

Node.js

# Session Management with Node.js

# What is a Session?

- Definition: A session is a logical conversation between a web server and a web browser that spans multiple requests and responses.
- Purpose: To maintain stateful information about a user or client across multiple interactions.
- Example: Storing user login details, shopping cart contents, or user preferences.
-

# Introduction to Session Management

- Session management is crucial for maintaining user state and data across multiple requests in a web application.
- Key Aspects:
  - - User Authentication
  - - State Preservation
  - - Security Handling

# Why Use Session Management?

- - Maintain user login status across pages
- - Track user activities
- - Improve security by managing sessions effectively

# Session Management Techniques

- 1. Using Cookies
- 2. Using Session IDs stored on the server
- 3. Token-based Authentication (JWT)

# Session Management in Node.js

- Popular libraries:

- `express-session`: The most commonly used session middleware for Express.js applications.
- `connect-mongo`: Stores sessions in a MongoDB database.
- `connect-redis`: Stores sessions in a Redis database.

- Key concepts:

- Session ID: A unique identifier assigned to each session.
- Session store: Where session data is stored.

# Implementing Session Management with Express.js

- 1. Install express-session:
  - `npm install express-session`
- 2. Sample Code:
- ```
app.use(session({  
  secret: 'your-secret-key', // Important for security  
  resave: false,  
  saveUninitialized: true,  
  store: new MongoStore({ // Or any other store of your choice  
    url: 'mongodb://localhost/sessions'  
  })  
}));
```
- Databases (MongoDB, Redis, etc.): Scalable and persistent, but require additional setup.



# Working Example

```
app.get('/', (req, res) => {  
  if (req.session.views) {  
    req.session.views++;  
    res.send(`Views: ${req.session.views}`);  
  } else {  
    req.session.views = 1;  
    res.send('Welcome to the session demo.  
Refresh!');  
  }  
});
```

# Best Practices for Session Management

- - Use secure cookies (HTTPS)
- - Set session expiration timeouts
- - Store minimal data in sessions
- - Use strong session secrets

# Conclusion

- Session management is vital for building secure and user-friendly web applications.
- Implementing sessions in Node.js with Express is straightforward and highly customizable.

# Node.js – Express

- Express js is a very popular web application framework built to create Node.js Web based applications.
- It provides an integrated environment to facilitate rapid development of Node based Web applications.
- Express framework is based on Connect middleware engine and used Jade html template framework for HTML templating.
- Core features of Express framework:
  - Allows to set up middlewares to respond to HTTP Requests.
  - Defines a routing table which is used to perform different action based on HTTP method and URL.
  - Allows to dynamically render HTML Pages based on passing arguments to templates.

# Installing Express

- Create a directory myapp which is under the nodejs\_workspace directory.  
(mkdir myapp, then cd myapp).  
[<http://expressjs.com/en/starter/installing.html>]
- Use the npm init command under the new myapp folder to create a package.json file for your application – myapp.  
npm init
- NOTE: you can hit RETURN to accept the defaults for most of them, except entry point is app.js:  
entry point: app.js
- Install express locally in the nodejs\_workspace directory, and save it in the dependencies list:  
npm install express --save
- Install the Express framework globally using npm if you want to create web application using node terminal.  
npm install express -g --save

# Node.js – Web Application with express

- <http://expressjs.com/en/starter/hello-world.html>
- In myapp directory, create a file named app.js and add the following codes:

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
    res.send('Hello World!');
});
app.listen(3000, function () {
    console.log('app.js listening to http://localhost:3000/');
});
```
- Run the app.js in the server:

```
node app.js
```
- Then, load <http://localhost:3000/> in a browser to see the output.

# Web application – get/post form (server.js)

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {           // To display index.html
    res.sendFile(__dirname + "/index.html");
});
app.get('/process_get', function (req, res) { // To process get method
    var response = { fname: req.query.fname,
        lname: req.query.lname }; // preparing the output in JSON format
    console.log(response);
    res.json(response);
});
```

# Web application – get/post form (server.js)

```
var bodyParser = require('body-parser');    // To process post method
var urlenParser = bodyParser.urlencoded({ extended: false}); // creating the
    application/x-www-form-urlencoded parser
app.post('/process_post', urlenParser, function (req, res) {
    var response = { fname: req.body.fname,
        lname: req.body.lname }; // preparing the output in JSON format
    console.log(response);
    res.end(JSON.stringify(response));
});
var server = app.listen(8081, function () {
    console.log('server.js is listening at http://127.0.0.1:8081/index.html or
        http://localhost:8081/index.html');
});
console.log('End of program');
```



# `app.use()` is used to mount middleware functions.

`app.use()` can be used in several ways:

## **Without a path:**

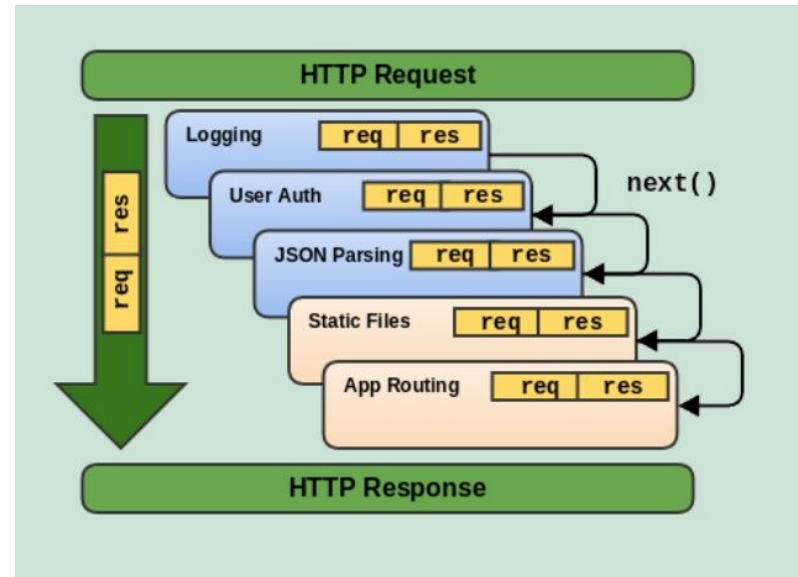
When used without a path, the middleware function will be executed for every request to the application.

## **With a path:**

When used with a path, the middleware function will be executed only for requests whose path matches the specified path.

## **With multiple middleware functions:**

`app.use()` can also take multiple middleware functions as arguments. In this case, the middleware functions will be executed in the order they are provided.

