das-purnendu

LANGCHAIN & LANGGRAPH INTERVIEW HANDBOOK 2026

From your first chain
to senior-level system design.

38 slides • Code-first • Built for juniors

Save this. You'll need it. ↓

— Purnendu Das

das-purnendu

HOW TO USE THIS HANDBOOK

Read it like a tutorial.

Revisit it like a cheatsheet.

EVERY TOPIC GIVES YOU 4 THINGS:

1. **What it is,** in plain English
2. **Why it matters** in production
3. **A code snippet** you can run
4. **The interview signal** it tests

Don't memorize. Build.

Open a notebook and break every snippet by hand.

1. What it is

Clear concepts in plain, simple English.

2. Why it matters

Understand the real-world impact and use cases.

3. Code you can run

Tested snippets. Copy, paste, and experiment.

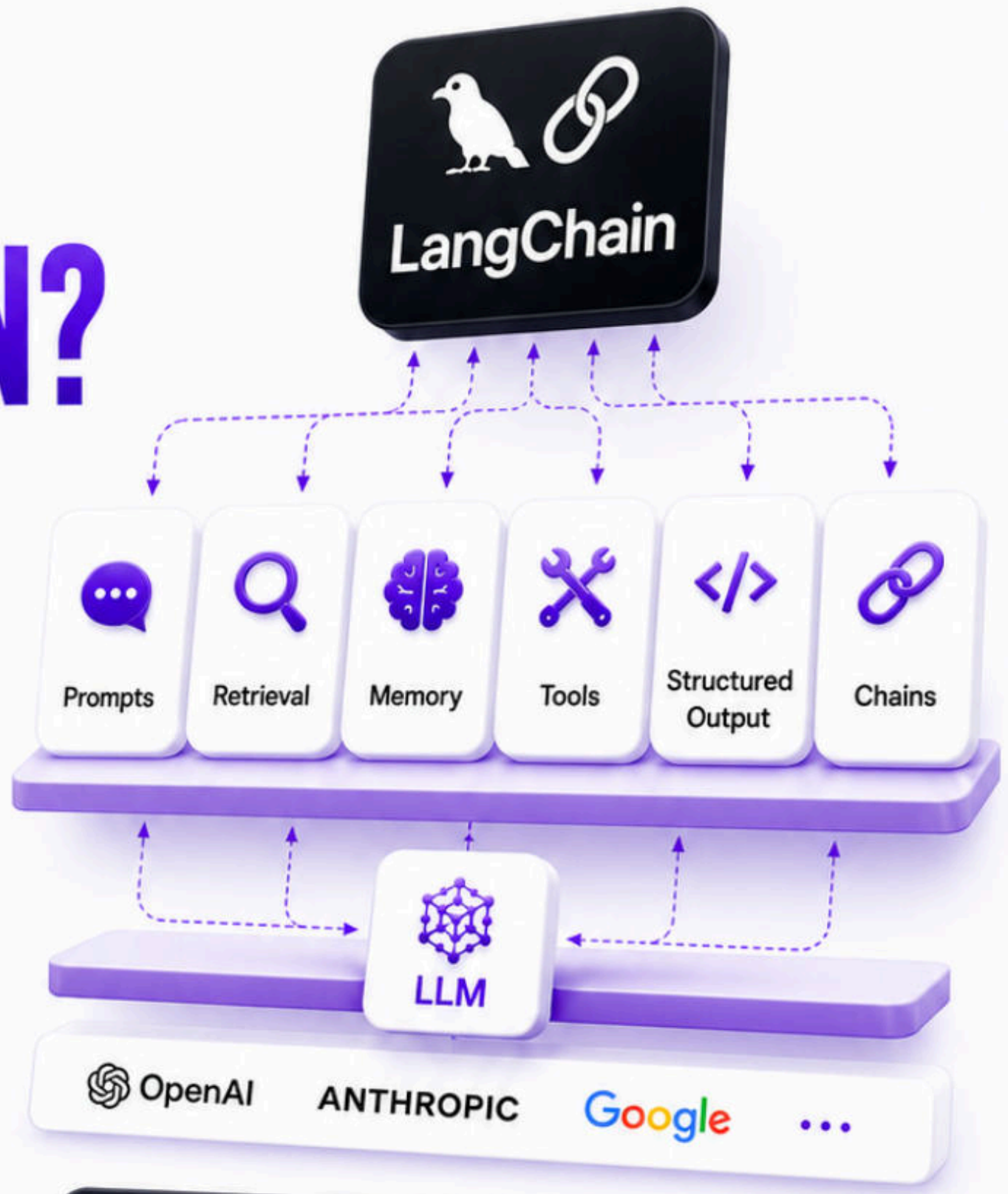
4. Interview signal

Know what interviewers are really testing behind the question.

This is your fast track to production-ready AI engineering.

das-purnendu

WHAT IS LANGCHAIN?



An LLM is just a function: text in, text out.

Real apps need: prompts, retrieval, memory, tools, structured output, multi-step chains.

LangChain gives you uniform abstractions for all of these — so you compose like Lego instead of writing glue code.

THE MENTAL MODEL

Every piece is a **Runnable.**

Pipe them with |

```python
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

chain = (
  {
    "context": retriever | format_docs,
    "question": RunnablePassthrough(),
  }
  | prompt
  | model
  | StrOutputParser()
)

chain.invoke({"question": "What is LangChain?"})
```

LangChain = The operating system for LLM applications.

Composable. Testable. Production-ready.

das-purnendu

LANGCHAIN ARCHITECTURE

5 layers, every project.

1 MODELS

OpenAI, Anthropic,
Bedrock, local and more.

2 PROMPTS

Templates and messages,
not f-strings.

3 RETRIEVAL

Loaders, splitters, embeddings,
vector stores.

4 COMPOSITION (LCEL)

The | operator.
Compose, parallelize, stream.

5 AGENTS & TOOLS

When LLMs need to do
real-world things.

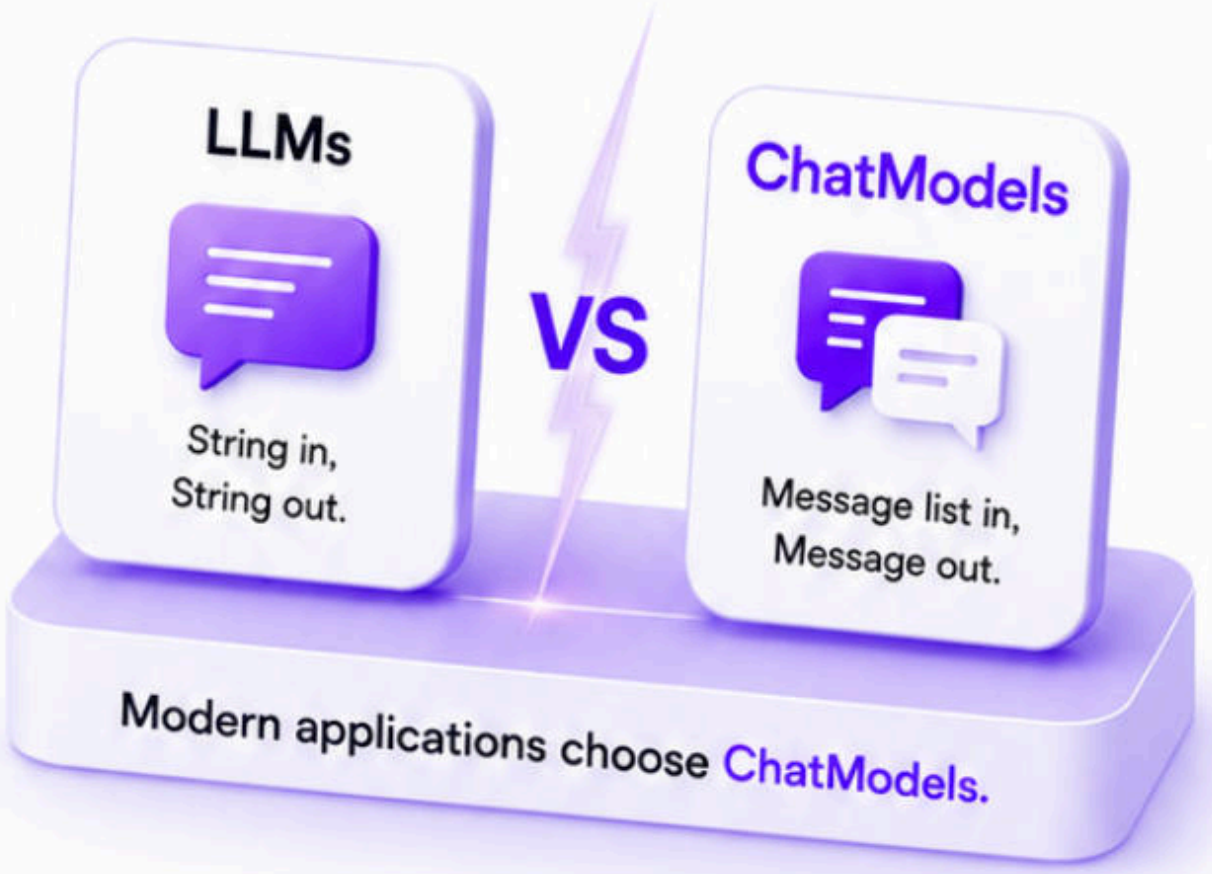
Interview rounds map to layers.
Know which one a question targets.

LangChain's design is simple on purpose —
so **you can focus on solving real problems, not wiring glue.**

5/38

das-purnendu

LLMS VS CHATMODELS



Two interfaces.
Most beginners pick the wrong one.

LLMs (Legacy)

- Single string input
- Single string output
- No role separation
- Harder to control behavior
- Mostly used in old codebases

ChatModels (Modern)

- List of messages as input
- Message output
- Roles: System, Human, AI
- Better instruction following
- The standard in 2026 and beyond

```python
from langchain_openai import ChatOpenAI
model = ChatOpenAI(model="gpt-4o-mini")
response = model.invoke("Hello, how are you?")
print(response.content)

# Always use this.
```

Input (messages)

System: You are a helpful assistant.

Human: Hello, how are you?

↓

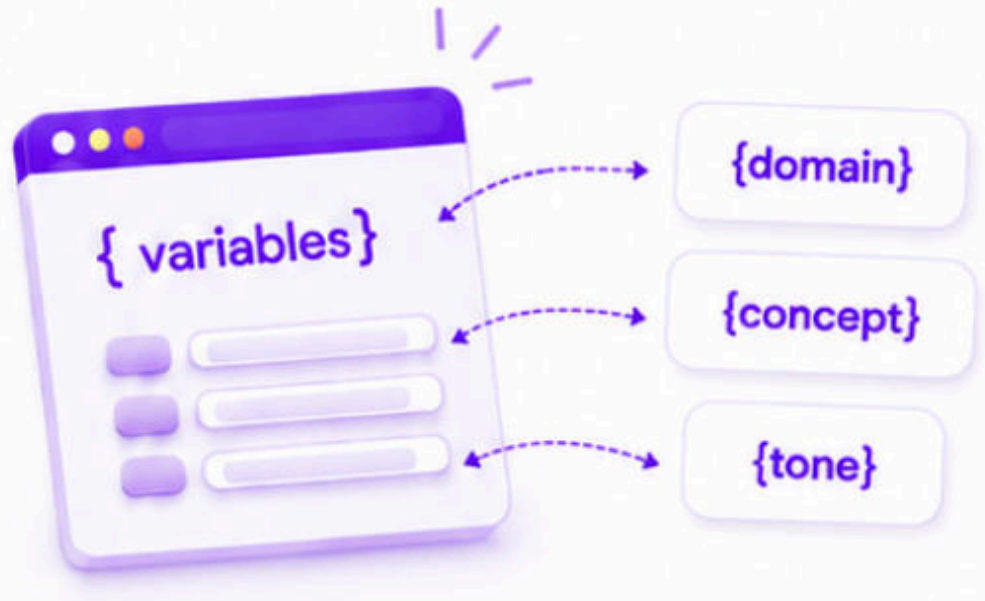
AI: I'm doing great, thanks!

Red flag:

Candidates reaching for OpenAI() for chat use cases in 2026 likely learned from old tutorials.

das-purnendu

PROMPTS & TEMPLATES



Stop hardcoding. Start templating.

```python
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert in {domain}."),
    ("human", "Explain {concept} in 2 lines."),
])

chain = prompt | model

chain.invoke({
    "domain": "Machine Learning",
    "concept": "Overfitting"
})
```

Rendered prompt (example)

SYSTEM
You are an expert in Machine Learning.

HUMAN
Explain Overfitting in 2 lines.

WHY IT MATTERS

Testable
Unit test prompts like any other code.

Versionable
Track changes, diff, review.

Swappable
Change prompts without changing code.

Shareable
Reuse across projects and teams.

BAD VS GOOD

Hardcoded (Bad)

```
query = f"""
 You are an expert
 in Machine Learning,
 Explain Overfitting
 in 2 lines.
"""
```

- ✗ Hard to test
- ✗ Hard to reuse
- ✗ Messy and brittle

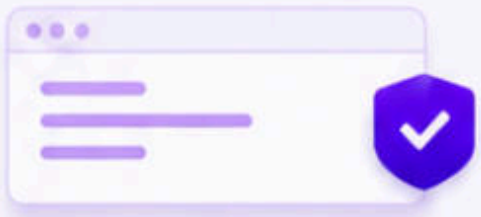
Templated (Good)

```
prompt.invoke({
  "domain": "ML",
  "concept": "Overfitting"
})
```

- ✓ Testable
- ✓ Reusable
- ✓ Clean and scalable

Anchor instructions in the system message.

RLHF models obey it more strongly.



Good prompts = better outputs. Invest time here.

→ Next: Output Parsers

das-purnendu

OUTPUT PARSERS

LLMs return strings.
Your app wants Pydantic objects.

The 2026 default:
`with_structured_output`

```python
from pydantic import BaseModel

class Joke(BaseModel):
    setup: str
    punchline: str

from langchain_openai import ChatOpenAI
model = ChatOpenAI(model="gpt-4o-mini")

structured = model.with_structured_output(Joke)

joke = structured.invoke("Tell me a joke")

print(joke)
# Joke(setup='...', punchline='...')
```

LLM (string)

```
{
  "setup": "Why did the developer go broke?",
  "punchline": "Because he used up all his cache!"
}
```

Parsed (Pydantic object)

```
Joke(
  setup='Why did the developer go broke?',
  punchline="Because he used up all his cache!"
)
```

Typed. Validated. Reliable.
No regex. No pain.

WHY THIS WINS

Uses native function-calling
More reliable than asking for JSON.

Auto-validates with Pydantic
Type-safe. Fewer runtime bugs.

Cleaner code
No manual `json.loads()`, no regex.

Better for production
Predictable outputs, easier monitoring.

