changes| Git["Git Repository\nGitHub/GitLab"] end subgraph "Deployment Server" Git -->|Clone/Pull| Deploy["Deployment Script\npython/bash"] Deploy -->|Configure| AppDir["Application Directory"] end subgraph "Application Container / VM" AppServer["Web Server\nGunicorn/Uvicorn"] -->|WSGI/ASGI| FastAPI["FastAPI Application"] FastAPI -->|Import| ApiModules["API Modules"] FastAPI -->|Import| ProcessingModules["Processing Modules"] FastAPI -->|Import| StorageModules["Storage Modules"] FastAPI -->|Serve static files| Static["Static Assets"] LLMProcess["LLM Process\nllama-cpp-python"] <-->|IPC| FastAPI Streamlit["Streamlit UI"] <-->|API calls| FastAPI end subgraph "Data Persistence" DocStore["Document Store\nFilesystem"] VectorDB["Vector Database\nFAISS/Chroma\nPersisted on Disk"] MetadataDB["Metadata Store\nSQLite/JSON"] ModelFiles["Model Files\nGGUF Format"] end FastAPI -->|Read/Write| DocStore FastAPI -->|Query| VectorDB FastAPI -->|Read/Write| MetadataDB LLMProcess -->|Load| ModelFiles subgraph "External User" User["End User"] -->|HTTP/HTTPS| AppServer AppServer -->|HTTP Response| User end

Key Deployment Components

1. **Developer Environment**
    - Local development workstations for coding and testing
    - Git repository for version control and collaboration
    - CI/CD pipelines can be added later for automated testing and deployment
2. **Deployment Server**
    - Handles the application deployment process
    - Configures environment variables and dependencies
    - Sets up necessary directory structures
3. **Application Container/VM**
    - Single-server deployment model for prototype
    - Web server (Gunicorn/Uvicorn) for handling HTTP requests
    - FastAPI application serving as the central service
    - Separate LLM process to avoid blocking API calls during inference
    - Streamlit UI running as a separate process but on same host
4. **Data Persistence**
    - Simple filesystem-based storage for documents
    - FAISS/Chroma vector indices persisted to disk
    - Lightweight database (SQLite) for metadata and tasks
    - Pre-downloaded model files accessed locally

Deployment Considerations

- **Hardware Requirements:**
    - CPU: 4+ cores (8+ recommended for faster inference)
    - RAM: 8GB minimum (16GB recommended)
    - Storage: 10GB for application + 5-10GB for model + space for documents
- **Networking:**
    - Default ports: 8000 (FastAPI) and 8501 (Streamlit)
    - Internal communication between services via localhost
    - Optional reverse proxy (Nginx) for production deployment
- **Scaling:**
    - Vertical scaling (increasing server resources) is the simplest approach for the prototype
    - Horizontal scaling would require separating components into microservices in future versions
- **Security:**
    - Local deployment with basic authentication for the prototype
    - No external APIs or cloud services required
    - Document access controls can be implemented later

This deployment architecture prioritizes simplicity and ease of setup, aligning with the 6-week prototype timeline. All components run on a single server or developer machine, minimizing deployment complexity while still providing appropriate separation of concerns.

---

8. Getting Started

8.1 Development Environment Setup

```
# Clone the repository
git clone https://github.com/your-org/TenderIQ.git
cd TenderIQ

# Create and activate virtual environment
python -m venv venv

# Windows
venv\Scripts\activate

# macOS/Linux
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt

# Download LLM model (if not using GPT4All which downloads automatically)
mkdir -p models
# Download LLaMA-2-7B-Chat 4-bit quantized model to models directory
# from https://huggingface.co/TheBloke/Llama-2-7B-Chat-GGUF/

# Run the application
python app.py
```

