

[Open in app](#) ↗

All your favorite parts of Medium are now in one sidebar for easy access.

Okay, got it

# AI Agents: Build an Agent from Scratch (Part-2)

10 min read · Feb 18, 2025



Vipra Singh

Follow



Listen



Share



More

*Discover AI agents, their design, and real-world applications.*

## Posts in this Series

1. [Introduction](#)
2. [Build an Agent from Scratch \(This Post\)](#)
3. [AI Agent Frameworks](#)
4. [Types of AI Agents](#)
5. [Workflow vs Agent](#)
6. [Agent Architectures](#)
7. [Multi-Agent Architectures](#)
8. [Building Multi-Agent System](#)
9. [Agentic Memory](#)
10. [Agentic RAG](#)

## 11. *Model Context Protocol (MCP)*

### 12. *Evaluation*

All your favorite parts of Medium are now in one sidebar for easy access.

## Table of Contents

- 1. What is an Agent?
- 2. Implementation
  - 2.1 Pre-requisites
  - 2.2 Implementation Steps
- 3. Conclusion

In our previous blog post, we provided a comprehensive overview of AI agents, discussing their characteristics, components, evolution, challenges, and future possibilities.

In this blog, we'll explore how to build an agent from scratch using Python. This agent will be capable of making decisions based on user input, selecting appropriate tools, and executing tasks accordingly. Let's get started!

## 1. What is an Agent?

An agent is an autonomous entity capable of perceiving its environment, making decisions, and taking actions to achieve specific goals. Agents can vary in complexity from simple reactive agents that respond to stimuli to more advanced intelligent agents that learn and adapt over time. Common types of agents include:

1. **Reactive Agents:** Respond directly to environmental changes without internal memory.
2. **Model-Based Agents:** Use internal models of the world to make decisions.
3. **Goal-Based Agents:** Plan actions based on achieving specific goals.
4. **Utility-Based Agents:** Evaluate potential actions based on a utility function to maximize outcomes.

Examples include chatbots, recommendation systems, and autonomous vehicles, each utilizing different types of agents to perform tasks efficiently and intelligently.

All your favorite parts of Medium are now in one sidebar for easy access.

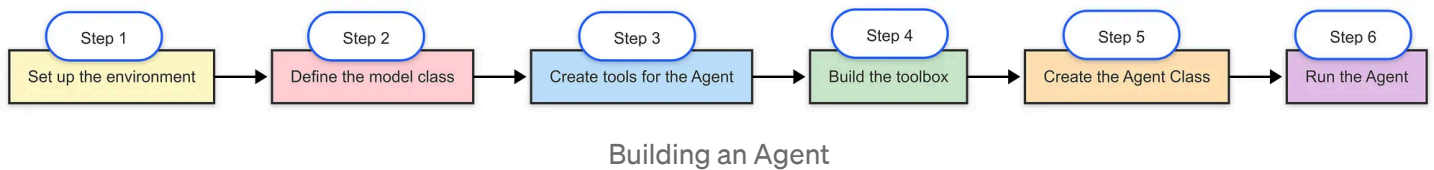
of our agent are:

of the agent, responsible for processing the input and  
ies.

- **Tools:** Pre-defined functions that the agent can execute based on the user's request.
- **Toolbox:** A collection of tools the agent has at its disposal.
- **System Prompt:** The instruction set that guides the agent on how to handle user input and choose the right tools.

## 2. Implementation

Now, let's roll up our sleeves and start building!



### 2.1 Pre-requisites

*The complete code for this tutorial is available in the AI Agents GitHub repository.*

*You can find the implementation here: [Build an Agent from Scratch](#).*

Before running the code, ensure that your system meets the following prerequisites:

#### 1. Python Environment Setup

You'll need Python installed to run the AI agent. Follow these steps to set up your environment:

##### Install Python (if not already installed)

- Download and install Python (3.8+ recommended) from [python.org](https://python.org).
- Verify installation:

```
python --version
```

All your favorite parts of Medium are now in one sidebar for easy access.

### Environment (Recommended)

environment to manage dependencies:

```
python -m venv ai_agents_env  
source ai_agents_env/bin/activate # On Windows: ai_agents_env\Scripts\activate
```

## Install Required Dependencies

Navigate to the repository directory and install dependencies:

```
pip install -r requirements.txt
```

## 2. Setup Ollama Locally

Ollama is used to run and manage local language models efficiently. Follow these steps to install and configure it:

### Download and Install Ollama

- Visit [Ollama's official site](#) and download the installer for your OS.
- Install it following the instructions for your platform.

### Verify Ollama Installation

Run the following command to check if Ollama is installed correctly:

```
ollama --version
```

### Pull a Model (if required)

Some agent implementations may require a specific model. You can pull a model

using:

All your favorite parts of Medium are now in one sidebar for easy access.

```
l # Replace 'mistral' with the model needed
```

