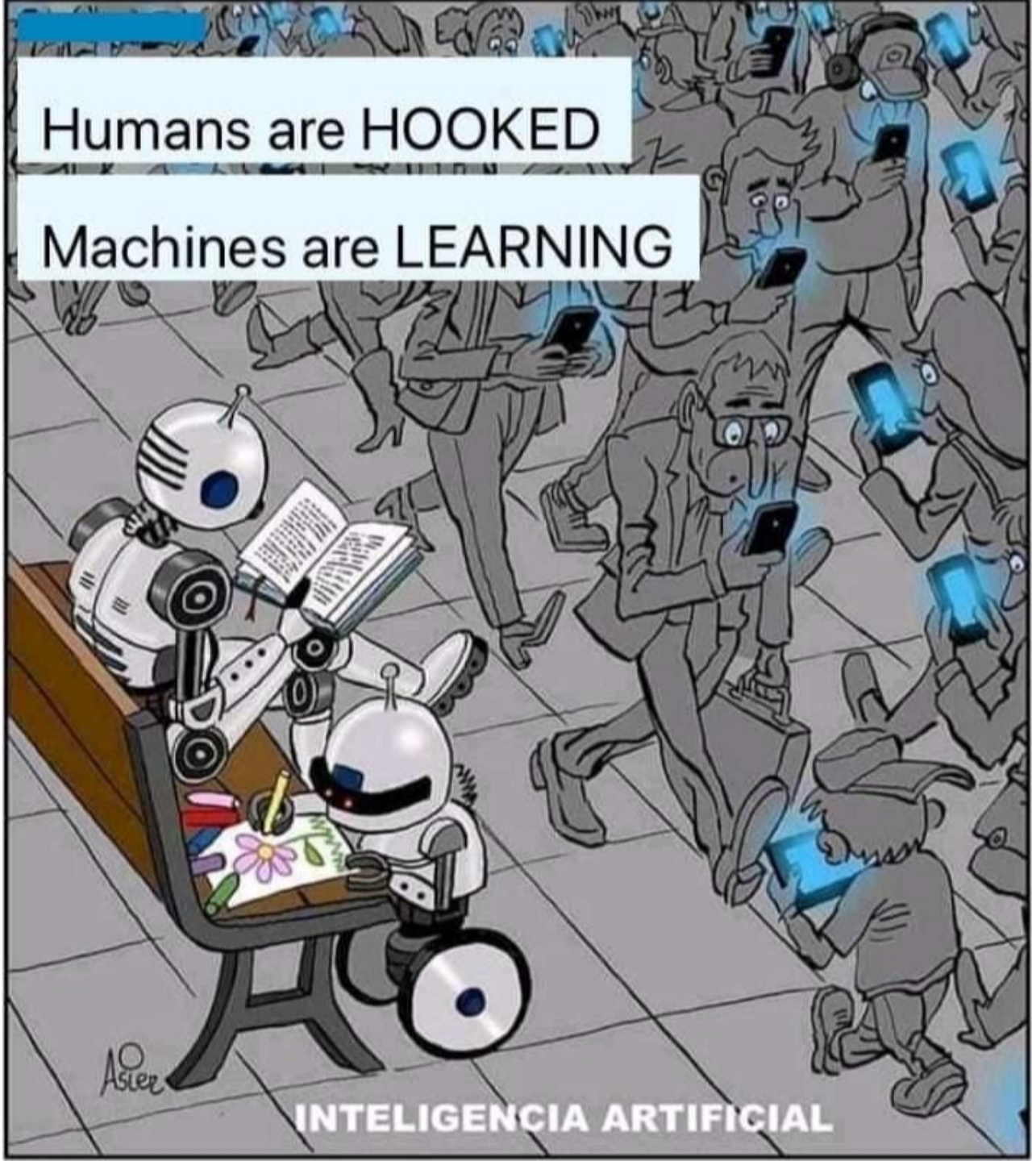
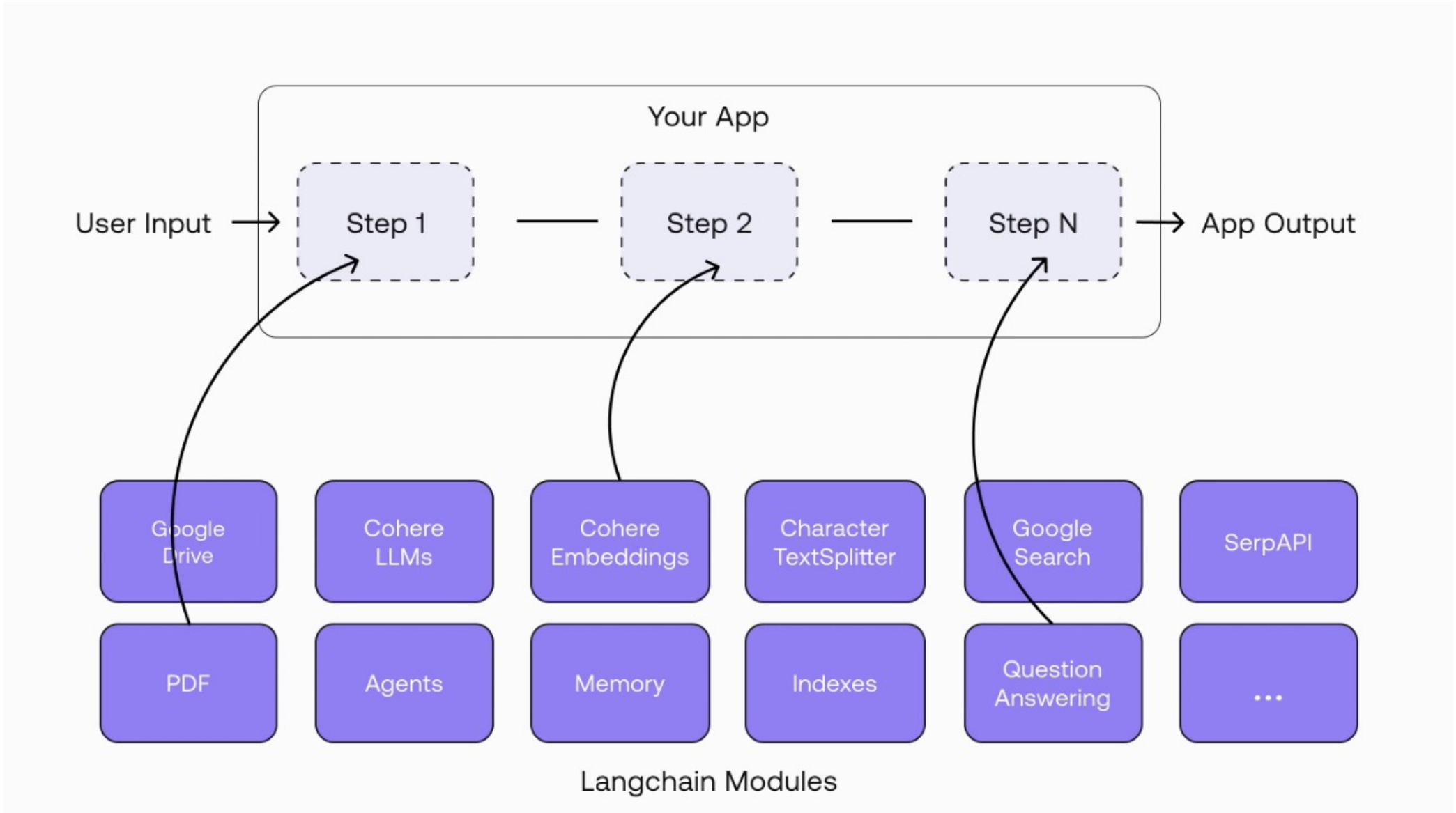


GenAI - LangChain

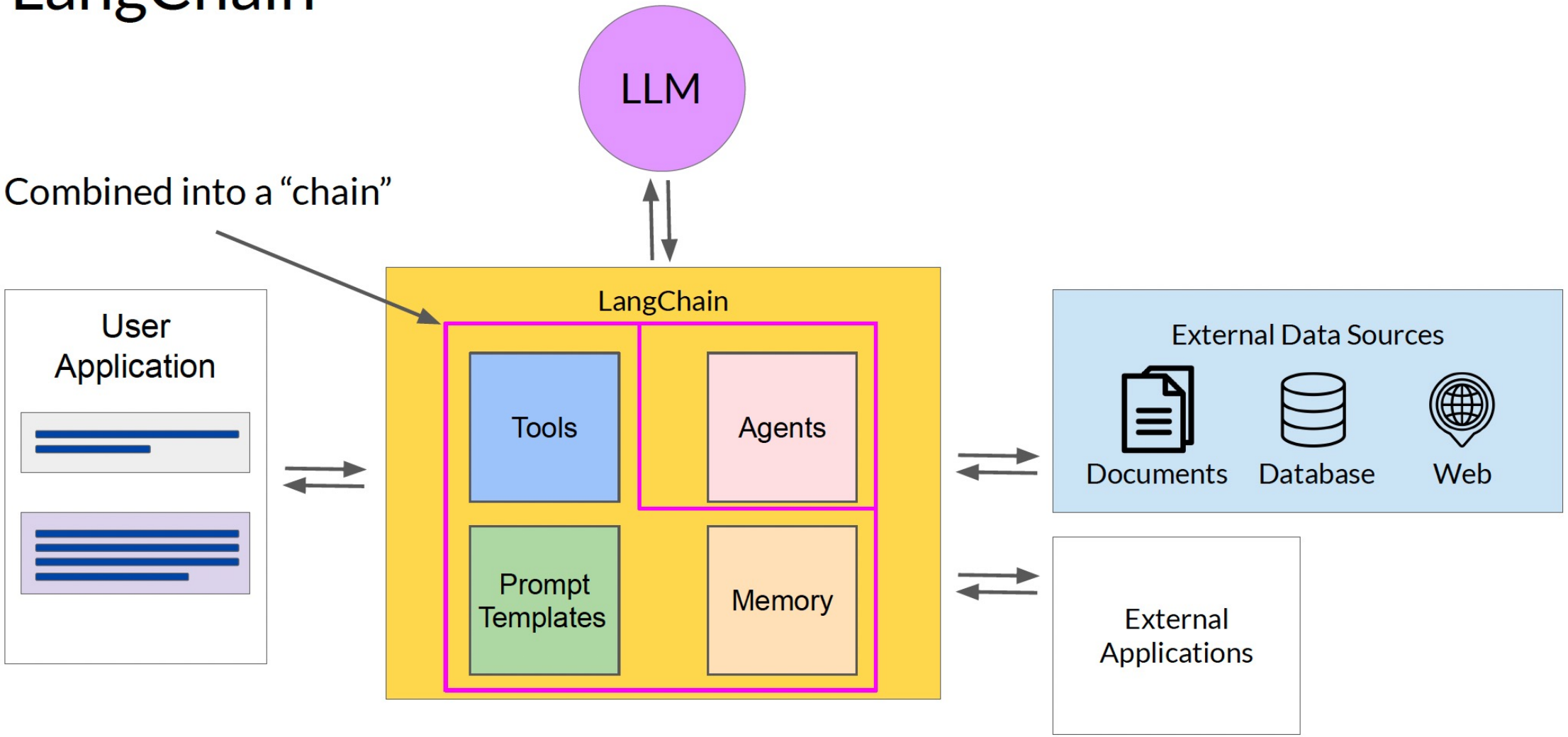
Naveen Kumar Bhansali
Co-Founder BlitzAI | Adjunct Faculty @ IIM Bangalore

LangChain

- LangChain is a framework for developing applications powered by language models.
- It enables applications that:
  - **Are context-aware**: connect a language model to sources of context (prompt instructions, few shot examples, content to ground its response in, etc.)
  - **Reason**: rely on a language model to reason (about how to answer based on provided context, what actions to take, etc.)


