Class 17: Retrieval Augmented Generation (RAG)

- **Review:** Model Selection
- **Review:** LLM Context
- Retrieval Augmented Generation (RAG)

Copyright 2026, Kyle J. Harms

Activity: Team Name

Pick up your stuff and move to sit with your team.

Come up with a name for your project's team.

Submit the team name to Canvas.

Review: Model Selection

When selecting a model for your application, consider the following factors:

- **Model Size:** Larger models generally have better performance but require more computational resources.
- **Training Data:** The quality and diversity of the training data can impact the model's performance and the types of responses it generates.
- **Inference Speed:** The time it takes for the model to generate a response can be important for user experience, especially in interactive applications.
- **Specialization:** Some models are trained for specific tasks or domains, which can improve performance in those areas.
- **Context Length:** The amount of context the model can consider when generating responses can affect its ability to provide relevant and coherent answers.

Review: Context Window

The **context window** is the maximum number of tokens a model can “see” at once.

Everything must fit inside the window:

- System prompt
- Chat history (**every** previous message)
- Context you provide (data, documents, etc.)
- User’s prompt
- **And** the model’s response

WebLLM: 4,096 tokens for all models (~3,000 English words)

Review: Providing Context in Apps

```
const games = [
  { name: "Catan", kind: "strategy", minPlayers: 3, maxPlayers: 4, minutes: 90 },
  { name: "Ticket to Ride", kind: "family", minPlayers: 2, maxPlayers: 5, minutes: 60 },
  { name: "Codenames", kind: "party", minPlayers: 2, maxPlayers: 8, minutes: 15 },
  { name: "Pandemic", kind: "cooperative", minPlayers: 2, maxPlayers: 4, minutes: 45 },
];
const matches = games.filter(
  (game) => game.minPlayers <= 4 && game.maxPlayers >= 4 && game.minutes <= 30
);
const messages = [
  {
    role: "system",
    content: "You are a board game expert. Only recommend games from this list:\n" +
      JSON.stringify(matches),
  },
  { role: "user", content: userPrompt },
];
```

Retrieval Augmented Generation

Retrieval Augmented Generation (RAG)

A technique that combines retrieval-based methods with generative models to improve the quality and relevance of generated responses.

1. **R**etrieve the data relevant to the user's task.
2. **A**ugment the prompt with that data.
3. **G**enerate a response with the LLM.

Idea: Provide additional context to the model, but only the context that is relevant to the user's task.

Activity: Syllabus Assistant

Open the **activity repository** as a codespace.

Syllabus Assistant is an app for answering questions about INFO 4340/5440 course policies.

Working with a **peer**, evaluate the quality of its responses.

How could your fix this?

Idea: Add Documents to System Prompt

Add all course documents to the system prompt. The model can now answer questions about the syllabus!

Caution!

- 12 documents, ~5,500 words (**~7,000 tokens**)
- Context window: **4,096 tokens** (system prompt + context + question + **response**)
- You can't `filter()` it: it's paragraphs, not fields like **Game Night**.

Implementing RAG

1. **Split** documents into chunks (e.g., paragraphs, sections, or pages).
2. **Index** chunks with a vector database / create embeddings.
3. **Retrieve** relevant chunks for a given user prompt.
4. **Augment** (append) the system prompt with the retrieved chunks.
5. **Generate** a response with the LLM.

Step 1: Split Documents into Chunks

Why split?

- Conserve space in the system prompt (and context window)
- Smaller chunks can make search more **precise**

Gotcha: too small may lose context; too big may fill up the context window.

Chunk Size

Chunk size	Each chunk is…	Problem
Too small (~100 characters)	A sentence fragment	Loses context: *"…will consume 1 slip day."* Of what?
Too big (~4,000 characters)	A whole document	3 chunks ≈ 3,000 tokens: fills the context window. One vector for many topics: search is less precise.
Goldilocks (~500–1,000 characters)	A paragraph or a section	One idea per chunk

Chunk Overlap

A split can cut an idea in half. **Overlap** repeats the end of one chunk at the start of the next chunk.

Rule of thumb: overlap ≈ 10–20% of the chunk size

Example: “Late Work” with `{ chunkSize: 200, chunkOverlap: 50 }`:

```
Chunk 2: Try your best to submit assignments on time. But don't panic if you need to submit
         an assignment late. You may submit an assignment late without penalty by using the
         provided slip day
Chunk 3: without penalty by using the provided slip day accommodations. Otherwise, in the
         interest of fairness to all students, late work is not accepted and will receive 0 credit.
```

Choosing a Text Splitter

The **splitter** takes a document and splits it into chunks.

Your documents	Splitter	Splits at
Plain text, PDF	`RecursiveCharacterTextSplitter` *(default)*	Paragraphs → lines → words
Markdown	`new MarkdownTextSplitter()`	Headings, then paragraphs
Web pages (HTML)	`RecursiveCharacterTextSplitter.fromLanguage("html")`	Block tags (`<p>` , `<div>` , …)
One known separator (FAQ entries, `---`)	`new CharacterTextSplitter({ separator: "---" })`	Only that separator

Step 2: Index Chunks / Create Embeddings

An **embedding model** turns text into **vectors**.

Texts with similar **meanings** get similar vectors.

Example:

Passage	Similar to *"Can I turn in homework late?"*
"You are provided with a total of 2 slip days..."	✅ Close
"Attendance is taken through in-class activity handouts."	❌ Far

It only makes vectors; it **can't** chat.

Code Walkthrough: Split Documents into Chunks

1. Import and read the documents.
2. Create embeddings (+ chunk, index/embed)

   chunk size: 800 (~200 tokens).

   embedding model: `snowflake-arctic-embed-s` (67 MB, 384 numbers per vector)

Step 3: Retrieve Chunks

A **hybrid search** combines two kinds of search:

- **Keyword:** matches the exact words ("slip day" → *Slip Day Accommodation*)
- **Meaning:** matches the idea ("turn in late" → *Slip Day Accommodation*)

`documentIndex.search(question, k)` returns the **top *k* chunks**:
`[{ title, text, score }, ...]` (most relevant first)

Activity: Chunk Retrieval Practice

Working with a peer, complete the handout.

1. For each prompt, select the **3 chunks** that best answer it.
2. Circle whether each chunk matches the prompt's **words** or its **meaning**.

Code Walkthrough: Retrieve Chunks

Search the index and get the chunks:

```
await documentIndex.search(prompt, 3);
```

Step 4: Augment the Prompt

Add the chunks to the system prompt:

```
const documents = chunks.map((chunk, i) => chunk.text.join("\n\n---\n\n");
```

Code Walkthrough: Augment the Prompt + Generate a Response

1. Append the chunks to the system prompt.

```
const documents = chunks.map((chunk, i) => chunk.text.join("\n\n---\n\n");
```

2. Use updated system prompt to generate a response.

Homework 5

Index a document (PDF) provided by the user.

Using RAG, answer questions about the document.

Display the answers and the chunks that were retrieved as the UI.

Not a ChatBot!