S

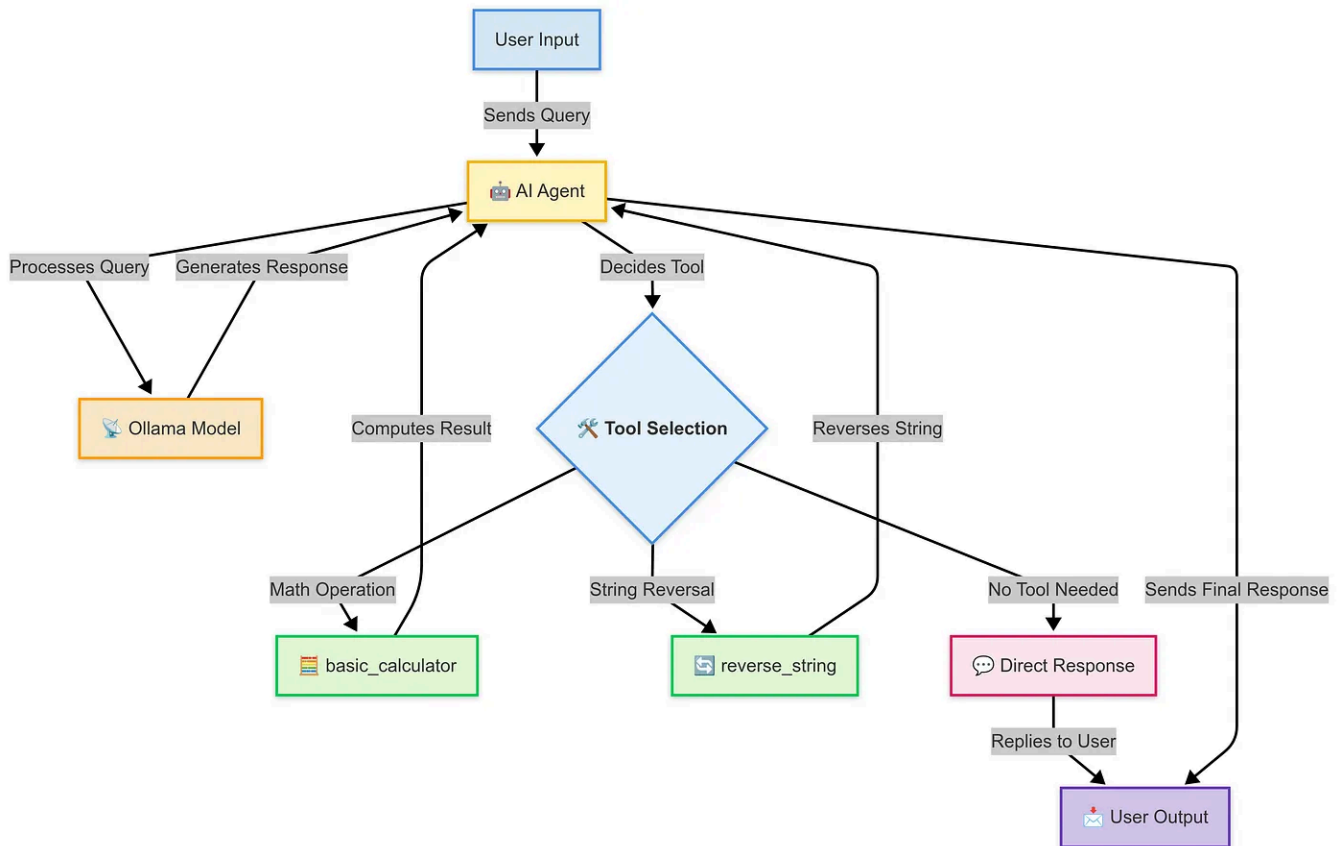


Image by Author

## Step 1: Setting Up the Environment

Along with Python, we'll also need to install some essential libraries. For this tutorial, we'll be using `requests`, `json`, and `termcolor`. Additionally, we will use `dotenv` to manage environment variables.

```
pip install requests termcolor python-dotenv
```

## Step 2: Defining the Model Class

The first thing we need is a model that will process user input. We'll create an `OllamaModel` class, which interacts with a local API to generate responses.

All your favorite parts of Medium are now in one sidebar for easy access.

entation:

```
from dotenv import load_dotenv
load_dotenv()
### Models
import requests
import json
import operator
```

```
class OllamaModel:
    def __init__(self, model, system_prompt, temperature=0, stop=None):
        """
        Initializes the OllamaModel with the given parameters.

        Parameters:
        model (str): The name of the model to use.
        system_prompt (str): The system prompt to use.
        temperature (float): The temperature setting for the model.
        stop (str): The stop token for the model.
        """
        self.model_endpoint = "http://localhost:11434/api/generate"
        self.temperature = temperature
        self.model = model
        self.system_prompt = system_prompt
        self.headers = {"Content-Type": "application/json"}
        self.stop = stop

    def generate_text(self, prompt):
        """
        Generates a response from the Ollama model based on the provided prompt.

        Parameters:
        prompt (str): The user query to generate a response for.

        Returns:
        dict: The response from the model as a dictionary.
        """
        payload = {
            "model": self.model,
            "format": "json",
            "prompt": prompt,
            "system": self.system_prompt,
```

```
"stream": False,
"temperature": self.temperature,
"stop": self.stop
```

All your favorite parts of Medium are now in one sidebar for easy access.

```
t_response = requests.post(
    lf.model_endpoint,
    headers=self.headers,
    data=json.dumps(payload)
)

print("REQUEST RESPONSE", request_response)
request_response_json = request_response.json()
response = request_response_json['response']
response_dict = json.loads(response)

print(f"\n\nResponse from Ollama model: {response_dict}")

return response_dict
except requests.RequestException as e:
    response = {"error": f"Error in invoking model! {str(e)}"}
    return response
```

This class initializes with the model's name, system prompt, temperature, and stop token. The `generate_text` method sends a request to the model API and returns the response.

### Step 3: Creating Tools for the Agent

The next step is to create tools that our agents can use. These tools are simple Python functions that perform specific tasks. Here's an example of a basic calculator and a string reverser:

```
def basic_calculator(input_str):
    """
    Perform a numeric operation on two numbers based on the input string or dict.

    Parameters:
    input_str (str or dict): Either a JSON string representing a dictionary with
        'num1', 'num2', and 'operation' keys, or a dictionary directly. Example: '{"num1": 5, "num2": 10, "operation": "add"}'
        or {"num1": 67869, "num2": 9030393, "operation": "c

    Returns:
    str: The formatted result of the operation.
```

Raises:

Exception: If an error occurs during the operation (e.g., division by zero)

ValueError: If an unsupported operation is requested or input is invalid.

All your favorite parts of

Medium are now in one

sidebar for easy access.

```

both dictionary and string inputs
nce(input_str, dict):
dict = input_str

# Clean and parse the input string
input_str_clean = input_str.replace("'", "\\'")
input_str_clean = input_str_clean.strip().strip("\\'")
input_dict = json.loads(input_str_clean)

# Validate required fields
if not all(key in input_dict for key in ['num1', 'num2', 'operation']):
    return "Error: Input must contain 'num1', 'num2', and 'operation'"

num1 = float(input_dict['num1']) # Convert to float to handle decimal
num2 = float(input_dict['num2'])
operation = input_dict['operation'].lower() # Make case-insensitive
except (json.JSONDecodeError, KeyError) as e:
    return "Invalid input format. Please provide valid numbers and operation"
except ValueError as e:
    return "Error: Please provide valid numerical values."

# Define the supported operations with error handling
operations = {
    'add': operator.add,
    'plus': operator.add, # Alternative word for add
    'subtract': operator.sub,
    'minus': operator.sub, # Alternative word for subtract
    'multiply': operator.mul,
    'times': operator.mul, # Alternative word for multiply
    'divide': operator.truediv,
    'floor_divide': operator.floordiv,
    'modulus': operator.mod,
    'power': operator.pow,
    'lt': operator.lt,
    'le': operator.le,
    'eq': operator.eq,
    'ne': operator.ne,
    'ge': operator.ge,
    'gt': operator.gt
}

# Check if the operation is supported
if operation not in operations:
    return f"Unsupported operation: '{operation}'. Supported operations are

try:
    # Special handling for division by zero
    if (operation in ['divide', 'floor_divide', 'modulus']) and num2 == 0:

```

```

        return "Error: Division by zero is not allowed"

    # Perform the operation
    operations[operation](num1, num2)

    # Result based on type
    if isinstance(result, bool):
        result_str = "True" if result else "False"
    elif isinstance(result, float):
        # Handle floating point precision
        result_str = f"{result:.6f}".rstrip('0').rstrip('.')
    else:
        result_str = str(result)

    return f"The answer is: {result_str}"
except Exception as e:
    return f"Error during calculation: {str(e)}"

def reverse_string(input_string):
    """
    Reverse the given string.

    Parameters:
    input_string (str): The string to be reversed.

    Returns:
    str: The reversed string.
    """
    # Check if input is a string
    if not isinstance(input_string, str):
        return "Error: Input must be a string"

    # Reverse the string using slicing
    reversed_string = input_string[::-1]

    # Format the output
    result = f"The reversed string is: {reversed_string}"

    return result

```

These functions are designed to perform specific tasks based on the input provided. The `basic_calculator` handles arithmetic operations, while `reverse_string` reverses a given string.

## Step 4: Building the Toolbox

The `ToolBox` class stores all the tools the agent can use and provides descriptions for each:

```

class ToolBox:
    def __init__(self):
        self._dict = {}

    def add(self, functions_list):
        """
        literal name and docstring of each function in the list.

        Parameters:
        functions_list (list): List of function objects to store.

        Returns:
        dict: Dictionary with function names as keys and their docstrings as values.
        """
        for func in functions_list:
            self.tools_dict[func.__name__] = func.__doc__
        return self.tools_dict

    def tools(self):
        """
        Returns the dictionary created in store as a text string.

        Returns:
        str: Dictionary of stored functions and their docstrings as a text string.
        """
        tools_str = ""
        for name, doc in self.tools_dict.items():
            tools_str += f"{name}: \"{doc}\"\\n"
        return tools_str.strip()

```

All your favorite parts of Medium are now in one sidebar for easy access.

This class will help the agent understand which tools are available and what each one does.

## Step 5: Creating the Agent Class

The agent needs to think, decide which tool to use, and execute it. Here's the Agent class:

```

agent_system_prompt_template = """
You are an intelligent AI assistant with access to specific tools. Your responses should be in the following JSON format:
{{
    "tool_choice": "name_of_the_tool",
    "tool_input": "inputs_to_the_tool"
}}

```

**TOOLS AND WHEN TO USE THEM:**

All your favorite parts of Medium are now in one sidebar for easy access.

**1. basic\_calculator:** Use for ANY mathematical calculations

```

    {"num1": number, "num2": number, "operation": "add/subtract
    rations: add/plus, subtract/minus, multiply/times, divide
    s and outputs:
    late 15 plus 7"
    ol_choice": "basic_calculator", "tool_input": {"num1": 15, "
  
```

Input: "What is 100 divided by 5?"

Output: {"tool\_choice": "basic\_calculator", "tool\_input": {"num1": 100,

**2. reverse\_string:** Use for ANY request involving reversing text

- Input format: Just the text to be reversed as a string
- ALWAYS use this tool when user mentions "reverse", "backwards", or asks to
- Example inputs and outputs:

Input: "Reverse of 'Howwww'?"

Output: {"tool\_choice": "reverse\_string", "tool\_input": "Howwww"}}

Input: "What is the reverse of Python?"

Output: {"tool\_choice": "reverse\_string", "tool\_input": "Python"}}

**3. no tool:** Use for general conversation and questions

- Example inputs and outputs:

Input: "Who are you?"

Output: {"tool\_choice": "no tool", "tool\_input": "I am an AI assistant th

Input: "How are you?"

Output: {"tool\_choice": "no tool", "tool\_input": "I'm functioning well, t

**STRICT RULES:****1. For questions about identity, capabilities, or feelings:**

- ALWAYS use "no tool"
- Provide a complete, friendly response
- Mention your capabilities

**2. For ANY text reversal request:**

- ALWAYS use "reverse\_string"
- Extract ONLY the text to be reversed
- Remove quotes, "reverse of", and other extra text

**3. For ANY math operations:**

- ALWAYS use "basic\_calculator"
- Extract the numbers and operation
- Convert text numbers to digits

Here is a list of your tools along with their descriptions:

```
{tool_descriptions}
```

Remember: Your response must ALWAYS be valid JSON with "tool\_choice" and "tool\_""

```
class Agent:
    def __init__(self, tools, model_service, model_name, stop=None):
```

All your favorite parts of  
Medium are now in one  
sidebar for easy access.

```
        # Initialize the agent with a list of tools and a model.
        :
        (t): List of tool functions.
        _ (class): The model service class with a generate_text method
        model_name (str): The name of the model to use.
        """
        self.tools = tools
        self.model_service = model_service
        self.model_name = model_name
        self.stop = stop

    def prepare_tools(self):
        """
        Stores the tools in the toolbox and returns their descriptions.

        Returns:
        str: Descriptions of the tools stored in the toolbox.
        """
        toolbox = ToolBox()
        toolbox.store(self.tools)
        tool_descriptions = toolbox.tools()
        return tool_descriptions

    def think(self, prompt):
        """
        Runs the generate_text method on the model using the system prompt template.

        Parameters:
        prompt (str): The user query to generate a response for.

        Returns:
        dict: The response from the model as a dictionary.
        """
        tool_descriptions = self.prepare_tools()
        agent_system_prompt = agent_system_prompt_template.format(tool_descriptions)

        # Create an instance of the model service with the system prompt

        if self.model_service == OllamaModel:
            model_instance = self.model_service(
                model=self.model_name,
                system_prompt=agent_system_prompt,
                temperature=0,
                stop=self.stop
            )
        else:
            model_instance = self.model_service(
```

```

model=self.model_name,
system_prompt=agent_system_prompt,
temperature=0

```

All your favorite parts of Medium are now in one sidebar for easy access.

```

and return the response dictionary
onse_dict = model_instance.generate_text(prompt)
nt_response_dict

```

```

def work(self, prompt):
    """
    Parses the dictionary returned from think and executes the appropriate

    Parameters:
    prompt (str): The user query to generate a response for.

    Returns:
    The response from executing the appropriate tool or the tool_input if r
    """
    agent_response_dict = self.think(prompt)
    tool_choice = agent_response_dict.get("tool_choice")
    tool_input = agent_response_dict.get("tool_input")

    for tool in self.tools:
        if tool.__name__ == tool_choice:
            response = tool(tool_input)
            print(colored(response, 'cyan'))
            return

    print(colored(tool_input, 'cyan'))
    return

```

This class has three main methods:

- **prepare\_tools:** Stores and returns the descriptions of the tools.
- **think:** Decide which tool to use based on the user prompt.
- **work:** Executes the chosen tool and returns the result.