8.2 Project Structure

```
TenderIQ/
├── app.py                    # Main application entry point
├── requirements.txt          # Project dependencies with versions
├── README.md                 # Project overview and setup instructions
├── config.py                 # Configuration parameters
│
├── models/                   # Directory for storing LLM models
│   └── llama-2-7b-chat.Q4_K_M.gguf  # Quantized LLaMA model
│
├── data/                     # Data storage
│   ├── uploads/              # Original uploaded documents
│   ├── chunks/               # Processed text chunks
│   ├── vector_stores/        # Vector indices
│   └── tasks/                # Extracted tasks
│
├── logs/                     # Application logs
│   ├── app.log               # Main application log
│   └── error.log             # Error log
│
├── src/                      # Source code
│   ├── api/                  # Backend API (Layer 2)
│   │   ├── __init__.py
│   │   ├── routes.py         # API endpoint definitions
│   │   ├── upload.py         # Document upload handlers
│   │   ├── query.py          # Question answering handlers
│   │   └── tasks.py          # Task management endpoints
│   │
│   ├── processing/           # Document processing (Layer 3)
│   │   ├── __init__.py
│   │   ├── parser.py         # PDF/DOCX extraction
│   │   ├── chunker.py        # Text chunking
│   │   └── metadata.py       # Metadata extraction
│   │
│   ├── embedding/            # Embedding pipeline (Layer 3)
│   │   ├── __init__.py
│   │   ├── model.py          # SentenceTransformer wrapper
│   │   └── index.py          # Vector index creation
│   │
│   ├── retrieval/            # Query processing (Layer 4)
│   │   ├── __init__.py
│   │   ├── search.py         # Vector similarity search
│   │   ├── context.py        # Context assembly
│   │   └── prompt.py         # Prompt construction
│   │
│   ├── llm/                  # LLM inference (Layer 4)
│   │   ├── __init__.py
│   │   ├── inference.py      # LLaMA/GPT4All wrapper
│   │   └── response.py       # Response formatting
│   │
│   ├── tasks/                # Task extraction (Layer 5)
│   │   ├── __init__.py
│   │   ├── extractor.py      # Task identification logic
│   │   └── formatter.py      # Task formatting and storage
│   │
│   ├── storage/              # Data persistence (Layer 6)
│   │   ├── __init__.py
│   │   ├── documents.py      # Document store interface
│   │   ├── chunks.py         # Chunk store operations
│   │   ├── vectors.py        # Vector DB management
│   │   ├── tasks.py          # Task database operations
│   │   └── cache.py          # Response caching
│   │
│   └── utils/                # Shared utilities
│       ├── __init__.py
│       ├── logging.py        # Logging configuration
│       ├── monitoring.py     # Performance monitoring
│       └── errors.py         # Error handling
│
└── ui/                       # User interface (Layer 1)
    ├── __init__.py
    ├── app.py                # Streamlit/Gradio main app
    ├── pages/                # UI view components
    │   ├── __init__.py
    │   ├── upload.py         # Document upload page
    │   ├── query.py          # Q&A interface
    │   └── tasks.py          # Task management view
    └── components/           # Reusable UI components
        ├── __init__.py
        ├── chat.py           # Chat interface
        └── document_viewer.py # Document preview
```

8.3 Key Documentation References

- **FastAPI Documentation**: https://fastapi.tiangolo.com/ (https://fastapi.tiangolo.com/)
- **Streamlit Documentation**: https://docs.streamlit.io/ (https://docs.streamlit.io/)
- **LangChain Text Splitting**: https://python.langchain.com/docs/modules/data_connection/document_transformers/ (https://python.langchain.com/docs/modules/data_connection/document_transformers/)
- **SentenceTransformers**: https://www.sbert.net/ (https://www.sbert.net/)
- **llama-cpp-python**: https://github.com/abetlen/llama-cpp-python (https://github.com/abetlen/llama-cpp-python)
- **FAISS Documentation**: https://faiss.ai/ (https://faiss.ai/)

9. Risk Management

Risk	Impact	Probability	Mitigation Strategy
Large LLM models exceed local hardware capacity	High	Medium	Use smaller or more quantized models (4-bit instead of 8-bit); set up fallback to rule-based approaches for specific tasks
Document parsing fails on complex PDFs	Medium	Medium	Implement robust error handling; provide feedback to user; build fallback text extraction methods
Slow LLM inference response times	Medium	High	Optimize prompts; set up caching for frequent queries; consider streaming responses; provide clear loading indicators
Inaccurate answers due to retrieval errors	High	Medium	Implement confidence scores; display multiple source chunks; allow users to browse all potentially relevant chunks
Poor RAM utilization causing crashes	High	Low	Implement batch processing; clear memory after document processing; build monitoring to detect memory issues early
Integration issues between components	Medium	Medium	Define clear APIs between modules; use dependency injection; write integration tests for critical paths
Complex tender documents extraction quality	High	Medium	Develop specialized preprocessing for tables, headers, and lists; create test suite with various document formats

9.1 Contingency Planning

If certain components prove too challenging within the 6-week timeframe:

1. **LLM Performance Issues**: Fall back to keyword-based or rule-based extraction for specific tasks
2. **Vector Search Scalability**: Limit document size or implement pagination for processing very large documents
3. **Task Extraction Complexity**: Provide manual tagging interface as backup to automatic extraction
4. **UI Development Constraints**: Focus on functional MVP components over visual polish

10. Testing Strategy

10.1 Testing Framework

Test Level	Tools	Scope	Responsibility
Unit Tests	pytest	Individual functions and classes	Individual developers
Integration Tests	pytest with fixtures	Component interactions	Module owners
End-to-End Tests	pytest & Selenium	Full user journeys	QA lead
Performance Tests	locust.io	API endpoints & critical paths	Technical lead
Usability Testing	Manual testing checklist	UI components	All team members

10.2 Test Coverage Requirements

- **Core Components**: 80%+ unit test coverage for:
    - Document parsing/chunking
    - Embedding generation
    - Retrieval logic
    - Task extraction algorithms
- **API Endpoints**: 70%+ coverage for:
    - Request validation
    - Response formatting
    - Error handling
- **UI**: Comprehensive tests for:
    - Document upload flow
    - Question input/output
    - Task list interaction
    - Error state displays

10.3 Test Data

- Create a diverse test corpus with at least:
    - 3 PDF documents of varying complexity
    - 3 DOCX files with different formatting
    - 1 sample with tables and charts
    - 1 sample with multiple languages

10.4 Test Automation

- Implement GitHub Actions workflow for Continuous Integration
- Run automated tests on each PR
- Include linting checks (flake8, black) in the CI pipeline
- Generate test coverage reports automatically

11. Development Guidelines

11.1 Collaboration Guidelines

Git Workflow

- **Branch Naming Convention**: `[type]/[short-description]`
    - Types: `feature`, `bugfix`, `refactor`, `docs`, `test`, `chore`
    - Example: `feature/document-upload` or `bugfix/pdf-extraction-error`
- **Commit Message Format**: `[type]: [short summary]`
    - Example: `feat: implement PDF text extraction` or `fix: handle malformed PDF files`
- **Pull Request Process**:
    1. Create branch from `develop`
    2. Implement changes with appropriate tests
    3. Submit PR with descriptive title and details
    4. Ensure CI checks pass
    5. Obtain at least one code review approval
    6. Squash and merge into `develop`
- **Release Process**:
    1. Create release branch from `develop` as `release/vX.Y.Z`
    2. Fix only critical bugs in release branch
    3. Merge to `main` and tag with version
    4. Merge release branch back to `develop`

Code Review Guidelines

- **Review Timeframe**: Within 24 hours of PR submission
- **Review Focus Areas**:

    - Functionality: Does it work as intended?
    - Tests: Are there appropriate tests?
    - Code quality: Is the code maintainable?
    - Performance: Any obvious inefficiencies?
    - Security: Any potential vulnerabilities?

11.2 Code Style Standards

Python Style

- **Linter**: flake8 for style checking
- **Formatter**: black with line length of 88 characters
- **Type Hints**: Use Python type hints for function signatures
- **Docstrings**: Google-style docstrings

```
def process_document(file_path: str, chunk_size: int = 500) -> List[DocumentChunk]:
    """
    Process a document file into chunks.

    Args:
        file_path: Path to the document file
        chunk_size: Size of text chunks in characters

    Returns:
        List of document chunks with text and metadata

    Raises:
        FileNotFoundError: If the file does not exist
        UnsupportedFormatError: If file format is not supported
    """
    # Implementation
```

Frontend Style

- **CSS**: Use utility-first approach (like Tailwind)
- **Components**: Prefer functional components with clear props interfaces
- **State Management**: Centralize state management for complex state

11.3 Decision Log

Date	Decision	Alternatives Considered	Rationale
2025-06-13	Use LangChain for document processing pipeline	Custom pipeline, LlamaIndex	LangChain has built-in integrations for our entire stack and strong community support
2025-06-13	Local LLM vs API-based models	OpenAI API, Azure OpenAI	Data privacy requirements mandate keeping all processing local
2025-06-13	FAISS vs Chroma for vector store	Pinecone, Weaviate, Qdrant	FAISS offers best performance for local deployment with moderate-sized indices
2025-06-13	Streamlit vs Gradio for UI	Flask+React, FastAPI+Vue	Rapid prototyping capabilities and minimal frontend expertise required
2025-06-14	Document chunk size of 800 tokens	500, 1000, 1500 tokens	Balances context length with precision in retrieval based on test results
2025-06-14	Quantized 4-bit model vs 8-bit	16-bit full precision	4-bit provides acceptable quality while running on lower-end hardware

