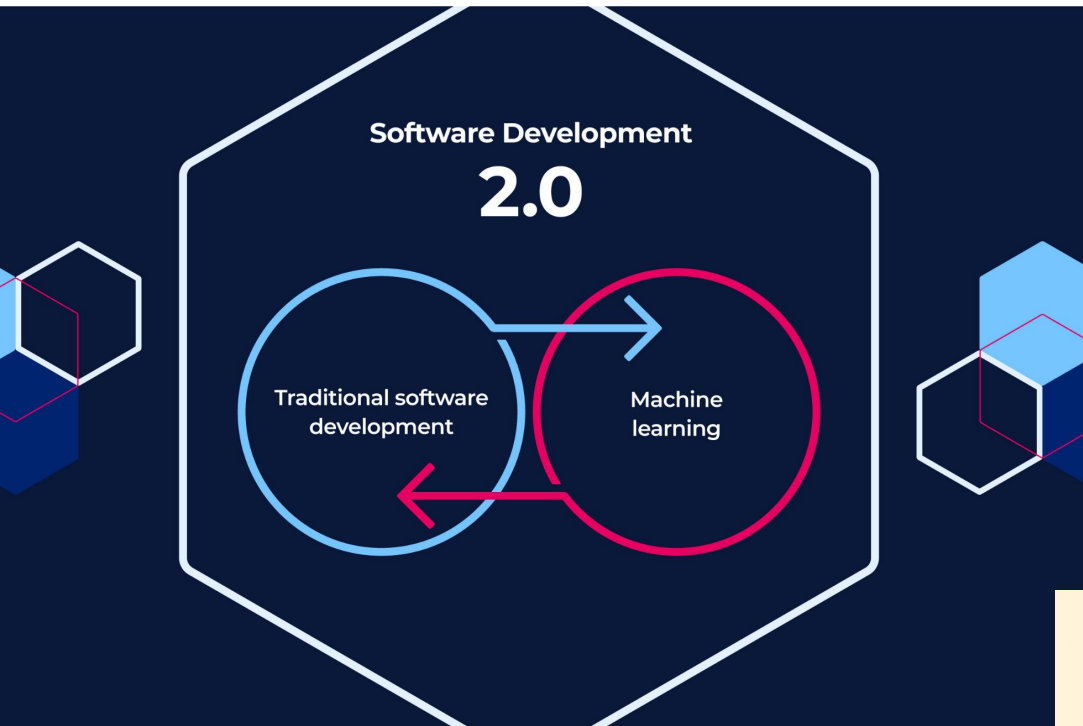
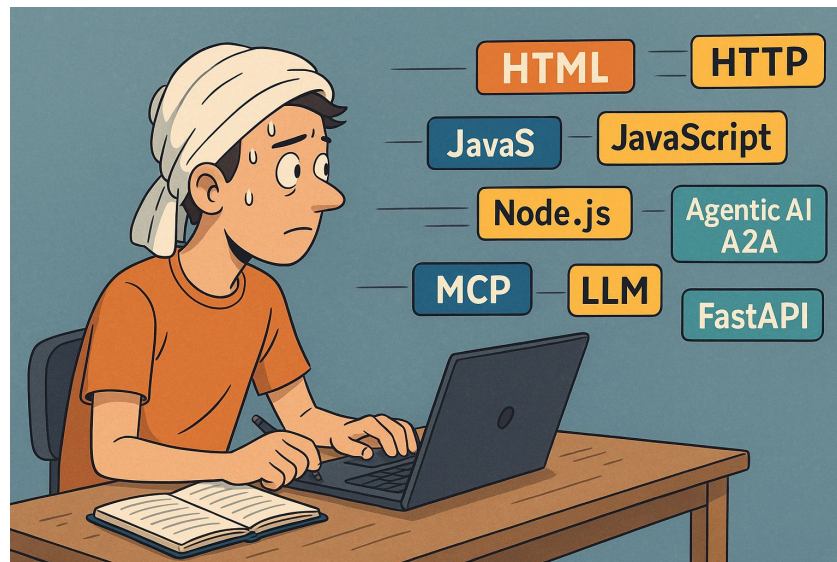


Knowledge will make you be free, Socrates



Coming 4 weeks are critical for success in the course

- You are learning HTML, HTTP, JavaScript, REST, Express, NodeJS, Agentic AI A2A, MCP, LLM, FastAPI etc
- make sure to code every day
-

