

Software Engineering with Generative AI

Week 7 · RAG, agent

Instructor: Jie “JW” Wu

CS 5090 · Fall 2026

Michigan Technological University

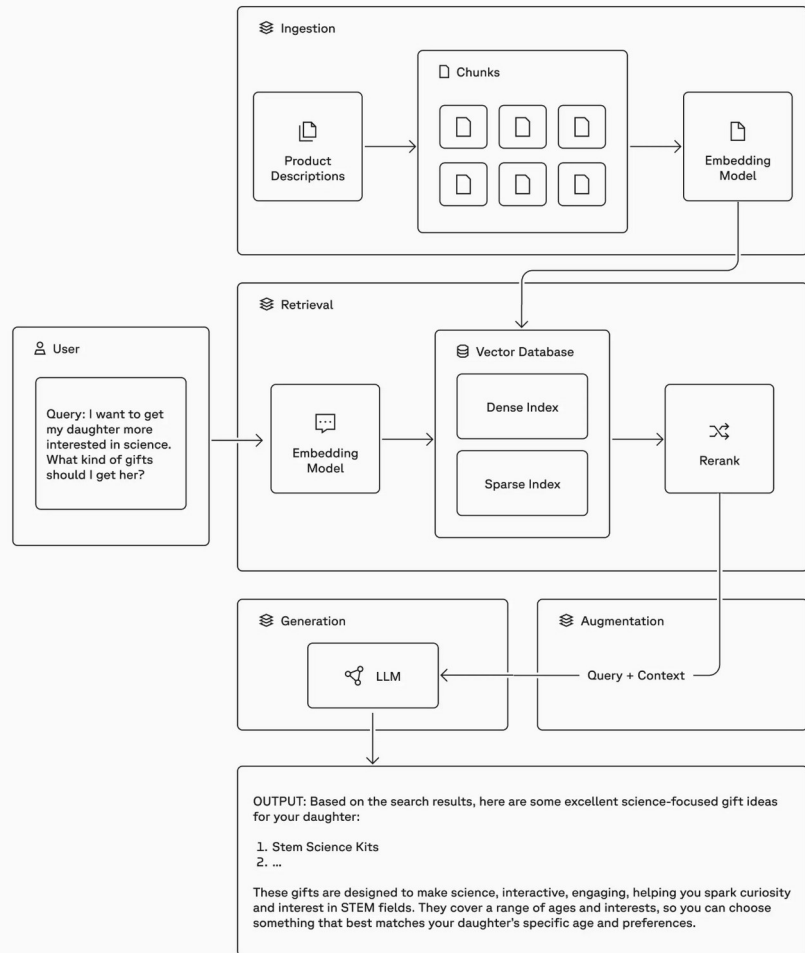


The problems

Problem:

- Knowledge cutoff
- Lack depth in domain-specific knowledge
- Lack private or proprietary data
- Loses trust
- Output generation is probabilistic

https://www.pinecone.io/learn/retrieval-augmented-generation/?utm_source=chatgpt.com

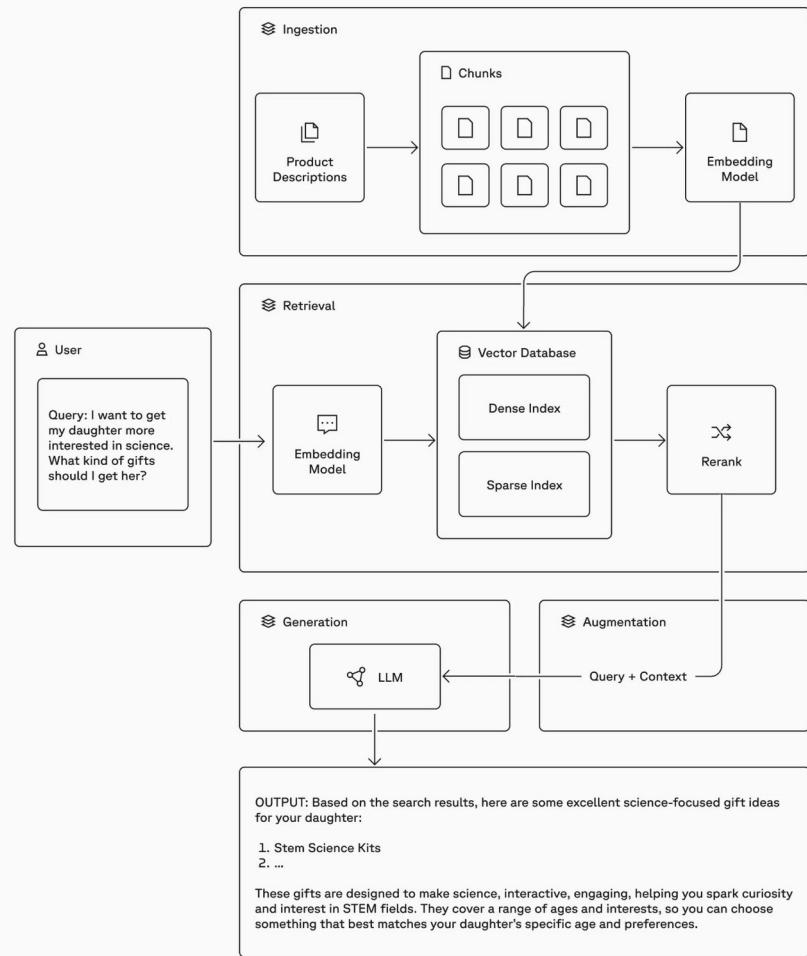


RAG

Retrieval-augmented generation, or RAG, is a technique that uses authoritative, external data to improve the accuracy, relevancy, and usefulness of a model's output. It does this through the following four core components, which we'll cover in more detail later in this article:

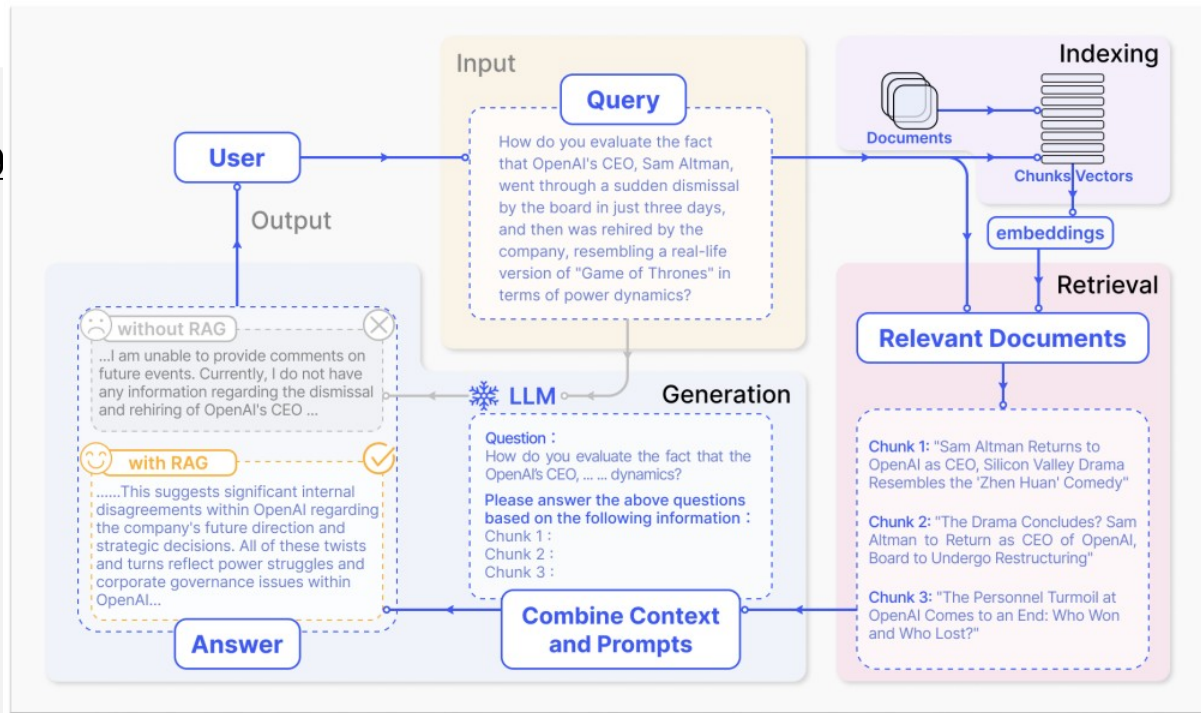
1. **Ingestion:** authoritative data like company proprietary data is loaded into a data source, like a Pinecone vector database
2. **Retrieval:** relevant data is retrieved from an external data source based on a user query
3. **Augmentation:** the retrieved data and the user query are combined into a prompt to provide the model with context for the generation step
4. **Generation:** the model generates output from the augmented prompt, using the context to drive a more accurate and relevant response.

https://www.pinecone.io/learn/retrieval-augmented-generation/?utm_source=chatgpt.com



RAG- example

<https://arxiv.org/pdf/2312.1099>



Advanced RAG: 3 paradigms of RAGs

Comparison between the three paradigms of RAG. (Left) Naive RAG mainly consists of three parts: indexing, retrieval and generation. (Middle) Advanced RAG proposes multiple optimization strategies around pre-retrieval and post-retrieval, with a process similar to the Naive RAG, still following a chain-like structure. (Right) Modular RAG inherits and develops from the previous paradigm, showcasing greater flexibility overall. This is evident in the introduction of multiple specific functional modules and the replacement of existing modules. The overall process is not limited to sequential retrieval and generation; it includes methods such as iterative and

