Introduction to LangChain and LLM Limitations

Large Language Models have transformed the way software applications interact with human knowledge. Instead of writing complex rule-based programs, developers can now rely on models that understand natural language and generate meaningful responses.

However, these models do not automatically know about private documents, company policies, or personal notes.

They are trained on general public data and therefore require additional mechanisms to access domain specific information. LangChain was created to bridge this gap between raw data sources and intelligent language models.

LangChain is an open source framework that provides building blocks for creating Generative AI applications. It includes modules for loading documents, splitting long text, generating embeddings, storing vectors, and constructing prompts.

Each module solves a practical engineering problem faced while integrating LLMs into real products. Without such a framework, developers would need to write hundreds of lines of custom code for every project. LangChain standardizes these tasks and allows rapid experimentation.

One of the biggest challenges with language models is the context window limitation. Models can process only a fixed number of tokens in a single request. Real documents such as research papers, legal contracts, or annual reports often contain tens of thousands of words. Sending the entire document to the model is impossible and even if it were possible, the cost and latency would be extremely high. Therefore, external data must be processed intelligently before it reaches the model.

Page 2 – Document Loaders and Data Ingestion

The first step in any LLM application is data ingestion. LangChain provides a variety of document loaders that understand different file formats. TextLoader reads plain text files, CSVLoader handles spreadsheets, PyPDFLoader extracts content from PDF pages, and WebBaseLoader downloads information from websites.

Each loader converts the original source into a common Document object that contains page content and metadata such as file name or page number. This unified structure allows the rest of the pipeline to remain independent of the data source.

Consider a company building an internal knowledge assistant. The information may exist in multiple places: product descriptions in PDFs, customer records in CSV files, technical guides in text documents, and help articles on the website. Manually combining all these formats would be extremely time consuming. With LangChain loaders, the same code can read every source and transform it into a consistent representation. This is the foundation for building scalable AI systems.

Metadata plays an important role during retrieval. When a PDF is loaded, the system remembers which page the text came from. When a CSV is loaded, the row number and column names are preserved. Later, when the model generates an answer, the application can show references to the original source. This increases user trust and makes debugging easier.

Page 3 – Text Splitting and Chunking Strategies

After loading documents, the next challenge is dividing them into smaller pieces. This process is called chunking. A naive approach would be to cut the text every 500 characters, but such fixed splitting often breaks sentences in the middle and destroys meaning. LangChain provides smarter text splitters that try to respect paragraphs, new lines, and semantic boundaries.

CharacterTextSplitter is the simplest method. It divides text purely based on length and optional overlap. This is useful for quick experiments but may not always produce high quality chunks. RecursiveCharacterTextSplitter is more advanced. It attempts to split by paragraphs first, then by sentences, and only finally by individual characters. This hierarchical strategy preserves readability and improves retrieval performance.

Chunk overlap is another important concept. When two adjacent chunks share a small portion of text, the model receives enough context to understand questions that span across boundaries. Without overlap, the beginning of one chunk may refer to an idea that ended in the previous chunk, causing the model to lose information. Selecting the right chunk size and overlap requires experimentation based on the use case.

Page 4 – Retrieval Augmented Generation

Retrieval Augmented Generation, commonly known as RAG, is the architecture used by most practical LLM applications. Instead of relying only on the model's internal knowledge, the system first searches for relevant information from external documents. The retrieved chunks are then inserted into the prompt and the model generates an answer grounded in real data. This approach reduces hallucinations and ensures up-to-date responses.

Vector databases are typically used to store chunk embeddings. When a user asks a question, the query is converted into an embedding and compared with stored vectors to find similar pieces of text. The quality of these results depends heavily on how well the original documents were loaded and split. Poor chunking leads to poor retrieval, which directly affects the final answer.

In enterprise environments, RAG systems can assist employees in finding policies, analyzing reports, summarizing meetings, and answering technical questions. The same architecture can power chatbots for education, healthcare, finance, and customer support. Understanding document loaders and text splitters is therefore not just an academic exercise but a critical professional skill.

Page 5 – Conclusion

Building reliable Generative AI applications requires more than simply calling an API. Developers must design a complete pipeline that handles data ingestion, preprocessing, retrieval, and prompt construction. LangChain provides practical tools for each of these stages and has become a standard framework in the industry.

This assignment demonstrates the core concepts behind that pipeline. By experimenting with different loaders and splitters, students learn how raw unstructured data can be transformed into model-friendly knowledge. These foundations will remain important even as new models and techniques emerge in the future.

1. Evolution of Large Language Model Applications

Large Language Models have introduced a new paradigm in software development where systems can understand intent instead of rigid commands. Traditional applications depended on structured databases and predefined queries. Any information outside those structures was almost impossible to access automatically. With the emergence of transformer-based models, computers can now read unstructured text and reason over it in a human-like manner. This capability opened opportunities in search, automation, education, and enterprise knowledge management.

However, these models alone are not sufficient for real business usage. They are trained on public internet data that may be outdated or irrelevant to a specific organization. A bank, hospital, or university cannot rely on generic knowledge when answering internal questions. Sensitive and domain-specific documents must be incorporated safely. This requirement led to the development of frameworks such as LangChain that connect private data with language models.