Your App
User Input
Step 1
Step 2
Step N
App Output
Google Drive
Cohere LLMs
Cohere Embeddings
Character TextSplitter
Google Search
SerpAPI
PDF
Agents
Memory
Indexes
Question Answering
...
Langchain Modules

LangChain


LLM
Combined into a “chain”
User Application
LangChain
Tools
Agents
Prompt Templates
Memory
External Data Sources
Documents
Database
Web
External Applications

Sequential Chains

Sequential chains allow you to connect multiple chains and compose them into pipelines that execute some specific scenario.

There are two types of sequential chains:

- SimpleSequentialChain:
  - Simplest form
  - Single input/output, and the output of one step is the input to the next.
- SequentialChain:
  - General Form.
  - Multiple inputs/outputs.

Router Chain

- Paradigm to create a chain that dynamically selects the next chain to use for a given input.

- Router chains are made up of two components:
  - The RouterChain itself (responsible for selecting the next chain to call).

  - Destination_chains: chains that the router chain can route to.

Data Loaders

- Document loaders deal with the specifics of
  - accessing and
  - converting data from a variety of different formats and sources into a standardized format (Which consists of content and then associated metadata).

- Variety of sources: PDFs and other document formats, webpages (HTML/JSON), YouTube, databases etc.
  - Also works on structured data.

Data Splitters

- Why should we split?
  - Documents we load can be very large.
  - While retrieval, we will need only the relevant part.

Data Splitters: Challenge

- When we split, we could end up with part of the sentence in one chunk, and the other part of the sentence in another chunk.
- So, when we answer a question, we actually don't have the right information in either chunk. Hence, we might not get the correct answer.
- Solution: Configure the splitter such that we get semantically relevant chunks together using parameters such as Chunk Size and Chunk Overlap.

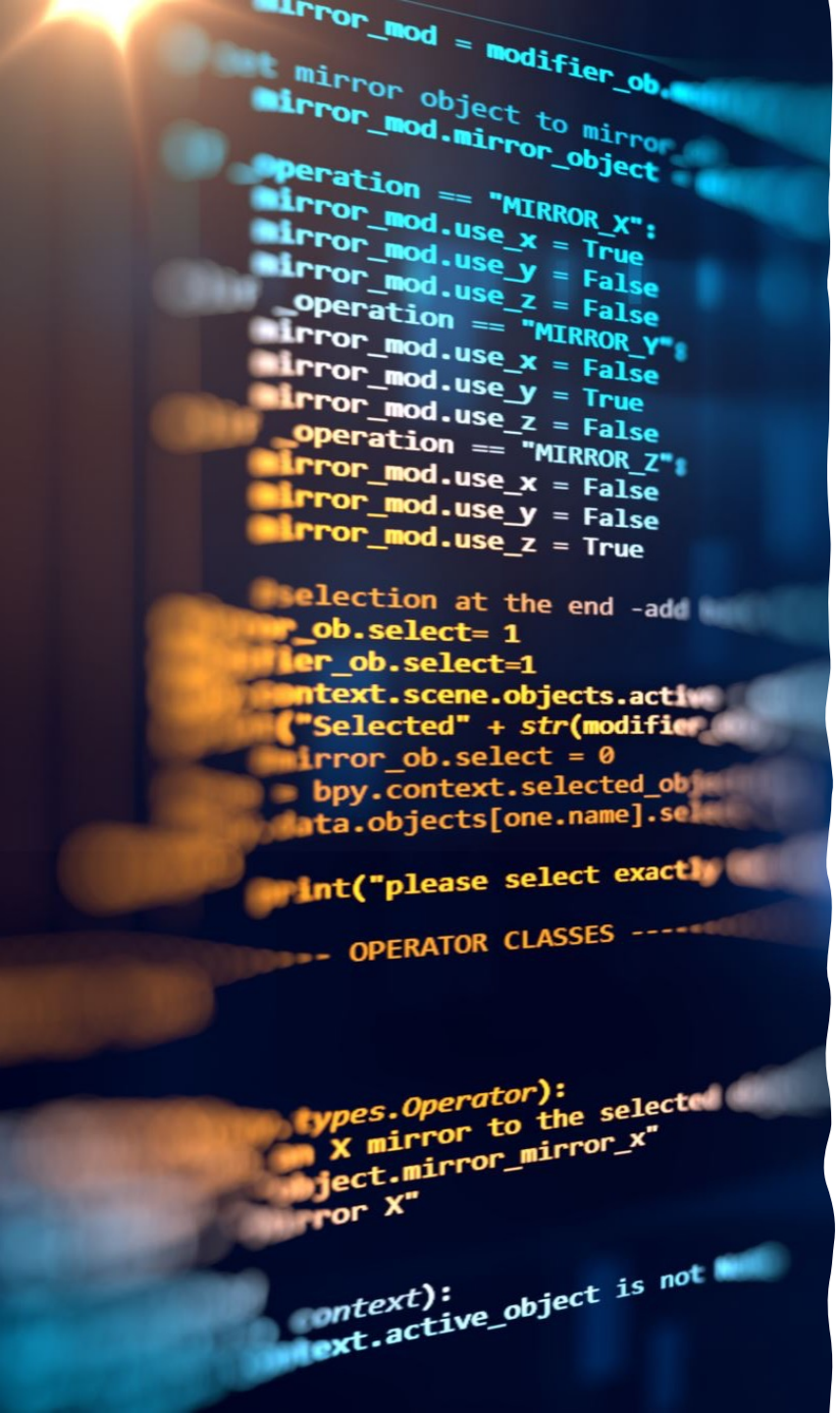
Data Splitters: Parameters

- Chunk Size: Size of the chunk w.r.t characters or tokens.

- Chunk Overlap: Overlap window between two chunks to ensure there is continuity in the flow of information.

Data Splitters: Internal working

- Split the text up into small, semantically meaningful chunks (often sentences).
- Start combining these small chunks into a larger chunk until you reach a certain size (as measured by some function).
- Once you reach that size, make that chunk its own piece of text and then start creating a new chunk of text with some overlap (to keep context between chunks).



Data Splitters: Types

- Split by character: This is the simplest method. This splits based on characters (by default "\n\n") and measure chunk length by number of characters.

- Split by code: CodeTextSplitter allows you to split your code with multiple languages supported.

- https://chunkviz.up.railway.app/ (Try 120 and 500).

Data Splitters: Types

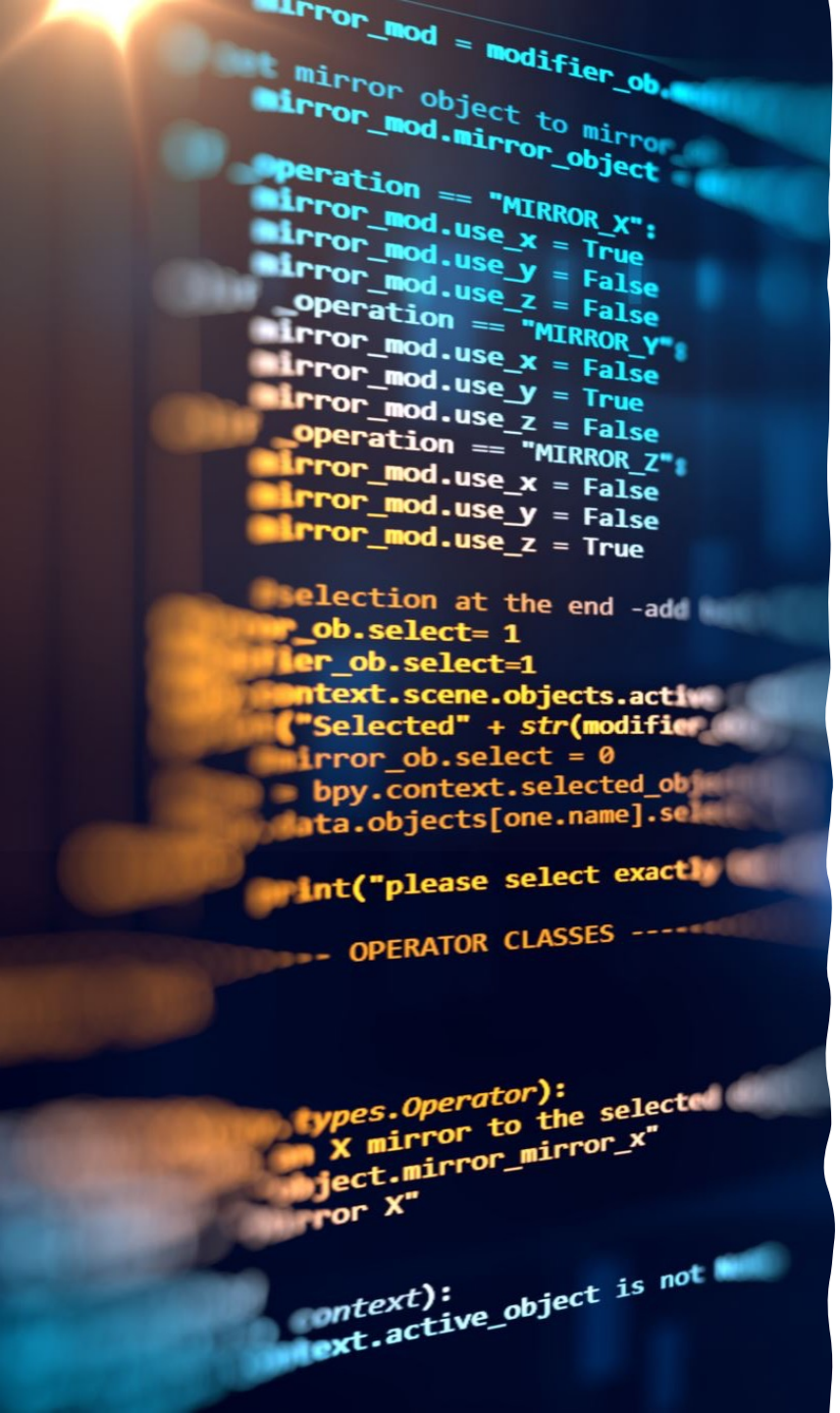
- Recursively split by character (**DEFAULT**): It tries to split on them in order until the chunks are small enough. The default list is ["\n\n", "\n", " ", ""].
    - This has the effect of trying to keep all paragraphs (and then sentences, and then words) together as long as possible, as those would generically seem to be the strongest semantically related pieces of text.
- Split by tokens: The text is split by character and the length is measured by “tiktoken” (python package) tokenizer.
    - Choose the tokenizer based on the language model opted.