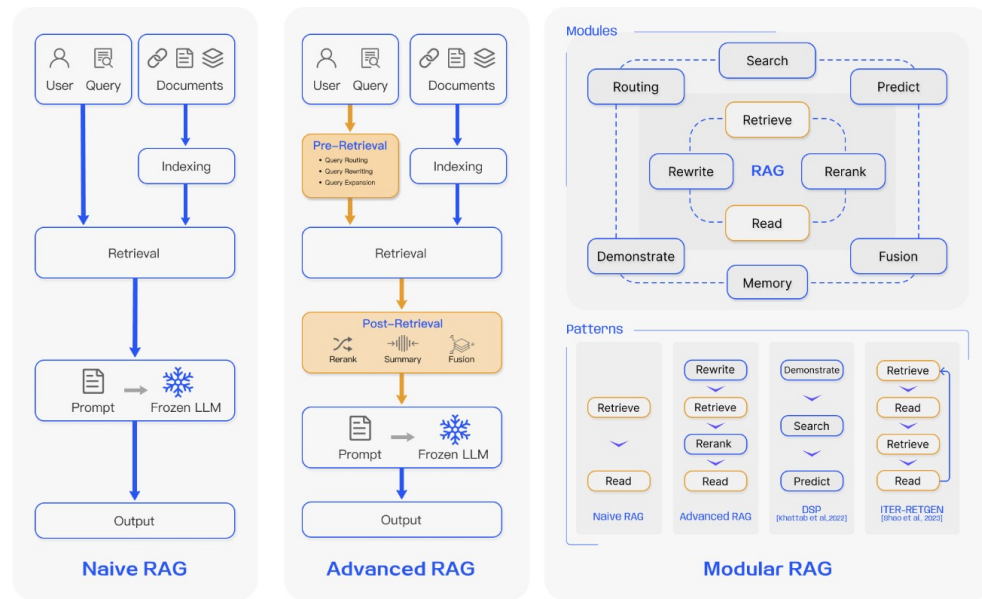
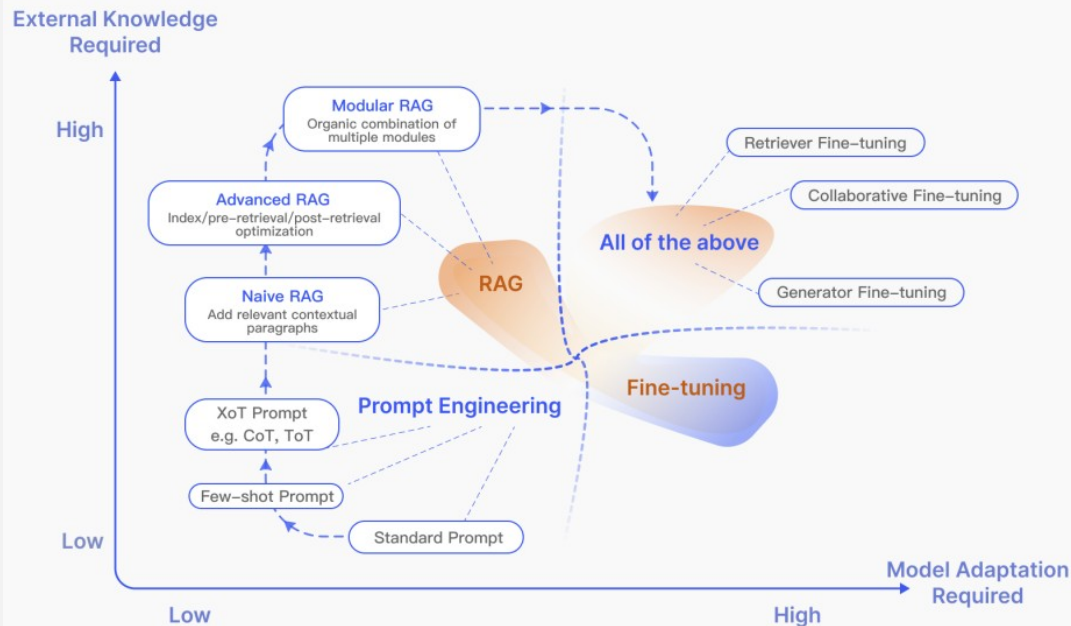


Fig. 3. Comparison between the three paradigms of RAG. (Left) Naive RAG mainly consists of three parts: indexing, retrieval and generation. (Middle) Advanced RAG proposes multiple optimization strategies around pre-retrieval and post-retrieval, with a process similar to the Naive RAG, still following a chain-like structure. (Right) Modular RAG inherits and develops from the previous paradigm, showcasing greater flexibility overall. This is evident in the introduction of multiple specific functional modules and the replacement of existing modules. The overall process is not limited to sequential retrieval and generation; it includes methods such as iterative and adaptive retrieval.

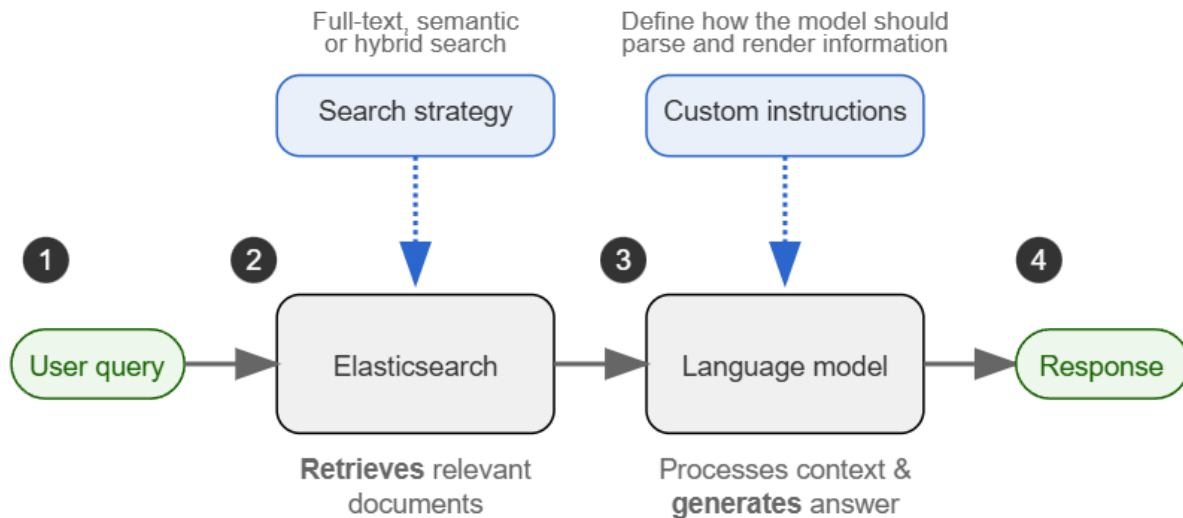
RAG compared w/ other methods

RAG compared with other model optimization methods in the aspects of “External Knowledge Required” and “Model Adaption Required”. Prompt Engineering requires low modifications to the model and external knowledge, focusing on harnessing the capabilities of LLMs themselves. Fine-tuning, on the other hand, involves further training the model. In the early stages of RAG (Naive RAG), there is a low demand for model modifications. As research progresses, Modular RAG has become more integrated with fine-tuning techniques.