BAD (Don't do this)

```
response = model.invoke("Tell me a joke")
data = json.loads(response.content)
joke = Joke(**data) # Can crash!
```

- ✗ Breaks on minor format changes
- ✗ Hard to debug
- ✗ Not future-proof

VS

GOOD (Do this)

```
structured = model.with_structured_output(Joke)
joke = structured.invoke("Tell me a joke")
# Always returns a Joke instance
```

- ✓ Robust
- ✓ Type-safe
- ✓ Production-ready

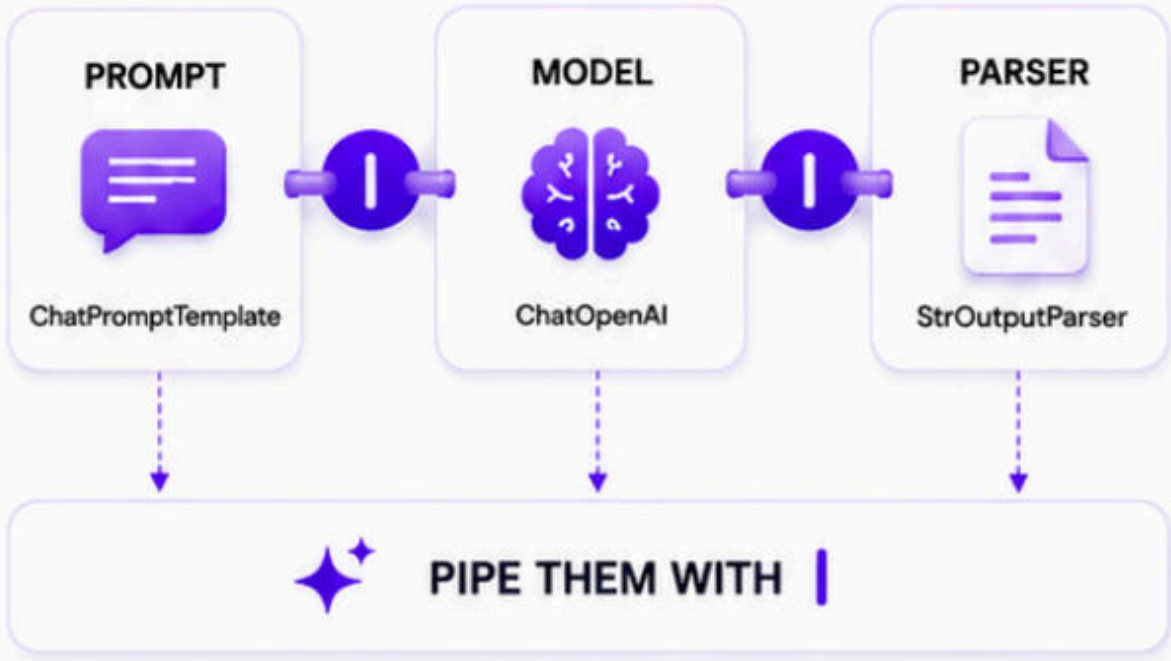
Think in schemas, not strings.
Define the shape once. Reuse everywhere.

8/38

das-purnendu

LCEL: THE PIPE OPERATOR

The system that killed LLMChain.



Readable. Composable. Powerful.

LCEL (LangChain Expression Language)
compose, run, stream. Effortless.

```python
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

# 1. Define components
prompt = ChatPromptTemplate.from_template("Tell me a {topic} in 2 lines.")
model = ChatOpenAI(model="gpt-4o-mini")
parser = StrOutputParser()

# 2. Compose with pipe operator (+ passthrough for observability)
chain = (
    {"topic": RunnablePassthrough()}   # passes input through for logging/debugging
    | prompt
    | model
    | parser
)
# 3. Invoke
result = chain.invoke("LCEL")
print(result)
```

WHAT'S HAPPENING?

- **Input goes into Prompt**
  Fills the template.
- **Prompt output goes to Model**
  LLM generates the response.
- **Model output goes to Parser**
  Converts to structured output.
- **You get the final result**
  Clean, usable, ready.

4 MODES. FREE. BUILT-IN.

INVOKE
Run once, get result.

```
chain.invoke(input)
```

Sync

AINVOKE
Async/await friendly.

```
await chain.ainvoke(input)
```

Async

BATCH
Run many inputs in parallel.

```
chain.batch([in1, in2, ...])
```

Parallel

STREAM
Stream tokens as they arrive.

```
for chunk in chain.stream(input):
    print(chunk, end="")
```

Streaming

Still writing LLMChain?
You're writing 2022 code.

```
from langchain.chains import LLMChain

chain = LLMChain(llm=model, prompt=prompt)
```

Don't do this.

LCEL > LLMChain
More control. More power. Less magic, more engineering.

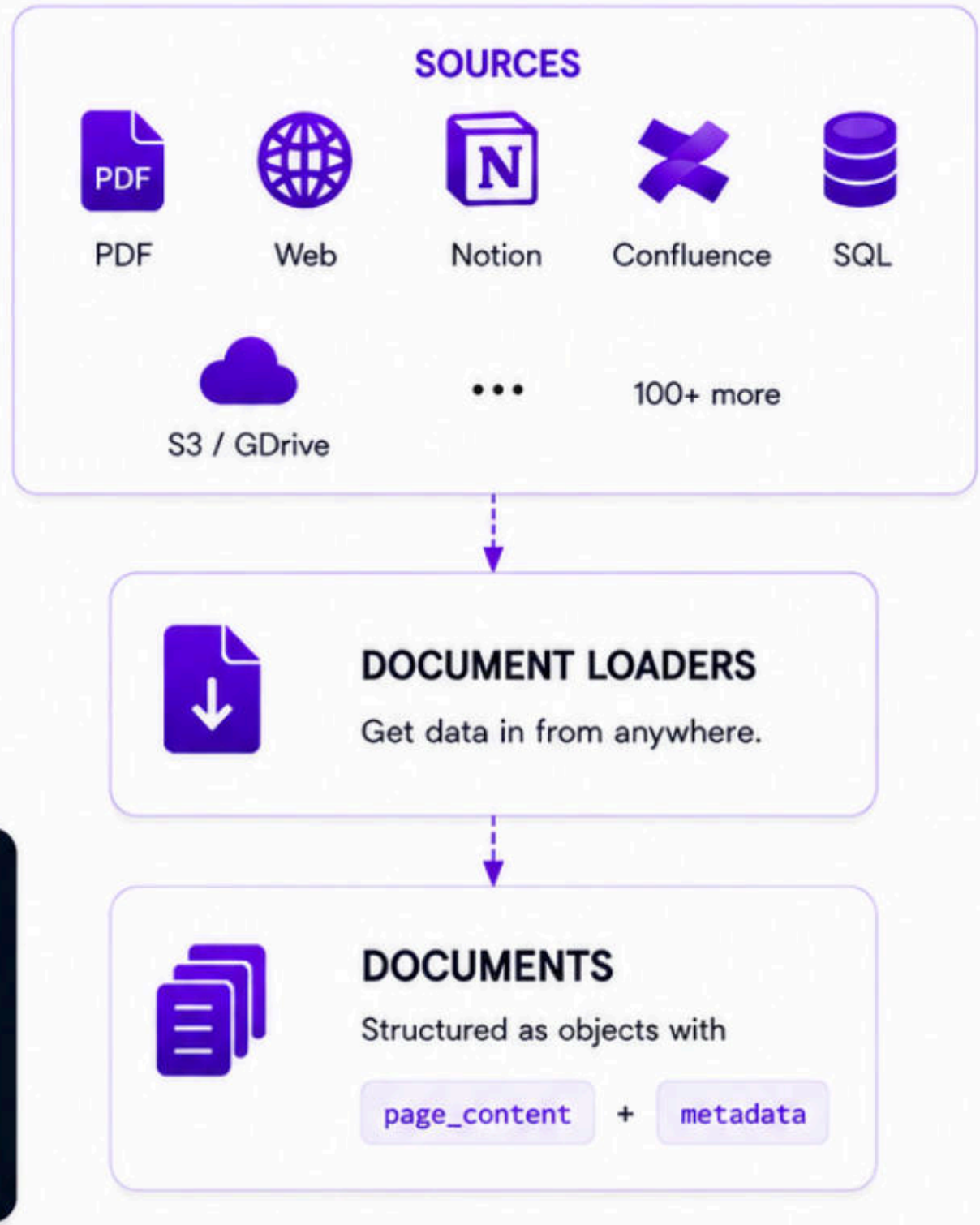
das-purnendu

DOCUMENT LOADERS

How your data gets in.

Loaders bring your data from anywhere into a standard Document format.

```python
from langchain_community.document_loaders import (
    PyPDFLoader, WebBaseLoader, DirectoryLoader
)

docs = PyPDFLoader("report.pdf").load()
# Each doc has: page_content + metadata
```



Loaders exist for: PDF, web, Notion, Confluence, S3, GDrive, SQL, and **100+** more.

WHAT YOU GET

Each document is an object with:

- `page_content`: the actual text
- `metadata`: where it came from and other useful context

```
{
   "page_content": "Q4 Summary shows strong growth ...",
   "metadata": {
      "source": "report.pdf",
      "page": 12,
      "title": "Q4 Summary",
      "author": "Finance Team",
      "created_at": "2026-05-12"
   }
}
```

WHY METADATA MATTERS

- Filter by source, date, author, section, etc.
- Cite the right page in answers
- Debug retrieval issues faster
- Build trustworthy, auditable RAG systems

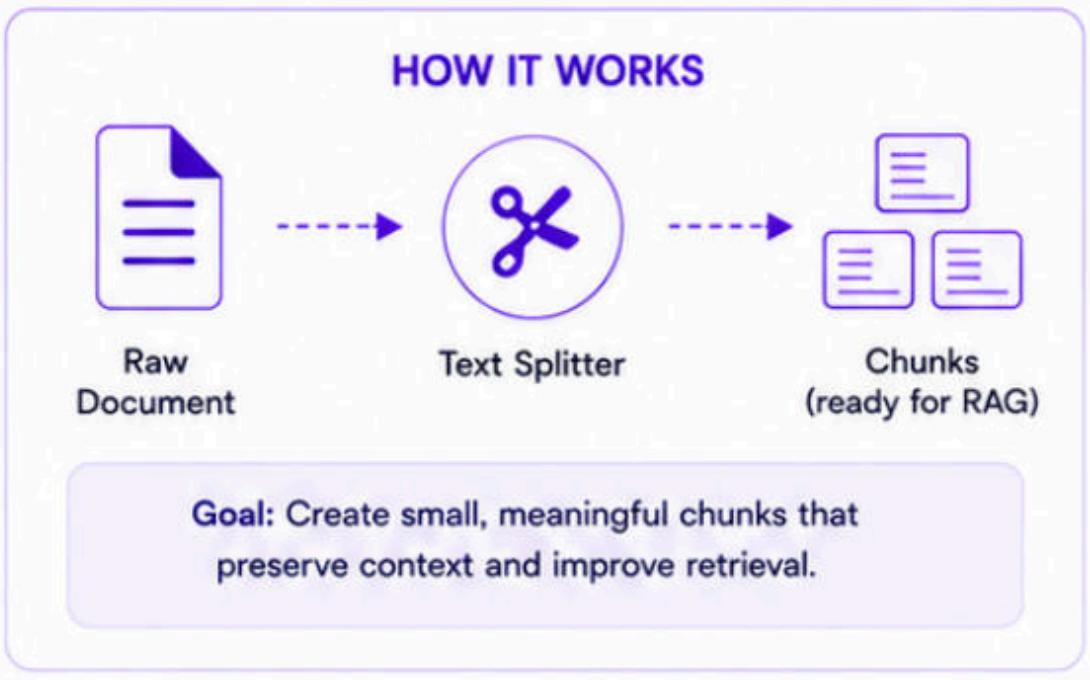
Never throw away metadata.

It's how you filter, cite, and debug retrieval.

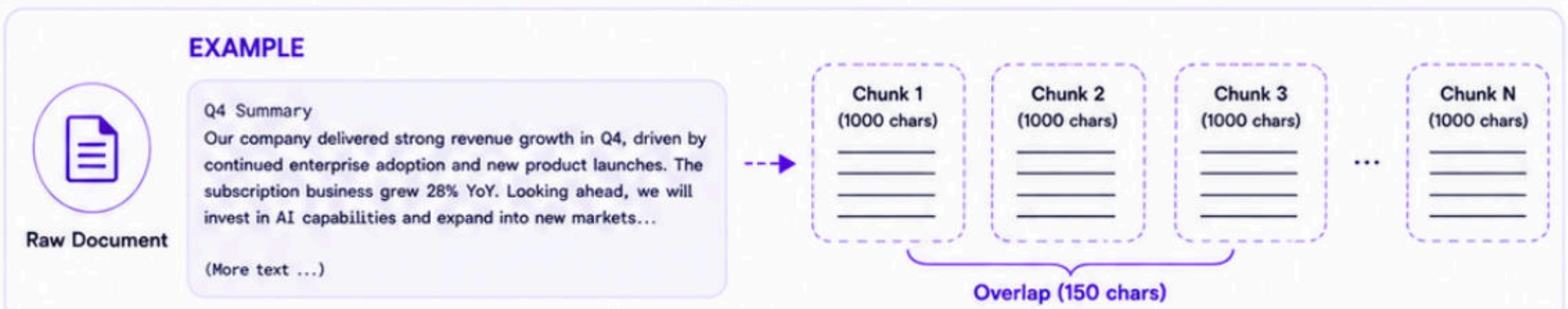
das-purnendu

TEXT SPLITTERS

Bad chunking destroys RAG.
Get this wrong, nothing else helps.




```python
from langchain_text_splitters import
    RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=150,
)

chunks = splitter.split_documents(docs)
```



CHUNK SIZE IS NOT ONE-SIZE-FITS-ALL.

"What size do you use?"
→ "Depends, we evaluated 3 sizes against our eval set."

PRACTICAL GUIDELINES

- Start with a range (e.g., 500, 1000, 1500).
- Evaluate on your eval set (retrieval metrics + downstream quality).
- Consider your model's context window.
- Balance: too small → lose context, too big → poor recall.

The right chunks = better retrieval = better answers.
Invest time in chunking. It pays off everywhere.

das-purnendu

EMBEDDINGS

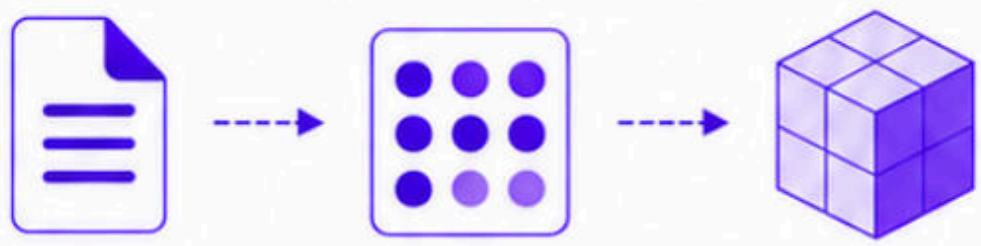
Embeddings = **meaning as math.**

> "Cancel my plan" and
> "unsubscribe from membership"
> land in nearly the same place
> in vector space.

```python
from langchain_openai import OpenAIEmbeddings

embedder = OpenAIEmbeddings(
    model="text-embedding-3-small"
)

vec = embedder.embed_query("Cancel my plan")
# 1536 floats representing meaning
```

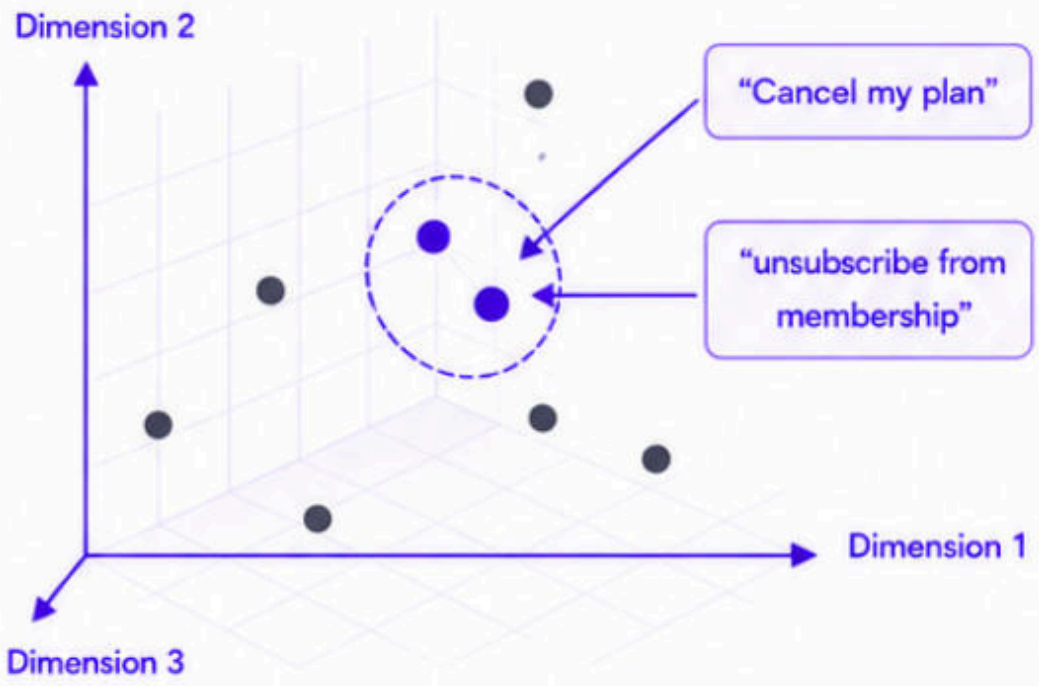
Output: List[float] of length **1536**

WHAT ARE EMBEDDINGS?