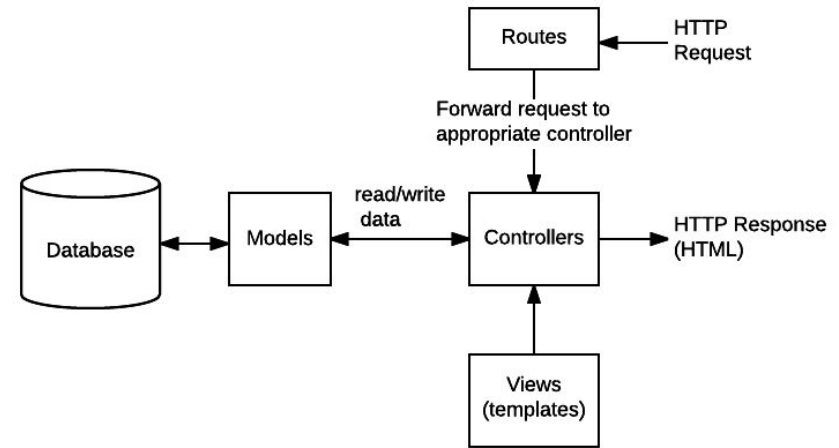


`app.use()` is commonly used for tasks such as:

- **Logging:** Logging incoming requests or other application events.
- **Authentication:** Verifying user credentials before allowing access to certain routes.
- **Authorization:** Checking if a user has the necessary permissions to access a resource.
- **Data parsing:** Parsing request bodies (e.g., JSON or URL-encoded data).
- **Serving static files:** Serving static files such as HTML, CSS, and JavaScript files.
- **Error handling:** Handling errors that occur during the request-response cycle.

# Basic Routing

- **Routing** refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.
- Route definition takes the following structure:  
`app.METHOD(PATH, HANDLER);`
- Where:
  - app is an instance of express.
  - METHOD is an HTTP request method, in lower case.
  - PATH is a path on the server
  - HANDLER is the function executed when the route is matched.



# Route methods

- A route method is derived from one of the HTTP methods, and is attached to an instance of the express class.
- The following code is an example of routes that are defined for the GET and POST methods to the root of the app.

- GET method route:

```
app.get('/', function (req, res) {  
    res.send('Get request to the homepage');  
});
```

- POST method route:

```
app.post('/', function (req, res) {  
    res.send('Post request to the homepage');  
});
```

# Route paths based on strings

- This route path will match requests to the root route, /.  

```
app.get('/', function (req, res) {  
  res.send('root'); });
```
- This route path will match requests to /about.  

```
app.get('/about', function (req, res) {  
  res.send('about'); });
```
- This route path will match requests to /random.  

```
app.get('/random', function (req, res) {  
  res.send('random'); });
```

# Route paths based on string patterns

- This route path will match `acd` and `abcd`.  
`app.get('/ab?cd', function(req, res) { res.send('ab?cd'); });`
- This route path will match `abcd`, `abbc`, `abbbcd`, and so on.  
`app.get('/ab+cd', function(req, res) { res.send('ab+cd'); });`
- This route path will match `abcd`, `abxcd`, `abRANDOMcd`, `ab123cd`, and so on.  
`app.get('/ab*cd', function(req, res) { res.send('ab*cd'); });`
- This route path will match `/abe` and `/abcde`.  
`app.get('/ab(cd)?e', function(req, res) { res.send('ab(cd)?e'); });`

# Route paths based on regular expressions

- This route path will match anything with an “a” in the route name.

```
app.get(/a/, function(req, res) {  
  res.send('/a/');  
});
```

- This route path will match butterfly and dragonfly, but not butterflyman, dragonfly man, and so on.

```
app.get(/.*fly$/, function(req, res) {  
  res.send('/.*fly$/');  
});
```

# Route Parameters

- Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

Route path: `/users/:userId/books/:bookId`

Request URL:

`http://localhost:3000/users/34/books/8989`    `req.params:`  
`{ "userId": "34", "bookId": "8989" }`

- To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', function(req, res) {  
    res.send(req.params);  
});
```

# Express Routers

- A router object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.
- A router behaves like middleware itself, so you can use it as an argument to [app.use\(\)](#) or as the argument to another router’s [use\(\)](#) method.
- The top-level express object has a [Router\(\)](#) method that creates a new router object.

```
var express = require('express');  
var app = express();  
var router = express.Router();
```



# Nested routers

1. A nested router in Express is just a router mounted under another router's path, so routes share a URL prefix (and often URL params).
2. Create routers:  
Use `express.Router()` to create separate router instances for different sections of your application.
3. Mount sub-routers:  
Use the `use()` method to mount a sub-router onto a parent router, specifying a path prefix.
4. Handle requests:  
Define routes within each router to handle specific requests for their respective paths.