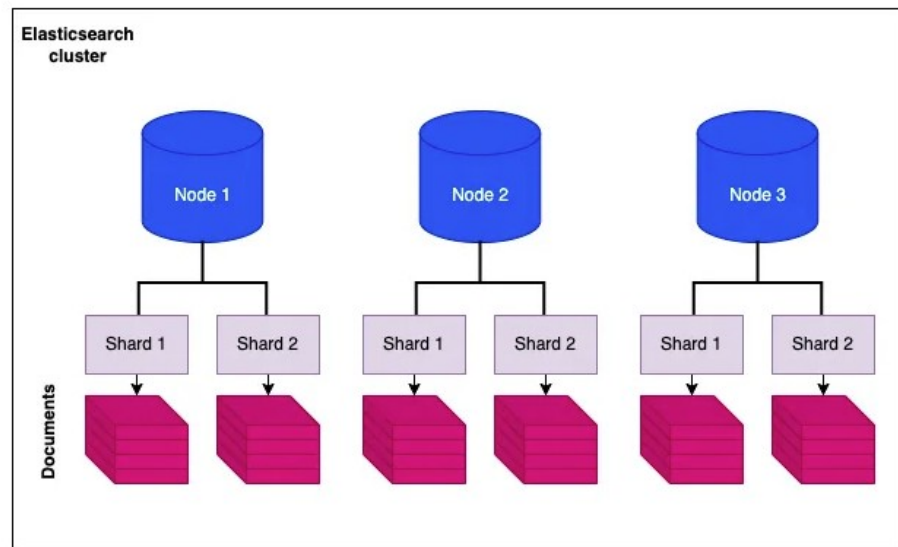
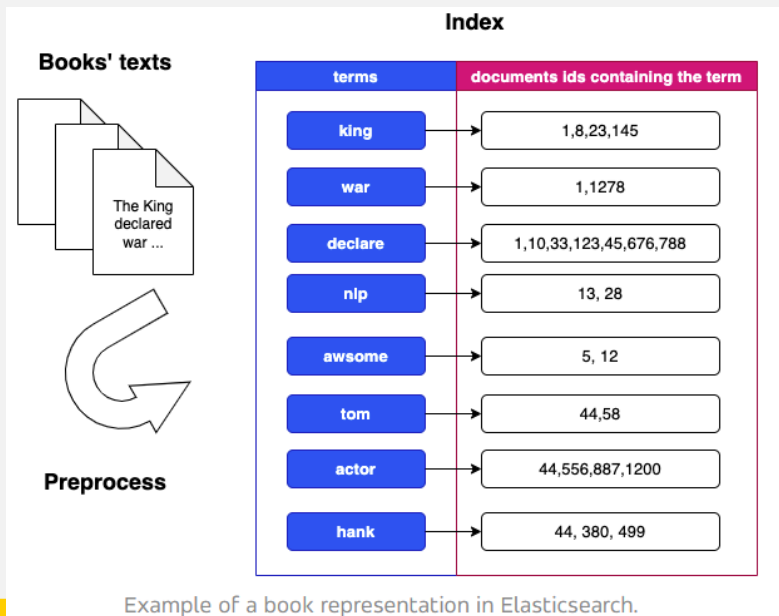
RAG in practice

<https://www.elastic.co/docs/solutions/search/rag>



ElasticSearch 101

<https://neurosyst.com/blog/elasticsearch-introduction>



What are the failure points of RAG?

<https://arxiv.org/pdf/2401.05856>

Seven Failure Points When Engineering a Retrieval Augmented Generation System

Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, Mohamed Abdelrazek
{scott.barnett,stefanus.kurniawan,srikanth.thudumu,zach.brannelly,mohamed.abdelrazek}@deakin.edu.au

Applied Artificial Intelligence Institute
Geelong, Australia

What are the failure points of RAG?

Seven Failure Points When Engineering a Retrieval Augmented Generation System

CAIN 2024, April 2024, Lisbon, Portugal

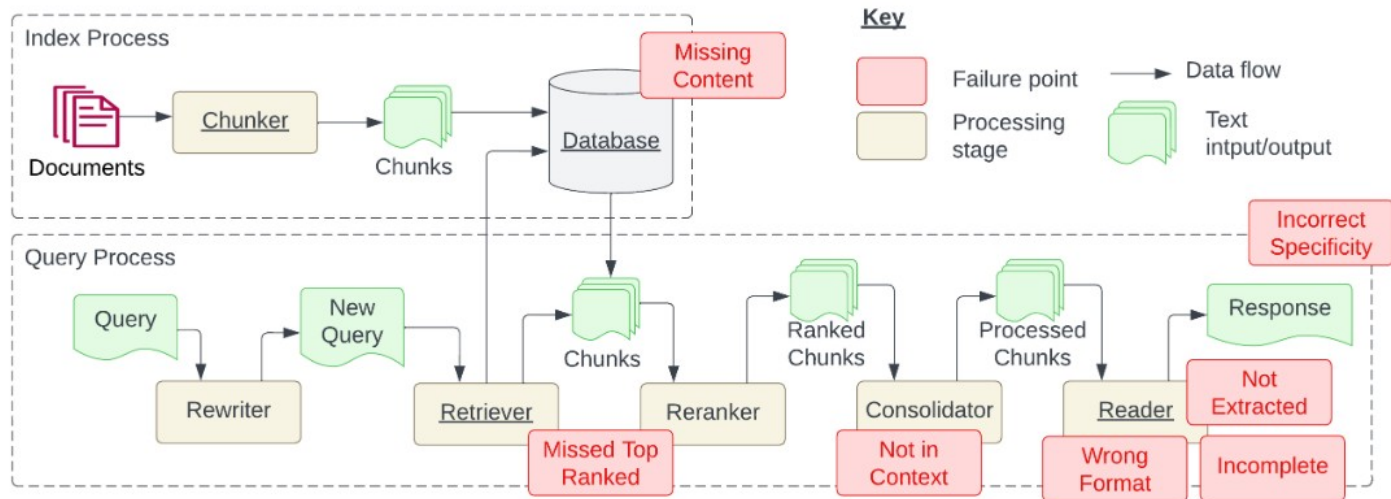


Figure 1: Indexing and Query processes required for creating a Retrieval Augmented Generation (RAG) system. The indexing process is typically done at development time and queries at runtime. Failure points identified in this study are shown in red boxes. All required stages are underlined. Figure expanded from [19].