LangChain acts as an orchestration layer between raw information and intelligent responses. It does not replace the model; instead it prepares the context so that the model can reason accurately. The framework emphasizes modular design where each component performs one responsibility: loading, splitting, embedding, retrieving, and prompting. Understanding these components is essential for any engineer working with Generative AI.

---

2. Data Ingestion Through Document Loaders

The first technical step in a RAG system is ingestion. Organizations store knowledge in many formats: invoices in PDF, reports in Word, analytics in CSV, and articles on websites. Manually converting all these into a single database would be unrealistic. Document loaders automate this process by reading each format and transforming it into a standard object.

A Document object in LangChain contains two main parts: the textual content and metadata. Metadata may include file name, page number, author, or URL. This information becomes extremely valuable later when the system needs to cite sources. For example, when a user asks "What is the refund policy?", the assistant can answer and also show "Source: policy.pdf – page 3". Such transparency is mandatory in professional environments.

Different loaders follow different strategies. The PDF loader extracts text page by page because PDFs are internally divided into pages. The CSV loader converts every row into a separate document so that tabular records can be searched individually. The web loader removes HTML tags and keeps meaningful paragraphs. By unifying all these sources, LangChain allows the same downstream pipeline to operate on heterogeneous data.

---

3. Need for Text Splitting

Even after loading, documents are usually too large for direct use. Language models have token limits and performance degrades when irrelevant information is included. Text splitting therefore becomes a critical preprocessing stage. The objective is to create chunks that are small enough for the model yet large enough to preserve meaning.

A poor splitting strategy can destroy the entire application. If a sentence is cut in the middle, the retriever may return incomplete facts. If chunks are extremely large, the vector search will be less

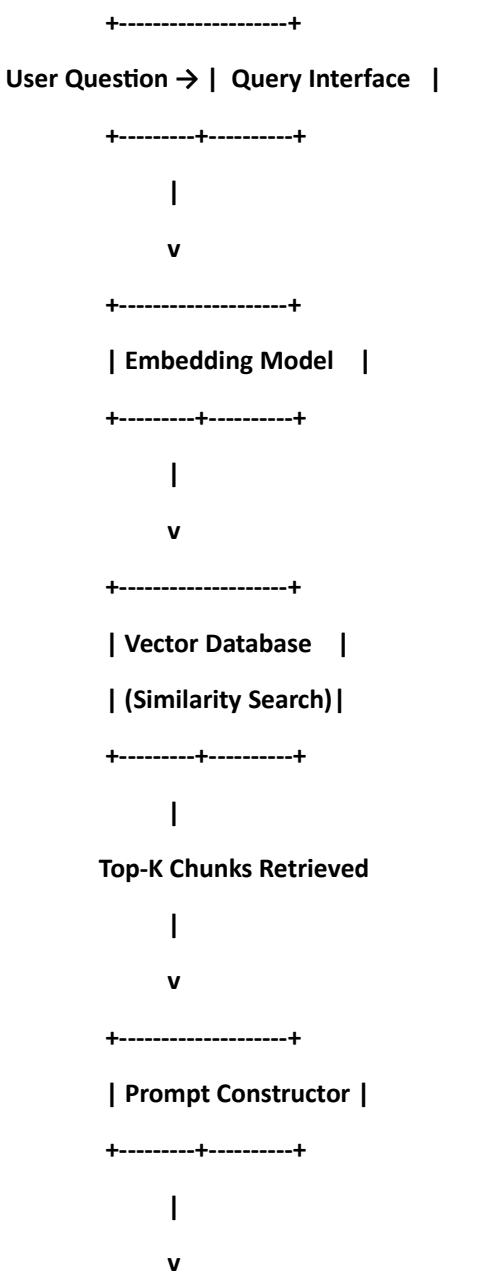
precise. Engineers must balance three factors: chunk size, overlap, and separator hierarchy. LangChain provides several splitters to experiment with these parameters.

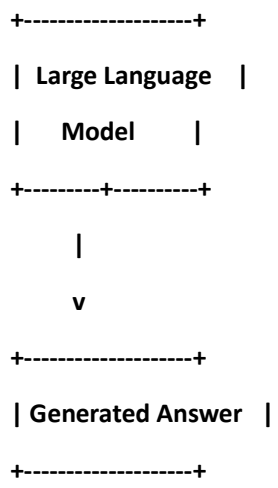
RecursiveCharacterTextSplitter is widely used because it respects natural language structure. It tries to split by double new lines first, then by single new lines, then by spaces. This mimics how humans organize ideas into paragraphs and sentences. Overlap of 50–100 characters ensures continuity across boundaries. These design choices directly influence retrieval quality.

---

4. RAG Architecture – Textual Diagram

Below is a conceptual architecture diagram expressed in text form that you can include in your report.