```
// app.js
app.use('/users',
  usersRouter);
```

```
// routes/users/index.js
router.use('/:userId/posts',
  postsRouter);
```

```
// routes/users/posts.js
const router =
  express.Router({
    mergeParams: true });
router.get('/', (req, res) => {
  res.json({ userId:
    req.params.userId, posts:
    [] });
});
```

# App == Router

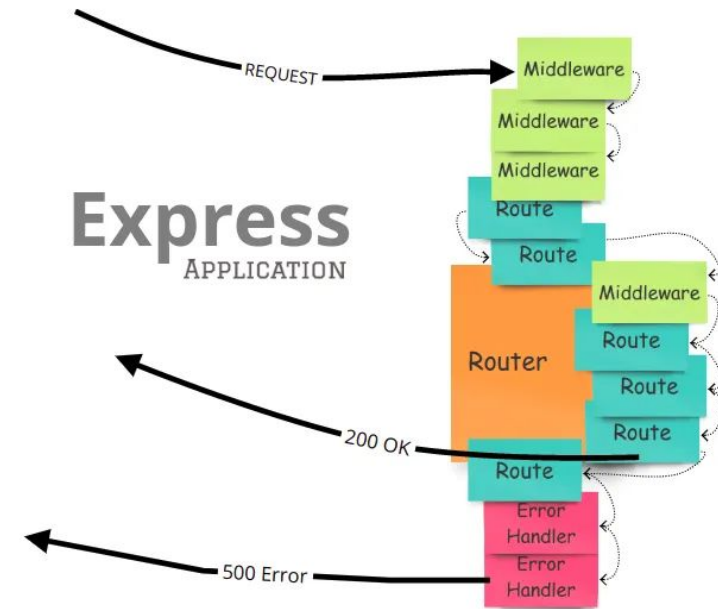
.use() method:

- It is used for registering Routers, Middlewares, Routes & Error handlers.
- It can be used with or without a path as first param
- Both an App and a Router have a .use() method

Every Express app at it's core is a Router.

When we create an app using `express()`, we are essentially creating a root-level Router.

- Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.



# Route handlers

- You can provide multiple callback functions that behave like [middleware](#) to handle a request.
- Route handlers can be in the form of a function, an array of functions, or combinations of both.

# Route handlers samples

- A single callback function can handle a route. For example:

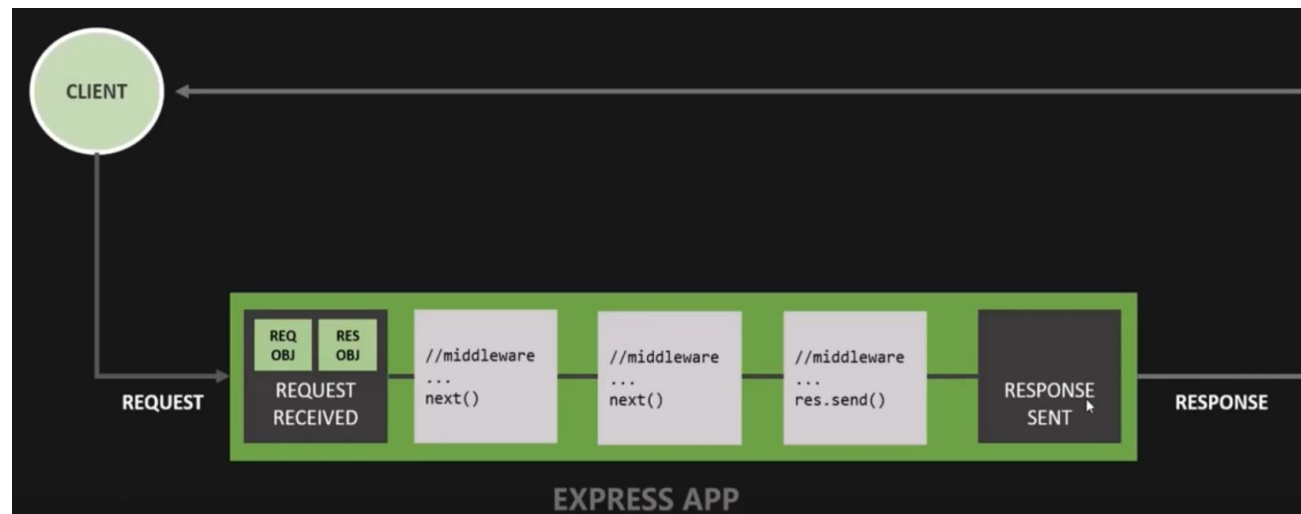
```
app.get('/example/a', function (req, res) {  
  res.send('Hello from A!'); });
```

- More than one callback function can handle a route (make sure you specify the next object). For example:

```
app.get('/example/b', function (req, res, next) {  
  console.log('the response will be sent by the next  
function ...');    next();  
}, function (req, res) { res.send('Hello from B!'); });
```

# Express Middleware

- **Middleware** functions are functions that have access to the [request object](#) (req), the [response object](#) (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.
- Middleware functions can perform the following tasks:
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.
- If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.



# Middleware sample

- <http://expressjs.com/en/guide/writing-middleware.html>
- Middleware function requestTime add the current time to the req (the request object).
- hkbu\_app\_middleware.js

# Router Methods

- `router.all(path, [callback, ...] callback)`
- This method is extremely useful for mapping “global” logic for specific path prefixes or arbitrary matches. For example, if you placed the following route at the top of all other route definitions, it would require that all routes from that point on would require authentication and automatically load a user. Keep in mind that these callbacks do not have to act as end points; `loadUser` can perform a task, then call `next()` to continue matching subsequent routes.

```
router.all('*', requireAuthentication, loadUser);
```

- Another example of this is white-listed “global” functionality. Here the example is much like before, but it only restricts paths prefixed with “/api”:

```
router.all('/api/*', requireAuthentication);
```

# Creating a RESTful service

## Steps

1. Define the domain and data
2. Organize the data in to groups
3. Create URI to resource mapping
4. Define the representations to the client  
(XML, HTML, CSS, ...)
5. Link data across resources (connectedness or hypermedia)
6. Create use cases to map events/usage
7. Plan for things going wrong

Top



Down



## CATALOG PAGE

|                          |       |
|--------------------------|-------|
| <input type="checkbox"/> | ===== |
| <input type="checkbox"/> | ===== |
| <input type="checkbox"/> | ===== |
| ⋮                        | ⋮     |

SELECT  
PRODUCT

## CART PAGE

| Qty                      | Desc  | Price | Total                |
|--------------------------|-------|-------|----------------------|
| <input type="checkbox"/> | ===== | \$-   | \$-                  |
| <input type="checkbox"/> | ===== | -     | -                    |
|                          |       |       | <input type="text"/> |

RECALC.

CONTINUE  
SHOPPING

CHECKOUT

Maybe show  
cart summary?

CHECKOUT?

Order Summary (1 line per item)

Name:

Address:

Payment:

PAY

## Product:

- name
- description
- image
- price

## Seller Details:

- login name
- password

## Cart:

|     |
|-----|
| • ? |
|-----|

## Order:

- buyer details
- payment details
- shipping status

## Line Item:

- product
- quantity
- price

a..n

1..n

# References

- <https://www.w3schools.com/nodejs/>
- <https://www.w3schools.in/express-js/routing>
- <https://www.tutorialspoint.com/expressjs/index.htm>
- [Express.com](https://express.com)
- [Tutorials Point](https://www.tutorialspoint.com)

# File Structure of Express

An Express.js file structure is important for a well-organized application because it enhances maintainability, scalability

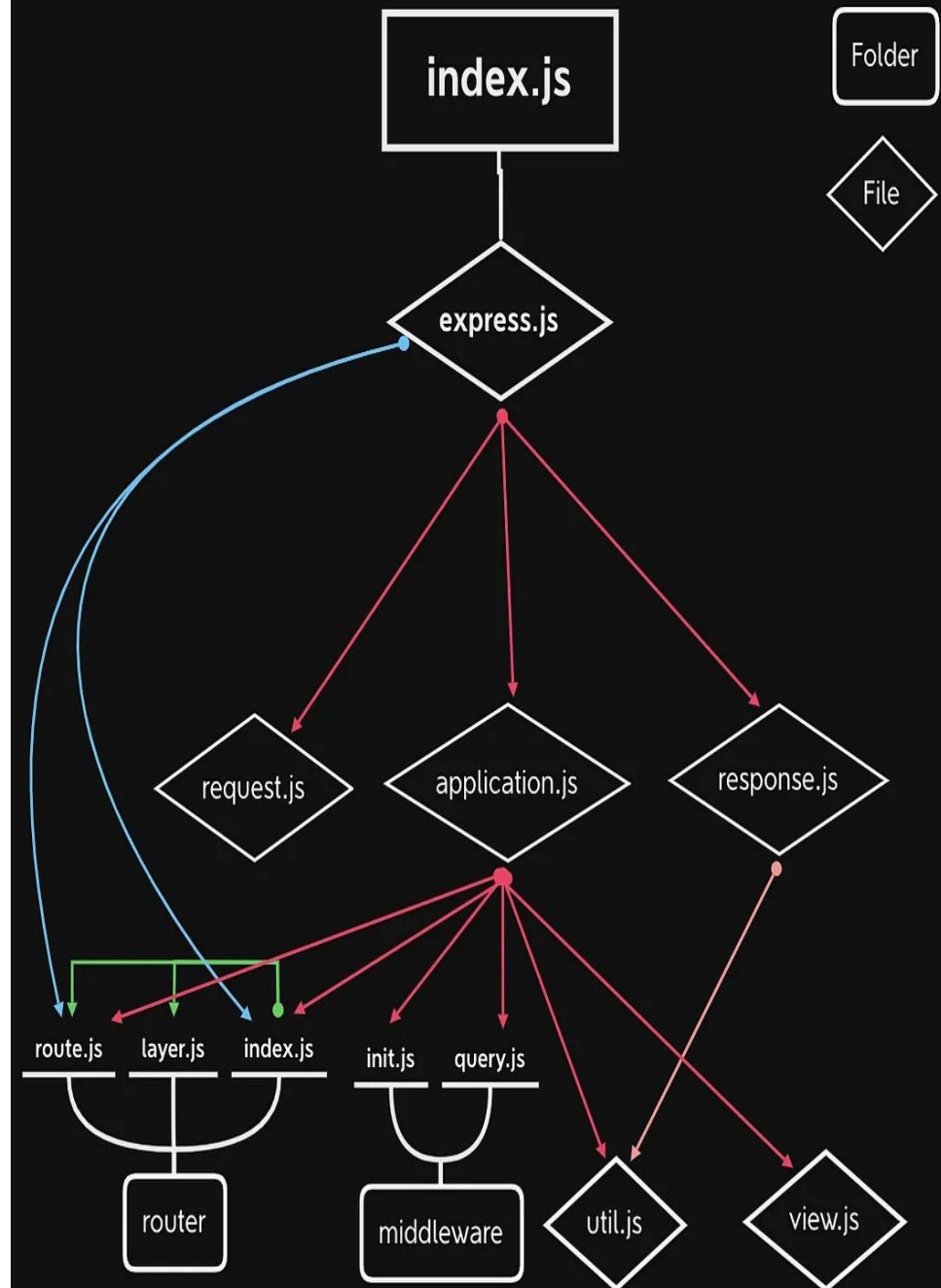
index.js : the entry point of the module

express.js: grouping different parts of the application into one object

application.js: app level properties and methods

router.js: router level properties and methods

route.js and layer.js: essential files for completing functionality for router



# Prompt Engineering — Definition

**Prompt Engineering** is the practice of crafting the input text (and optional parameters) that guide a model's behavior for a task.

It focuses on **what you say**: task framing, constraints, output format, style, and evaluation cues.

Task + Constraints + Output Format (+ Hints/Examples)

# Prompt Engineering

- Why is it Important?
  - Helps optimize model performance.
  - Used to evaluate model output and improve safety measures.
  - Requires experimentation and iteration—there's no perfect prompt.

# The Prompt (single instruction)

A **prompt** contains just enough to perform one task.

```
Summarize this article in 3 bullet points, focusing on the causes.
```

**Strengths:** fast, minimal tokens.

**Limits:** ambiguity, higher variance, no examples, no memory.

# Elements of a Prompt

- A prompt is composed with the following components:

- Instructions
- Context
- Input data
- Output indicator

Classify the text into neutral, negative or positive

Text: I think the food was okay.

Sentiment:

# Prompt (structured instructions + examples)

This **prompt** combines the task with examples, schemas, and format specs.

**Role:** Senior analyst

**Task:** Summarize the article into exactly **3 bullets**: [Cause], [Evidence], [Impact].

**Style:** Neutral, factual. Avoid speculation.

**Format:** Return only the 3 bullets, nothing else.

**Example:**

**Input:** "Storm caused power outage: s..."

**Output:**

- [Cause] Severe storm front
- [Evidence] NWS alerts
- [Impact] 20k without power

**Now analyze:**

{article\_text}

**Benefit:** lower variance, more control. **Cost:** more tokens.



# Prompt Programming (cognitive tools)

You define mini-functions (cognitive tools) with strict I/O, then “call” them in order inside your prompt.

Treat reusable reasoning patterns like **functions**