What are the failure points of RAG? (summary)

- FP1: Missing Content – The answer is not in the available documents, but the system may generate an unsupported response.
- FP2: Missed Top-Ranked Documents – Relevant documents are not included in the top-K results.
- FP3: Context Consolidation Limitations – Retrieved answers are lost during context selection or consolidation.
- FP4: Extraction Failure – The answer is in the context, but the LLM fails to extract it correctly due to noise or contradictions.
- FP5: Wrong Format – The response does not follow the requested format, such as a table or list.
- FP6: Incorrect Specificity – The answer is too broad or too detailed to meet the user's needs.

What are the failure points of RAG? (lessons learnt)

FP	Lesson	Description	Case Studies
FP4	Larger context get better results (Context refers to a particular setting or situation in which the content occurs)	A larger context enabled more accurate responses (8K vs 4K). Contrary to prior work with GPT-3.5 [13]	AI Tutor
FP1	Semantic caching drives cost and latency down	RAG systems struggle with concurrent users due to rate limits and the cost of LLMs. Prepopulate the semantic cache with frequently asked questions [1].	AI Tutor
FP5-7	Jailbreaks bypass the RAG system and hit the safety training.	Research suggests fine-tuning LLMs reverses safety training [11], test all fine-tuned LLMs for RAG system.	AI Tutor
FP2, FP4	Adding meta-data improves retrieval.	Adding the file name and chunk number into the retrieved context helped the reader extract the required information. Useful for chat dialogue.	AI Tutor
FP2, FP4-7	Open source embedding models perform better for small text.	Opensource sentence embedding models performed as well as closed source alternatives on small text.	BioASQ, AI Tutor
FP2-7	RAG systems require continuous calibration.	RAG systems receive unknown input at runtime requiring constant monitoring.	AI Tutor, BioASQ
FP1, FP2	Implement a RAG pipeline for configuration.	A RAG system requires calibrating chunk size, embedding strategy, chunking strategy, retrieval strategy, consolidation strategy, context size, and prompts.	Cognitive Reviewer, AI Tutor, BioASQ
FP2, FP4	RAG pipelines created by assembling bespoke solutions are suboptima.	End-to-end training enhances domain adaptation in RAG systems [18].	BioASQ, AI Tutor
FP2-7	Testing performance characteristics are only possible at runtime.	Offline evaluation techniques such as G-Evals [14] look promising but are premised on having access to labelled question and answer pairs.	Cognitive Reviewer, AI Tutor

Table 2: The lessons learned from the three case studies with key takeaways for future RAG implementations

Key SE Research Directions for RAG Systems

- Chunking & Embeddings: Improve document segmentation, embedding quality, and retrieval accuracy.
- RAG vs. Fine-Tuning: Compare accuracy, cost, latency, security, and flexibility to determine the best approach.
- Testing & Evaluation: Develop domain-specific test datasets and reliable quality metrics for RAG systems.
- Monitoring & Adaptation: Monitor performance changes across model updates and adapt systems to maintain reliability.
- Future Research: Establish standardized evaluation methods and optimize retrieval, context selection, and overall system performance.

DocPrompting: GENERATING CODE BY RETRIEVING THE DOCS

<https://arxiv.org/pdf/2207.05987>

Publicly available source-code libraries are continuously growing and changing. This makes it impossible for models of code to keep current with all available APIs by simply training these models on existing code repositories. Thus, existing models *inherently cannot generalize* to using unseen functions and libraries, because these would never appear in their training data. In contrast, when human programmers use functions and libraries for the first time, they frequently refer to textual resources such as code manuals and documentation, to explore and understand the available functionality. Inspired by this observation, we introduce DocPrompting: a natural-language-to-code generation approach that explicitly leverages code documentation by (1) retrieving the relevant documentation pieces given a natural language (NL) intent, and (2) generating code based on the NL intent and the retrieved documentation. DocPrompting is general: it can be applied to any programming language, and is agnostic to the underlying neural model. We demonstrate that DocPrompting consistently improves NL-to-code models: DocPrompting improves strong base models such as CodeT5 by 2.85% in pass@1 (52% relative gain) and 4.39% in pass@10 (30% relative gain) in execution-based evaluation on the popular Python CoNaLa benchmark; on a new Bash dataset `tlldr`, DocPrompting improves CodeT5 and GPT-Neo-1.3B by up to absolute 6.9% exact match.¹