Embeddings & Vector Stores

Embeddings & Vector Stores

- Once the document splits up into small, semantically meaningful chunks, we generally put these chunks into an index, whereby we can easily retrieve them when it comes to answer questions about this corpus of data.

- For that we use embeddings and vector stores.



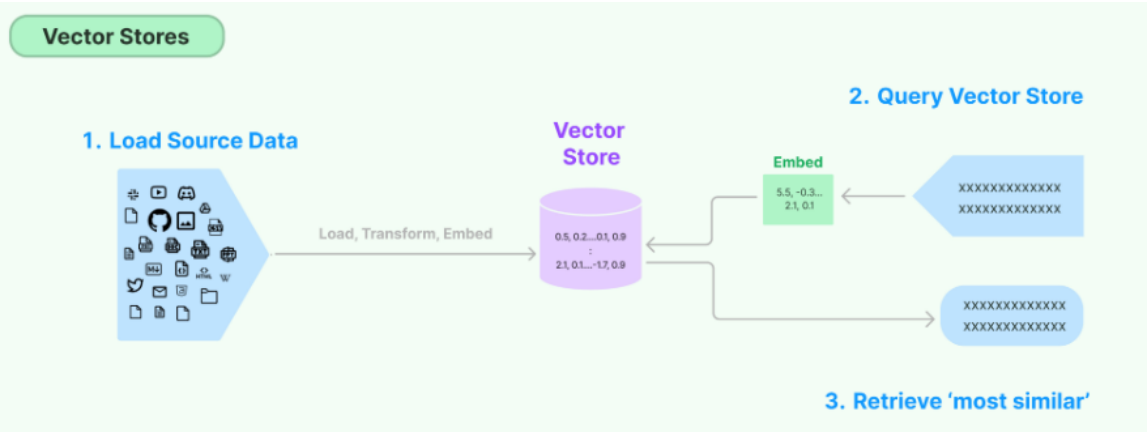
Embeddings

- Embeddings are numerical (vector) representation of the text which capture both syntactical and semantical meaning of the text.
- This is useful because it means we can think about text in the vector space and do things like semantic search where we look for pieces of text that are most similar in the vector space.
- The base Embeddings class in LangChain provides two methods:
  - one for embedding documents and
  - one for embedding a query.

Vector Store

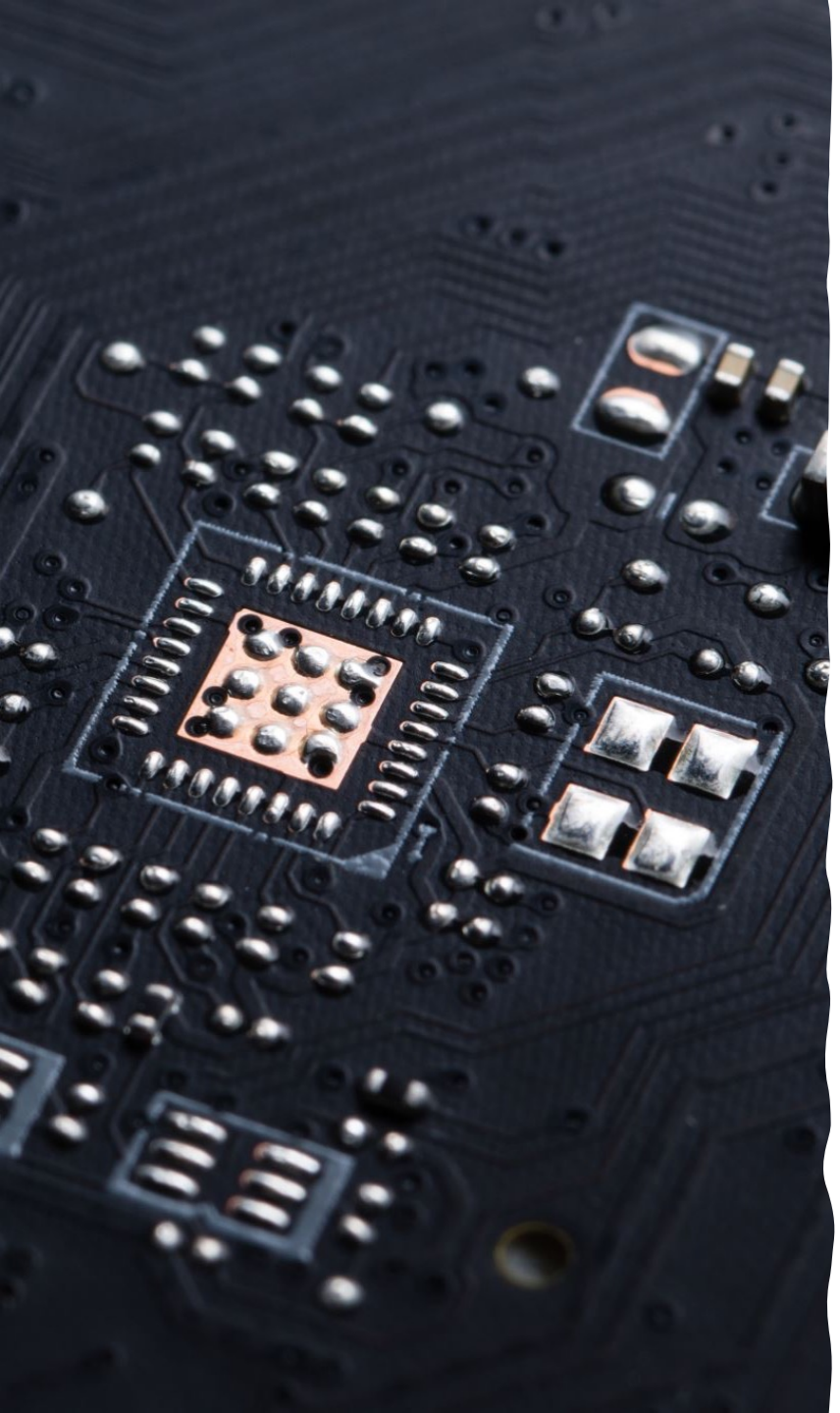
- One of the most common ways to store and search over unstructured data is to embed it and store the resulting embedding vectors, and then at query time to embed the unstructured query and retrieve the embedding vectors that are **'most similar'** to the embedded query.

- A vector store takes care of storing embedded data and performing vector search for you.



Some popular vector stores

- **Chroma**, an open-source embeddings store
- **Elasticsearch**, a popular search/analytics engine and vector database
- **Milvus**, a vector database built for scalable similarity search
- **Pinecone**, a fully managed vector database
- **Qdrant**, a vector search engine
- **Redis** as a vector database



Memory

Memory

- Large Language Models are stateless.
- Each transaction is independent.
- Chatbots appear to have memory by providing the full conversation as 'context’.
- And this memory storage is used as input or additional context to the LLM so that they can generate an output as if it's just having the next conversational turn, knowing what's been said before.

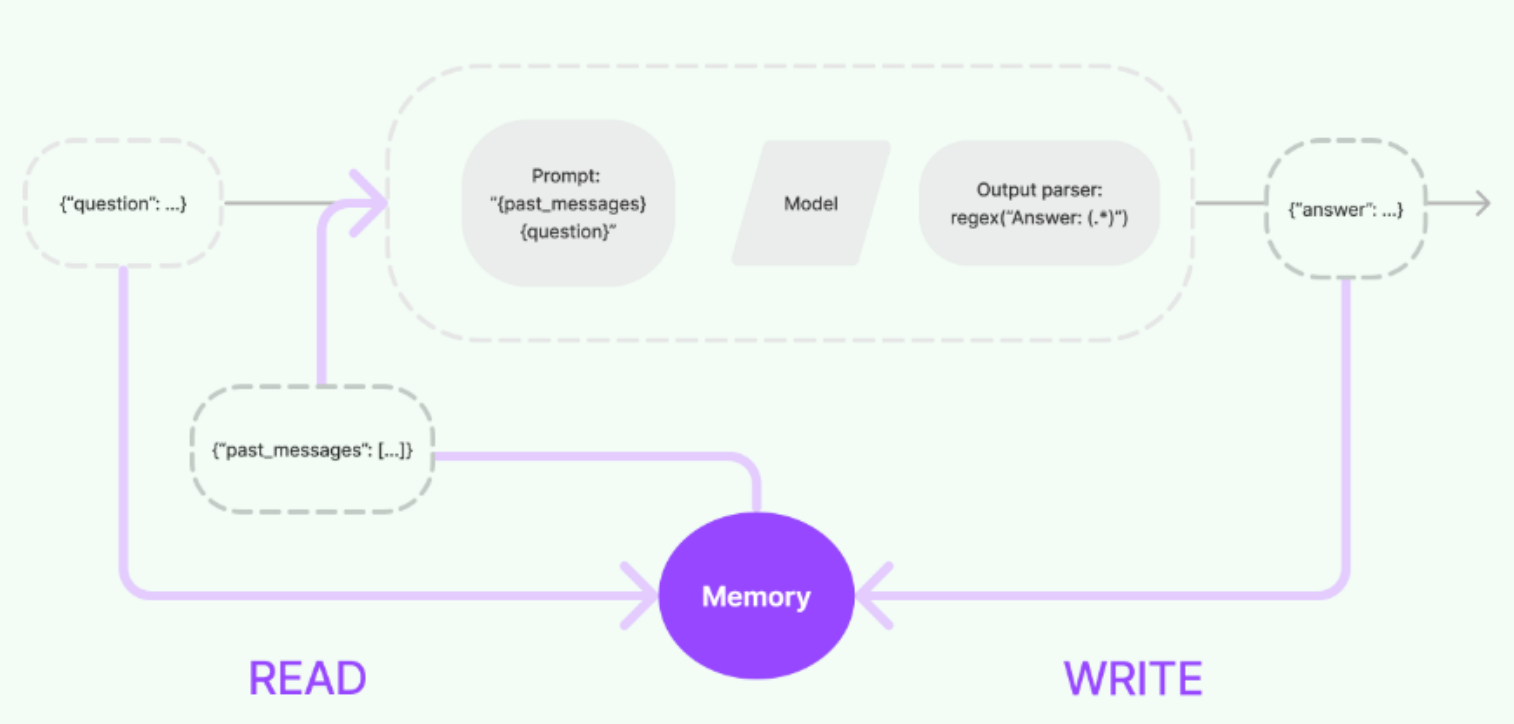
Memory

- And as the conversation becomes long, the amount of memory needed becomes huge and the cost of sending a lot of tokens to the LLM, which usually charges based on the number of tokens it needs to process, will also become more expensive.

