


Final Implementation

User Prompt

Create a RAG from scratch.

Plan

 Create a RAG (Retrieval-Augmented Generation) System from Scratch

 Step-by-Step Plan

1. **Define Requirements**
 - Identify the domain (e.g., Q&A, summarization) and the type of documents (e.g., articles, reports).
 - Determine the query types (e.g., factual, open-ended).
2. **Collect and Preprocess Data**
 - Gather a dataset of documents relevant to the domain.
 - Clean the data: remove duplicates, tokenize, and normalize text (e.g., lowercasing, removing stopwords).
3. **Build a Document Index**
 - Convert documents into a searchable format (e.g., TF-IDF, inverted index, or vector embeddings).
 - Use tools like **Elasticsearch**, **FAISS**, or a custom solution for efficient retrieval.
4. **Implement a Retrieval Model**
 - Choose a retriever:
 - **Traditional**: BM25 for keyword-based retrieval.
 - **Neural**: DPR (Dense Retrieval) for semantic similarity using embeddings (e.g., BERT).
 - Train or fine-tune the retriever on your dataset.
5. **Train a Generation Model**
 - Use a sequence model (e.g., Transformer-based models like **T5**, **BART**, or **GPT**).
 - Fine-tune the model on a dataset of questions and answers (if available) or use a pre-trained model.
6. **Integrate Retrieval and Generation**
 - Design a pipeline:
 - For a query, retrieve relevant documents using the retriever.
 - Feed the retrieved documents + query into the generation model to produce an answer.
7. **Evaluate the System**
 - Test with sample queries and measure:
 - Relevance of retrieved documents.
 - Accuracy and coherence of generated answers.
 - Use metrics like ROUGE, BLEU, or human evaluation.
8. **Optimize and Deploy**
 - Tune hyperparameters (e.g., retrieval top-K, generation temperature).
 - Optimize for speed and scalability (e.g., caching, parallelization).
 - Deploy the system (e.g., as a web service, API, or embedded in an app).

This plan provides a foundation for building a RAG system. Adjust steps based on your specific use case and available resources.

Architecture

 Folder Structure & Main Files

 Root Directory

```
rag-system/ |—— data/ # Preprocessed documents and queries |—— models/ # Trained models (retriever,
generator) |—— utils/ # Helper functions (text processing, embeddings) |—— config/ # Configuration files
(hyperparameters, paths) |—— scripts/ # Scripts for training/inference |—— logs/ # Application logs |——
requirements.txt # Dependency list
```

 Key Files

```
### **Data Processing**
- `data/preprocess.py`: Clean and tokenize documents/queries.
- `data/split_dataset.py`: Split data into training/validation/test sets.

### **Retrieval Model**
- `models/retriever.py`: Implementation of BM25 or DPR.
- `models/retriever_train.py`: Script to train the retriever.

### **Generation Model**
- `models/generator.py`: Load/fine-tune a transformer model (e.g., T5, BART).
- `models/generator_train.py`: Script to train the generator.

### **Integration Pipeline**
- `scripts/run_rag_pipeline.py`: Main script to combine retrieval and generation.
- `scripts/eval_rag.py`: Script for evaluating the system (ROUGE, BLEU, etc.).

### **Utilities**
- `utils/embeddings.py`: Generate sentence embeddings (e.g., BERT, Sentence-BERT).
- `utils/text_utils.py`: Functions for tokenization, truncation, etc.

### **Config Files**
- `config/settings.yaml`: Paths to data/models, hyperparameters (e.g., `retrieval_top_k`, `generator_temperature`).

### **Dependencies**
- `requirements.txt`:
```

scikit-learn elasticsearch transformers==4.29 torch sentence-transformers

```
## 🧰 Tech Stack
- **Retrieval**: BM25, DPR, Elasticsearch, FAISS.
- **Generation**: T5, BART, GPT-2 (fine-tuned).
- **Embeddings**: BERT, Sentence-BERT, or custom models.
- **Data**: JSON, CSV, or raw text files.
```

This structure allows modularity, scalability, and easy debugging. Adjust based on your specific use case! ✨



Code (excerpt)

snippet_1.txt

```
# 🧩 TypeScript Implementation for RAG System

## 📁 Folder Structure...
```

snippet_2.txt

```
## 📄 Key Files

### **data/preprocess.ts**...
```

snippet_3.txt

```
### **models/retriever.ts**...
```

snippet_4.txt

```
### **models/generator.ts**...
```

snippet_5.txt

```
### **utils/embeddings.ts**...
```

snippet_6.txt

```
### **scripts/run_rag_pipeline.ts**...
```

snippet_7.txt

```
### **config/settings.ts**...
```

snippet_8.txt

```
## 📄 Requirements...
```

snippet_9.txt

This implementation provides a basic framework for a RAG system in TypeScript. You can expand it with:

1. Real BM25/DPR implementations
2. Transformer model loading (e.g., Hugging Face)
3. Embedding...

Review

The implementation provides a solid foundation for a RAG system in TypeScript, with proper separation of concerns and modular design. Here's a breakdown of key improvements and considerations:

1. TypeScript Structure:

- The folder structure follows best practices for TypeScript projects with explicit typing.
- The `tsconfig.json` would need to be configured with proper compiler options (e.g., `moduleResolution`, `target`, `lib`).