DocPrompting: GENERATING CODE BY RETRIEVING THE DOCS

From S: Code generation for new or unseen library

<https://arxiv.org/pdf/2207.05987>

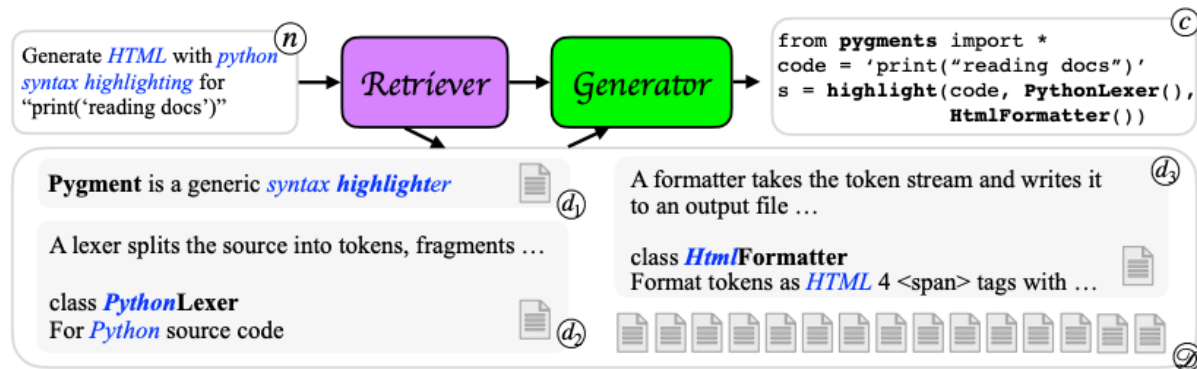


Figure 1: DocPrompting: given an NL intent $\textcircled{n}$, the retriever retrieves a set of relevant documentation $\{ \textcircled{d_1}, \textcircled{d_2}, \textcircled{d_3} \}$ from a documentation pool $\textcircled{D}$. Then, the generator generates the code $\textcircled{c}$ based on the NL and retrieved docs. DocPrompting allows the model to generalize to previously unseen usages by reading those docs. *Italic blue* highlights the shared tokens between NL and docs; **Bold** shows shared tokens between docs and the code snippet.

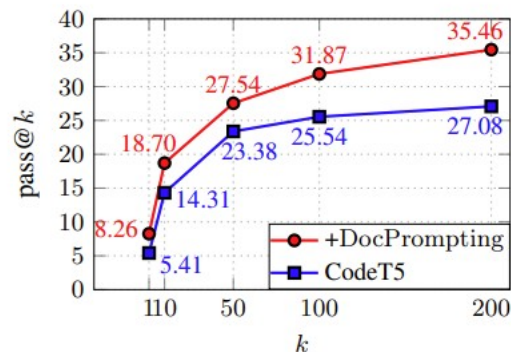


Figure 3: Pass@k of CodeT5 with and without DocPrompting on 100 CoNaLa examples.

Why Does Documentation Improve Code Generation Accuracy?

- **Bridges Natural Language and Code:** Documentation connects user intentions with relevant code through natural-language descriptions and function signatures.
- **Improves Code Understanding:** Retrieved documentation provides additional context to help LLMs map user requests to the correct implementation.
- **Validates Through N-Gram Overlap:** Compares textual overlap between natural-language intents and code, both with and without the top-10 retrieved documents.
- **Key Takeaway:** Documentation helps LLMs generate more accurate code by reducing the gap between user intent and code implementation.

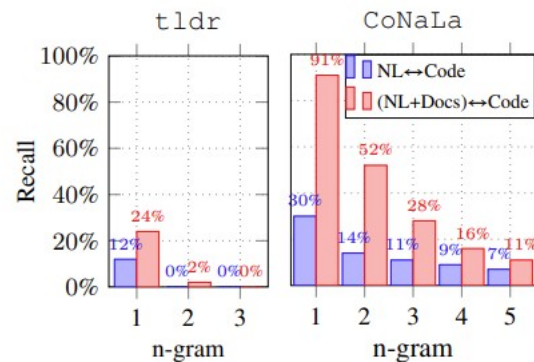


Figure 4: Using documentation significantly increases the n -gram overlap recall between the input and the output, in tldr and CoNaLa.

Technical presentation guide

Markus Püschel, “Small Guide to Giving Presentations”

<https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Seminars/AS%202016/guide-presentations.pdf>

Simon Peyton Jones, “How to Give a Great Research Talk”

<http://ccr.sigcomm.org/online/files/p83-keshavA.pdf>

“How to Read a Paper” S. Keshav

What is Agentic AI

Definition:

Agentic AI is AI that works toward a goal by choosing actions, using tools, and adjusting its approach based on feedback.

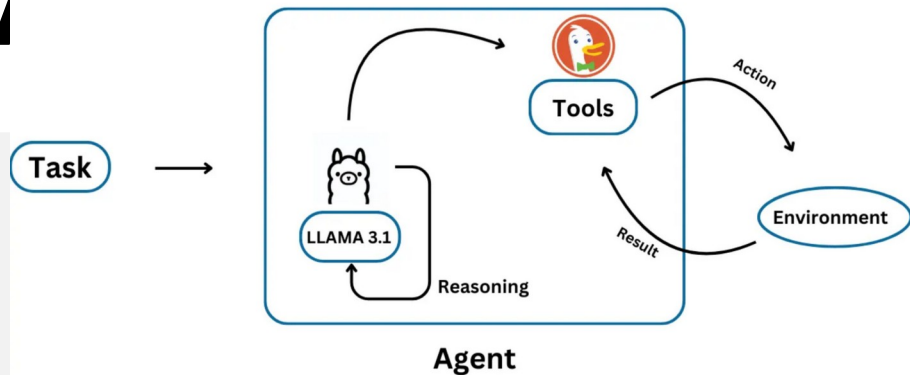
Rather than producing a single response, it can carry out a sequence of steps.

For example, you ask: **“Fix this bug in my repository.”**

An AI agent might:

1. **Inspect** the relevant files and error report.
2. **Plan** a possible fix.
3. **Act** by editing the code.
4. **Check** the result by running tests.
5. **Adapt** if tests fail, then try again or ask you for clarification.

The central idea is a **decision–action–feedback loop**: what happens after an action influences what the agent does next.