```
define_tool("understand_question", returns=[core_task, constraints, assumptions])
define_tool("plan_steps", returns=[numbered_plan])
define_tool("verify", returns=[issues, fixes])
```

Problem: {problem}

- 1) Call `understand_question` on the problem.
- 2) Call `plan_steps` using the result.
- 3) Execute the plan briefly.
- 4) Call `verify`; if issues, correct and show final answer. Output: Final answer only.

**Why it helps:** modular structure, explicit steps, easier auditing and reuse.

# Patterns You'll Reuse Often

- **Role/Persona**: “You are a tax law tutor...”
- **Explicit Format**: “Return JSON: { 'claim': string, 'citations': string[] }”
- **Constraints**: word limits, tone, style, banned content
- **Decomposition**: “List subproblems first, then solve #1...”
- **Self-check**: “Validate answer against constraints; revise if needed.”

# Prompt Pitfalls (and Fixes)

- **Ambiguity** → Add definitions, acceptance criteria, examples
- **Format drift** → Provide a strict schema and an invalid-output reminder
- **Hallucination** → Ask for citations, allow “I don’t know,” provide context
- **Over-prompting** → Keep only what changes behavior measurably
- **Hidden goals** → Make evaluation rubric explicit

# How Prompting Affects Scenarios (Examples)

## Scenario A — Classification

- **Atomic:** “Is this review positive or negative?”
- **Few-shot:** Add 3–5 labeled examples → higher consistency on edge cases
- **Schema:** “Return `{'label': 'positive' | 'negative', 'rationale': string}`”
- **Expected effect:** fewer borderline flip-flops, better rationale quality

## Scenario B — Transformation

- **Atomic:** “Rewrite in formal tone.”
- **Constraints + Style Guide:** avoids slang leakage, keeps key facts intact

# Context Engineering — Definition

**Context Engineering** designs **everything the model sees at inference time**, not just the prompt: instructions, examples, **retrieved documents**, **tools**, **memory/state**, and **control flow**.

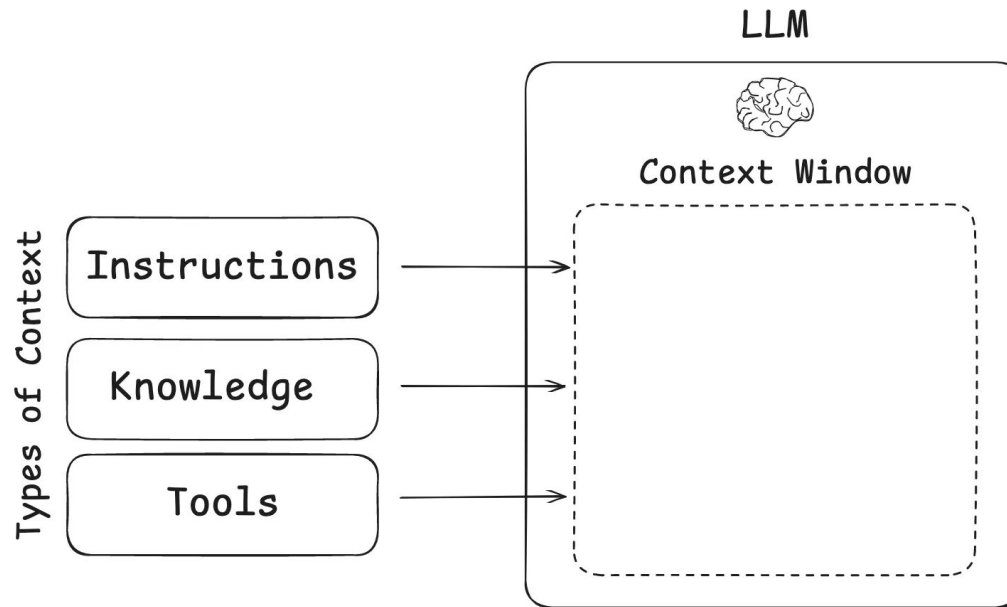
Prompt Engineering = “what you say”

Context Engineering = “everything else the model sees”

(docs, tools, memory...)

# Context Engineering

*[Context engineering is the] "...delicate art and science of filling the context window with just the right information for the next step."*





Andrej Karpathy ✓

@karpathy



+1 for "context engineering" over "prompt engineering".

People associate prompts with short task descriptions you'd give an LLM in your day-to-day use. When in every industrial-strength LLM app, context engineering is the delicate art and science of filling the context window with just the right information for the next step. Science because doing this right involves task descriptions and explanations, few shot examples, RAG, related (possibly multimodal) data, tools, state and history, compacting... Too little or of the wrong form and the LLM doesn't have the right context for optimal performance. Too much or too irrelevant and the LLM costs might go up and performance might come down. Doing this well is highly non-trivial. And art because of the guiding intuition around LLM psychology of people spirits.

On top of context engineering itself, an LLM app has to:

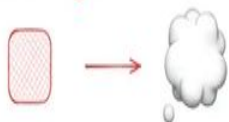
- break up problems just right into control flows
- pack the context windows just right
- dispatch calls to LLMs of the right kind and capability
- handle generation-verification UIUX flows
- a lot more - guardrails, security, evals, parallelism, prefetching, ...

So context engineering is just one small piece of an emerging thick layer of non-trivial software that coordinates individual LLM calls (and a lot more) into full LLM apps. The term "ChatGPT wrapper" is tired and really, really wrong.