- So LangChain provides several convenient kinds of memory to store and accumulate the conversation.

Memory System

A memory system needs to support two basic actions: reading and writing.



Building Memory into a system

- The two core design decisions in any memory system are:
  - **How state is stored**
    - List of chat messages are stored in In-memory lists to persistent databases.
  - **How state is queried**
    - Memory Types -> Data structures and algorithms on top of chat messages

Memory Types

- ConversationBufferMemory:

  This memory allows for storing of messages and then extracts the messages in a variable.

- ConversationBufferWindowMemory:

  This memory keeps a list of the interactions of the conversation over time. It only uses the last K interactions.

Memory Types

- ConversationTokenBufferMemory:

  This memory keeps a buffer of recent interactions in memory and uses token length rather than number of interactions to determine when to flush interactions.

- ConversationSummaryMemory:

  This memory creates a summary of the conversation over time.

Retrieval: Different ways

- Accessing/Indexing the data in the vector store:
  - Semantic Search
  - Maximal Marginal Relevance
  - Including Metadata

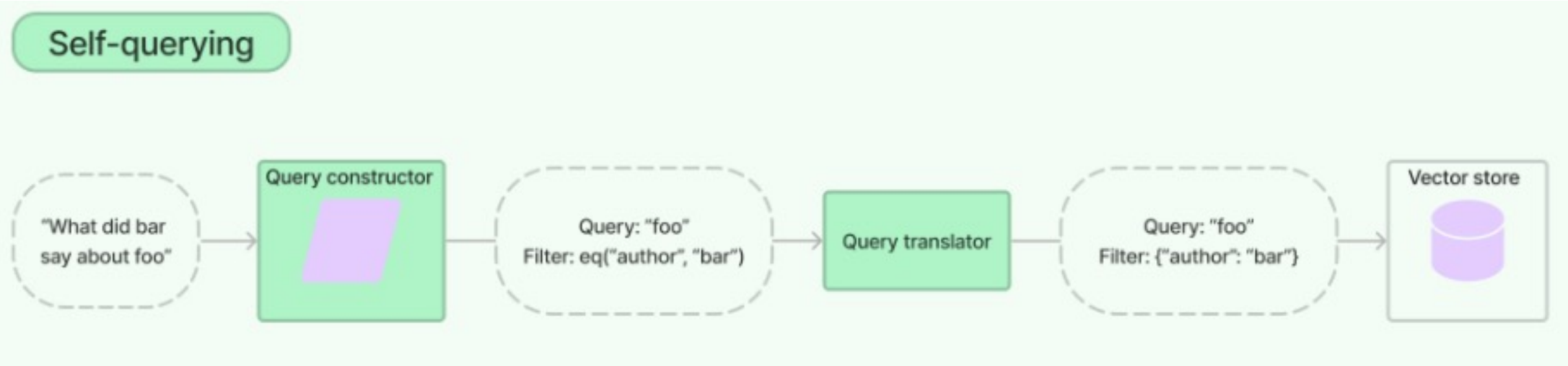
- Self querying - LLM aided Retrieval

Retrieval – Maximum Marginal Relevance

- You may not always want to choose the most similar responses i.e., you may want to add variety.

- Algorithm:
  - Query the vector store
  - Top ‘fetch_k’ most similar responses
  - Within those responses choose the ‘k’ most diverse.

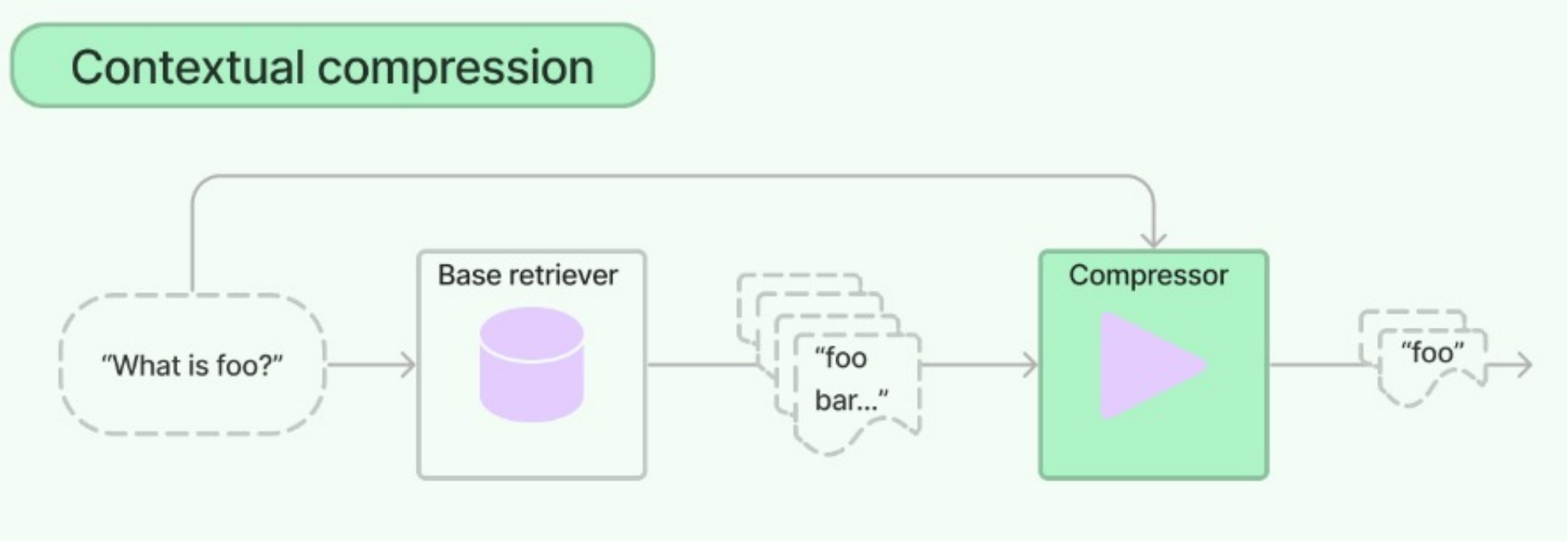
Retrieval – Self-querying



Retrieval – Self Querying

- A self-querying retriever is one that, as the name suggests, has the ability to query itself.
- Specifically, given any natural language query, the retriever uses a query-constructing LLM chain to write a structured query and then applies that structured query to its underlying VectorStore.
- This allows the retriever to not only use the user-input query for semantic similarity comparison with the contents of stored documents but to also extract filters from the user query on the metadata of stored documents and to execute those filters.

Retrieval – Contextual Compression



Retrieval – Contextual Compression

- One challenge with retrieval is that usually you don't know the specific queries your document storage system will face when you ingest data into the system.

- This means that the information most relevant to a query may be buried in a document with a lot of irrelevant text. Passing that full document through your application can lead to more expensive LLM calls and poorer responses.

- Contextual compression is meant to fix this.

Retrieval – Contextual Compression

- **The idea is simple**: instead of immediately returning retrieved documents as-is, you can compress them using the context of the given query, so that only the relevant information is returned.

- "**Compressing**" here refers to both compressing the contents of an individual document and filtering out documents wholesale.