Transforms text into a list of numbers (vectors) that capture its meaning.

VECTOR SPACE VISUALIZATION



Semantically similar text lands close together.

USE CASES

Semantic Search
Find relevant content by meaning, not keywords.

Semantic Matching
Match user queries to intents / documents.

Clustering
Group similar documents automatically.

Recommendations
Recommend based on semantic similarity.

EXAMPLE

Query: "Cancel my plan" → `[ -0.021, 0.413, -0.177, ..., 0.248 ]`

Query: "unsubscribe from membership" → `[ -0.018, 0.407, -0.180, ..., 0.251 ]`

Cosine Similarity
≈ 0.98
(Very similar)

CRITICAL: Consistency is non-negotiable.

Query and document embeddings MUST come from the same model.

Mix two = random noise retrieval.

Same model ✓
text-embedding-3-small ↔ text-embedding-3-small
Good results

Different models ✗
text-embedding-3-small ↔ text-embedding-ada-002
Poor / random results

das-purnendu

VECTOR STORES

Where your **embeddings** live and get **searched.**

EMBEDDINGS LIVE HERE

Store once. Search many.

Local: FAISS, Chroma — prototypes

Great for local development and small-scale experiments.

Postgres: pgvector — already on Postgres? Use this

Keep your data and vectors in the same trusted database.

Enterprise: OpenSearch — best for hybrid search at scale

Powerful, scalable, and built for complex search workloads.

Managed: Pinecone, Weaviate, Qdrant

Fully managed vector databases — focus on building, not infra.

Interview red flag:

Naming Pinecone reflexively without context = filtered out.

Choose based on needs: scale, latency, filters, hybrid search, deployment, cost.

```python
from langchain_community.vectorstores import FAISS
vs = FAISS.from_documents(chunks, embedder)
results = vs.similarity_search(query, k=4)
# results -> List[Document]
```

Pro tip

Good chunks + good embeddings + right vector store = great results.

Bad any one of them = disappointing results.

das-purnendu

RETRIEVERS ≠ VECTOR STORES

A common confusion that costs interviews.

VectorStore

stores + searches vectors

- Owns the index and data
- Knows how to add, delete, and search
- Examples: FAISS, Chroma, pgvector, Pinecone, Qdrant, Weaviate, OpenSearch

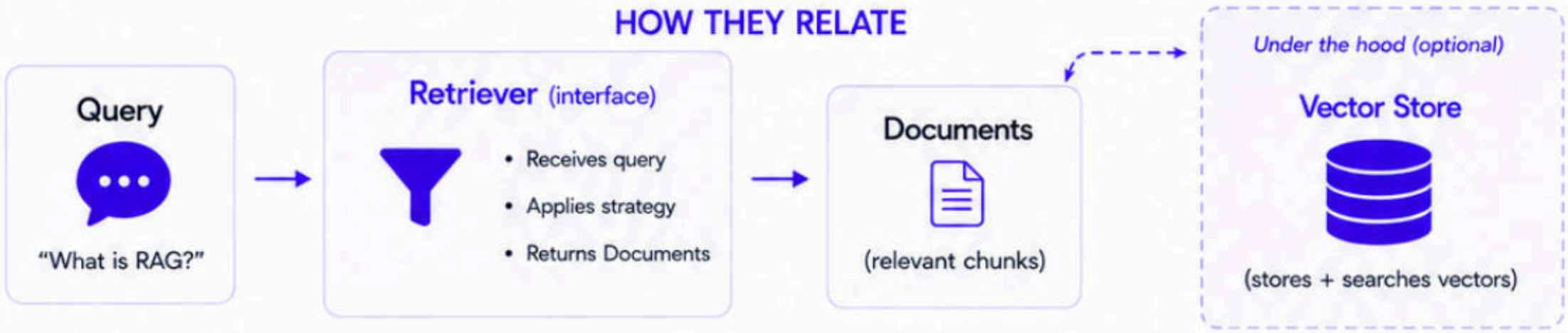
Retriever

interface that returns Documents for a query

- Defines how to get relevant documents
- May use a vector store (or not)
- Examples: VectorStoreRetriever, BM25Retriever, EnsembleRetriever, ParentDocumentRetriever

Every vector store can become a retriever.
But a retriever **doesn't** need a vector store.

HOW THEY RELATE



RETRIEVERS CAN USE MANY STRATEGIES

Vector Search	BM25 / Keyword	Ensemble	Parent / Multi-step	Custom
(via VectorStore)	(sparse search)	(hybrid / fusion)	(advanced retrieval)	(your logic)

```python
retriever = vs.as_retriever(search_kwargs={"k": 4})
# Or BM25, ensemble, custom – same interface
```

Key takeaway

Think of retriever as the "brain" for fetching.
It decides **how, not where.**

This separation is one of LangChain's best ideas.

- Pluggable strategies
- Easier testing
- Swap without changing your chain

das-purnendu

BASIC RAG PIPELINE

The 7-step shape every senior knows by heart:

1 Load	2 Split	3 Embed	4 Index	5 Retrieve	6 Augment prompt	7 Generate
Load raw documents from source.	Split into small, overlapping chunks.	Convert chunks to embedding vectors.	Store vectors in a vector store.	Retrieve relevant chunks for the user query.	Add retrieved context to the user prompt.	LLM generates answer using context.
Files, DBs, APIs, Web	Chunk size, overlap	Embedding Model	Vector Store (FAISS, pgvector, Pinecone, ...)	Top-K similarity	Prompt Template	LLM (Model)

python

```python
rag_chain = (
    {"context": retriever | format_docs,
     "question": RunnablePassthrough()}
    | prompt | model | StrOutputParser()
)
```

`retriever`
Fetches relevant documents (steps 1–5)

`format_docs`
Formats docs into a single context string

`prompt`
Injects context + question into a prompt template

`model`
LLM that produces the answer

`StrOutputParser()`
Parses model output as a string

What flows where?

User question
↓
Retriever finds relevant chunks
↓
Context is added to prompt
↓
LLM generates answer
↓
Answer to user

Every “production RAG” question
= how to make these 7 steps better.

- Better data → Better chunks
- Better retrieval → Better context
- Better prompts → Better answers
- Better evaluation → Trust & reliability

15/38

das-purnendu

ADVANCED RETRIEVERS

GOAL

Find the **most relevant** context, not just any context.

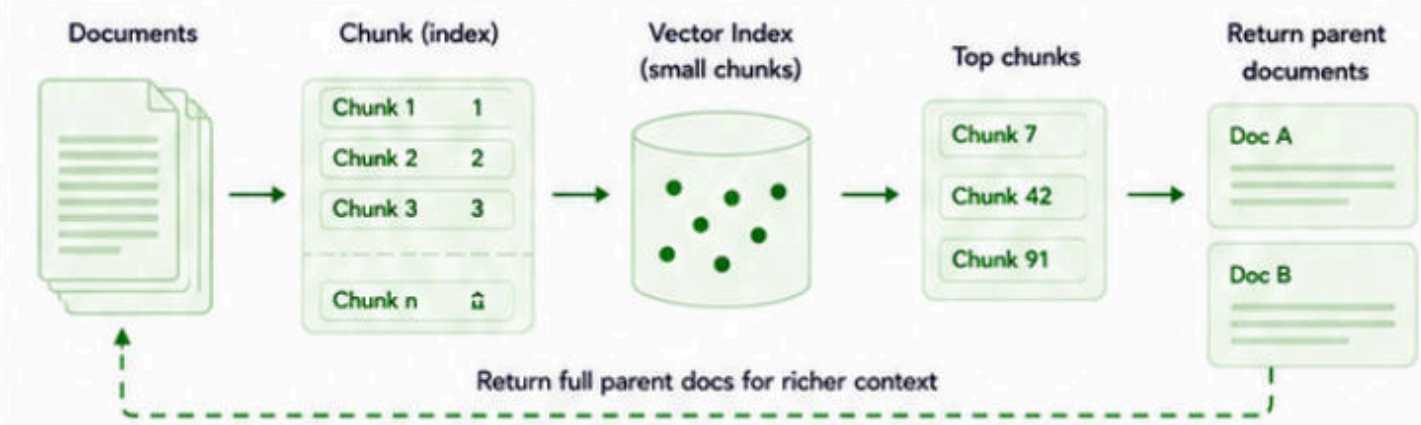
Where **production RAG** actually wins.

1. MultiQuery

Rephrases query N ways.
Solves vocab mismatch.

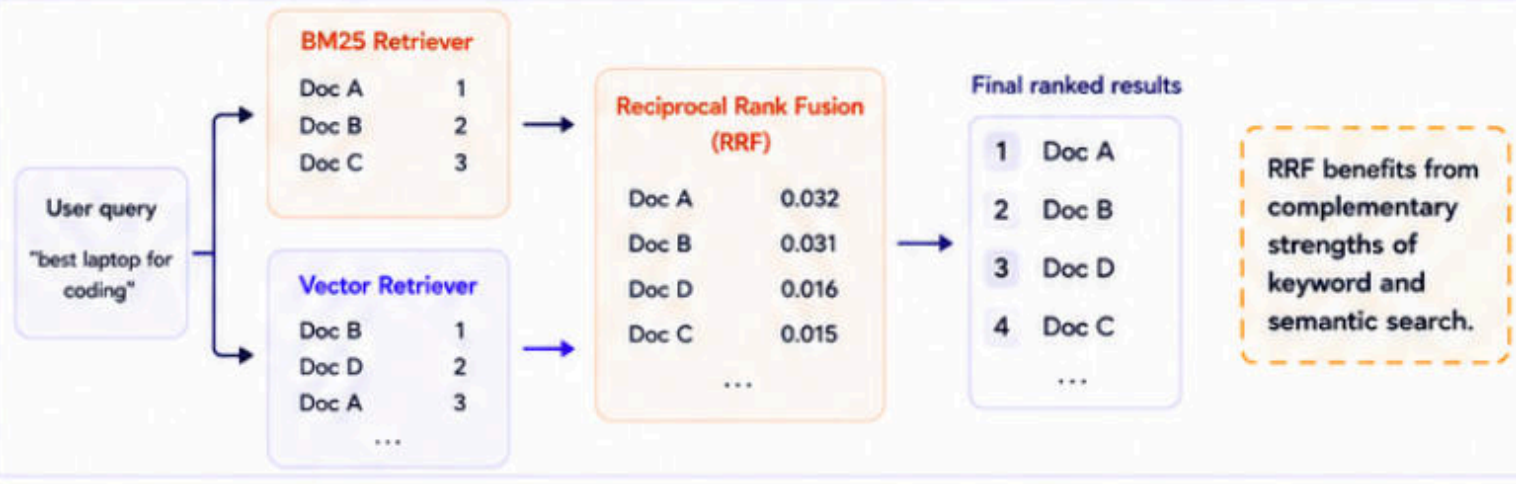
2. ParentDocument

Index small chunks,
return big parents.
Best of both worlds.



3. Ensemble

BM25 + vectors via
reciprocal rank fusion.



Reranking is the final 10% of magic.

Cohere Rerank or BGE reranker on the top 20 candidates → pick best 4.

Cheap latency, huge quality gain.

WHY IT MATTERS

Higher recall
Catch more relevant information across different phrasings.

Better context
Keep full meaning with larger parent content.

Stronger results
Combine strengths of multiple retrieval methods.

Higher precision
Reranking puts the most relevant at the top.

PRO TIP: Start advanced, but measure always. Offline eval (Recall@K, MRR) + online eval (user feedback, task success).

das-purnendu

MEMORY IN LANGCHAIN

LLMs are **stateless**.

Memory **injects past turns** into the next prompt.

WHY MEMORY?

WHAT MEMORY DOES

- Remembers past messages
- Maintains conversation context
- Improves relevance & coherence
- Enables personalized interactions

HOW IT WORKS


```python
from langchain_core.runnables.history import
RunnableWithMessageHistory

with_memory = RunnableWithMessageHistory(
    chain, get_history,
    input_messages_key="input",
    history_messages_key="history",
)
```

Key point

Memory isn't magic.
It's just the right messages
in the right place.

LONG CONVERSATIONS EXPLODE TOKENS.

Three proven strategies to manage memory:

1 Truncation

Keep the most recent N turns.
Simple and effective.

Use when

Conversations are short-medium and recency matters most.

2 Summarization

Summarize older turns into a compact summary.
Keep the summary + recent turns.

Use when

Conversations are long and details from the past still matter.

3 Retrieval-Based Memory

Store past turns in a vector store.
Retrieve only what's relevant to the current query.

Use when

Conversations are very long or you need precise, relevant recall.

Beyond single-turn chatbots, prefer LangGraph state + checkpointer. Way more powerful.

- Durable execution across restarts
- Human-in-the-loop & branching
- Persistent state for complex workflows
- Production-ready at scale

COMMON MEMORY BACKENDS

- **In-Memory** (ConversationBufferMemory) — Fast & simple. Lost on restart.
- **Redis** — Shared across instances. Good for scale.
- **Postgres / SQL** — Structured, reliable, great for production.
- **MongoDB** — Flexible schema for complex apps.
- **Custom** — Roll your own (files, KV stores, cloud DBs, etc.).

Pro tip

Always use a `session_id` to isolate memory per user / conversation.

17/38

das-purnendu

TOOLS & FUNCTION CALLING

How LLMs do things.

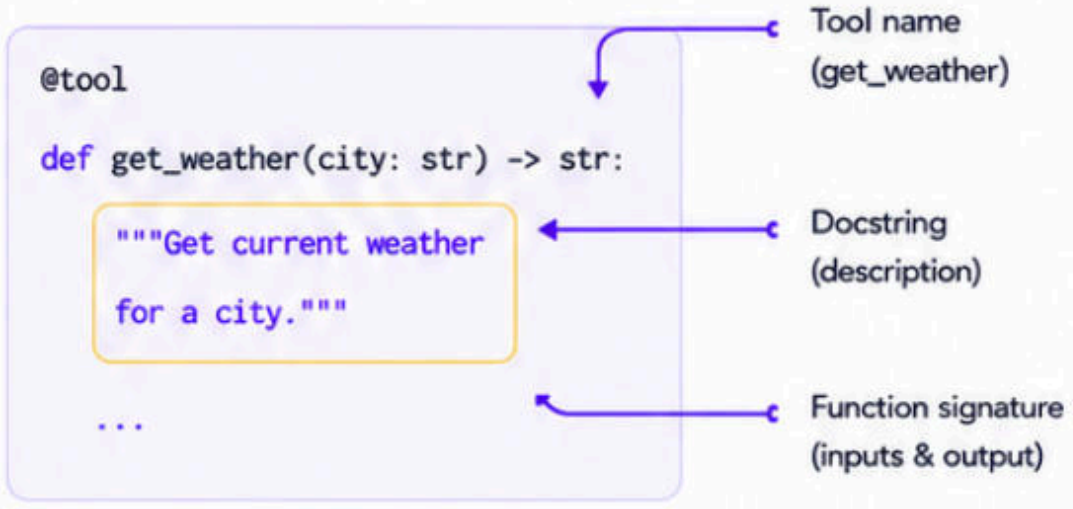
```python
from langchain_core.tools import tool

@tool
def get_weather(city: str) -> str:
    """Get current weather for a city."""
    return f"28°C in {city)"

model_with_tools = model.bind_tools([get_weather])
```

What's happening?

- You define a Python function and decorate it with @tool.
- LangChain converts it to a tool the model can call.
- The model decides when to call it (based on your prompt and the tool description).
- The tool runs, result is sent back, model uses it to answer.