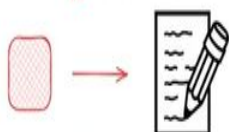


## Write Context

Long-term memories  
(across agent sessions)



Scratchpad  
(within agent session)



State  
(within agent session)



## Select Context

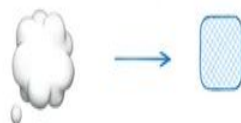
Retrieve relevant tools



Retrieve from scratchpad



Retrieve long-term memory



Retrieve relevant knowledge



## Compress Context

Summarize context  
to retain relevant tokens



Trim context to  
remove irrelevant tokens

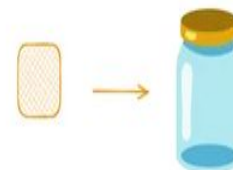


## Isolate Context

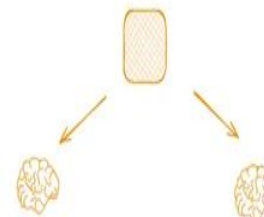
Partition context in state



Hold in environment/sandbox

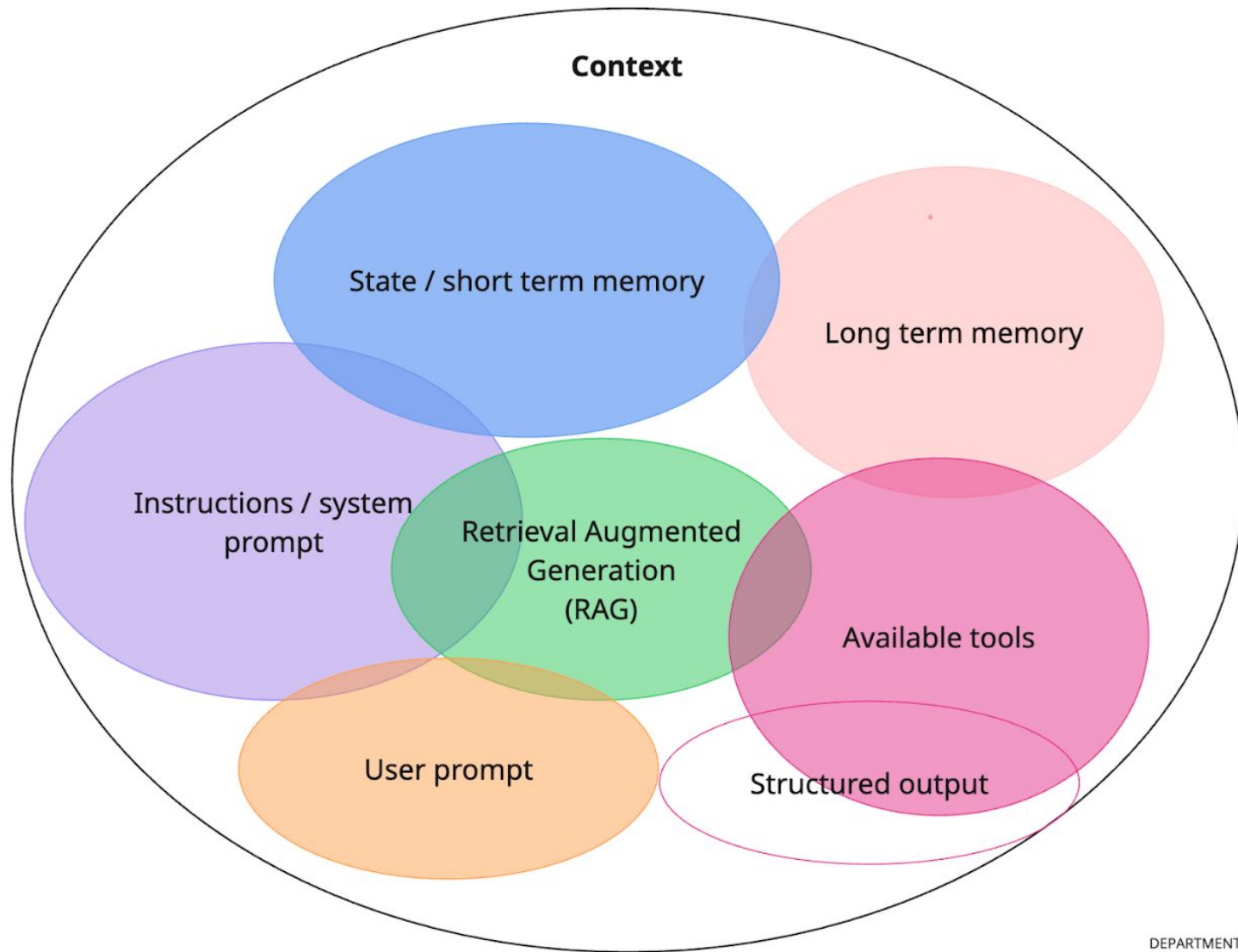


Partition across multi-agent



 = LLM Context

# Context Engineering Explained



# Why Context Engineering Matters

- **Different scope:** Prompting shapes *instructions*; context engineering shapes the *information environment* (evidence, tools, state).
- **Reliability:** Better grounding → fewer hallucinations → more reproducible outputs.
- **Scale:** With more data, tools, and steps, **planning the context** becomes the main driver of quality, latency, and cost.
- **Ownership:** Teams can **version, test, and review** context kits like code (PRs, diffs, rollbacks).

# Whole Picture: Prompt vs Context

| Aspect    | Prompt Engineering               | Context Engineering                                          |
|-----------|----------------------------------|--------------------------------------------------------------|
| Goal      | Express the task clearly         | Supply the <i>right</i> info, tools, and state               |
| Unit      | Instruction text<br>(+few demos) | Context kit: snippets, demos,<br>tools, memory, control flow |
| Risks     | Ambiguity, drift                 | Irrelevant/noisy docs, stale state,<br>tool misuse           |
| Scaling   | Add structure/examples           | Retrieval, summarization, pruning,<br>staged pipelines       |
| Artifacts | Prompt templates                 | Context packs, retrieval graphs,<br>tool policies            |

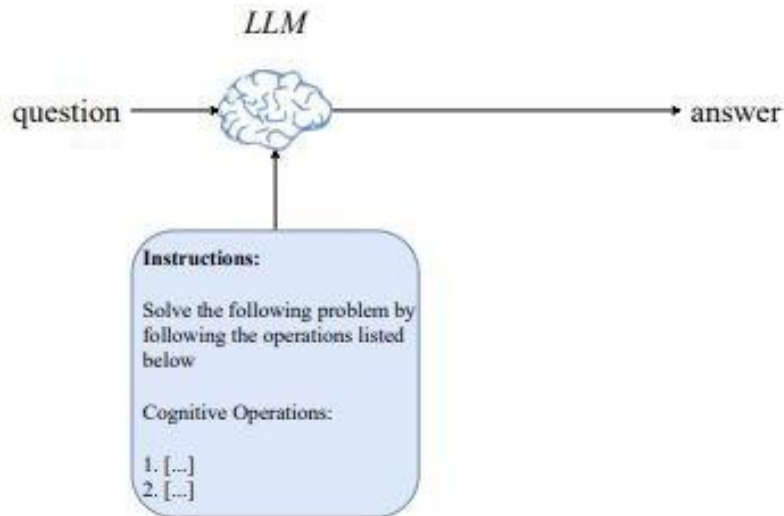
# From Simple → Large-Scale Work

**Simple task:** one page summary → 1–2 demos + 2–3 snippets is enough.

**Large system:** code agents, multi-repo RAG, long-running workflows → needs:

- **Top-k retrieval + reranking** (semantic + keyword hybrids)
- **Step-wise contexts** (Research → Plan → Implement → Verify)
- **Interface contracts** (schemas for inputs/outputs per step)
- **Memory strategy** (session summaries, decision logs)
- **Tooling policy** (which tools, when, with what arguments)

## A) Cognitive Prompting



## B) Cognitive Tools

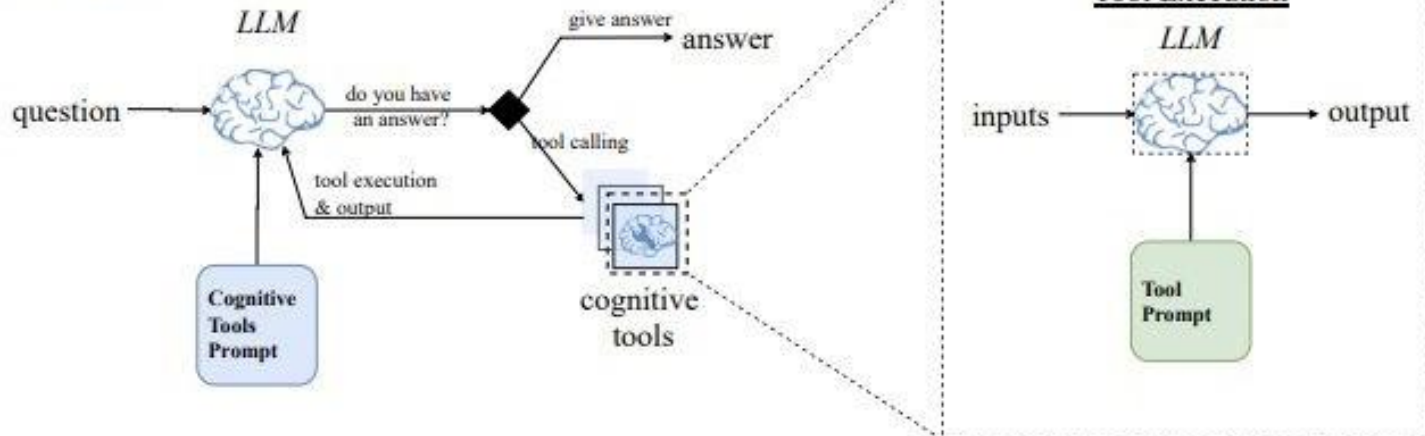


Figure 1: Overview of our Cognitive Tools pipeline vs Cognitive Prompting

# Design Principles (Context Planning)

- **Define the decision**: what must the model decide/produce at this step?
- **Identify evidence**: which sources justify that decision?
- **Choose demo types**: positive, counter-example, edge case
- **Set format contract**: JSON schema or structured template
- **Limit the tool belt**: only the tools needed for the decision
- **Prune aggressively**: remove anything not used by this step
- Treat each step like a function with a minimal, sufficient signature.

# More Impactful

1 Bad Line of Code == 1 Bad Line of Code

1 Bad Line of Plan == 10-100 Bad Lines of Code

Wrong Solution

1 Bad Line of Research == 1000+ Bad Lines of Code

Misunderstanding the System

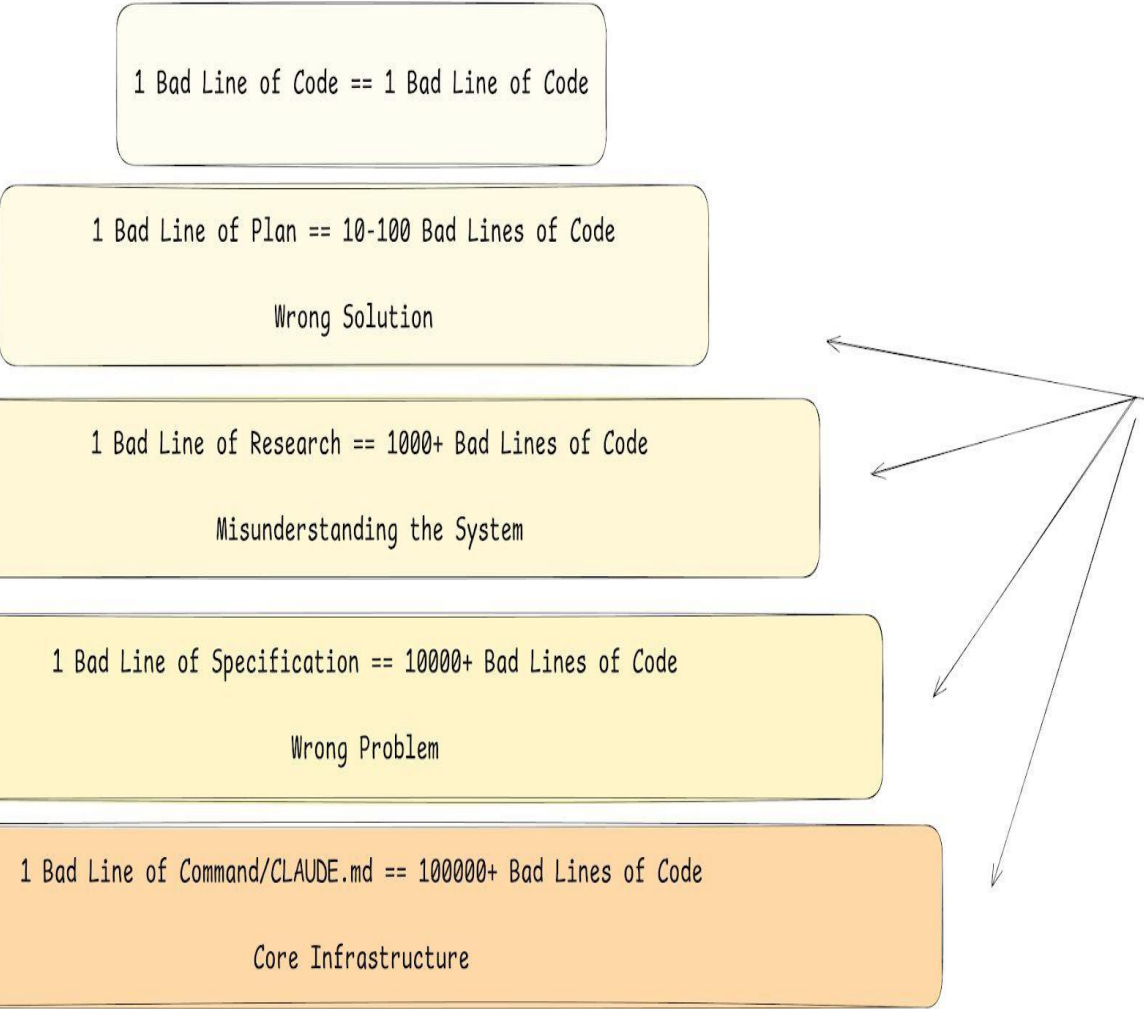
1 Bad Line of Specification == 10000+ Bad Lines of Code

Wrong Problem

1 Bad Line of Command/CLAUDE.md == 100000+ Bad Lines of Code

Core Infrastructure

Human Effort and Focus  
on the HIGHEST LEVERAGE  
parts of the pipeline