## Step 6: Running the Agent

Finally, let's put everything together and run our agent. In your script's `main` section, initialize the agent and start accepting user input:

## # Example usage

```
if __name__ == "__main__":
```

All your favorite parts of Medium are now in one sidebar for easy access.

or using this agent:

as you can try:

operations:

e.g. "15 plus 7"

- "What is 100 divided by 5?"
- "Multiply 23 and 4"

### 2. String reversal:

- "Reverse the word 'hello world'"
- "Can you reverse 'Python Programming'?"

### 3. General questions (will get direct responses):

- "Who are you?"
- "What can you help me with?"

Ollama Commands (run these in terminal):

- Check available models: `'ollama list'`
- Check running models: `'ps aux | grep ollama'`
- List model tags: `'curl http://localhost:11434/api/tags'`
- Pull a new model: `'ollama pull mistral'`
- Run model server: `'ollama serve'`

```
tools = [basic_calculator, reverse_string]
```

```
# Uncomment below to run with OpenAI
```

```
# model_service = OpenAIModel
```

```
# model_name = 'gpt-3.5-turbo'
```

```
# stop = None
```

```
# Using Ollama with llama2 model
```

```
model_service = OllamaModel
```

```
model_name = "llama2" # Can be changed to other models like 'mistral', 'co
```

```
stop = "<|eot_id|>"
```

```
agent = Agent(tools=tools, model_service=model_service, model_name=model_na
```

```
print("\nWelcome to the AI Agent! Type 'exit' to quit.")
```

```
print("You can ask me to:")
```

```
print("1. Perform calculations (e.g., 'Calculate 15 plus 7')")
```

```
print("2. Reverse strings (e.g., 'Reverse hello world')")
```

```
print("3. Answer general questions\n")
```

```
while True:
```

```
    prompt = input("Ask me anything: ")
```

```
    if prompt.lower() == "exit":
```

```
        break
```

```
agent.work(prompt)
```

All your favorite parts of Medium are now in one sidebar for easy access.

explored the understanding of what an agent is to implement it up the environment, defined the model, created essential tools, and built a structured toolbox to support our agent's functionality. Finally, we brought everything together by running the agent in action.

This structured approach provides a solid foundation for building intelligent, interactive agents capable of automating tasks and making informed decisions. As AI agents continue to evolve, their applications will expand across industries, driving efficiency and innovation. Stay tuned for more insights and enhancements to take your AI agents to the next level!

## Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentation, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

## Thank you for reading!

If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 🖐️ or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.
- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides our future posts.

## Connect with me!

Vipra

Ai Agent

Llm

Llm Applications

Tools

Prompt



All your favorite parts of Medium are now in one sidebar for easy access.

ingh

}

Follow

## Responses (23)



Van Thinh Nguyen

What are your thoughts?



Rafsan Bhuiyan

Feb 26



Thank you! Please continue. When I see articles about Ai agents. I bookmark immediately



19

[Reply](#)



Lovelyn David

Mar 5



Excellent read



4

[Reply](#)



Arvind Shastry

Mar 24 (edited)



Thanks for sharing, Vipra. Built my first AI agent following the above steps. Have a couple of queries (1) Can Agents build their own tools on-demand if one is not found on the tool library to perform certain operations (2) How do we know if the... [more](#)



3



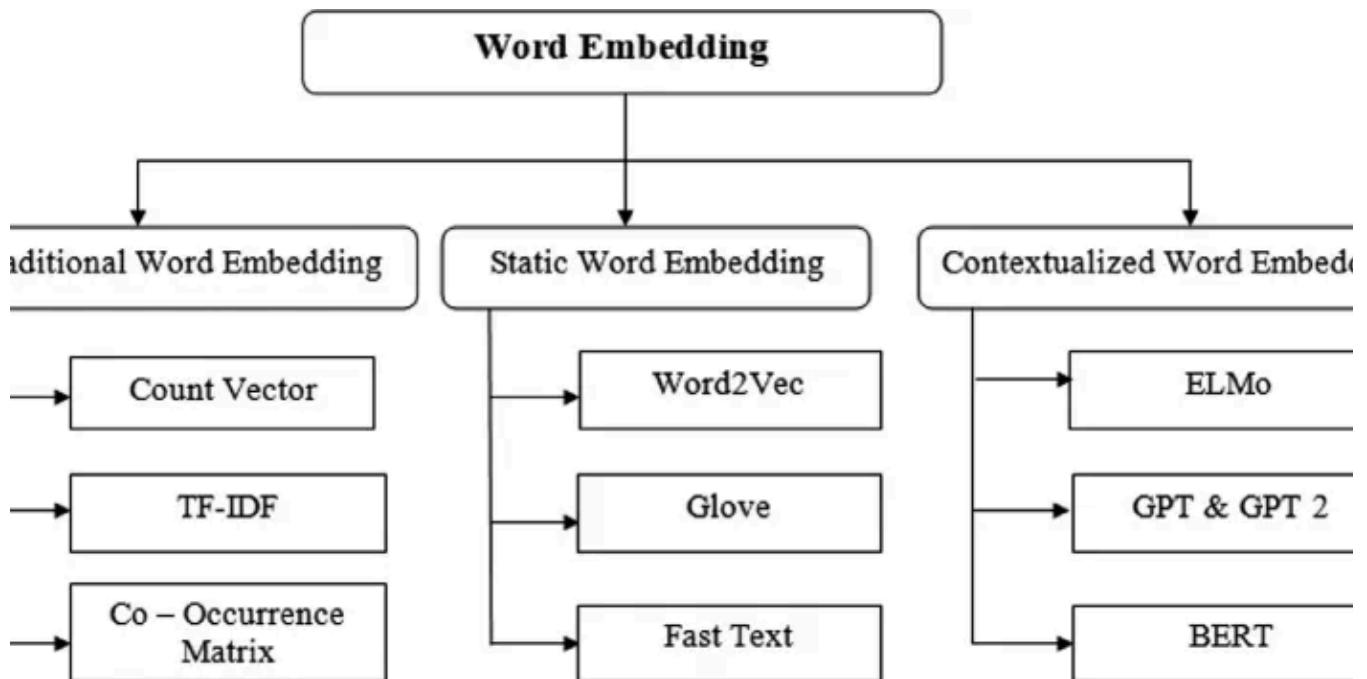
1 reply

[Reply](#)