10.4 Accessibility Considerations

- Ensure UI meets WCAG 2.1 AA standards:
    - Proper color contrast (minimum 4.5:1 for normal text)
    - Keyboard navigation support
    - Screen reader compatibility for critical functions
    - Descriptive aria-labels for UI components
    - Responsive design for various screen sizes
- Implementation guidelines:
    - Include alt text for all images
    - Use semantic HTML elements

    - Implement focus states for interactive elements
    - Provide feedback for loading states and actions
    - Test with keyboard-only navigation

10.5 Data Privacy & Security Guidelines

- **Document Handling**:
    - All uploaded documents remain local to the user's machine
    - No data sent to external services
    - Implement auto-cleanup of uploaded documents after session ends
- **Security Measures**:
    - Input validation for all API endpoints
    - Sanitize file names and content before processing
    - Implement file size and type restrictions
    - Run with minimal required permissions
- **Development Practices**:
    - No hardcoded secrets in code
    - Use environment variables for configuration
    - Regularly update dependencies for security patches
    - Follow OWASP Top 10 security practices

10.6 Dependency Management

Core Dependencies

```
# requirements.txt

# API Framework
fastapi==0.103.1
uvicorn==0.23.2

# Document Processing
pymupdf==1.22.5
python-docx==0.8.11
langchain==0.0.267

# Embeddings & Vector Search
sentence-transformers==2.2.2
faiss-cpu==1.7.4
chroma-core==0.4.6

# LLM
llama-cpp-python==0.1.77
gpt4all==1.0.5

# UI
streamlit==1.26.0
gradio==3.40.1

# Testing
pytest==7.4.0
locust==2.16.1

# Code Quality
black==23.7.0
flake8==6.1.0
mypy==1.5.1
```

Dependency Update Policy

- Major version updates: Only during sprint boundaries after thorough testing
- Minor/patch updates: As needed to fix bugs or security issues

- Vulnerability scanning: Weekly automated check for security issues

10.7 Performance Benchmarks

Targets for the prototype's performance metrics:

Operation	Target Performance	Acceptable Minimum	Measurement Method
Document Upload & Processing	< 5 seconds per page	< 10 seconds per page	End-to-end timing from upload to chunks stored
Embedding Generation	< 2 seconds per chunk	< 5 seconds per chunk	Timing of SentenceTransformer inference
Query Response Time	< 5 seconds	< 15 seconds	End-to-end from query submission to answer display
UI Responsiveness	< 200ms for interactions	< 500ms	Lighthouse performance metrics
Memory Usage	Peak < 6GB	Peak < 12GB	Monitor during processing of 50-page document
Storage Requirements	< 100MB per document	< 250MB per document	Measure vector DB + document size

Performance Testing Plan

- Run benchmark tests at the end of each sprint
- Establish baseline in Sprint 1, track improvements through development
- Use profile tools (cProfile, memory_profiler) to identify bottlenecks
- Implement caching strategies for frequently accessed data

11. Future Roadmap

Potential enhancements beyond the initial 6-week prototype:

11.1 Short-term Improvements (1-3 months)

- **Multi-document comparison**: Analyze similarities and differences across multiple tenders
- **Custom training data**: Fine-tune embeddings on domain-specific tender documents
- **Export functionality**: Generate summary reports in PDF/DOCX formats
- **Notification system**: Email alerts for deadlines extracted from documents
- **Dark mode**: Alternative UI theme for reduced eye strain

11.2 Medium-term Features (3-6 months)

- **Collaborative mode**: Multiple users can annotate and comment on the same document
- **Image/diagram extraction**: Process visual information from tender documents
- **Template creation**: Generate response templates based on requirements
- **Integration with CRM**: Connect with existing customer relationship systems
- **Progressive Web App**: Enable offline functionality

11.3 Long-term Vision (6+ months)

- **Automatic response drafting**: Generate initial responses to tender requirements
- **Compliance checking**: Verify if company capabilities meet tender requirements
- **Historical analysis**: Learn from past successful/unsuccessful tender responses
- **Mobile application**: Native mobile experience for on-the-go access
- **Multi-language support**: Process and analyze documents in multiple languages

---

12. Version History

Version	Date	Description	Author
0.1	2025-06-13	Initial draft with requirements and architecture	UTL
0.2	2025-06-13	Updated with tech stack confirmation and formatted content	UTL
1.0	2025-06-14	Complete project plan with implementation details and getting started guide	UTL

Copyright Notice