```
graph TD; A[1 Bad Line of Code == 1 Bad Line of Code] --> B[1 Bad Line of Plan == 10-100 Bad Lines of Code  
Wrong Solution]; B --> C[1 Bad Line of Research == 1000+ Bad Lines of Code  
Misunderstanding the System]; C --> D[1 Bad Line of Specification == 10000+ Bad Lines of Code  
Wrong Problem]; D --> E[1 Bad Line of Command/CLAUDE.md == 100000+ Bad Lines of Code  
Core Infrastructure];
```



# Failure Modes → Bad Code

- **Missing constraints** → model “fills gaps” with guesses (wrong APIs, types)
- **Stale docs** → uses deprecated functions → compile/runtime errors
- **No dependency map** → incorrect import order, version clashes
- **Unscoped retrieval** → pulls unrelated repos → off-target code patterns
- **Tool misuse** → lack of preconditions → shell/code runners execute wrong cmds
- **No verification step** → syntax/format errors slip through

**Solution:** enforce **schema-first outputs**, **repo-scoped retrieval**, **compile/test hooks**, and **self-check prompts**.

# RAG (Retrieval-Augmented Generation) in 3 Steps

1. **Retrieve** top-k passages for the query
2. **Construct** a compact context pack (citations, snippets)
3. **Generate** with grounding, refusing when insufficient evidence

**Effect:** fewer hallucinations, better factuality; tokens spent on **relevant** info.

# Worked Example — Zero vs Few vs Multi

- **Task:** Convert unstructured incident notes into a JSON schema.
- **Zero-shot prompt:** “Parse the note into JSON:{{incident\_id,  
date, severity, systems, summary}}”
- **Expected output:** Sometimes missing fields or inconsistent casing.
- **Few-shot:** add 3 labeled notes → **consistent keys & severity scale.**
- **Multi-shot staged:** Step 1 extract spans → Step 2 normalize enums → Step 3 emit JSON
- **Effect:** best compliance to schema + fewer nulls for required fields

# Code Template — JSON-First Prompt (Python)

tells it to return **only** a single, compact (minified) JSON object that conforms to it. The “Steps” lines are *internal instructions*—the model should do them silently and emit just the final JSON.