[See all responses](#)

All your favorite parts of Medium are now in one sidebar for easy access.

more from Vipra Singh



 Vipra Singh

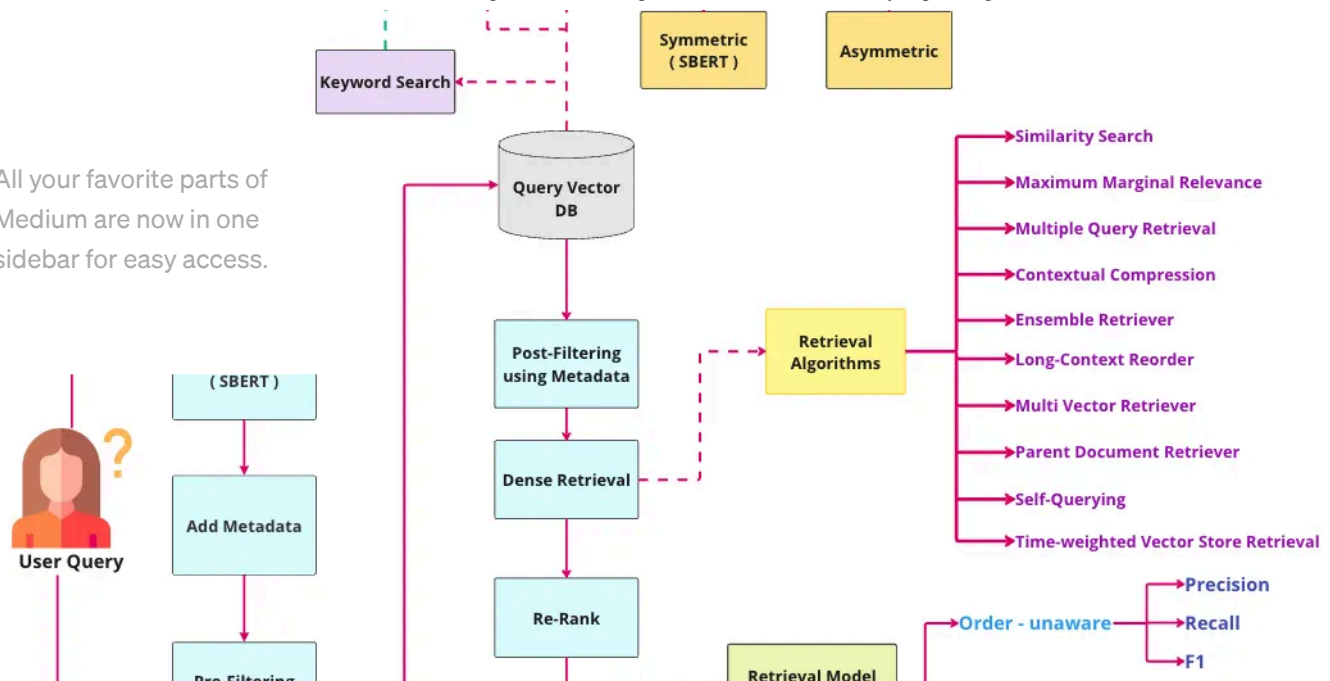
## LLM Architectures Explained: Word Embeddings (Part 2)


Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

★ Aug 19, 2024 🖱 785 💬 11

🔖 ⋮

All your favorite parts of Medium are now in one sidebar for easy access.

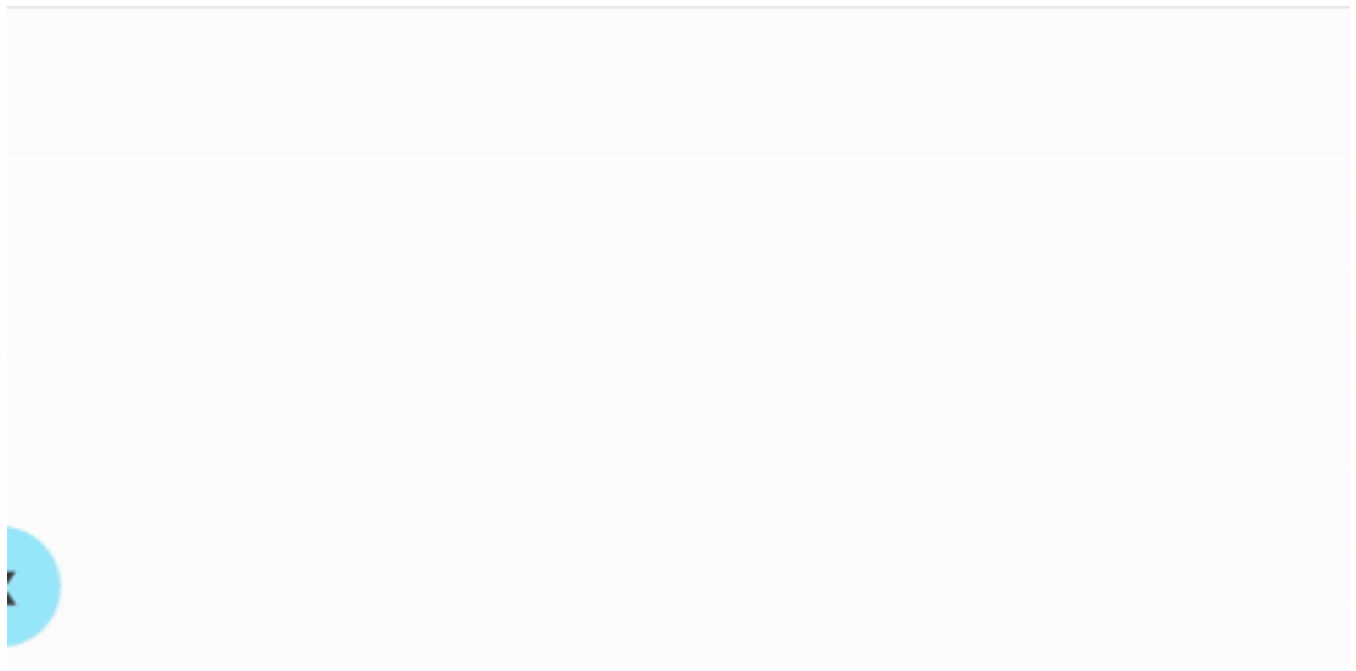


 Vipra Singh

## Building LLM Applications: Search & Retrieval (Part 5)

Learn Large Language Models ( LLM ) through the lens of a Retrieval Augmented Generation ( RAG ) Application.