THE FLOW

TOOL ANATOMY

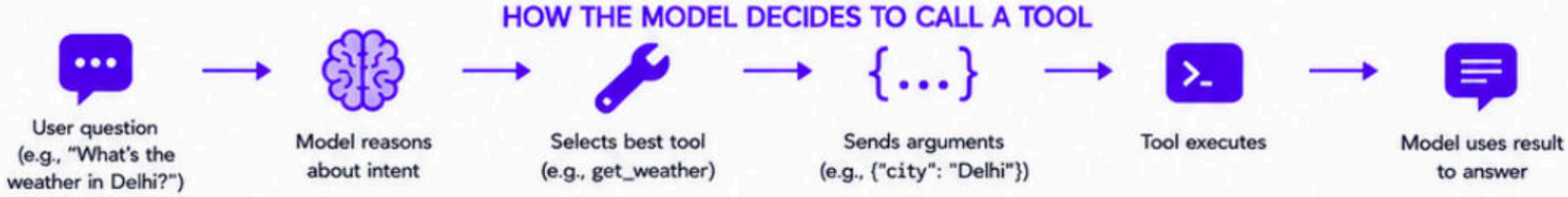


The docstring IS the prompt the model sees.
Vague docstring = bad tool selection.

COMMON USE CASES

Weather | Stock prices | Web search | Calculator
Calendar | Email | Database | Any API

HOW THE MODEL DECIDES TO CALL A TOOL



A TOOL DEFINITION EXAMPLE (UNDER THE HOOD)

```
{
  "name": "get_weather",
  "description": "Get current weather for a city.",
  "parameters": {
    "type": "object",
    "properties": {
      "city": {"type": "string", "description": "City name"}
},   "required": ["city"]
```

- This schema is what the model sees.
- Keep names clear. Descriptions tight.
- Fewer, focused tools = better decisions.

Biggest mistake: too many tools.
Models confuse at 10+.

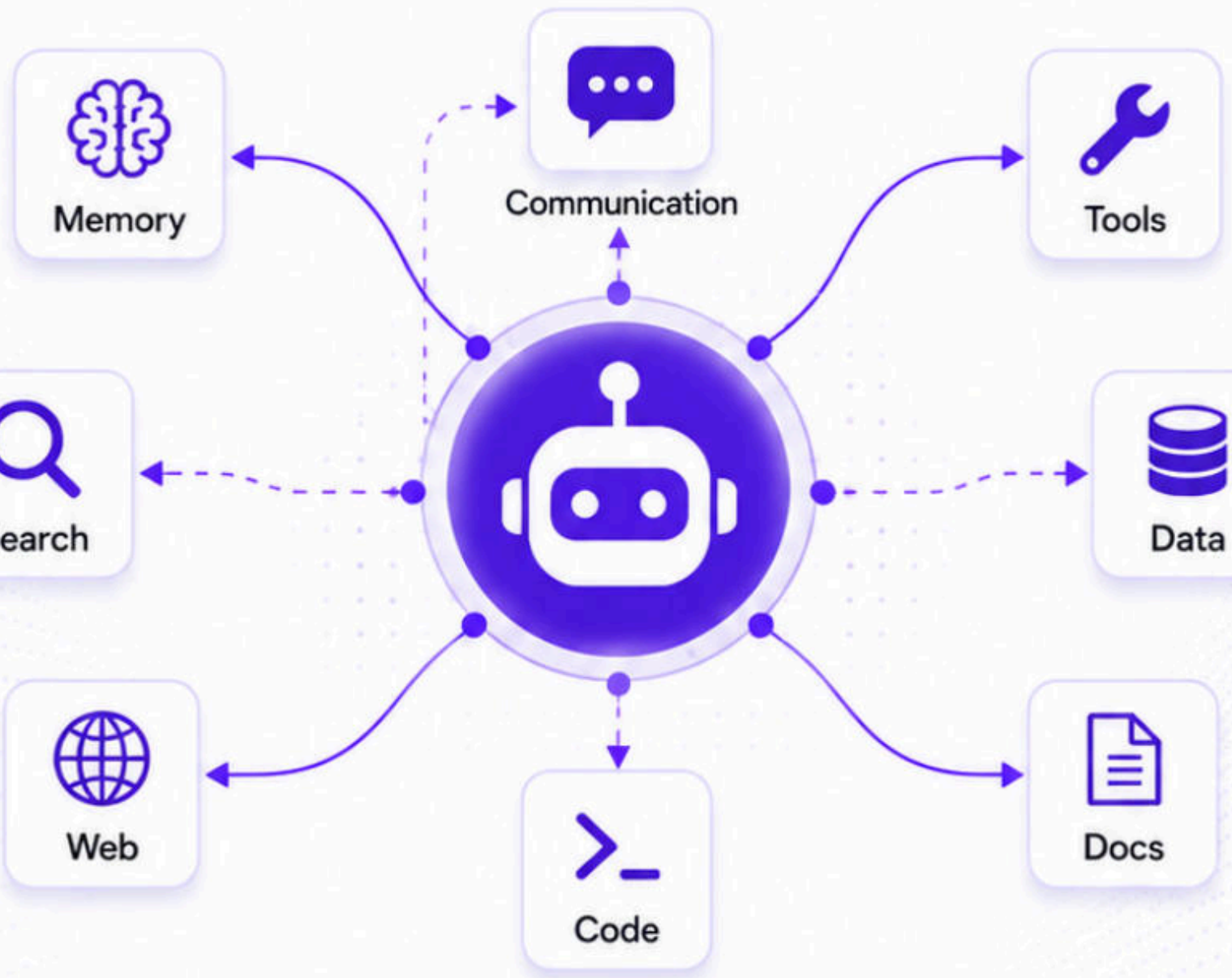
- Build semantic wrappers (combine related actions).
- Use MCP (slide 31) for discoverability at scale.
- Quality > Quantity.

das-purnendu

PART 2

LangGraph

Where chains end and real agents begin.



Continue swiping →

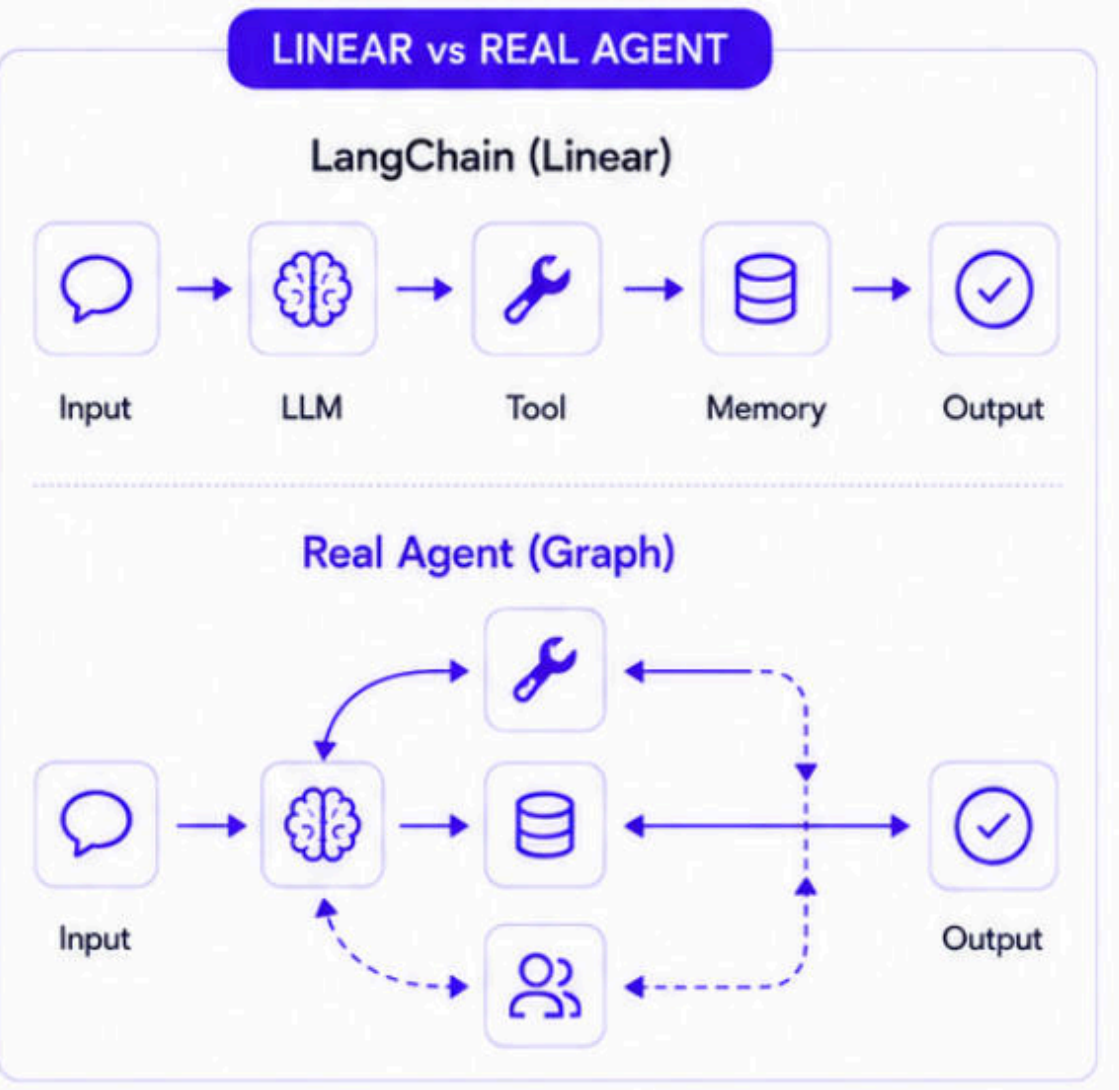
das-purnendu

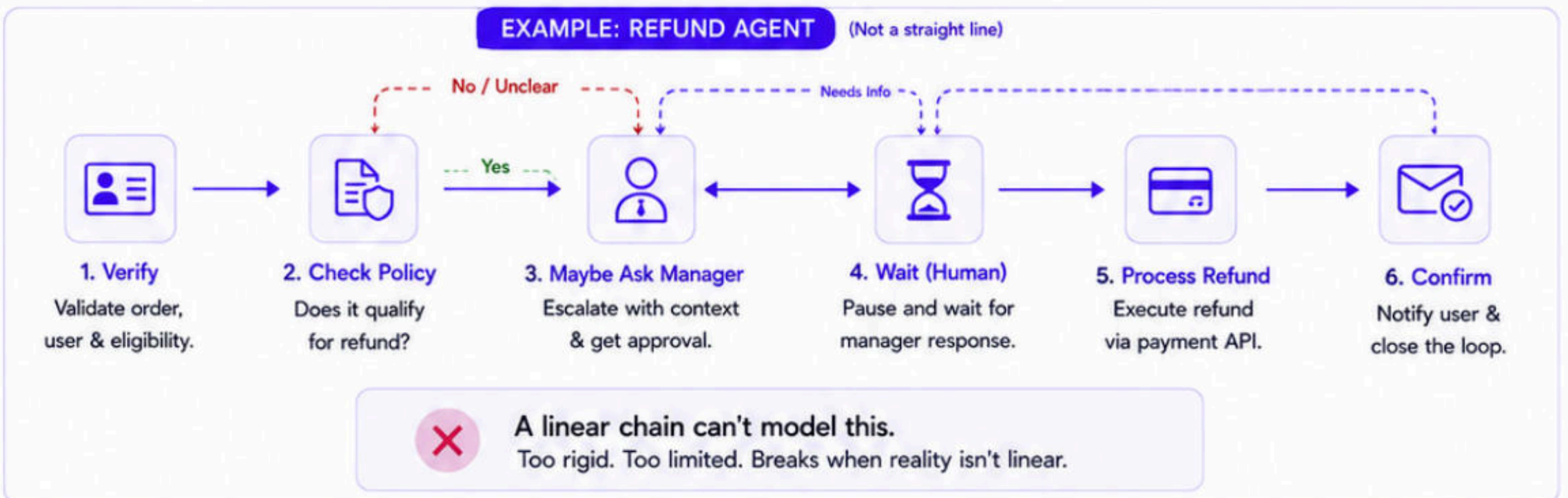
Why LangGraph?

LangChain composes in a straight line.

A real agent isn't a line. It's a graph.

It loops, branches, waits for humans, runs things in parallel.





A linear chain can't model this.
Too rigid. Too limited. Breaks when reality isn't linear.

LangGraph = explicit state machine.

Every transition, every state update is code you control.

Deterministic
You decide how the agent moves.

Durable State
State persists across steps & crashes.

Human-in-the-Loop
Pause, review, then resume.

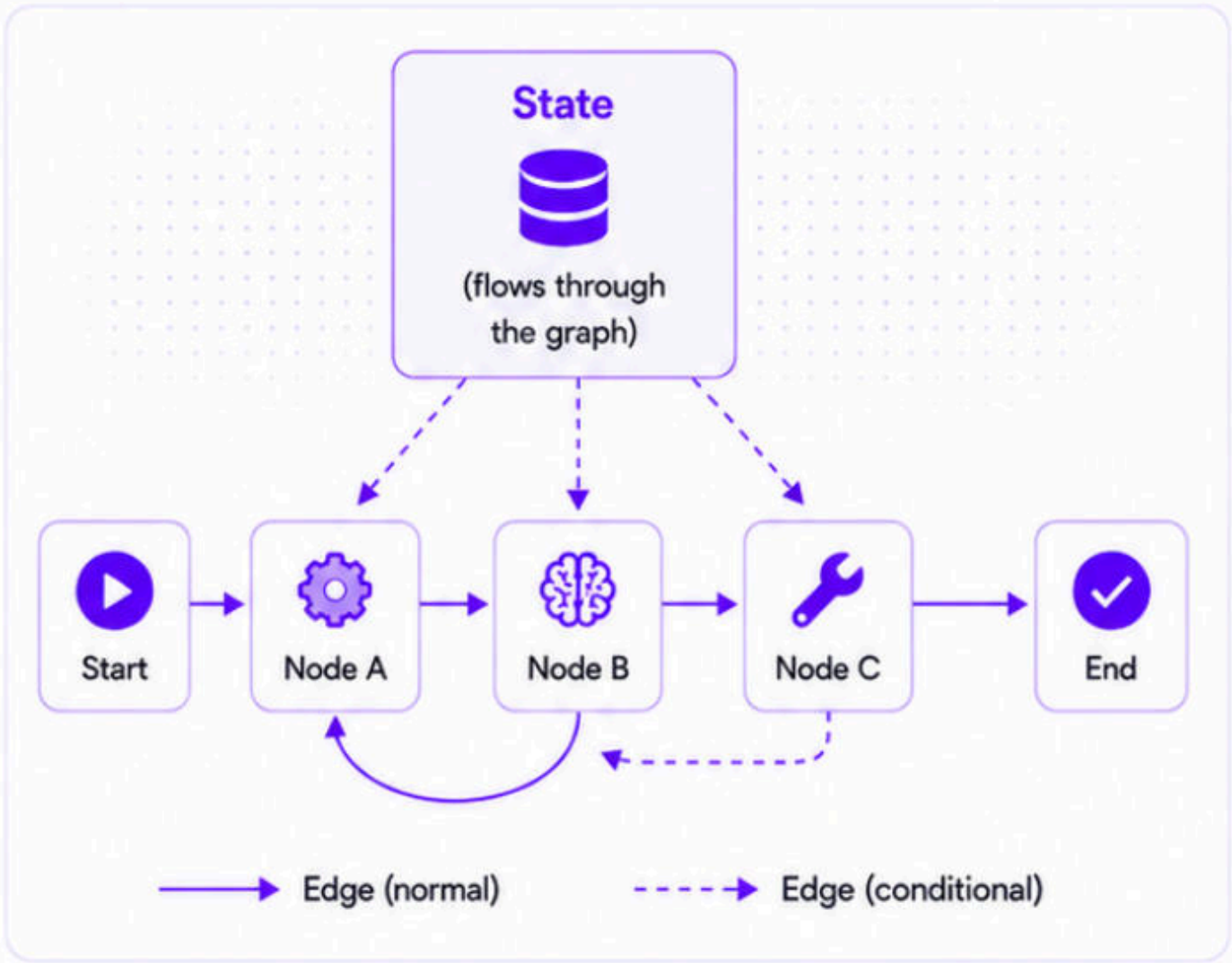
Parallel & Fast
Run multiple things at once.