```
SCHEMA = {{
    "type":"object",
    "properties":{{ "incident_id":{{"type":"string"}}},
        "date":{{"type":"string","format":"date"}}},
        "severity":{{"type":"string","enum":["low","medium","high","critical"]}},
        "systems":{{"type":"array","items":{{"type":"string"}}}},
        "summary":{{"type":"string"}}
    }},
    "required":["incident_id","date","severity","summary"]
}}
```

```
PROMPT = f'''
Role: Data wrangler
Task: Extract fields and output ONLY compact JSON valid to this schema. Schema
(JSON Schema): {{SCHEMA}}

Steps:
1) Extract candidate values (do not output).
2) Normalize values to schema (do not output).
3) Output final JSON ONLY.

Input:
{{{note_text}}}}
'''
```

# Prompt QA & Self-Check

Add a final guard step:

Validate:

- Output format exactly matches schema
- No unknown keys
- Required fields present & non-empty If validation fails, fix and re-emit.

**Result:** higher reliability without extra tools.

# Context QA — Evidence Blocks

Require citing passages used for an answer:

Use ONLY these  
snippets:

[1] "... neural networks can ... "

[2] "... support vector machines ..."

For each claim, add (source: [n]). If info is missing, say  
"insufficient evidence".

**Result:** better faithfulness; easier manual review.

# Measuring Impact (Rubric)

- **Accuracy** (task-specific)
- **Faithfulness** (to given evidence)
- **Format compliance** (schema, JSON)
- **Efficiency** (tokens, latency)
- **Robustness** (edge cases)
- **Repeatability** (variance across runs)

Track before/after when changing prompts or context kits.

# LlamaIndex

**What it is:** A framework for building Retrieval-Augmented apps: ingest data → create **Nodes** (chunks) → embed → store → **retrieve** (optionally generate later).

**Why Nodes matter:** first-class objects carrying text and metadata; produced by **node parsers** (splitters) and consumed by retrievers & indexes.

**Swappable parts:** embedding models, vector stores (in-memory or FAISS/Qdrant/etc.), chunkers, retrievers.



# RAG: Overview & Architecture

**Definition:** combine a parametric model with a **non-parametric memory** (retriever + index) so outputs can be grounded in external, up-to-date knowledge.

**Two stages:** (1) **Indexing** (ingest → chunk → embed → store) and (2) **Retrieval + generation** (fetch top-k, then prompt an LLM).

**Why it helps:** improves specificity and factuality vs. parametric-only models.

# Evaluating RAG (what to measure)

**Retriever:** *context recall* (did we fetch the best facts?)  
and *context precision* (is fetched context relevant?).

**Generator:** *faithfulness* (claims supported by context)  
and *answer relevancy*. **Workflow:** small QA set → log  
top-k & scores → compute metrics → iterate on  
embeddings/chunking/k/reranking.

# Chunking Techniques for RAG

- **LangChain** and **LlamaIndex**. Both frameworks are designed to handle document ingestion, splitting, indexing, and chaining together steps for seamless RAG workflows
- LlamaIndex → handle ingestion, indexing, & retrieval (its sweet spot for RAG).
- LangChain → handle orchestration/agents, prompts, and tool use around that retriever.
- **LlamaIndex (formerly GPT Index)**
  - Specializes in **data ingestion, indexing, and retrieval**.
  - Makes it easier to connect LLMs to your **structured and unstructured data sources** (databases, PDFs, APIs, etc.).
  - Provides retrieval-augmented generation (RAG) pipelines: you load data → index it → query it with an LLM.
- **LangChain**
  - Specializes in **orchestration and chaining** of LLM calls.
  - Provides abstractions for **prompts, tools, agents, memory, and workflows**.
  - Lets you combine LLMs with other software systems (search engines, APIs, calculators, etc.) into multi-step reasoning chains.

# The LlamaIndex pipeline

- **Indexing** is offline: chunk → embed → store.
- **Retrieval** is online: embed query → similarity search → return **top-k** with scores.

# SentenceSplitter

**Goal:** keep sentences/paragraphs intact to preserve coherence; reduces “hanging sentence fragments.”

**You control:** target chunk size and overlap; allows natural boundaries while staying near the size you set.

**When to use:** narrative or expository text where meaning spans full sentences.

# TokenTextSplitter

**Goal:** split by **tokens** (not characters) for precise control of model context limits. **Trade-off:** perfect budget control, but may split mid-sentence if needed to respect token count.

**When to use:** strict LLM window budgets; mixed-length inputs.



# SemanticSplitter

**Goal:** choose breakpoints using **embedding similarity** between adjacent sentences → chunks are **semantically coherent**.

**Controls:** look-ahead/buffer and threshold/percentile for when to split.

**When to use:** long, topic-shifting paragraphs; noisy boundaries.



Berlin is the capital and largest city of Germany, both by area and by population. Its more than 3.85 million inhabitants make it the European Union's most populous city, as measured by population within city limits. The city is also one of the states of Germany, and is the third smallest state in the country in terms of area. Berlin is a state of Brandenburg, and Brandenburg's capital. The urban area of Berlin has a population of over 4.5 million and is therefore the most populous urban area

**Embed sentence,  
but return context**

# SentenceWindow

**Goal:** index **single sentences** but attach a **window** of neighbor sentences as metadata, so retrieval stays fine-grained yet carries context.

**Use it when:** you want pinpoint matches and still provide surrounding context at answer time.

# RAG Quality Levers (retrieval side)

**Embeddings model:** pick for domain/language; better sentence encoders improve top-k recall and margins (dimensionality  $\neq$  quality).

**Chunking strategy:** sentence/recursive for coherence; token for strict budget;

**semantic** for content-aware splits; hierarchies for multi-scale retrieval.

**Retriever settings:** k, metric (cosine/dot/L2), **MMR/diversity**, and optional **rerankers** (cross-encoders) for higher precision.

# Questions?