🌟 Jan 28, 2024 🖱️ 771 💬 1



 Vipra Singh

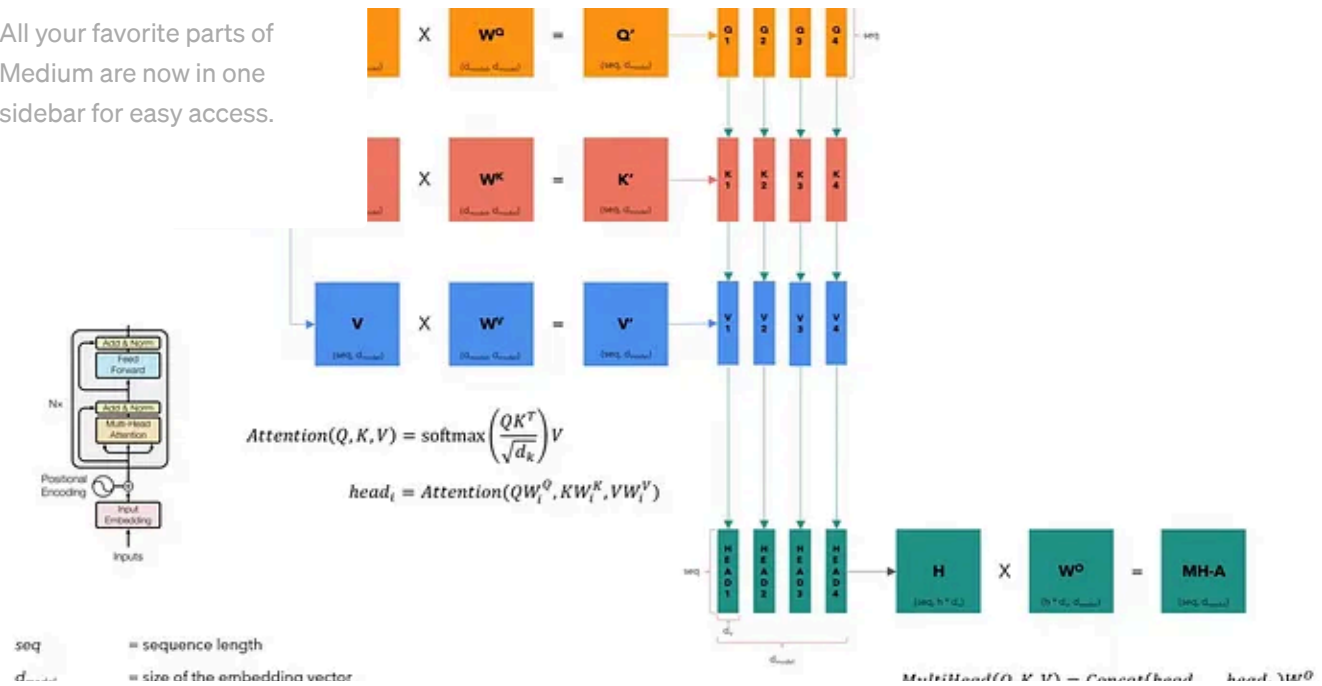
## LLM Architectures Explained: RNNs, LSTMs & GRUs (Part 3)

Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

★ Sep 9, 2024 🖱 665 💬 1



All your favorite parts of Medium are now in one sidebar for easy access.



Vipra Singh

## LLM Architectures Explained: Coding a Transformer (Part 7)

Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

★ Nov 10, 2024 🖱 456 💬 3



See all from Vipra Singh

## Recommended from Medium



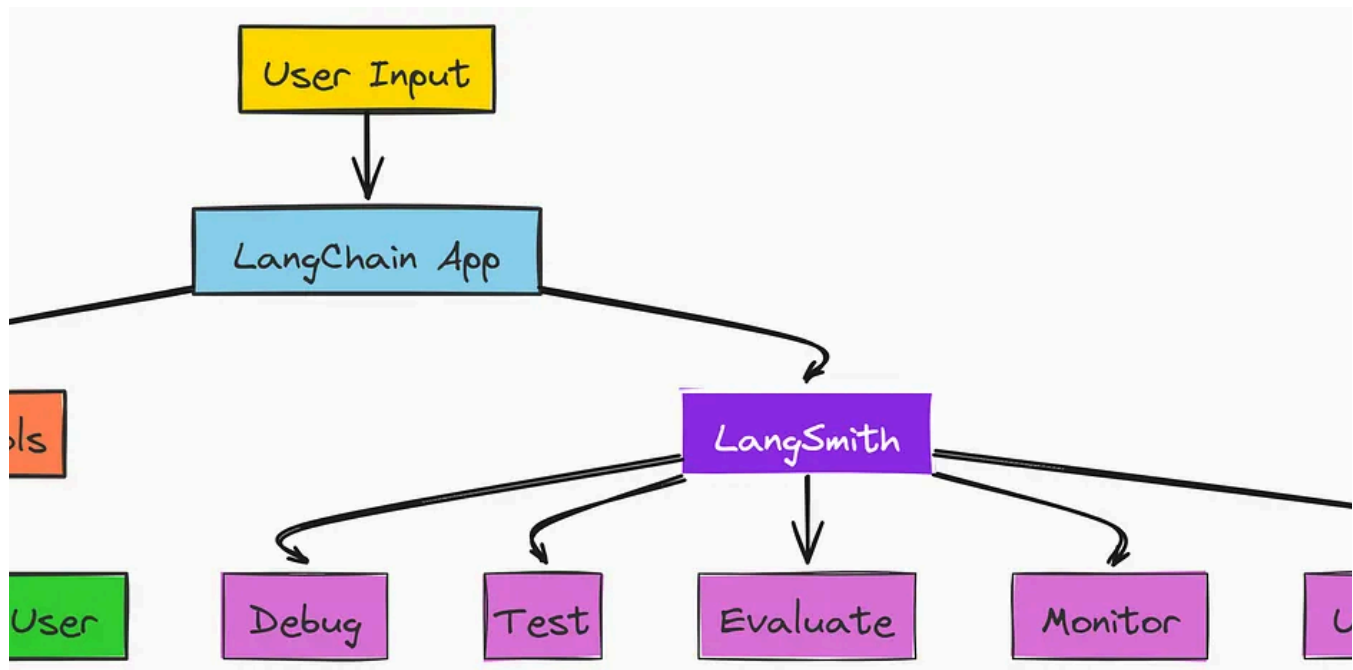
Saravana Thiagarajan K



## How to Keep Your AI Agents Accountable: A Practical Guide to Evaluation & Monitoring

It's easy to celebrate an AI demo that dazzles on launch day—the perfect answers, the smooth conversations, the promise of automation.