Flow Explanation

1. **User submits a natural language question.**
2. **The query is converted into an embedding vector.**
3. **Vector database searches for most similar document chunks.**
4. **Retrieved chunks are injected into a prompt template.**
5. **The LLM generates an answer grounded in those chunks.**
6. **The system optionally returns source references.**

---

5. Case Study – University Knowledge Assistant

Imagine a university that stores regulations, syllabus, hostel rules, and exam schedules in hundreds of PDFs. Students continuously email the administration asking repetitive questions such as:

- **“What is the attendance requirement?”**
- **“How many credits are needed to graduate?”**
- **“Is calculator allowed in statistics exam?”**

A RAG system can automate this support.

Step 1 – Ingestion

All PDFs, CSV schedules, and website notices are loaded using appropriate loaders.

Step 2 – Splitting

Documents are chunked with size 600 and overlap 80 to preserve context.

Step 3 – Embedding

OpenAI or local embedding model converts chunks into vectors.

Step 4 – Retrieval

When a student asks a question, the system fetches top 5 relevant chunks.

Step 5 – Generation

The model answers with citations like:
"According to Examination_Rules.pdf page 12, calculators are allowed only for engineering mathematics."

This assistant reduces workload of staff and provides 24/7 support.

---

6. Evaluation Metrics

To judge such a system, several metrics are used:

1. **Retrieval Precision – Are the returned chunks actually relevant?**
2. **Answer Faithfulness – Is the response supported by sources?**
3. **Context Recall – Did the system miss any important chunk?**
4. **Latency – Time from question to answer.**
5. **Cost – Tokens consumed per query.**

Engineers perform experiments with different chunk sizes and embedding models to maximize these scores. Logging user feedback further improves the pipeline.

---

7. Common Challenges and Solutions

Hallucination

Models sometimes invent facts.
Solution: Strict grounding, citation check, temperature control.

Noisy PDFs

Scanned PDFs produce bad text.
Solution: Use OCR preprocessing before loading.

Large Tables

Splitters break tables incorrectly.
Solution: Use table-aware loaders or convert to CSV.

Security

Private data must not leak.
Solution: On-prem embeddings and access control.

---

8. Implementation Best Practices

- **Use metadata to track sources**

- **Keep chunk size between 400–800**
- **Always add overlap**
- **Normalize text (remove headers/footers)**
- **Monitor retrieval results**
- **Provide “I don’t know” fallback**

These practices convert a simple demo into production-grade software.

---

9. Future Directions

The field is rapidly evolving. Semantic chunking using transformer boundaries, hybrid search combining keywords and vectors, and agentic workflows are becoming popular. Nevertheless, the fundamental stages of loading and splitting remain unchanged. Mastery of these basics ensures adaptability to any future model.

---

10. Conclusion

This report explained how external knowledge can be integrated with Large Language Models through LangChain. Document loaders convert heterogeneous data into a unified format, while text splitters prepare model-friendly chunks. Retrieval Augmented Generation then enables accurate, source-grounded answers. These concepts form the backbone of modern AI assistants used in industry today.

LangChain is an open-source framework designed to simplify the creation of applications using large language models (LLMs). It provides a standard interface for integrating with other tools and end-to-end chains for common applications. It helps AI developers connect LLMs such as GPT-4 with external data and computation. This framework comes for both Python and JavaScript.

Key benefits include:

- **Modular Workflow: Simplifies chaining LLMs together for reusable and efficient workflows.**
- **Prompt Management: Offers tools for effective prompt engineering and memory handling.**
- **Ease of Integration: Streamlines the process of building LLM-powered applications**

Key Components of LangChain

1. Chains: Chains define sequences of actions, where each step can involve querying an LLM, manipulating data or interacting with external tools. There are two types:

- **Simple Chains: A single LLM invocation.**
- **Multi-step Chains: Multiple LLMs or actions combined, where each step can take the output from the previous step.**

2. Prompt Management: LangChain facilitates managing and customizing prompts passed to the LLM. Developers can use PromptTemplates to define how inputs and outputs are formatted before being passed to the model. It also simplifies tasks like handling dynamic variables and prompt engineering, making it easier to control the LLM's behavior.

3. Agents: Agents are autonomous systems within LangChain that take actions based on input data. They can call external APIs or query databases dynamically, making decisions based on the situation. These agents leverage LLMs for decision-making, allowing them to respond intelligently to changing input.

4. Vector Database: LangChain integrates with a vector database which is used to store and search high-dimensional vector representations of data. This is important for performing similarity searches, where the LLM converts a query into a vector and compares it against the vectors in the database to retrieve relevant information.

Vector database plays a key role in tasks like document retrieval, knowledge base integration or context-based search providing the model with dynamic, real-time data to enhance responses.

5. Models: LangChain is model-agnostic meaning it can integrate with different LLMs such as OpenAI's GPT, Hugging Face models, DeepSeek R1 and more. This flexibility allows developers to choose the best model for their use case while benefiting from LangChain's architecture.

6. Memory Management: LangChain supports memory management allowing the LLM to "remember" context from previous interactions. This is especially useful for creating conversational agents that need context across multiple inputs. The memory allows the model to handle sequential conversations, keeping track of prior exchanges to ensure the system responds appropriately.