Bottom line: LangChain gets you from A to B. **LangGraph gets you anywhere.**

das-purnendu

State, Nodes, Edges

Three primitives. That's it.




```python
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages

class AgentState(TypedDict):
    messages: Annotated[list, add_messages]
    user_query: str
    final_answer: str | None
```

State

Typed dict that flows through the graph.

```
AgentState
```

Nodes

Functions that read state, return updates.

```
def node_x(state) -> dict:
    return {"key": value}
```

Edges

Connect nodes (conditional or not).

```
add_edge(a, b)
add_conditional_edges(
    a, condition_fn)
```

Nodes return updates, not new state.
Mixing this up = "my agent forgets" bugs.

Golden Rule

Read state → do work → return updates
LangGraph merges updates into state.

das-purnendu

Your First Graph

Let's build the canonical self-corrective RAG loop.

```python
from langgraph.graph import StateGraph, START, END

g = StateGraph(State)
g.add_node("retrieve", retrieve)
g.add_node("generate", generate)
g.add_node("grade", grade)

g.add_edge(START, "retrieve")
g.add_edge("retrieve", "generate")
g.add_edge("generate", "grade")
g.add_conditional_edges("grade", route)

app = g.compile()
```

retrieve
Fetch relevant documents.

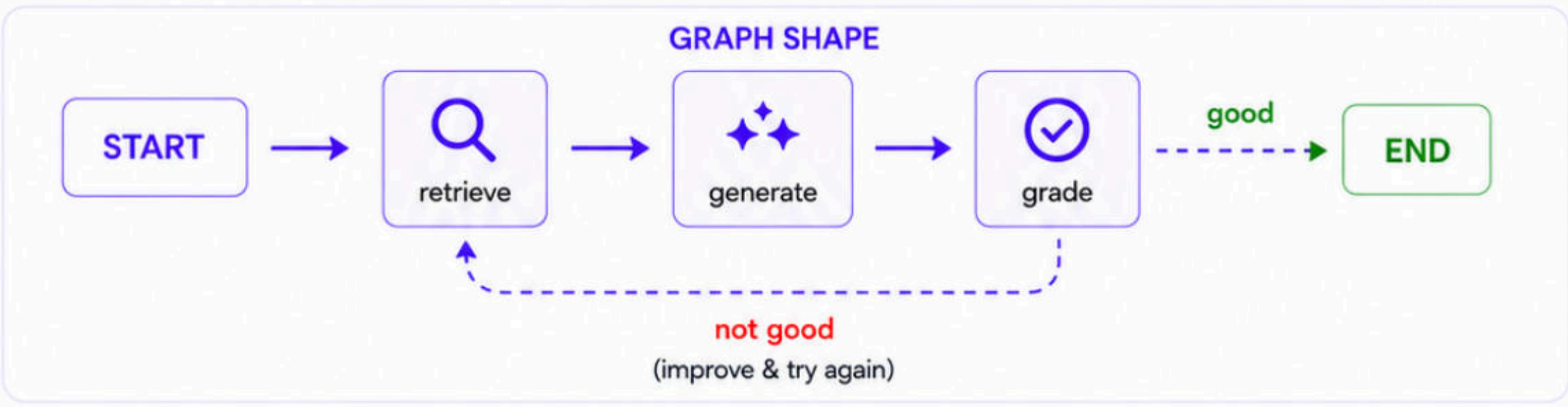
generate
Answer using context + user query.

grade
Check answer quality. Good enough?

loop
If not, go back and improve.

What's happening?

- START kicks things off.
- We move linearly: `retrieve` → `generate` → `grade`
- At `grade`, we conditionally route:
  - Good answer → `END`
  - Not good → back to `retrieve`
- The loop continues until the answer is good.



This shape — `retrieve` → `generate` → `grade` → loop — is the canonical "self-corrective RAG" pattern.

Memorize it.

Why it works

- Retrieval grounds the answer.
- Generation creates it.
- Grading verifies it.
- Looping improves it.

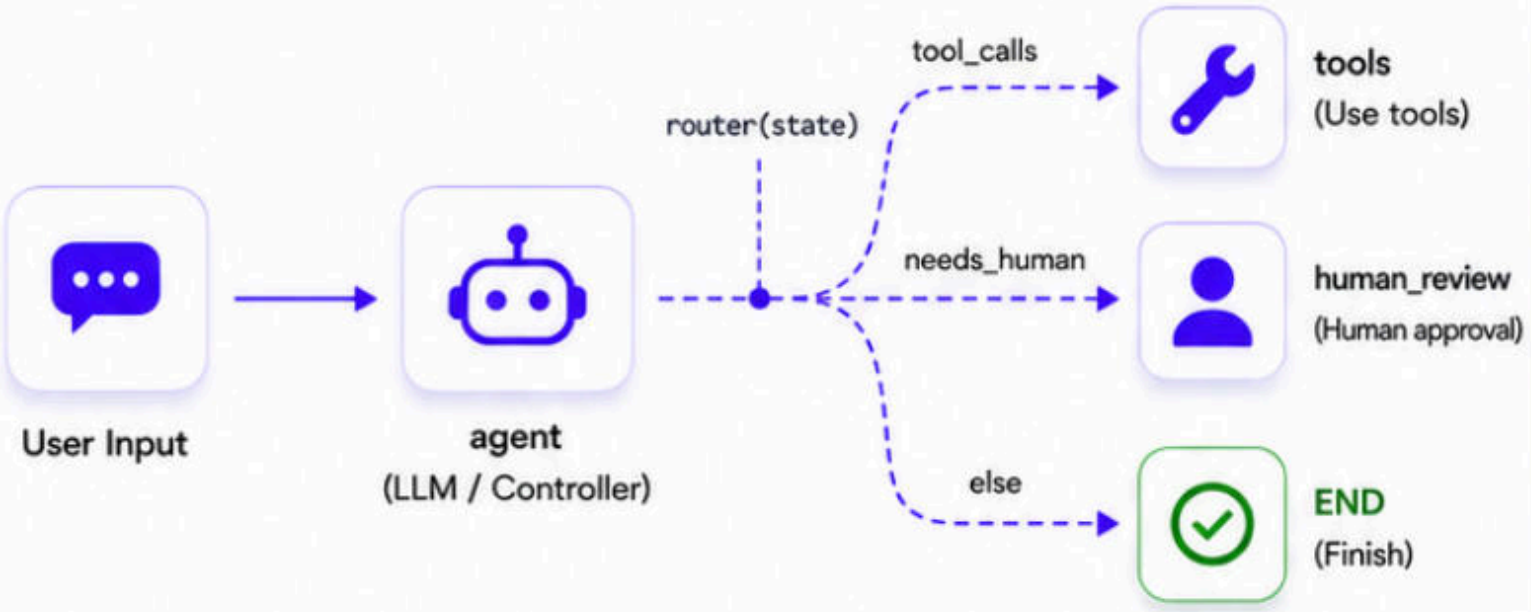
das-purnendu

Conditional Edges

How your graph decides where to go next.

```python
def router(state) -> str:
    if state["messages"][-1].tool_calls:
        return "tools"
    if state.get("needs_human"):
        return "human_review"
    return END

graph.add_conditional_edges("agent", router)
```

EXAMPLE: AGENT NODE ROUTING



How it works

1. Agent runs and updates state.
2. `router(state)` inspects state and returns a label.
3. The graph follows the edge that matches that label.

Best Practices

- Make router pure & deterministic. No side effects.
- Return ONLY known labels (or END). Be explicit.
- Keep labels small, stable, and named semantically.
- Log every decision with context. Future you will thank present you.

```python
# Pin labels explictily with a mapping
graph.add_conditional_edges(
    "agent",
    router,
    {
        "tools": "tools",
        "human_review": "human_review",
        END: END,  # default / fallback
    }
)
```

Same labels. Same truth. No surprises.

Logging example

```python
def router(state) -> str:
    ...
    label = ...  # compute label
    logger.info({
        "event": "router_decision",
        "label": label,
        "state_summary": summarize(state),
        "last_message": state["messages"][-1],
    })
    return label
```

Good router hygiene

- Pin valid labels (see below).
- Provide a default (else).
- Log inputs, decision, and label.
- Test every branch.

Routers fail SILENTLY on unknown labels.
The bug surfaces 3 nodes downstream.

Do this always:

- Pin labels with an explicit dict.
- Return only known labels or END.
- Log every decision with state context.

das-purnendu

Reducers Deep Dive

The single most misunderstood concept.

```python
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
import operator

class State(TypedDict):
    current_step: str                                  # overwrite
    messages: Annotated[list, add_messages]            # append
    findings: Annotated[list[str], operator.add]       # concat
```

By default,
returning `{"key": value}` **overwrites.**

A reducer
defines how parallel updates to the same key are **merged.**

Common Reducers

Reducer	What it does	Example
(default) overwrite	Keeps the last value	`{"k": "last"}`
operator.add	Concatenates lists	`{"k": [1, 2, 3]}`
add_messages	Appends messages with role/ID awareness	`{"messages": [...]}`
custom reducer	Any merge logic you define	`{"k": merged}`

Why reducers matter: parallel writes

No reducer (conflict)

state
findings: []

Node A: return {"findings": ["A"]}
Node B: return {"findings": ["B"]}

⚠ InvalidUpdateError
Multiple updates to key 'findings' in the same step.
Your graph crashes — it refuses to run.

LangGraph detects the conflict during the step and fails fast.

It does NOT silently overwrite.

VS.

With reducer (merge)

state
findings: []

Node A: return {"findings": ["A"]}
Node B: return {"findings": ["B"]}

State after merge
findings: ["A", "B"] ✓
(Both preserved)

Reducer merges both updates.
Deterministic.
Correct.

#1 reducer mistake in LangGraph:
parallel branches updating the same key without a reducer.
LangGraph detects the conflict and crashes with InvalidUpdateError.

Reducers are mandatory for parallel state writes.
No reducer → your graph crashes loudly.

Pro tip | Think of reducers as the merge strategy that makes parallel updates deterministic. | No reducer = invalid update. | Reducer = deterministic merge.

Checkpointers

Persist graph state at every step.

Without persistence, your graph has no memory and no future.

```python
from langgraph.checkpoint.postgres import PostgresSaver

with PostgresSaver.from_conn_string("postgresql://...") as pg:
    pg.setup()
    app = graph.compile(checkpointer=pg)

config = {"configurable": {"thread_id": "user-42"}}

app.invoke({"messages": [...]}, config)

# Future calls with same thread_id = full continuity
```

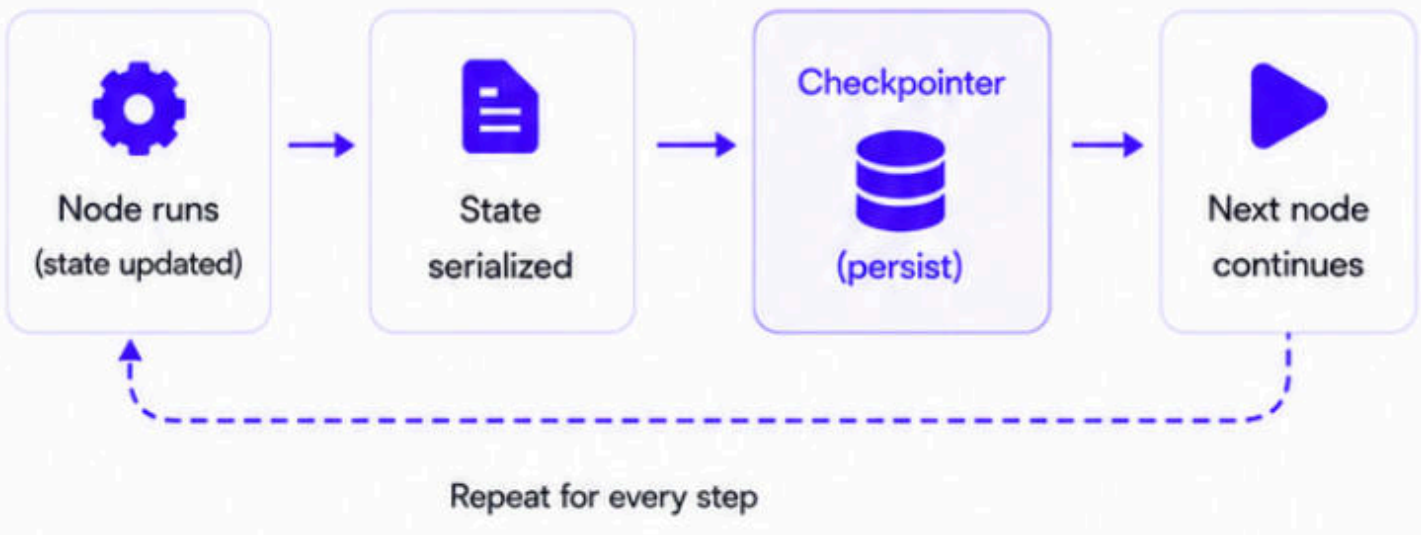
What is a checkpointer?

A checkpointer saves the entire graph state after every step (node execution).

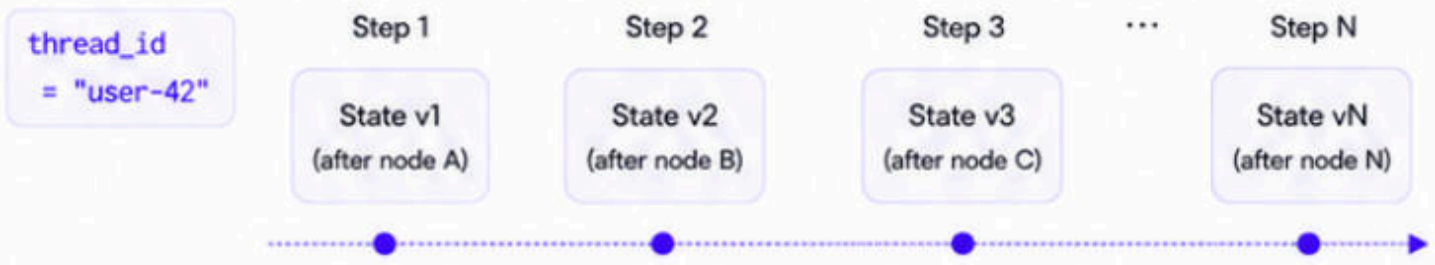
Why it matters

- Survive crashes & restarts
- Resume conversations seamlessly
- Enable HITL (Human-in-the-Loop)
- Time-travel debugging
- Audit & observability

How it works



Thread continuity (the magic)



Any future call with the same `thread_id`

- Loads latest state
- Continues exactly where it left off
- Full memory, full context, zero hacks

Example flow

PostgresSaver advantages

- Durable & ACID
- Multi-process / multi-pod safe
- Scalable
- Queryable & inspectable
- Production-grade

MemorySaver in production = guaranteed incident on the first pod restart.

PostgresSaver. Always.

Why?

- MemorySaver = in-memory only
- Lost on crash, restart, deploy
- You WILL lose user state

Pro tip: Use a meaningful thread_id (e.g., user_id, session_id, or conversation_id). It's the key that unlocks continuity.

Think long-term: Your users will.