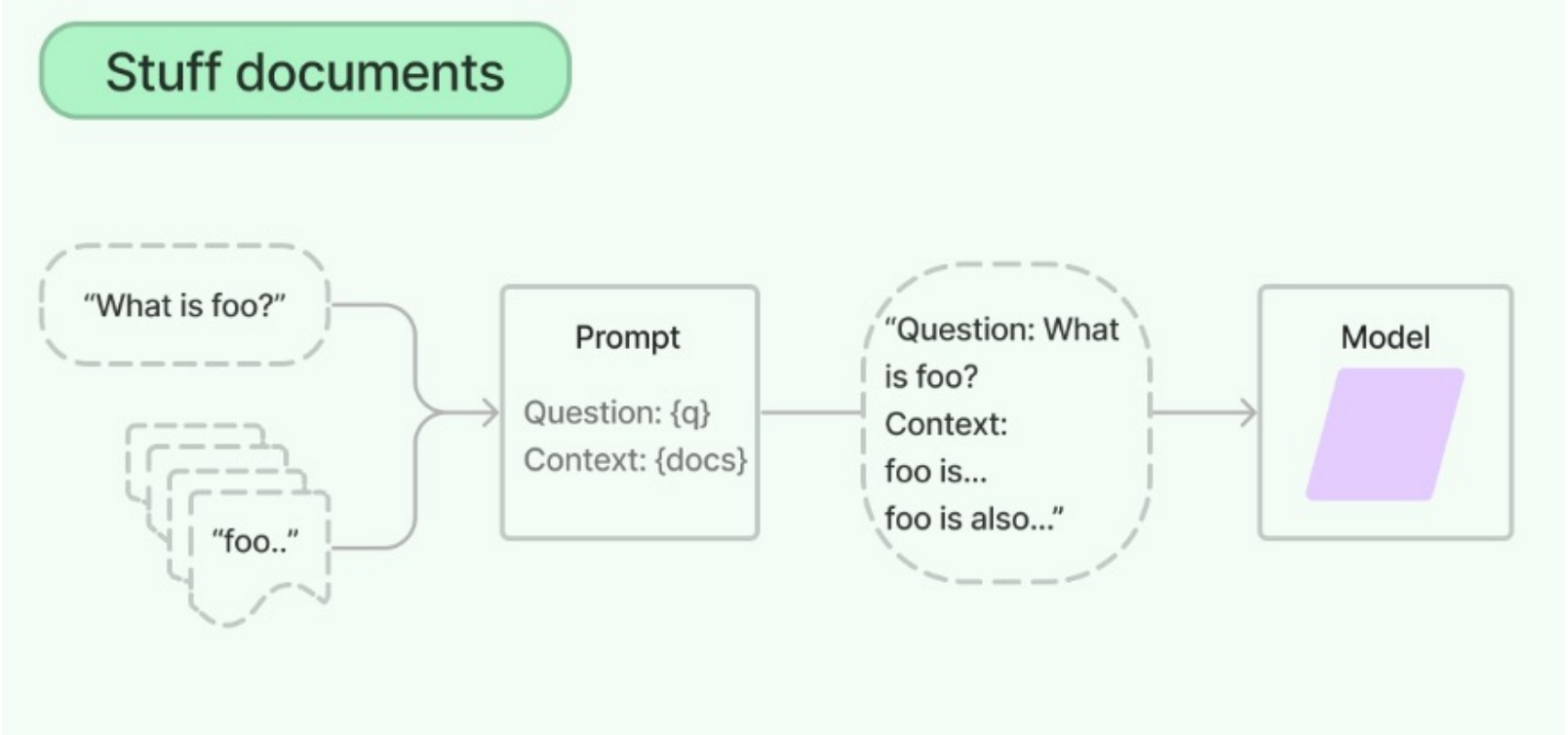
Retrieval Strategy (For Documents)

Retrieval -- Documents

- The following are the core chains for working with documents.
  - Stuff
  - Map-Reduce
  - Refine
  - Map re-rank

- They are useful for summarizing documents, answering questions over documents, extracting information from documents, and more.

Stuff


Stuff documents
"What is foo?"
"foo.."
Prompt
Question: {q}
Context: {docs}
"Question: What is foo?
Context:
foo is...
foo is also..."
Model

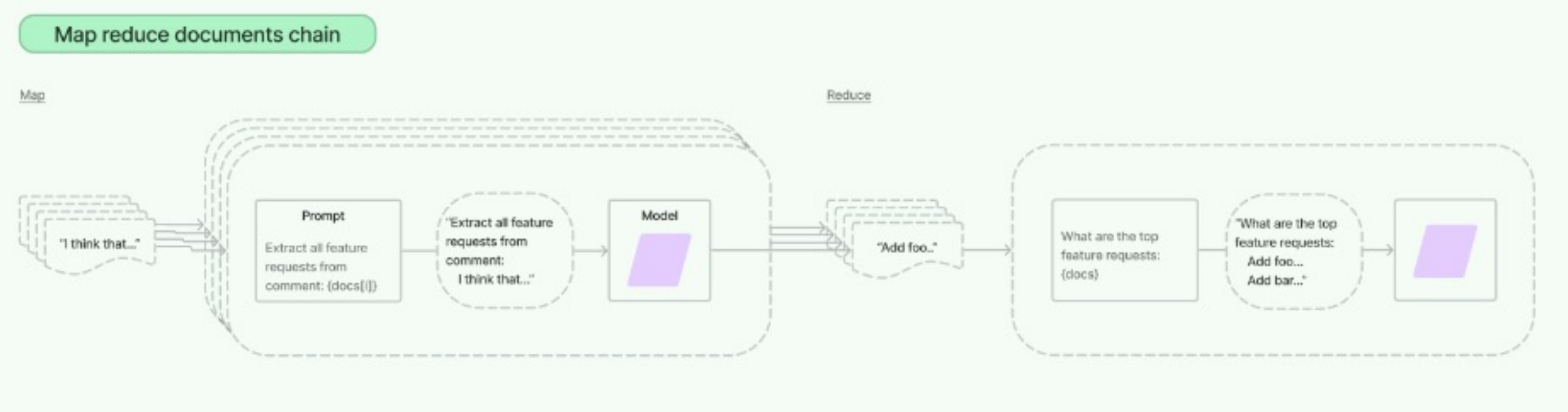
Stuff

- The stuff documents chain ("stuff" as in "to stuff" or "to fill") is the most straightforward of the document chains.
  - It takes a list of documents, inserts them all into a prompt and passes that prompt to an LLM.

- This chain is well-suited for applications where documents are small and only a few are passed in for most calls.
  - It only involves one call to the language model.

- However, this does have the **limitation** that if there's too many documents, they may not all be able to fit inside the context window.

Map reduce


Map reduce documents chain
Map
Reduce
"I think that..."
Prompt
Extract all feature requests from comment: {docs[i]}
"Extract all feature requests from comment:
I think that..."
Model
"Add foo.."
What are the top feature requests: {docs}
"What are the top feature requests:
Add foo...
Add bar..."

Map reduce

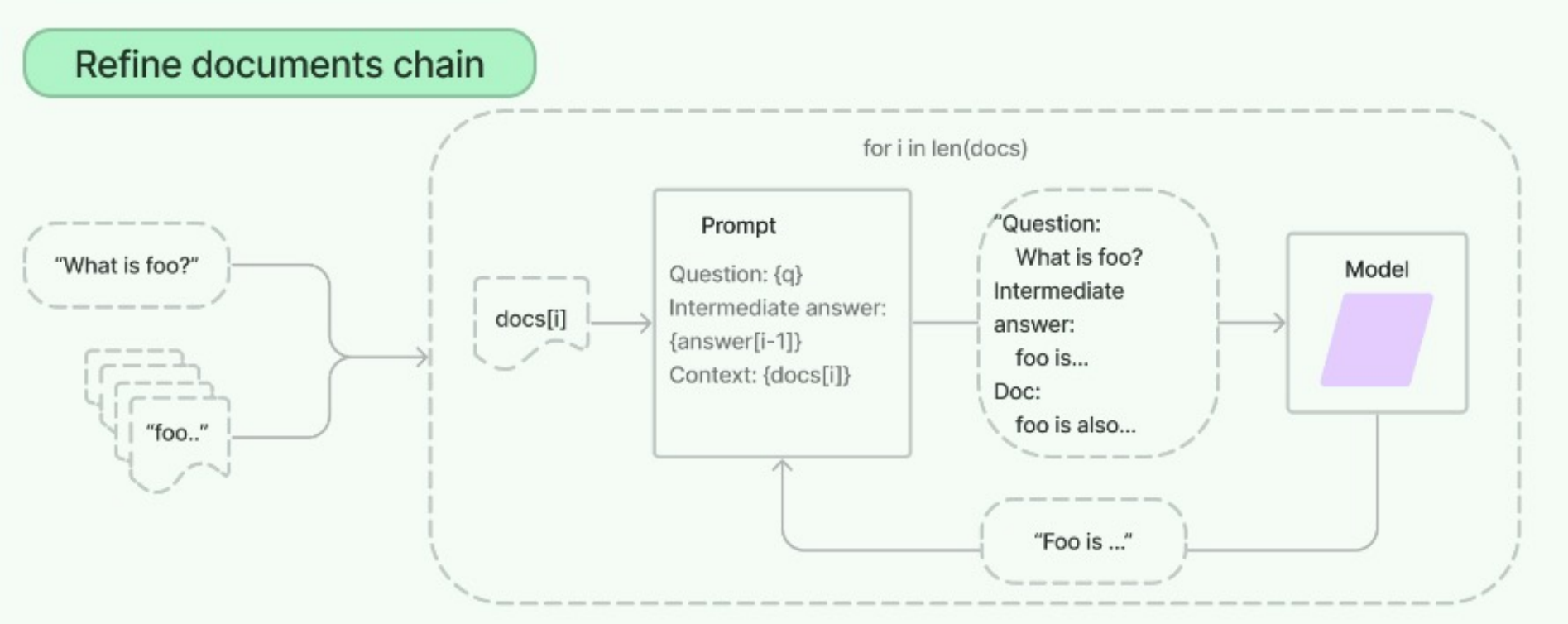
- The map reduce documents chain first applies an LLM chain to each document individually (the **Map** step), treating the chain output as a new document.

- It then passes all the new documents to a separate combine documents chain to get a single output (the **Reduce** step).

- This involves many more calls to the language model, but it does have the advantage in that it can operate over arbitrarily many documents.

- It can be slow.

Refine

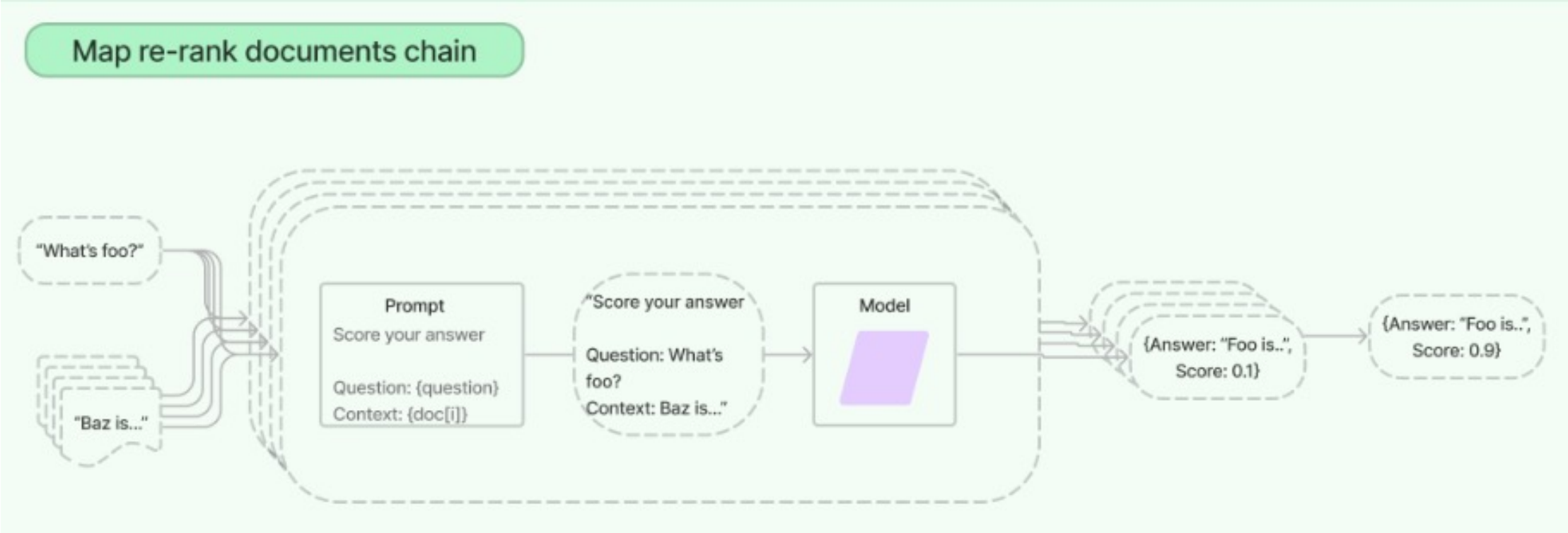

Refine documents chain
for i in len(docs)
"What is foo?"
"foo.."
docs[i]
Prompt
Question: {q}
Intermediate answer:
{answer[i-1]}
Context: {docs[i]}
"Question:
What is foo?
Intermediate
answer:
foo is...
Doc:
foo is also...
Model
"Foo is ..."