ReAct

Understand how agents operate

<https://arxiv.org/pdf/2210.03629>

Loop of Observe → Plan → Act

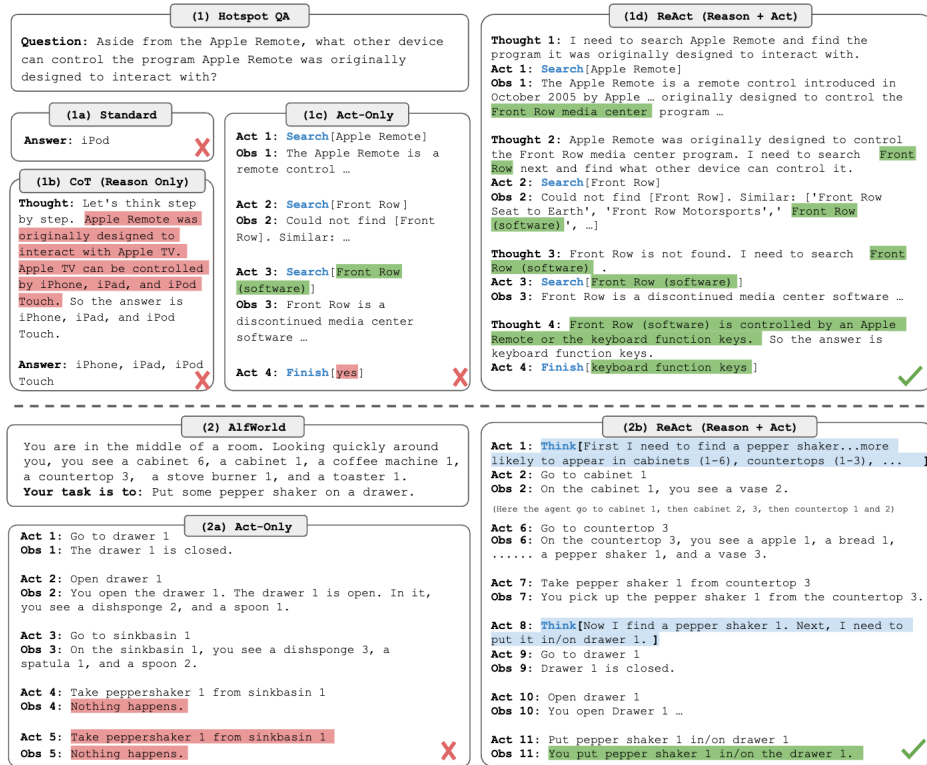


Figure 1: (1) Comparison of 4 prompting methods, (a) Standard, (b) Chain-of-thought (CoT, Reason Only), (c) Act-only, and (d) ReAct (Reason+Act), solving a HotpotQA (Yang et al., 2018) question. (2) Comparison of (a) Act-only and (b) ReAct prompting to solve an AlfWorld (Shridhar et al., 2020b) game. In both domains, we omit in-context examples in the prompt, and only show task solving trajectories generated by the model (Act, Thought) and the environment (Obs).

Tool Use

A tool is an interface through which a language model can invoke an external computer program.

- Access information or actions unavailable in the model's parameters.
- Delegate operations external programs perform more reliably or efficiently.

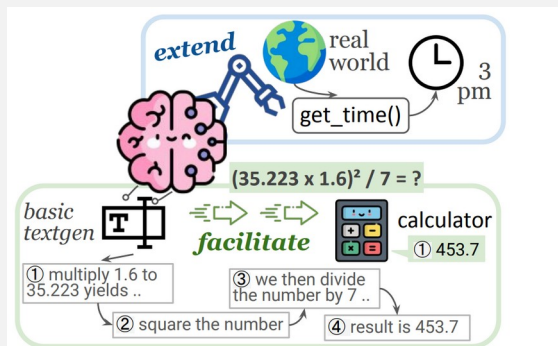


Figure 1: Illustration of tools extending and facilitating LM task-solving.

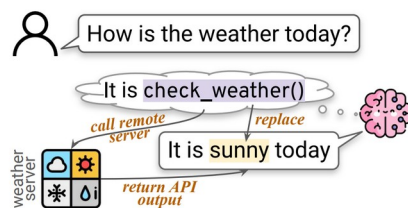


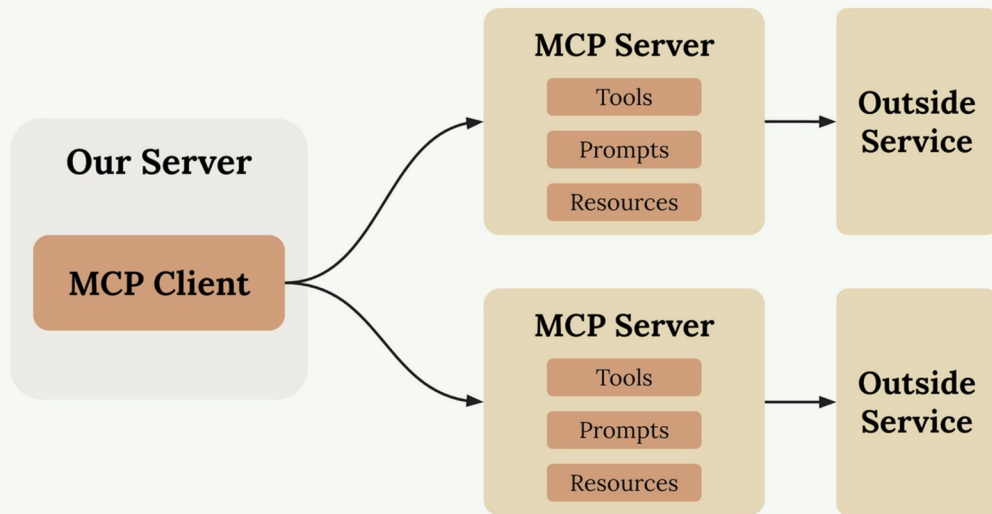
Figure 2: The basic tool use paradigm. LM calls `check_weather` tool by generating text tokens. This call triggers the server to execute the call and return the output `sunny`, using which the LM replaces the API call tokens in the response to the user.