2. Key Implementation Details:

- **Preprocessing:** The `preprocess.ts` file demonstrates how to load and split data, which can be extended with more sophisticated text cleaning.
- **Retrieval:** The `BM25Retriever` is a placeholder but can be replaced with a real implementation using libraries like `bm25-ts` or `dpr`.
- **Generation:** The `Generator` class uses a placeholder for transformer models. In practice, this would load a model from Hugging Face using `transformers` or `@huggingface/inference`.
- **Embeddings:** The `embeddings.ts` file is a stub that could be replaced with a real embedding model (e.g., `sentence-transformers`).

3. Pipeline Integration:

- The `run_rag_pipeline.ts` script demonstrates the workflow: preprocess → retrieve → generate.
- This can be extended with evaluation metrics (e.g., ROUGE, BLEU) and logging.

4. Config Management:

- The `settings.ts` file provides a centralized configuration point, which is good practice for production systems.

5. Dependencies:

- The `requirements.txt` includes necessary packages for TypeScript and transformer models. The user would need to install these via `npm install`.

Suggested Improvements:

- Add error handling and type guards in the code.
- Implement a real BM25 or DPR model for retrieval.
- Use a proper embedding model (e.g., `sentence-transformers`) for the generator.
- Add logging and metrics tracking.

This implementation is a good starting point and can be expanded based on specific requirements.

Next Steps

The implementation is a solid foundation for a RAG system in TypeScript/Node.js, with proper structure and modular design. Here's a summary of the key components and their purposes:

Folder Structure

- **data/**: Contains preprocessed documents and queries (e.g., `data.json`).
- **models/**: Houses retrieval and generation models (e.g., `retriever.ts`, `generator.ts`).
- **utils/**: Includes helper functions (e.g., `embeddings.ts`, `text_utils.ts`).
- **scripts/**: Contains the main pipeline script (`run_rag_pipeline.ts`).
- **config/**: Stores configuration settings (`settings.ts`).
- **requirements.txt**: Lists dependencies (e.g., `transformers`, `sentence-transformers`).

Key Implementation Details

1. Data Preprocessing

- `preprocess.ts` loads and splits data into documents and queries.
- Example: `splitData()` is a placeholder for more sophisticated splitting logic.

2. Retrieval

- `retriever.ts` implements a BM25-based retriever (placeholder).
- In practice, this would use a library like `bm25-ts` or `dpr` for semantic retrieval.

3. Generation

- `generator.ts` uses a placeholder for a transformer model (e.g., T5, BART).
- In production, this would load a model from Hugging Face using `transformers` or `@huggingface/inference`.

4. Embeddings

- `embeddings.ts` is a stub for generating sentence embeddings.
- Replace with a real model (e.g., `sentence-transformers` for semantic similarity).

5. Pipeline

- `run_rag_pipeline.ts` orchestrates the RAG workflow:
 - Preprocess → Retrieve → Generate → Output.
- This script can be extended with evaluation metrics (e.g., ROUGE, BLEU) and logging.

6. Configuration

- `settings.ts` centralizes paths, hyperparameters, and model settings.

Example Workflow

```
# Install dependencies
npm install transformers sentence-transformers
```

```
# Run the RAG pipeline
node scripts/run_rag_pipeline.ts
```

Strengths

- **Modular Design**: Separates concerns (data, retrieval, generation, utilities).
- **Scalable**: Can be extended with real models, logging, and metrics.
- **TypeScript**: Enforces type safety and better code structure.

🚧 Areas for Improvement

1. **Error Handling:** Add try/catch blocks for file I/O and model loading.
2. **Real Models:** Replace placeholders with actual models (e.g., `bm25-ts`, `transformers`).
3. **Logging:** Add `console.log()` or use a logging library for debugging.
4. **Evaluation:** Integrate evaluation scripts for ROUGE, BLEU, or human evaluation.

This implementation is a good starting point and can be expanded for production use. Let me know if you'd like help integrating real models or adding evaluation metrics! 🚀