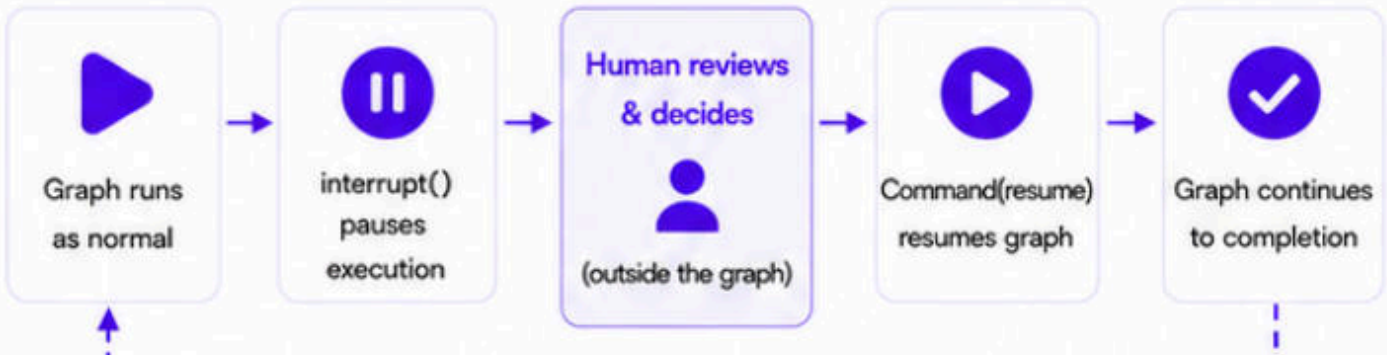
das-purnendu

Human-in-the-Loop

Pause the graph.
Wait. Resume with human input.

```python
from langgraph.types import interrupt, Command

def request_approval(state):
    decision = interrupt({
        "action": "refund",
        "amount": state["amount"],
    })
    return {"approved": decision["approved"]}

# Later, when human responds:
app.invoke(Command(resume={"approved": True}), config)
```

How it works



Example: Refund approval

- Agent decides a refund is needed (amount = $250).
- interrupt() sends the request to a human.
- Human approves or rejects.
- Command(resume=...) sends the response back.
- Graph resumes exactly where it left off.

What interrupt() does

- Saves the current state (requires a checkpoint)
- Pauses node execution and returns control
- Waits for an external human/system response
- Resumes from the exact point with Command

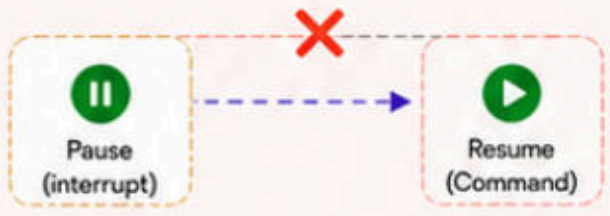
What Command(resume=...) does

- Delivers the human response back into the graph
- Unblocks the paused node
- Graph continues with full state & context

Non-negotiable: a checkpointer.
Without persistence,
there's nothing to resume from.

Without a checkpointer:

- The process is lost on crash/restart
- The human response has nowhere to go
- You cannot resume. You have to start over.



The evolution (don't mix them up)

2024 and earlier

interrupt_before

Break before a node runs
- Less flexible
- Coarse-grained control

2026 and beyond (recommended)

interrupt() + Command

- Pause anywhere in a node
- Pass rich data to humans
- Resume precisely with full control

Best practices

- Always use a persistent checkpointer (e.g., PostgresSaver).
- Design clear human tasks (what, why, options).
- Validate human input on resume.
- Timeouts & escalation for stale tasks.
- Log every interrupt and resume.

Pro tip Model human tasks as data, not code.
Make it observable, auditable, and deterministic.

Who	What	Why	Options	Deadline
Approver (User/Role)	Action & Details	Reason & Context	Approve / Reject / Request changes	SLA for response

das-purnendu

Subgraphs

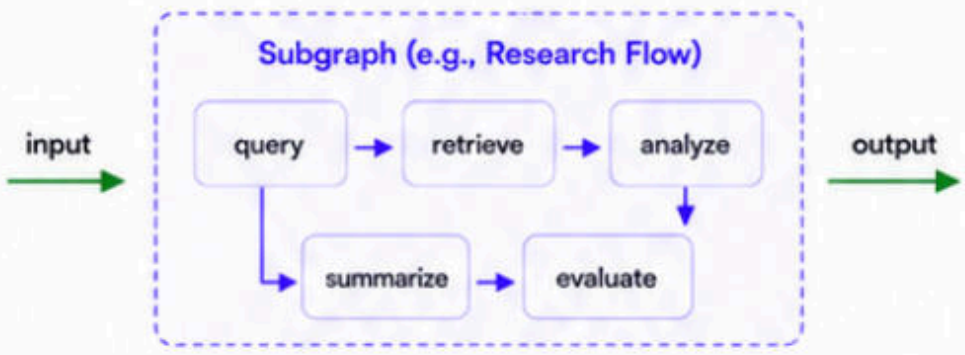
Compose complex graphs from smaller ones.

A compiled graph used as a node in a parent graph.

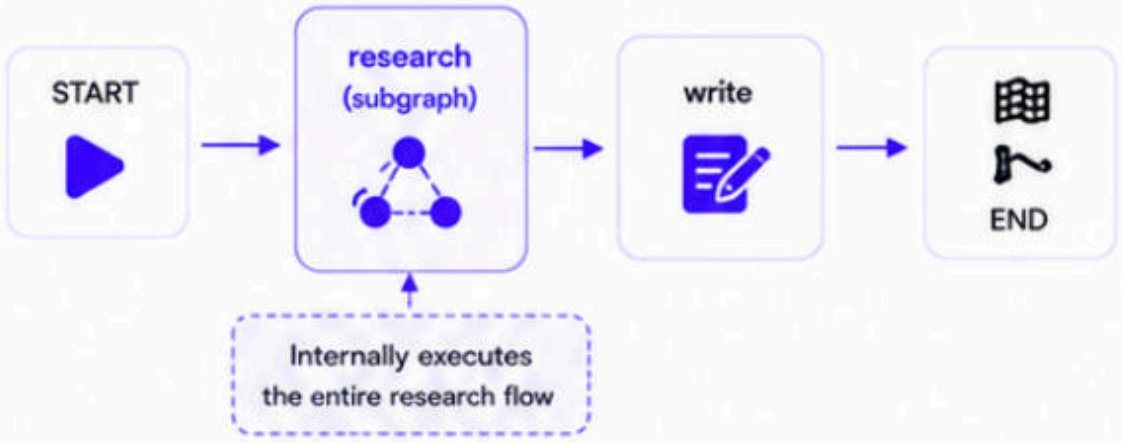
```python
# Compile a subgraph
research_app = research_graph.compile()

# Use the compiled subgraph as a node in a parent graph
main_graph.add_node("research", research_app)
main_graph.add_node("write", write_node)

main_graph.add_edge(START, "research")
main_graph.add_edge("research", "write")
main_graph.add_edge("write", END)

app = main_graph.compile(checkpointer=pg)
```

What is a subgraph?

A subgraph is a complete, compiled graph that behaves like a single node in a larger (parent) graph.



Used as a node in Parent Graph



When to use subgraphs?

Domain has its own state machine
Encapsulate complexity with its own flow and state.

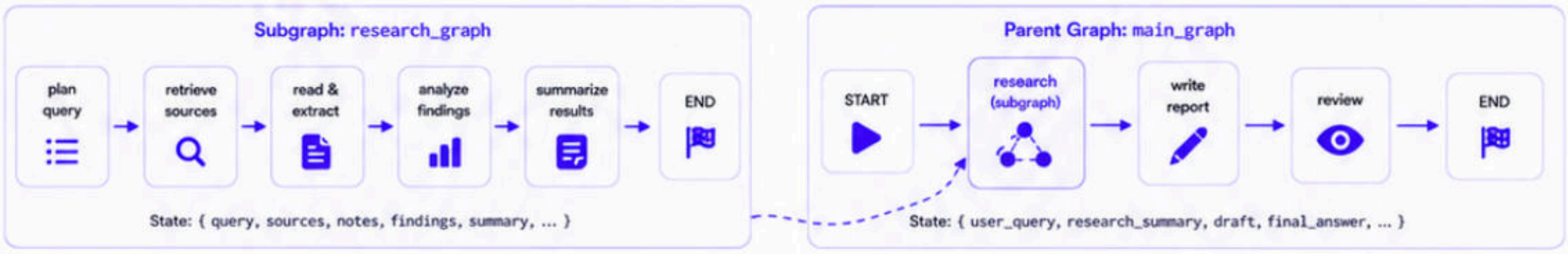
Reuse a flow
Build once, use many times across different parts of your system.

Supervisor with workers
Orchestrate worker subgraphs (specialists) from a supervisor graph.

Separation of concerns
Keep graphs small, focused and easier to reason about.

If your nodes.py is 800 lines and you're not using subgraphs — that's the answer.

Example: Research Assistant



Benefits

- **Modularity** — Smaller pieces are easier to build, test and maintain.
- **Reusability** — Plug the same subgraph into multiple parents.
- **Testability** — Test subgraphs in isolation with their own inputs/outputs.
- **Scalability** — Build libraries of subgraphs and compose at will.
- **Team velocity** — Different teams can own different subgraphs independently.

Pro tip

Design clean inputs & outputs for your subgraphs.
Treat them like functions: clear contract, no surprises.

Common anti-patterns

- Putting everything in one giant graph
- Deeply nesting subgraphs without reason
- Leaky state between parent and child
- Subgraphs with unclear input/output contract

Golden rule

Use subgraphs to manage complexity.
Compose, don't concatenate.

Send API

Map-reduce inside a graph.

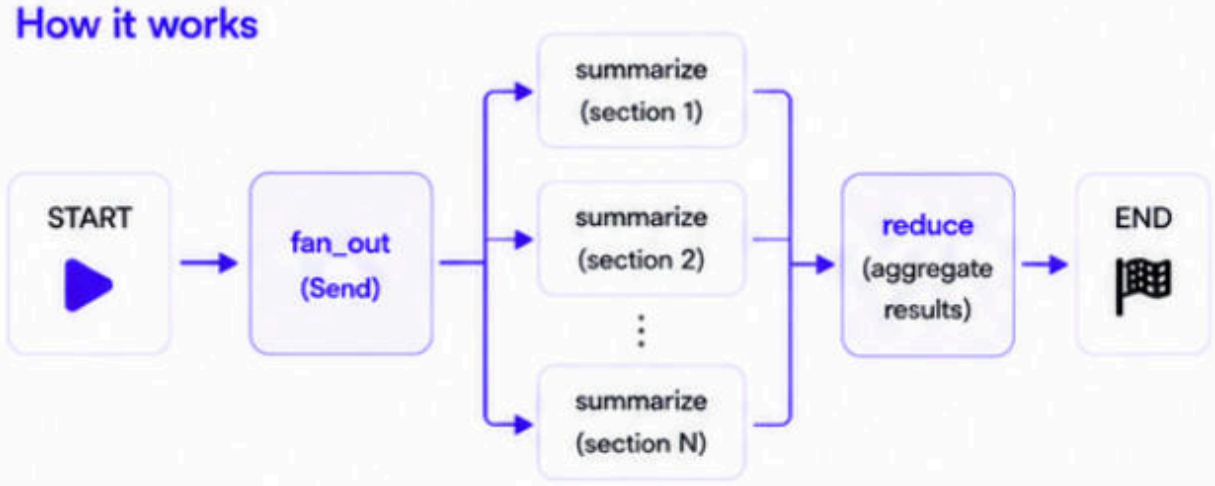
Dispatches N parallel invocations, each with its own state slice.

Three parallel sub-agents in 3s beats one sequential agent in 9s.

The whole point of a graph runtime.

```python
from langgraph.types import Send

def fan_out(state):
    return [
        Send("summarize", {"section": s})
        for s in state["sections"]
    ]

# Each Send creates a parallel child invocation
# that runs the 'summarize' node with its own state slice
```

How it works



1 → N (parallel) → 1

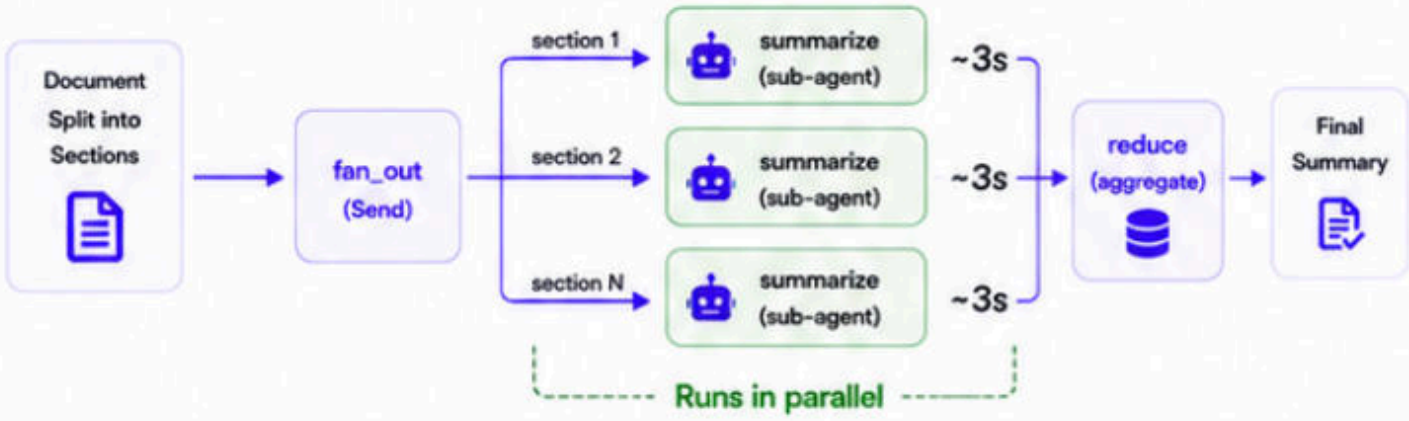
What actually happens

1. fan_out returns a list of Send objects.
2. The graph runtime spawns N parallel invocations.
3. Each invocation runs the target node ("summarize") with its own state slice.
4. All results are returned to the parent.
5. A reducer merges them into a single state.

Sequential (N=3) ~ 9s (3 × 3s) >>> Parallel with Send (N=3) ~ 3s (max of 3s)

Example: Summarize a long document

```python
def fan_out(state):
    return [Send("summarize", {"section": s})
        for s in state["sections"]]

def reduce(state):
    # 'summaries' is collected from all parallel results
    combined = "\n\n".join(state["summaries"])
    return {"final_summary": combined}
```



Why Send is powerful

- True parallelism inside the graph runtime
- Clean, declarative map step
- Works with any node (LLM, tool, subgraph, etc.)
- Scales to hundreds of parallel tasks
- Perfect for RAG chunks, sections, items, candidates, ...

Common use cases

- Chunked document summarization
- Parallel RAG retrieval & reranking
- Batch tool calls / API fan-out
- Multi-candidate evaluation
- Data enrichment pipelines

Reducers are non-negotiable

Parallel branches produce multiple values for the same key.
You MUST define how to merge them.

```python
class State(TypedDict):
    summaries: Annotated[list[str], operator.add]
    # or use add_messages for chat messages
    # or custom reducer
```

Without Send, candidates write Python for-loops and silently kill parallelism.

- For-loops = sequential execution
- Wastes latency, compute, and budget
- Harder to observe and reason about

Always use Send + a reducer.

Pro tip — Send does not block the parent node. The runtime handles orchestration, scheduling, and result collection for you.

Return Send objects → Runtime spawns tasks → Results flow back → Reducer merges into state → Graph continues

das-purnendu

Streaming Modes

4 modes. Pick the right combo for your UI.

```python
# Stream with multiple modes
for evt in app.stream(
    input_data,
    stream_mode=["messages", "updates"]
):
    handle(evt)

# Each evt is a dict with a 'type' key
# and mode-specific payload
```

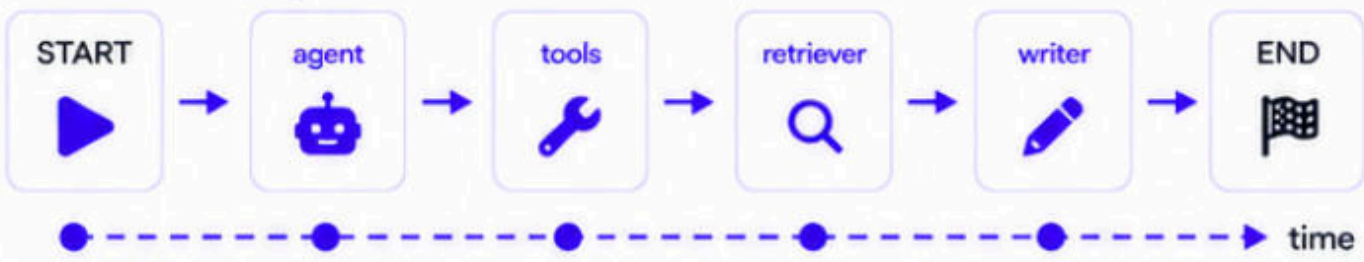
values
Full state after each step.
Great for dashboards, time-travel and debugging state.

updates
Diff per node (which fired, what changed).
Perfect for UI status & progress.

messages
Token-by-token LLM output.
Streams the assistant's words as they are generated.

debug
Everything, very loud.
All events, internals, timings, state, errors, metadata.