Oct 10 3



In Level Up Coding by Fareed Khan

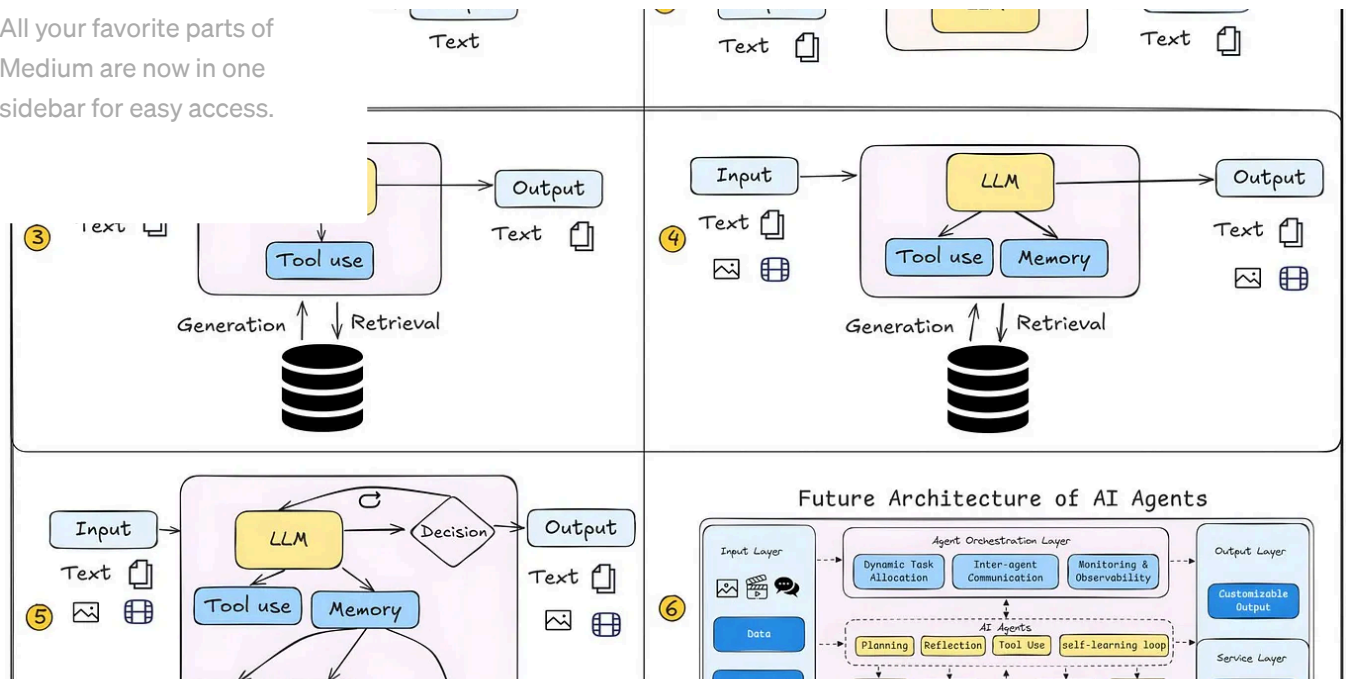
## Building a Multi-Agent AI System with LangGraph and LangSmith

A step-by-step guide to creating smarter AI with sub-agents

✦ Jun 2 🖱 1.6K 💬 24



All your favorite parts of Medium are now in one sidebar for easy access.

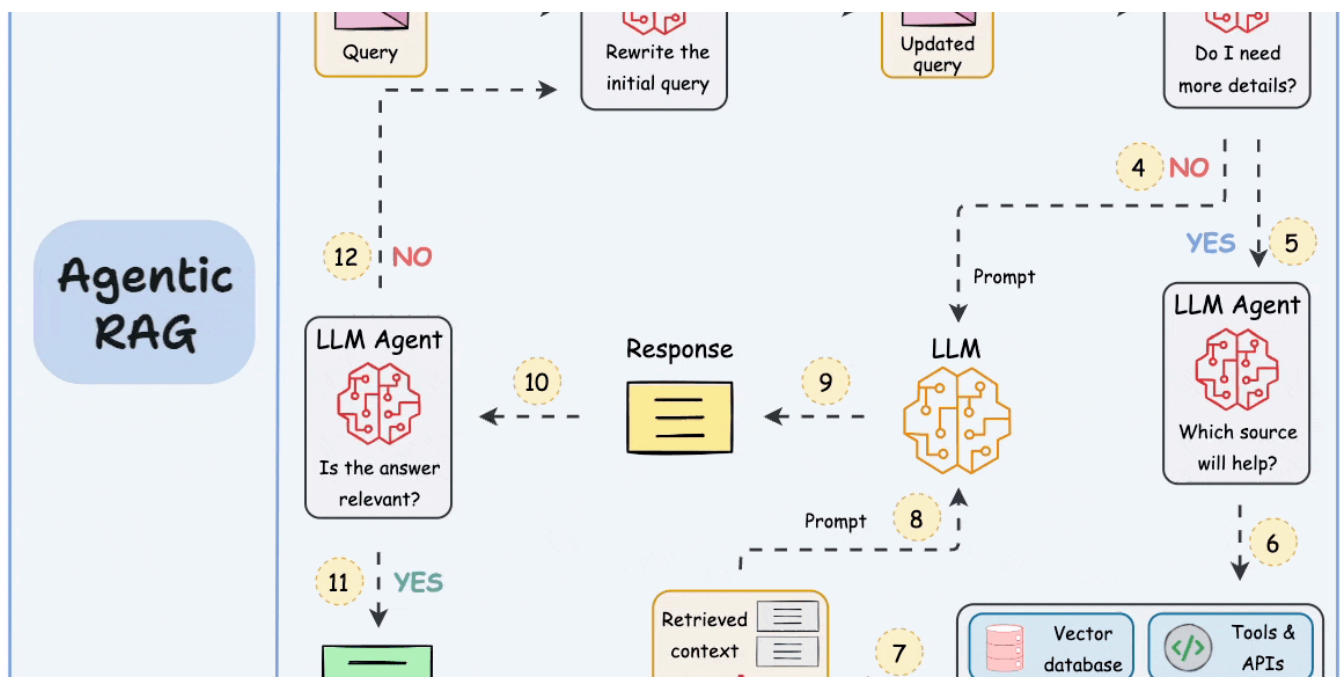


**DSC** In Data Science Collective by Paolo Perrone

## Why Most AI Agents Fail in Production (And How to Build Ones That Don't)

I'm a 8+ years Machine Learning Engineer building AI agents in production.