Refine

- The Refine documents chain constructs a response by looping over the input documents and iteratively updating its answer.
  - For each document, it passes all non-document inputs, the current document, and the latest intermediate answer to an LLM chain to get a new answer.

- You might get better result than the MapReduce chain.
  - That's because using the refined chain does allow you to combine information, albeit sequentially, and it encourages more carrying over of information than the MapReduce chain.

Map re-rank

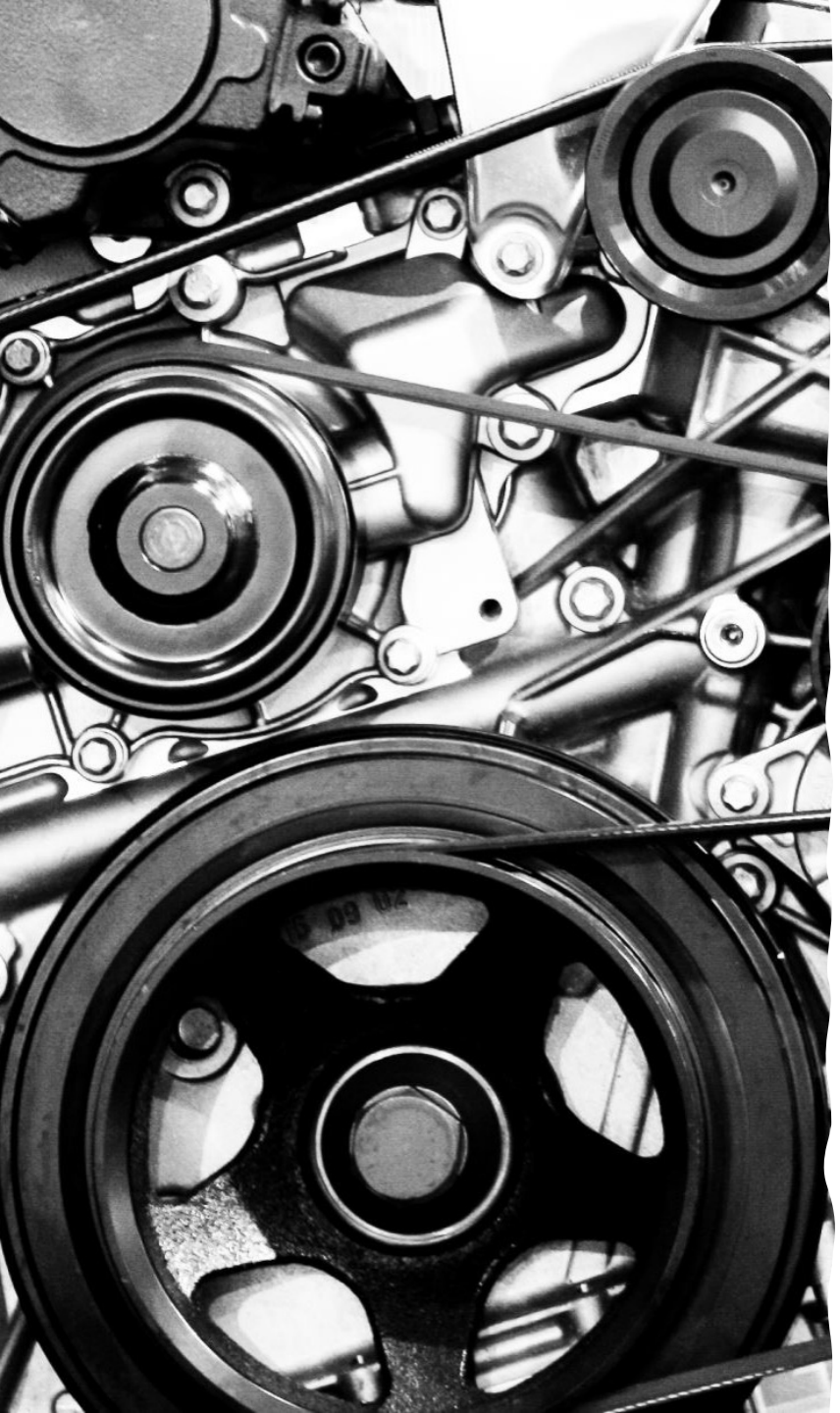

Map re-rank documents chain
"What's foo?"
"Baz is..."
Prompt
Score your answer
Question: {question}
Context: {doc[i]}
"Score your answer
Question: What's foo?
Context: Baz is..."
Model
{Answer: "Foo is..",
Score: 0.1}
{Answer: "Foo is..",
Score: 0.9}

Map re-rank

- The map re-rank documents chain runs an initial prompt on each document, that not only tries to complete a task but also gives a score for how certain it is in its answer.

- The highest scoring response is returned.

Agents



Agents

- The core idea of agents is to use an LLM to choose a sequence of actions to take.
- In chains, a sequence of actions is hardcoded (in code).
- In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

Agent

- This is the chain responsible for deciding what step to take next (powered by a language model and a prompt).
- The inputs to this chain are:
  - List of available tools; User input and Any previously executed steps.
- This chain then returns either the next action to take or the final response to send to the user.
- Different agents have different prompting styles for reasoning, different ways of encoding input, and different ways of parsing the output.

Agent Type

- Agents use an LLM to determine which actions to take and in what order. An action can either be using a tool and observing its output or returning a response to the user.

- Few Agents:
  - Zero-shot ReAct (Most general-purpose action agent) –
    - Uses the ReAct framework to determine which tool to use based solely on the tool's description.
  - Conversational
    - It uses the ReAct framework to decide which tool to use and uses memory to remember the previous conversation interactions.

Tools

- Tools are functions that an agent calls. There are two important considerations here:
  - Giving the agent access to the right tools
  - Describing the tools in a way that is most helpful to the agent.

- Few Examples
  - ChatGPT Plugins
  - Search – Bing, DuckDuckGo
  - Wikipedia

AgentExecutor

- The agent executor is the runtime for an agent. This is what actually calls the agent and executes the actions it chooses.
- Pseudocode for this runtime is below:

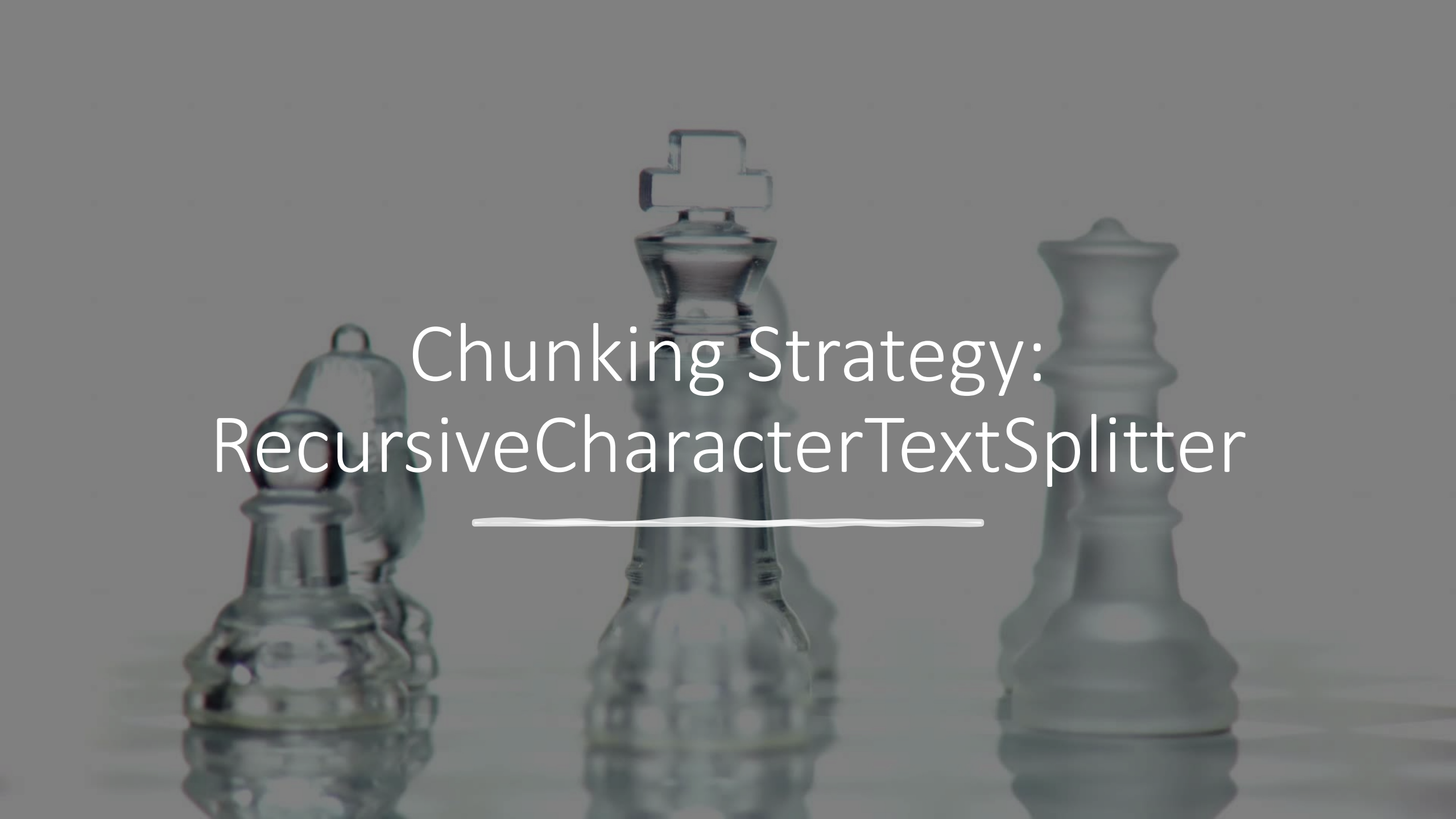
```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action, observation)
return next_action
```

Other types of agent runtimes

- The AgentExecutor class is the main agent runtime supported by LangChain. However, there are other, more experimental runtimes we also support.