How streaming works

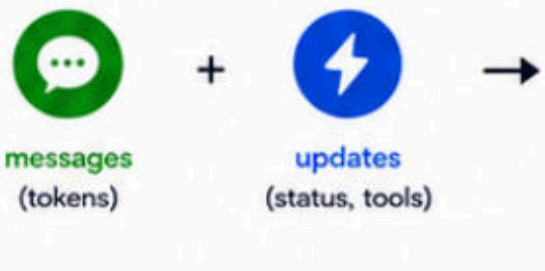


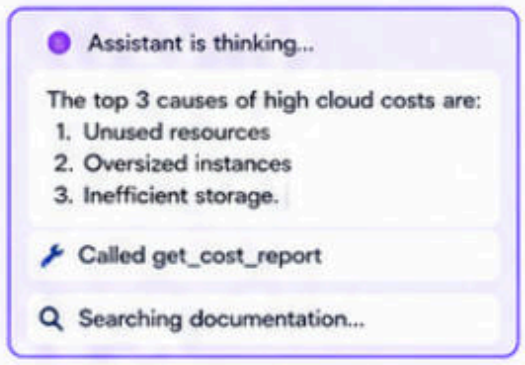
After each step completes,
the runtime emits events according to the selected modes.
Your UI consumes the stream in real time.

What each mode emits (example)

Mode	Emits	Example (abridged)
values	Full state snapshot after each step	`{"type": "values", "step": "agent", "state": {"messages": [...], "facts": [...], "needs_human": false}}`
updates	Diff: which node fired and what changed	`{"type": "updates", "node": "tools", "changes": {"tools_output": {...}}}`
messages	Token-by-token LLM output	`{"type": "messages", "node": "agent", "token": "Hel", "is_final": false}`
debug	Everything (very verbose)	`{"type": "debug", "event": "node_start", "node": "retriever", "ts": 171343.21, ...}`

For real chat UIs





Smooth tokens + rich status = great UX

Other useful combos

- values + updates → Observability & dashboards
- updates + debug → Troubleshooting & dev tools
- messages only → Minimal chat (tokens only)

```python
for evt in app.stream(input, stream_mode=["messages", "updates"]):
    ...
```

← *Most common*

Streaming mode is a UX decision, not a backend one.

- Don't default to *debug* in production.
- Don't stream values to a chat UI.
- Pick the minimal combo that fits the UI.

Pro tips
- Use *messages* for token streaming.
- Use *updates* for tool calls, progress, errors.
- Turn on *debug* only when you need it.
- Keep the UI responsive and incremental.

Mental model
The graph runs step by step.
You choose what the UI sees, and how chatty it should be.

Graph executes steps → Runtime emits events → Streaming mode filters events → UI renders in real time

das-purnendu

Multi-Agent Supervisor

In 2026, "multi-agent"
= **supervisor pattern**.
Not agents-talk-to-agents chaos.

```python
g.add_node("supervisor", supervisor)
g.add_node("researcher", researcher_subgraph)
g.add_node("writer", writer_subgraph)

g.add_conditional_edges("supervisor", route)
for w in ["researcher", "writer"]:
    g.add_edge(w, "supervisor")
```

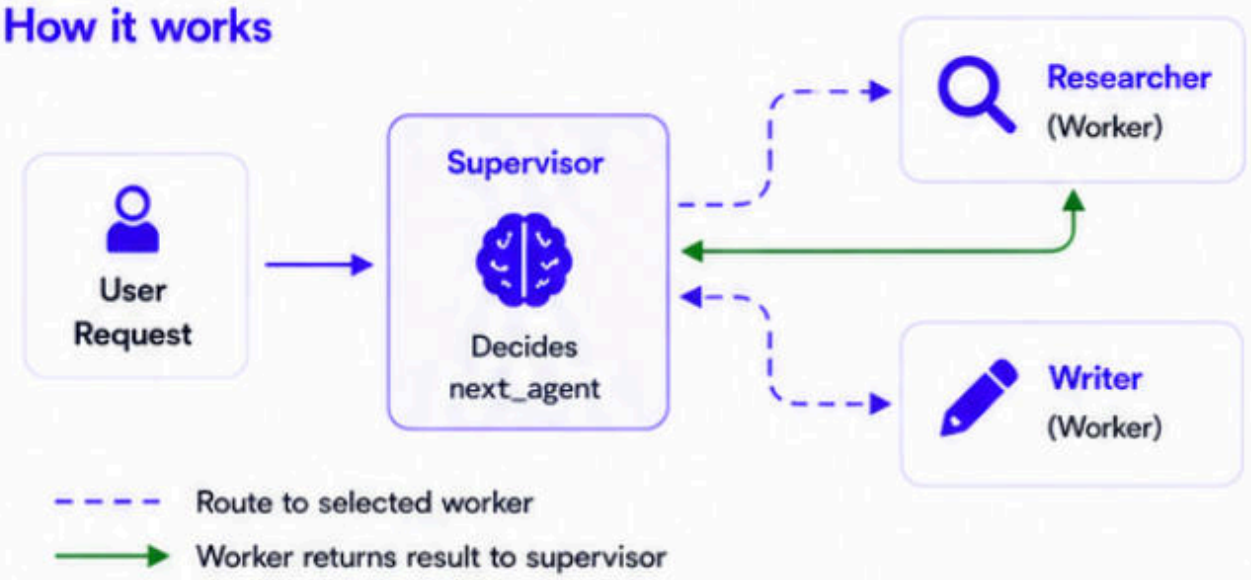
What is the supervisor pattern?

A supervisor node decides which specialized agent (worker) should act next based on the current state and goals.

- Supervisor = brain / router
- Workers = specialists that do the work
- Workers return control to supervisor when done

This creates a **controlled loop**, not agent-to-agent chaos.

How it works

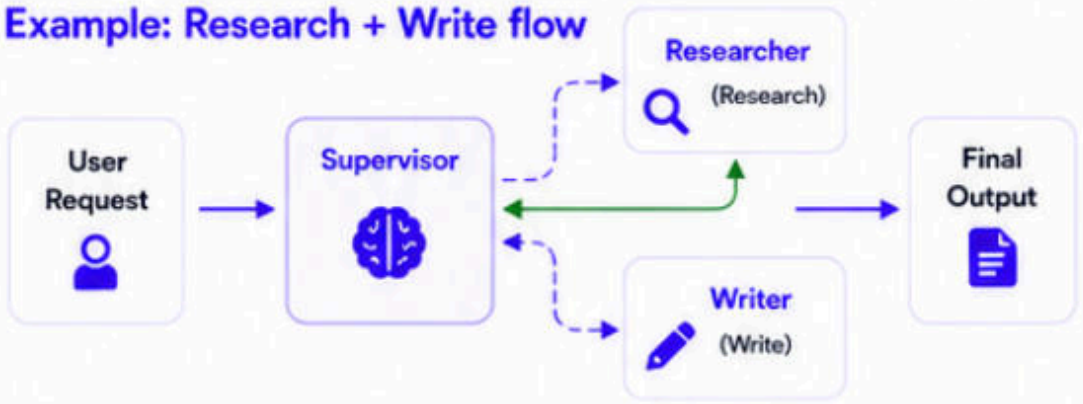


Repeat until the supervisor decides the task is complete or `max_iterations` is reached.

Supervisor loop (high level)

1. Supervisor looks at state, history, and goal.
2. Chooses next agent (e.g., researcher).
3. Routes to that worker (subgraph).
4. Worker does work and returns result + updates.
5. Control returns to supervisor. Loop.
6. Stop when done or `max_iterations` hit.

Example: Research + Write flow



Possible sequence
Supervisor → Researcher → Supervisor → Writer → Supervisor → Done

Production must-haves

Workers are subgraphs
Encapsulated state machines. Reusable, testable, isolated.

Hard max_iterations cap
Prevent infinite loops and runaway costs.

Every routing decision logged
Log: `current_state_summary`, `chosen_next_agent`, `reason`, `iteration`, `timestamp`.

Deterministic routing logic
No randomness in routing. Makes debugging and evaluation possible.

What NOT to do

- Agents calling each other directly.
- Unbounded loops with no iteration limit.
- Hidden routing logic with no logs.
- Workers as plain functions (hard to reuse and test).

Do this instead

Use a supervisor.
Route explicitly.
Loop intentionally.
Log everything.

Great work! You now know the core.

Build with confidence

Ship to production

Iterate and improve

das-purnendu

PART 3: Production & Interview Prep

The 3 a.m. on-call edition.

- **Ship it. Monitor it. Don't get paged.**
- **Real-world patterns that survive 3 a.m.**
- **Interview questions that actually get asked**
- **Anti-patterns that kill production systems**

What's coming in Part 3

Production Hardening
What to ship & what to avoid

Observability & Monitoring
Logs, metrics, traces

Reliability Patterns
Retries, timeouts, circuit breakers

Interview Prep
System design + code Qs

On-call Playbook
3 a.m. survival guide

"The difference between a toy graph and a production system is everything that **breaks at 3 a.m.**"

Final swipes →

das-purnendu

MCP Integration

The 2026 question almost no candidate gets right.

```python
from langchain_mcp_adapters.client import MultiServerMCPClient

client = MultiServerMCPClient({
  "github": {
     "command": "npx",
     "args": ["-y", "@modelcontextprotocol/server-github"],
  }
})

tools = await client.get_tools()
agent = create_react_agent("openai:gpt-4o-mini", tools)
```

Note (2026): The official GitHub MCP server has moved. Development is now at github.com/github/github-mcp-server. The package @modelcontextprotocol/server-github is deprecated but still works via npx (may be removed in the future).

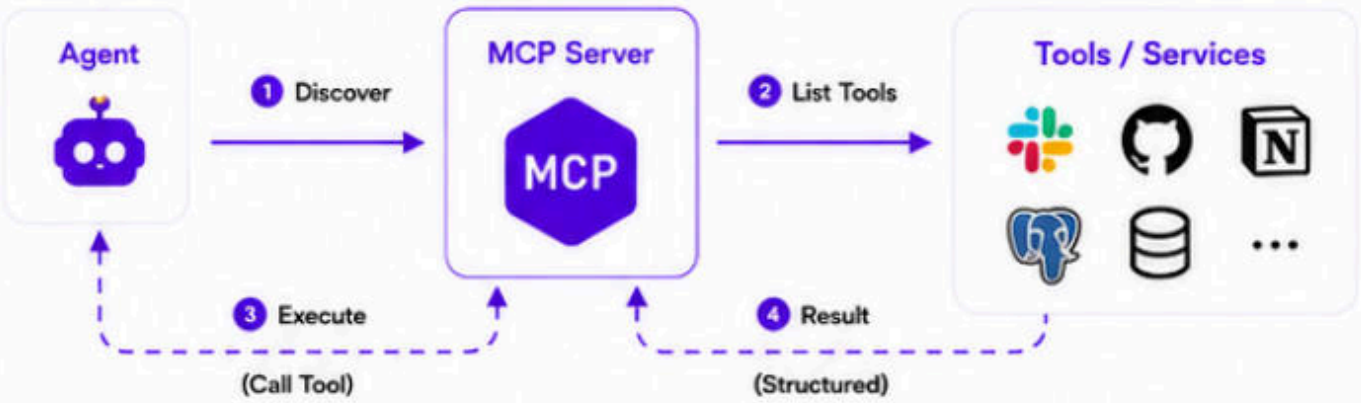
MCP = Model Context Protocol. "USB port for agents."

Standardizes how agents discover and call tools — Slack, GitHub, Postgres, Notion, anything.

Why it matters

- One protocol. Unlimited tools.
- Tools become first-class, discoverable capabilities.
- No more custom glue for every service.

How MCP works



MCP standardizes discovery, invocation, auth, and results.
Plug any service. Swap any implementation.

Real-world example

- Ask agent: "Create a PR for the bug you just analyzed"
- Agent discovers GitHub tool via MCP
- Calls create_pull_request via MCP
- Gets structured result back
- Continues the plan

Without MCP

- Custom integrations per tool
- Different auth, formats, errors
- Hard to scale. Hard to maintain

MCP benefits

Plug-and-play
Add new tools in minutes.

Secure by design
Auth, scopes, auditing built-in.

Standard interface
Consistent I/O across tools.

Scales infinitely
One protocol for all your tools.

Ecosystem growing
More servers. More power.

Bonus (rare signal):

A2A Protocol for agent-to-agent

Agents need to talk to other agents, not just tools.

- A2A = Agent-to-Agent protocol (emerging standard)
- Enables discovery, capabilities exchange, and task delegation
- Think "microservices for agents"
- Almost no one mentions this. You should.

Interview tip

Name MCP + one real tool (e.g., GitHub, Postgres) + mention A2A.

What they want to hear

"I don't build custom integrations. I use standards that scale."

Ship faster. Integrate once. Scale forever.

Production is about leverage.

das-purnendu

LangSmith & Debugging

Stop guessing. Start **tracing.**

```python
import os
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_API_KEY"] = "..."

# That's it. Every chain.invoke() is now traced.
```

What you get

Every step traced
See the full execution graph: inputs, outputs, latency, tokens, errors.

Replay any past run
Re-run exactly what happened with the same inputs & model versions.

Curate datasets
Turn real prod traffic into datasets for evals and regressions.

Run automated evals
LLM-as-judge, exact match, embedding similarity, custom evaluators.

Track quality over time
Dashboards, eval trends, latency, cost, token usage, success rate.

Trace everything

LangSmith

- Projects
- Traces
- Runs
- Datasets
- Evaluations
- Playground
- Monitor
- Settings

Traces › research_agent_run_7f3a... SUCCESS 3.21s 1,243 tokens $0.0123

Trace | Timeline | LLM Calls | Logs | Metadata | Scores

research_agent	3.21s
plan	0.21s
search_web	1.02s
Tavily Search	1.02s
extract	0.48s
summarize	0.67s
gpt-4o-mini	0.67s
final_answer	0.39s

Input | Output | Metadata | Logs

```
{
  "query": "Latest LangChain releases in 2026",
  "results": [
    { "title": "LangChain v1.0.5 Overview",
      "url": "..." ,
    { "title": "LangGraph v1.0.2 Deep Dive",
      "url": "..."
  ],
  "answer": "LangChain v1.0.5 introduced ..."
}
```

Run Details

Run ID
7f3a9e20-82e1-...

Start Time
May 12, 2026 10:42:11

Model
openai:gpt-4o-mini

Total Tokens
1,243

Total Cost
$0.0123

Latency

✓ Success

Debug smarter, not harder

See where it failed
Errors, timeouts, bad outputs.

Inspect latency
Find slow nodes in seconds.

Add tags & metadata
Version prompts, models, experiments.

Slice & filter
By user, model, route, error type.

Export & share
Share traces, runs, or datasets.

Red flag in interviews

Candidates who say

> "I add print statements"

have never been on-call for an agent.

> Without distributed tracing, debugging is **guessing.**
> The bug is always 2 nodes deeper than you think.

Pro tips

- Use meaningful run names.
- Log custom metadata.
- Track cost per run.
- Set alerts on error rate spikes.
- Build evals early, not at the end.

The goal
Make production issues reproducible, observable, and fixable.

Ship with confidence.
Sleep through the night.

das-purnendu

Common Production Bugs

The bugs that cost real engineers real hours:

1 

Parallel writes without reducers
→ **InvalidUpdateError (graph crashes)**

LangGraph detects conflicting writes to the same key in the same step and **refuses to run.**

Why it hurts
Your graph crashes loudly. Nothing runs. No results.

2 

MemorySaver in prod
→ **state dies on restart**

Process restarts, deploys, crashes = bye bye state.

Why it hurts
Users lose conversations. Support tickets spike.

3 

No max_iterations
→ **supervisor loops, burns tokens**

One bad route and it loops forever.