✦ Jun 16 🖱 2.1K 💬 37



**AI** In Artificial Intelligence in Plain English by Piyush Agnihotri

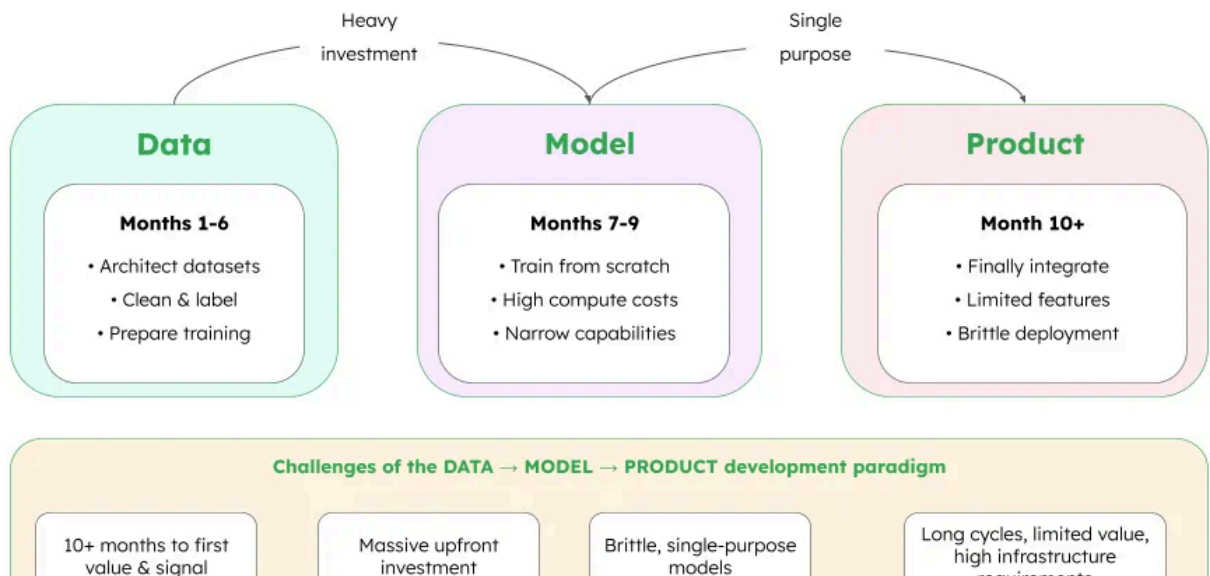
# Building Agentic RAG with LangGraph: Mastering Adaptive RAG for Production

All your favorite parts of Medium are now in one sidebar for easy access.

ems that know when to retrieve documents, search the web, or tly



## Traditional AI: Pre-Foundation Model Era



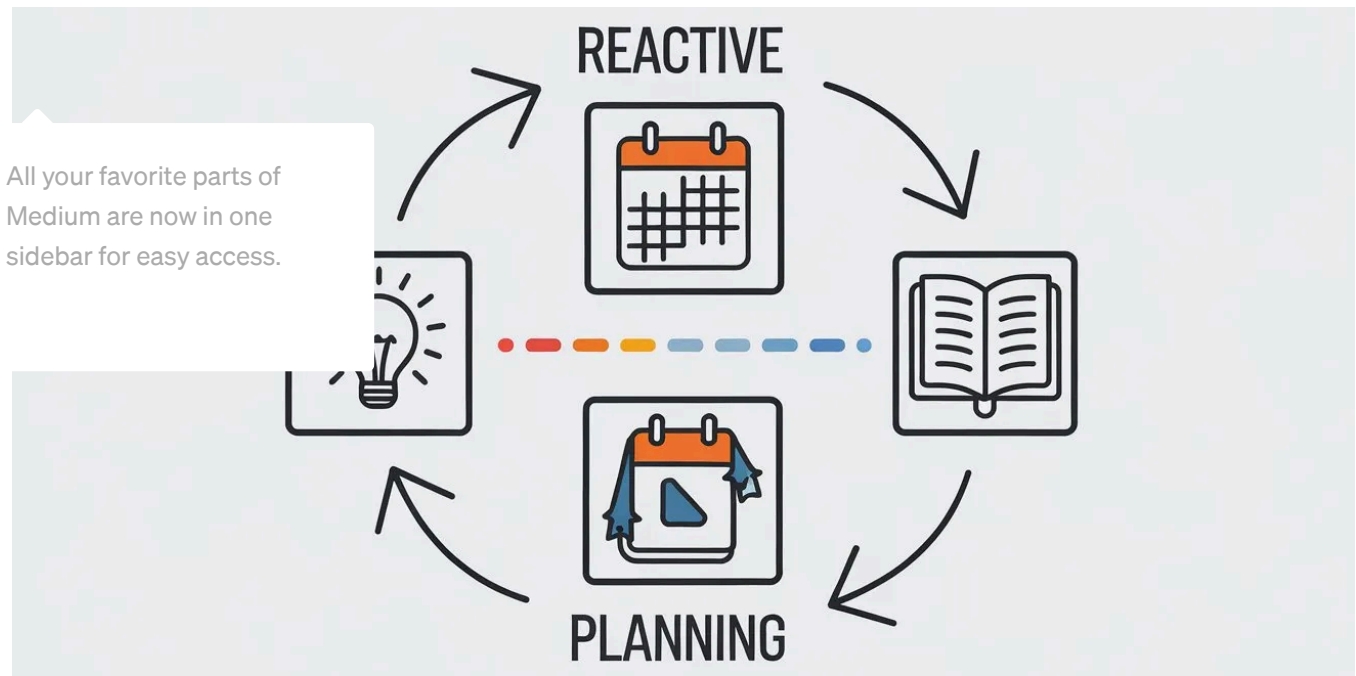
In MongoDB by MongoDB

## Build AI Agents Worth Keeping: The Canvas Framework

This article was written by Mikiko Bazeley.

Oct 8 108 5





 Aman Raghuvanshi

## Agentic AI #4—Understanding the Different Types of AI Agents: Reactive, Planning, and More

Learn about different types of AI agents—Reactive, Goal-Based, Utility-Based, Learning & Hybrid—in this deep dive into the architecture

Jun 5  58



See more recommendations