<https://arxiv.org/pdf/2403.15452>

MCP (Model Context Protocol)

Model Context Protocol (MCP) is a communication layer that provides Claude with context and tools without requiring you to write a bunch of tedious integration code. Think of it as a way to shift the burden of tool definitions and execution away from your server to specialized MCP servers.

<https://academy.claude.com/courses/introduction-to-model-context-protocol/introducing-mcp>



Reflexion

Understand how feedback and memory improve behavior
<https://arxiv.org/pdf/2303.11366>

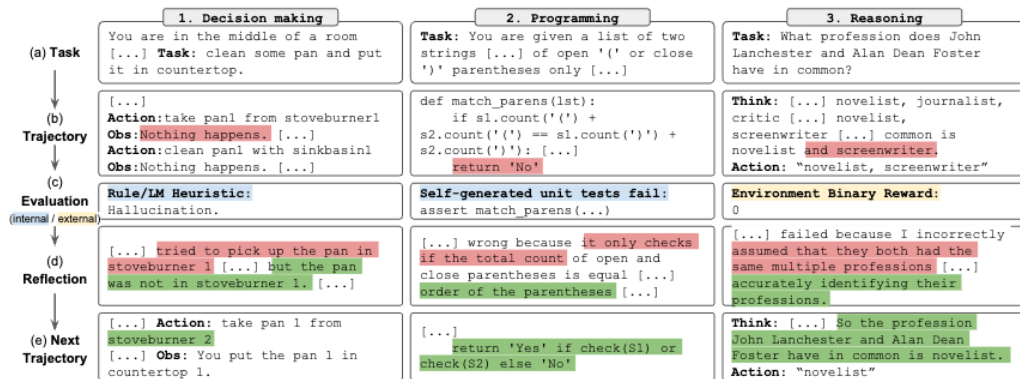


Figure 1: Reflexion works on decision-making 4.1, programming 4.3, and reasoning 4.2 tasks.

Related work on reasoning and decision-making					
Approach	Self refine	Hidden constraints	Decision making	Binary reward	Memory
Self-refine [15]	✓	✗	✗	✗	✗
Beam search [27]	✓	✓	✓	✓	✗
Reflexion (ours)	✓	✓	✓	✓	✓

Related work on programming					
Approach	Test execution	Debugging	Self-generated tests	Multiple languages	Self-reflection
AlphaCode [14]	✓	✗	✗	✓	✗
CodeT [5]	✓	✗	✓	✗	✗
Self-debugging [7]	✓	✓	✗	✗	✗
CodeRL [12]	✓	✓	✗	✗	✗
Reflexion (ours)	✓	✓	✓	✓	✓

SWE-Agent

Agent-Computer Interface

<https://arxiv.org/pdf/2405.15793>

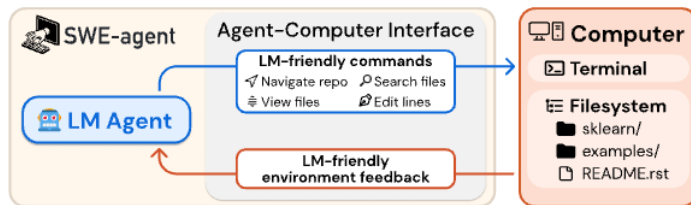


Figure 1: SWE-agent is an LM interacting with a computer through an agent-computer interface (ACI), which includes the commands the agent uses and the format of the feedback from the computer.

KISS agent

The Philosophy: Radical Simplicity

<https://arxiv.org/pdf/2604.23822>

https://dev.to/koushik_sen_d549bf321e6fb/meet-the-kiss-agent-framework-2ij6

Table 1: KISS Sorcar compared with Claude Code and Cursor, reproduced from the project README.

Capability	KISS Sorcar	Claude Code	Cursor
Interfaces	CLI + VS Code extension + web/mobile app	CLI + mobile app	Custom VS Code
AI Discovery	✓ simply via prompt	×	×
GEPA Prompt Optimization	✓ simply via prompt	×	×
Multiple models from multiple vendors in the same task	✓ Mix OpenAI, Anthropic, Gemini, Together, Z.AI, Moonshot AI, OpenRouter, Claude Code CLI, and Codex CLI	×	×
Primary focus	✓ Quality : rigorous review, end-to-end tests	Speed and developer ergonomics	Speed
Core Agents # LoC	~2900	Unknown	Unknown
Models in bundled catalog	504 across 9 provider categories	Claude family only	Subset chosen by Cursor
Bring your own API key / endpoint	✓ Yes: keys stay on your machine	✓ Anthropic key	△ Routed through Cursor backend
Open source	✓ Apache-2.0	×	×
Price	Free framework; pay only your chosen model provider	Subscription / API usage	Subscription
Run on top of Claude Code / Codex CLI	✓ cc/* and codex/* namespaces	N/A	×
Messaging and communication channels	✓ 23 third-party agents, including Slack, Gmail, Phone Control, SMS, and WhatsApp	Partial: Slack, mobile Remote Control, and research-preview channels for Telegram, Discord, and iMessage; no documented built-in Gmail, WhatsApp, phone-call, or SMS channel	Partial: Slack and Microsoft Teams Cloud Agent integrations; no documented built-in Gmail, WhatsApp, phone-call, or SMS channel
Terminal Bench 2.0 score	62.2%	58%	61.7% (Cursor agent)