- These include:
  - Plan-and-execute Agent
  - Baby AGI
  - **Auto GPT**

Appendix



Chunking Strategy: RecursiveCharacterTextSplitter

Chunking Strategy: Sample Text

What I Worked On

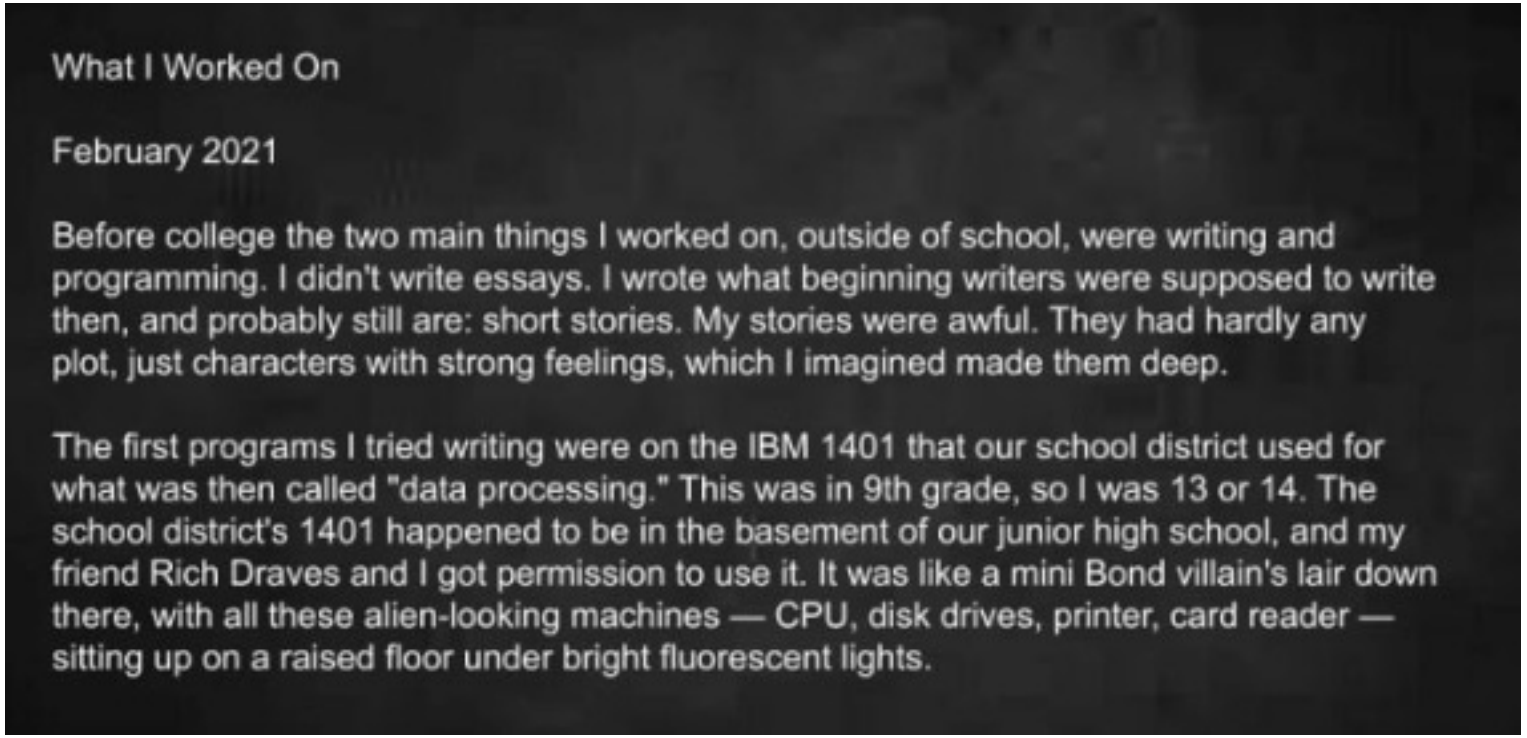
February 2021

Before college the two main things I worked on, outside of school, were writing and programming. I didn't write essays. I wrote what beginning writers were supposed to write then, and probably still are: short stories. My stories were awful. They had hardly any plot, just characters with strong feelings, which I imagined made them deep.

The first programs I tried writing were on the IBM 1401 that our school district used for what was then called "data processing." This was in 9th grade, so I was 13 or 14. The school district's 1401 happened to be in the basement of our junior high school, and my friend Rich Draves and I got permission to use it. It was like a mini Bond villain's lair down there, with all these alien-looking machines — CPU, disk drives, printer, card reader — sitting up on a raised floor under bright fluorescent lights.

https://chunkviz.up.railway.app/ (Try 120 and 500).

Chunking Strategy: Original Text



What I Worked On

February 2021

Before college the two main things I worked on, outside of school, were writing and programming. I didn't write essays. I wrote what beginning writers were supposed to write then, and probably still are: short stories. My stories were awful. They had hardly any plot, just characters with strong feelings, which I imagined made them deep.

The first programs I tried writing were on the IBM 1401 that our school district used for what was then called "data processing." This was in 9th grade, so I was 13 or 14. The school district's 1401 happened to be in the basement of our junior high school, and my friend Rich Draves and I got permission to use it. It was like a mini Bond villain's lair down there, with all these alien-looking machines — CPU, disk drives, printer, card reader — sitting up on a raised floor under bright fluorescent lights.

RecursiveCharacterTextSplitter Configuration

- We create a RecursiveCharacterTextSplitter instance, configuring it with a
  - chunk_size of 100 and
  - chunk_overlap value of zero.

- Our approach involves using the length function to measure each chunk based on its character count.

Chunking Strategy: Original Text with “\n\n”

\n\nWhat I Worked On\n\nFebruary 2021\n\nBefore college the two main things I worked on, outside of school, were writing and programming. I didn\'t write essays. I wrote what beginning writers were supposed to write then, and probably still are: short stories. My stories were awful. They had hardly any plot, just characters with strong feelings, which I imagined made them deep.\n\nThe first programs I tried writing were on the IBM 1401 that our school district used for what was then called "data processing." This was in 9th grade, so I was 13 or 14. The school district\'s 1401 happened to be in the basement of our junior high school, and my friend Rich Draves and I got permission to use it. It was like a mini Bond villain\'s lair down there, with all these alien-looking machines — CPU, disk drives, printer, card reader — sitting up on a raised floor under bright fluorescent lights.

Chunking Strategy – Recursive split: Split on “\n\n”

Chunking Strategy – Recursive split: Split on “\n\n”

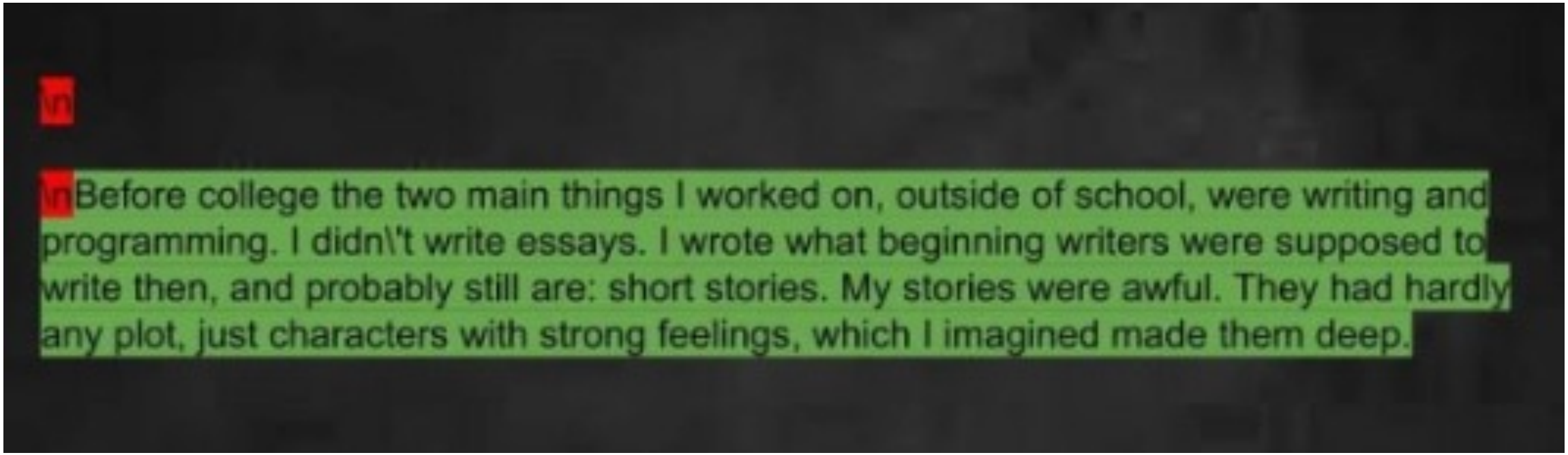
- Presently, we have four splits.
- Our next step involves assessing each split to check whether they meet the condition of being smaller than our specified chunk size, which is set at 100 characters.
- The first two splits satisfy this condition, thus earning them the label of good splits.
- Since both segments consist of fewer than 100 characters, we can combine them to create our initial chunk.

Chunking Strategy – Recursive split -> First Split after Merging

Chunking Strategy – Recursive split -> Second Split

- Proceeding to the second split, we find ourselves in a situation where further reduction isn't achievable using the \n\n character.
- Therefore, we proceed to the **next character**: \n.
- Our objective here is to execute a split using the \n character and determine if we can achieve a reduction in the split's size.