Why it hurts
Runaway costs. *Rate limits.* Angry users.

4 

Embedding model swap, no re-index
→ **garbage retrieval**

Old vectors + new model = irrelevant results.

Why it hurts
Bad answers. You chase phantom "prompt" issues.

5 

Router returning unmapped label
→ **wrong path, silent**

Falls back to default or None. You won't notice.

Why it hurts
Wrong agent runs. Hard to reproduce.

6 

Tool docstrings vague
→ **model picks wrong tool**

LLM guesses. Guessing is not a strategy.

Why it hurts
Wrong actions. Side effects. Hard to debug.

7 

Mixed query/doc embedding models
→ **noise**

Different spaces. Similarity becomes garbage.

Why it hurts
Retrieval quality tanks silently.

Save this slide.

You will hit at least 3 of these.
Now you know what to fix first.

Pro tip

- Add tests for routing + tools
- Set max_iterations on every loop
- Monitor token usage & error rate
- Log everything. Trust nothing.

Bugs hide in the edges. Observability finds them.

das-purnendu

RAG vs Fine-tuning vs Prompting

The decision framework interviewers want:

Prompting

Use the model's existing knowledge.

- Knowledge fits in context window
- Behavior can be controlled by instructions/examples
- Fastest to try and iterate
- Lowest cost & complexity

Start here.
Most problems end here.

RAG

Retrieve external knowledge at runtime.

- Knowledge is large or dynamic
- Data changes frequently
- Proprietary / private knowledge
- Need citations & traceability
- Enterprise default

Default for enterprise.
Scales knowledge, not parameters.

Fine-tuning

Adapt the model's weights.

- Need a specific style, tone, or format consistently
- Behavior is not achievable reliably via prompting
- High volume + consistent patterns
- Expensive, slower, harder to iterate
- Risk of overfitting

Rarely the right first answer.
Use only when proven necessary.

How to decide (in order)

Can I solve this with prompting?
- Knowledge fits in context?
- Behavior controllable via instructions/examples?

YES → **Use Prompting**
Fastest, cheapest, simplest.

NO → **Is the knowledge large, dynamic, or proprietary?**
- Can't fit in context
- Changes over time

YES → **Use RAG**
Retrieve, ground, cite, repeat.

NO → **Do I need a specific style, format, or behavior the model can't reliably follow?**

YES → **Consider Fine-tuning**
Escalate only when prompting + RAG aren't enough.

→ **Re-evaluate the problem**
You might be solving the wrong problem.

Golden order
prompting → RAG → fine-tuning
Escalate only when needed.

Red flag in interviews

"I'd fine-tune."
as the first answer = junior.

The order is almost always:
prompting → RAG → fine-tuning,
in that order, only escalating when needed.

Pro tips

Measure first.
Use evals to validate before escalating.

RAG quality > model size
Chunking, retrieval, and reranking matter more.

Keep it simple
Simple pipelines are easier to debug & scale.

Cost matters
Prompting < RAG << Fine-tuning.

das-purnendu

Rapid-Fire Interview Qs (1)

High-signal answers. Short, correct, memorable.

1

Q: When LangGraph over LangChain agent?

A: Cycles, branches, durable state, HITL, multi-agent.

2

Q: What's a reducer?

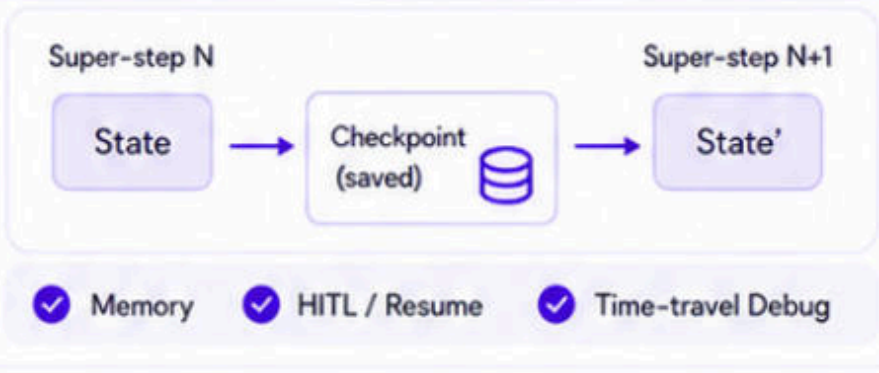
A: How state updates merge.
Default = overwrite.
add_messages = append.

```python
# Default behavior
state["messages"] = new_messages   # overwrite

# With reducer
def add_messages(left, right):
    return left + right            # append
```

3

Q: How do checkpointers work?

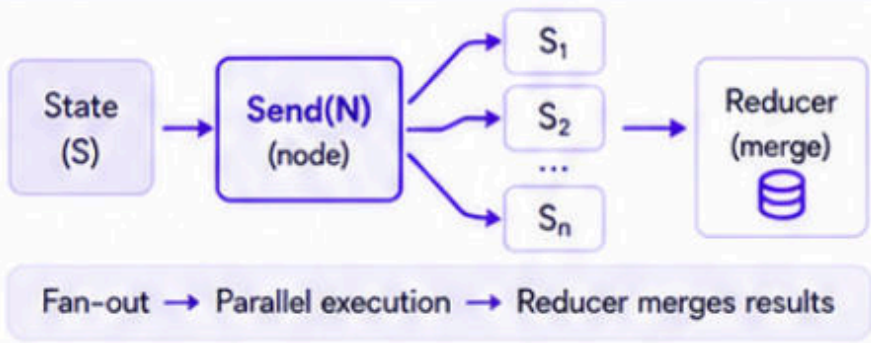
A: Persist state at every super-step.
Enables memory, HITL, time-travel.



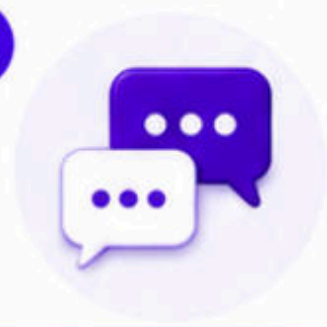
4

Q: Send API?

A: Dispatch N parallel invocations of a node, each with its own state slice.
Pair with reducer.



5



Q: Streaming for chat UIs?

A: Combine messages + updates.

```python
async for event in graph.astream(
    inputs,
    stream_mode=["messages", "updates"]
):
    if event["type"] == "messages":
        render_token(event["content"])
    elif event["type"] == "updates":
        update_status(event["data"])  # <-- correct
```

Interview tip

Answer in one line first. Then add one crisp detail or example.

das-purnendu

Rapid-Fire Interview Qs (2)

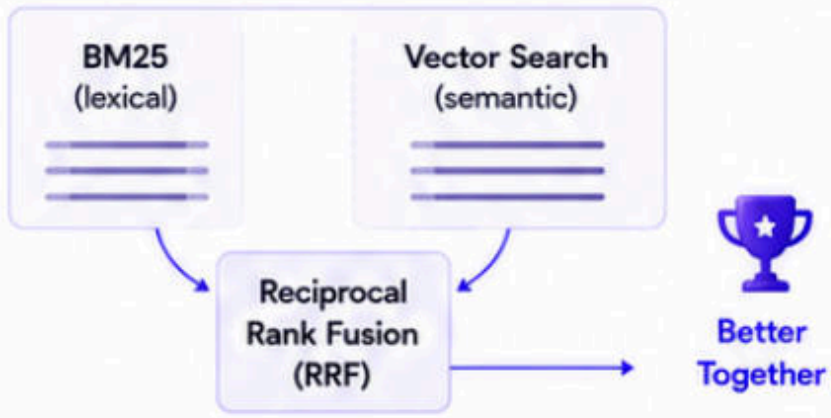
High-signal answers. Short, correct, memorable.

1

Q: Hybrid retrieval?

A: BM25 + vectors → reciprocal rank fusion. Beats either alone.

Higher Recall | Higher Precision



2

Q: How to debug an agent in prod?

A: LangSmith. Trace, replay, evaluate. Print statements don't scale.

3

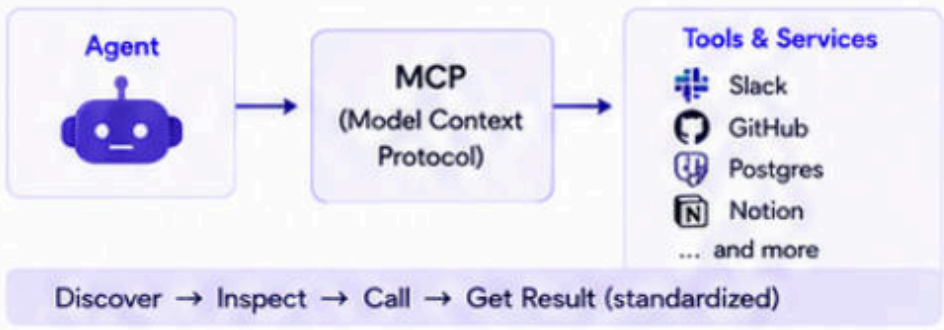
Q: Why with_structured_output?

A: Native function-calling. More reliable than parsing JSON from text.

4

Q: What's MCP?

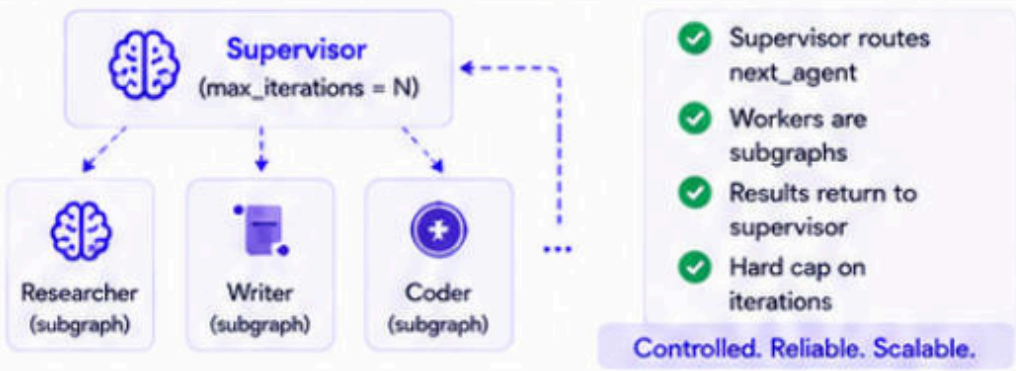
A: Standard protocol for agents to discover and call tools.



5

Q: First multi-agent pattern to reach for?

A: Supervisor with subgraph workers + max_iterations cap.



Interview tip

Structure your answers: one line → why → how → trade-offs.
Be precise. Be confident. Be memorable.

das-purnendu

System Design:
Customer Support Agent

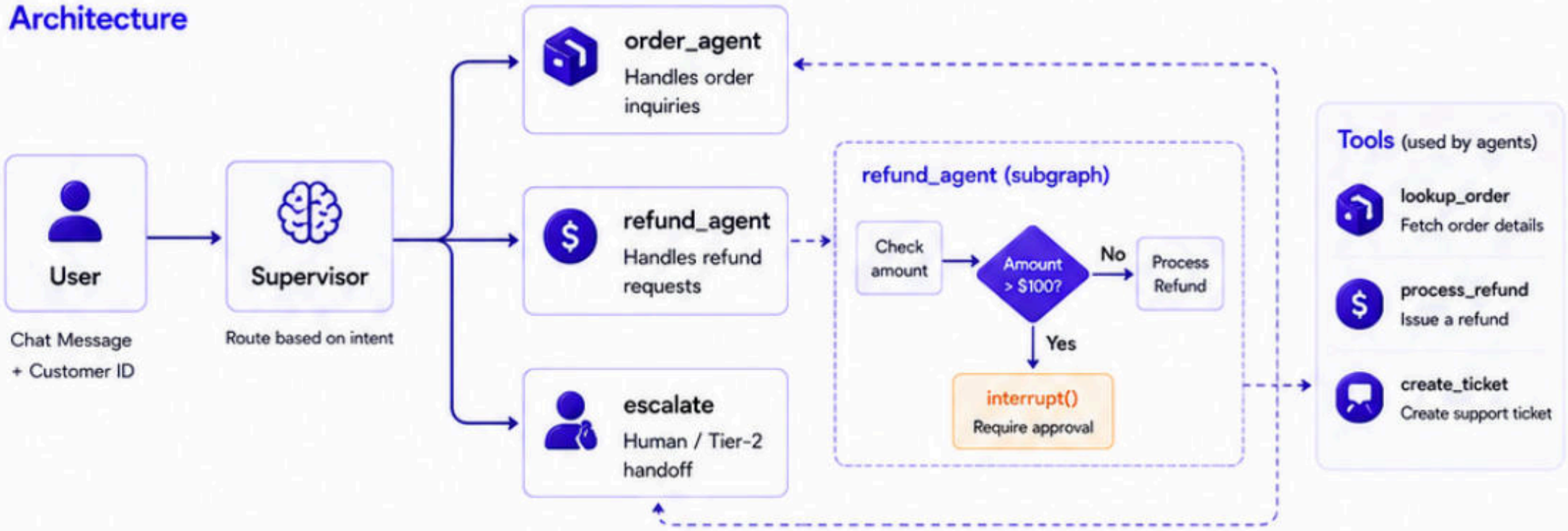
Intelligent, safe, and observable support — from chat to resolution.

Inputs
- chat message
- customer ID

Tools
- lookup_order
- process_refund
- create_ticket

Goals
- Resolve issues accurately
- Escalate when needed
- Keep customers informed

Architecture



State Management

State
(per conversation)
Reducers

messages	add_messages	(append-only)
customer	latest	(overwrite)
intent	latest	(overwrite)
action_log	list reducer	(append)

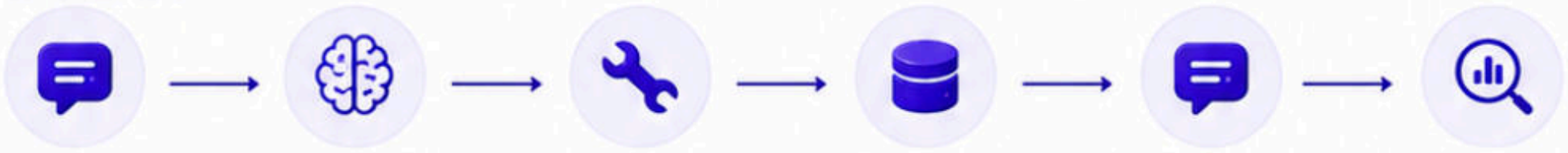
Checkpointer
PostgresSaver
thread_id = conversation_id
(enables resume & replay)

Observability
LangSmith
- Tag every run with customer_id
- Trace agents, tools, LLM calls
- Monitor latency, errors, retries

Guards & Safety
- max_iterations = 10
- Output validator (schema + rules)
- PII redaction (inputs & outputs)
- Toxicity check & safe fallbacks

Quality Signals
- Resolution rate
- Escalation rate
- CSAT
- Time to first response

End-to-End Flow



das-purnendu

Red Flags

What interviewers watch for —
and what you should do instead.

Red Flags Interviewers Watch For

Uses LLMChain and OpenAI() (legacy)
Shows outdated knowledge. Use LangChain Expression Language + ChatOpenAI.

Names Pinecone without asking about scale
Vector DB choice depends on scale, latency, freshness, and cost. Ask first.

No mention of reducers in multi-agent design
State bugs, message loss, and race conditions await.

"I'd fine-tune" as first answer to a knowledge problem
RAG or routing usually beats fine-tuning.

No checkpointer in HITL design
You can't recover, resume, or audit without it.

Vague on when to use LangGraph vs LangChain agent
Different tools. Different guarantees. Know when to pick which.

Save This One

If you save one slide, make it #23 (reducers).

That bug has cost more engineers more hours than any other.

From Theory to Production

I write production GenAI weekly — RAG, multi-agent, LLMOps.

Follow Purnendu Das for more battle-tested content.

Real systems. Real lessons. No fluff.

Let's Learn Together.

Comment the question that stumped you most.
I'll deep-dive the top one.

Purnendu Das
das-purnendu

Save | Like | Comment | Share