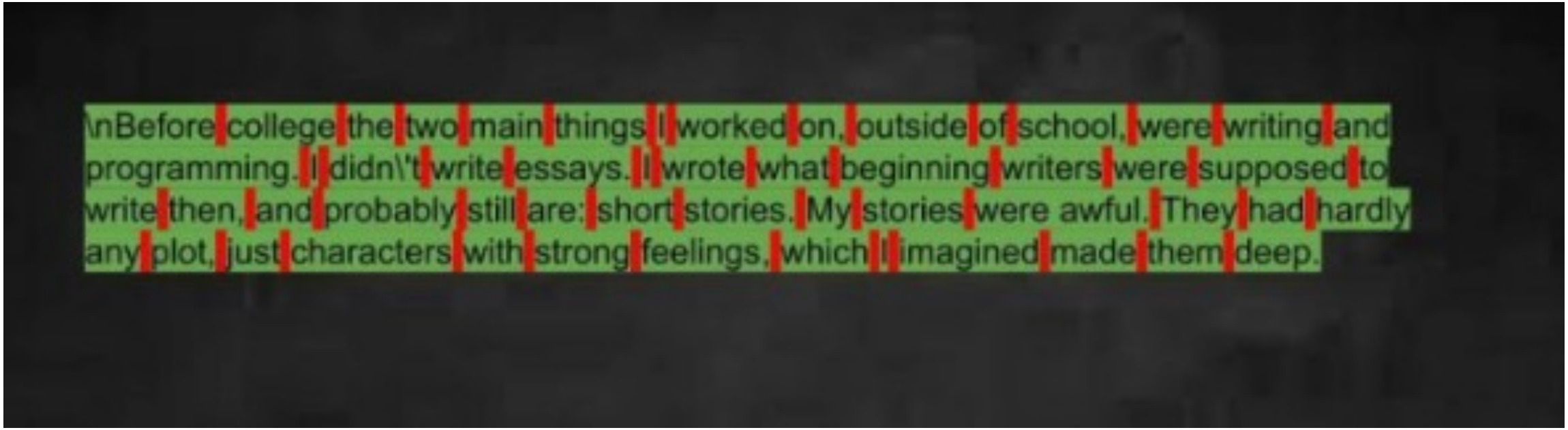
Chunking Strategy – Recursive split -> Second Split



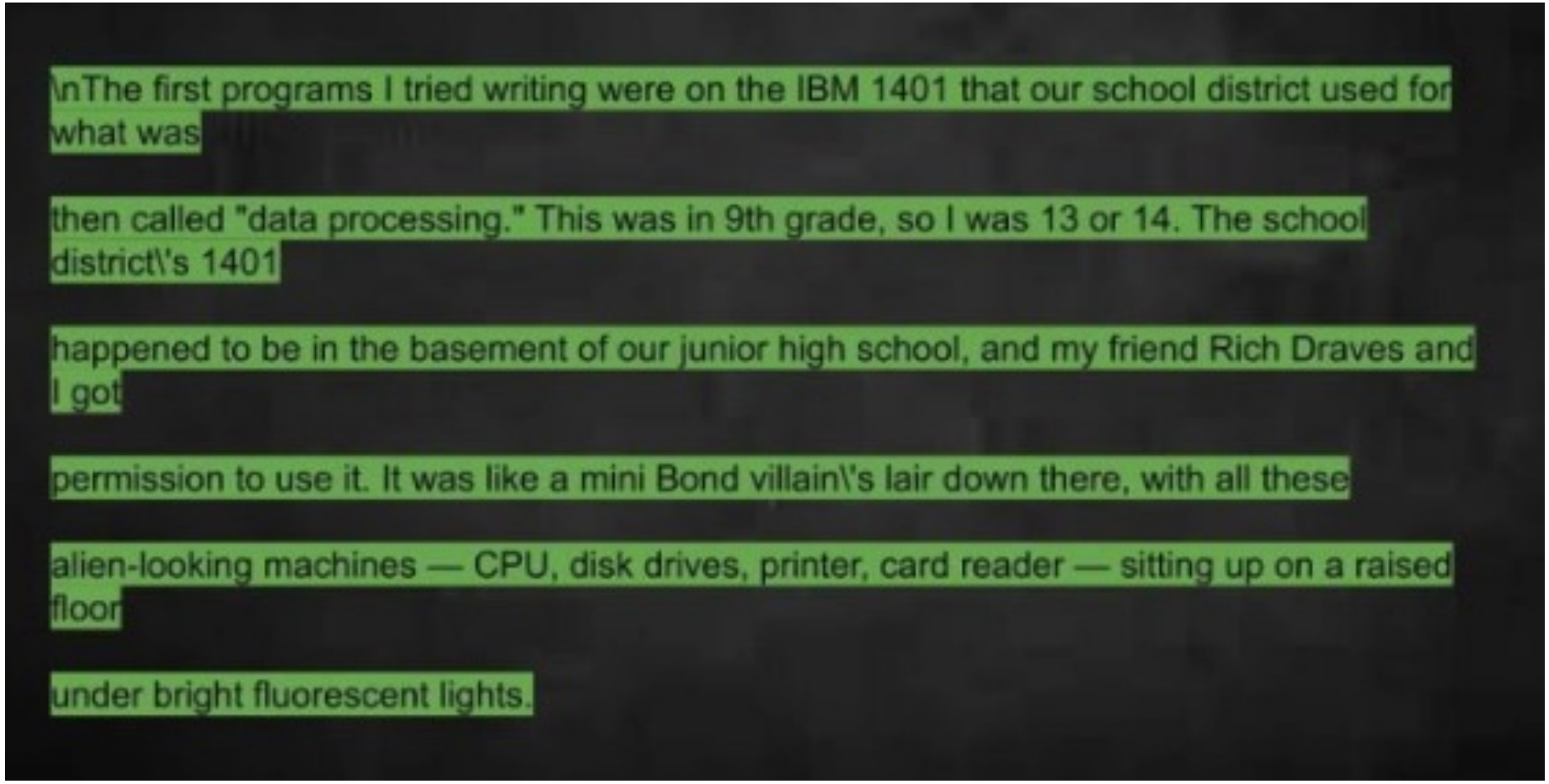
Chunking Strategy – Recursive split -> Split of spaces

- Upon executing the split using the \n character, we end up with two splits.
- The first split qualifies as a good split, given that it contains only one character.
- However, the second split surpasses our designated chunk size.
- Consequently, we need to invoke the split_text method on this particular split once again.
- However, this time we'll employ a split using the next character in our character list, which happens to be the ' ' character.

Chunking Strategy – Recursive split -> Split on spaces



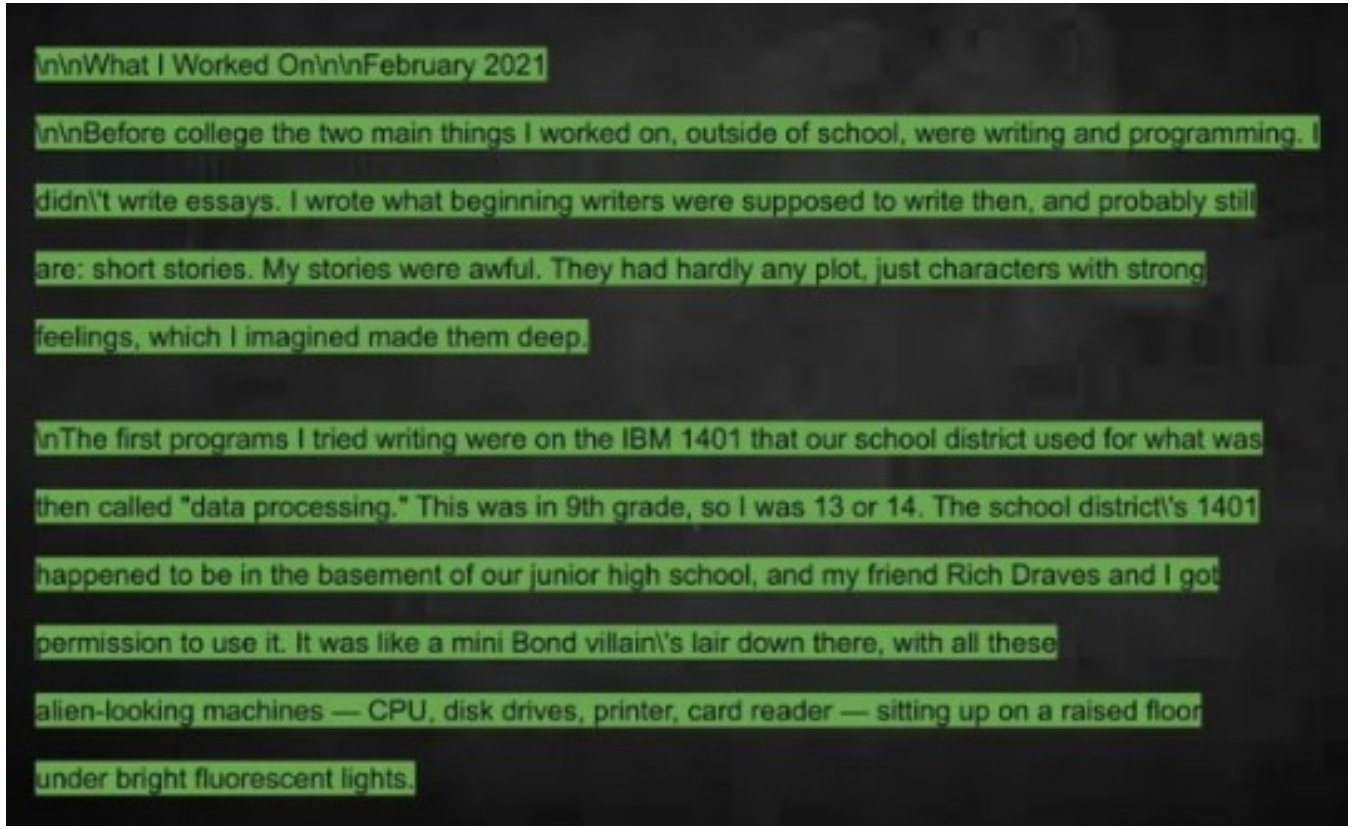
Chunking Strategy – Recursive split -> Split on spaces



Chunking Strategy – Recursive split -> Merging

- Next, we proceed with a merge, ensuring that no merged segments exceed the defined chunk size.
- After going through the entire process, we arrive at generating eleven individual chunks.
- Each of these eleven chunks successfully adheres to the 100-character limit.
- This outcome aligns precisely with what we achieved programmatically.

Chunking Strategy – Recursive split -> Final output



- https://chunkviz.up.railway.app/ -- Use 100 as chunk size