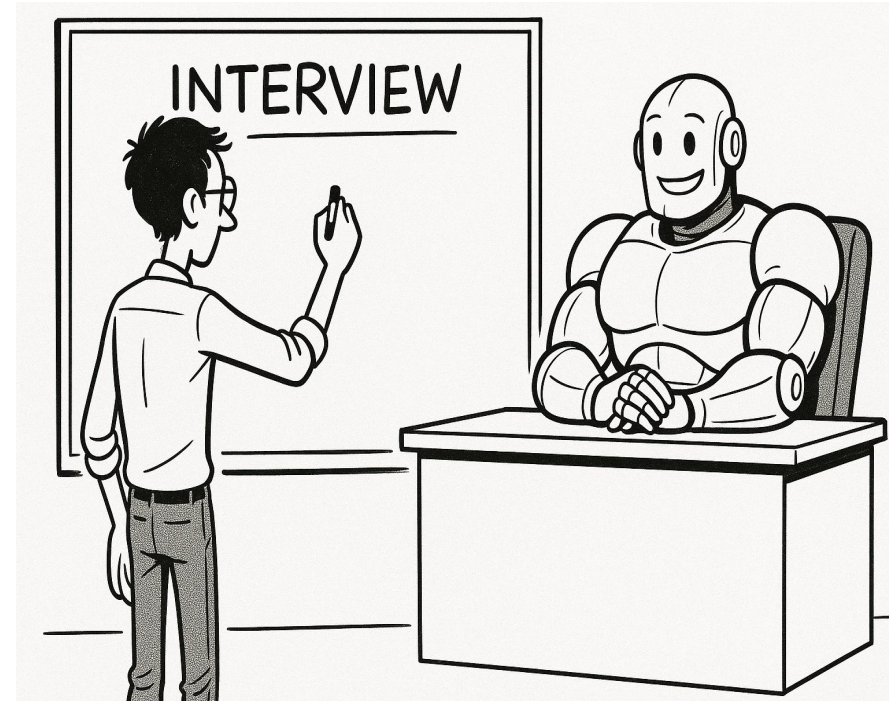


Advice for white board style interviews

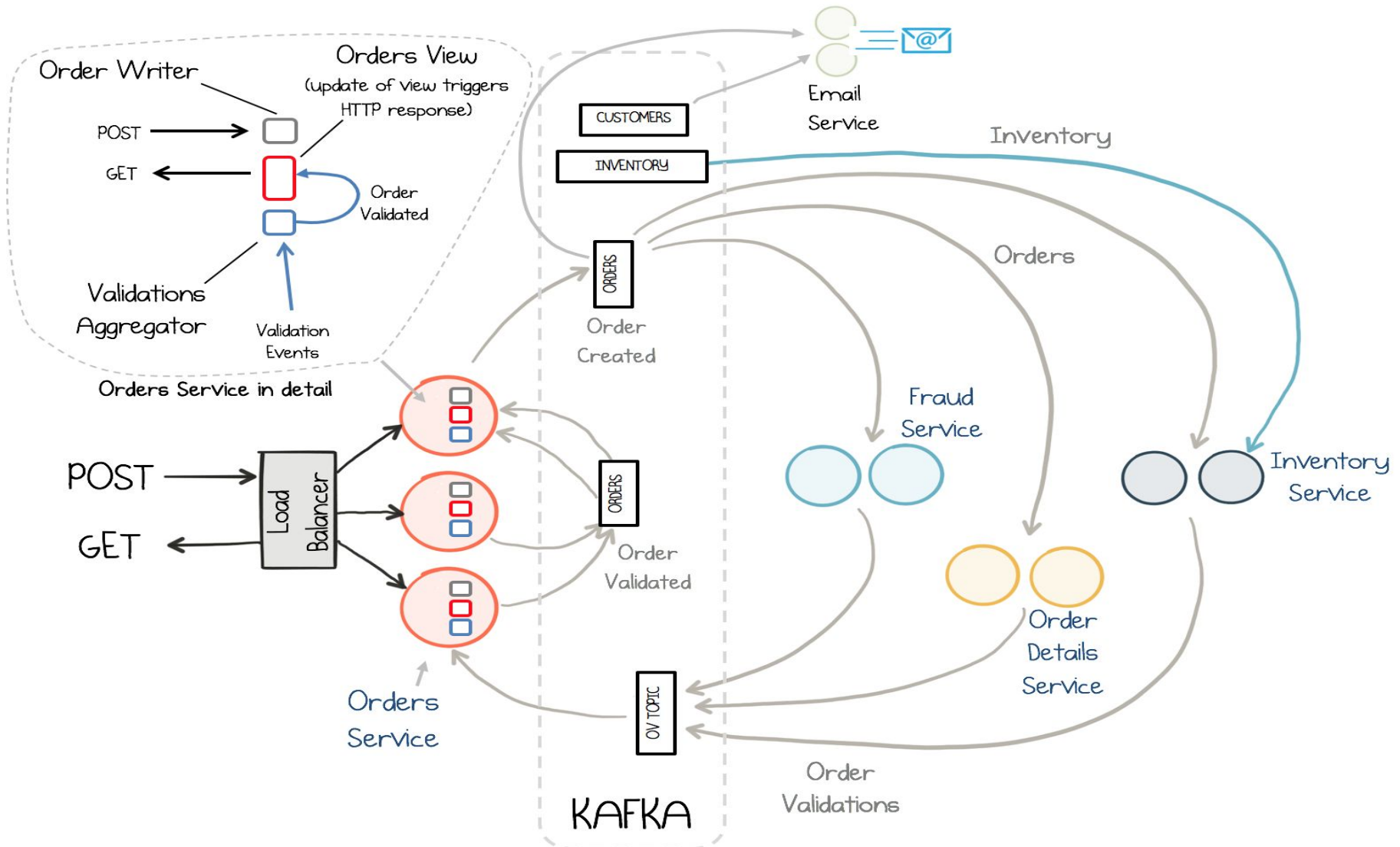
- Practice with Whiteboard
- Ask, ask, ask questions
- Talk Through Your Thought Process
- organize your work
- ask for feedback

After the Interview

1. Reflect on the Experience
2. Identify Areas for Improvement

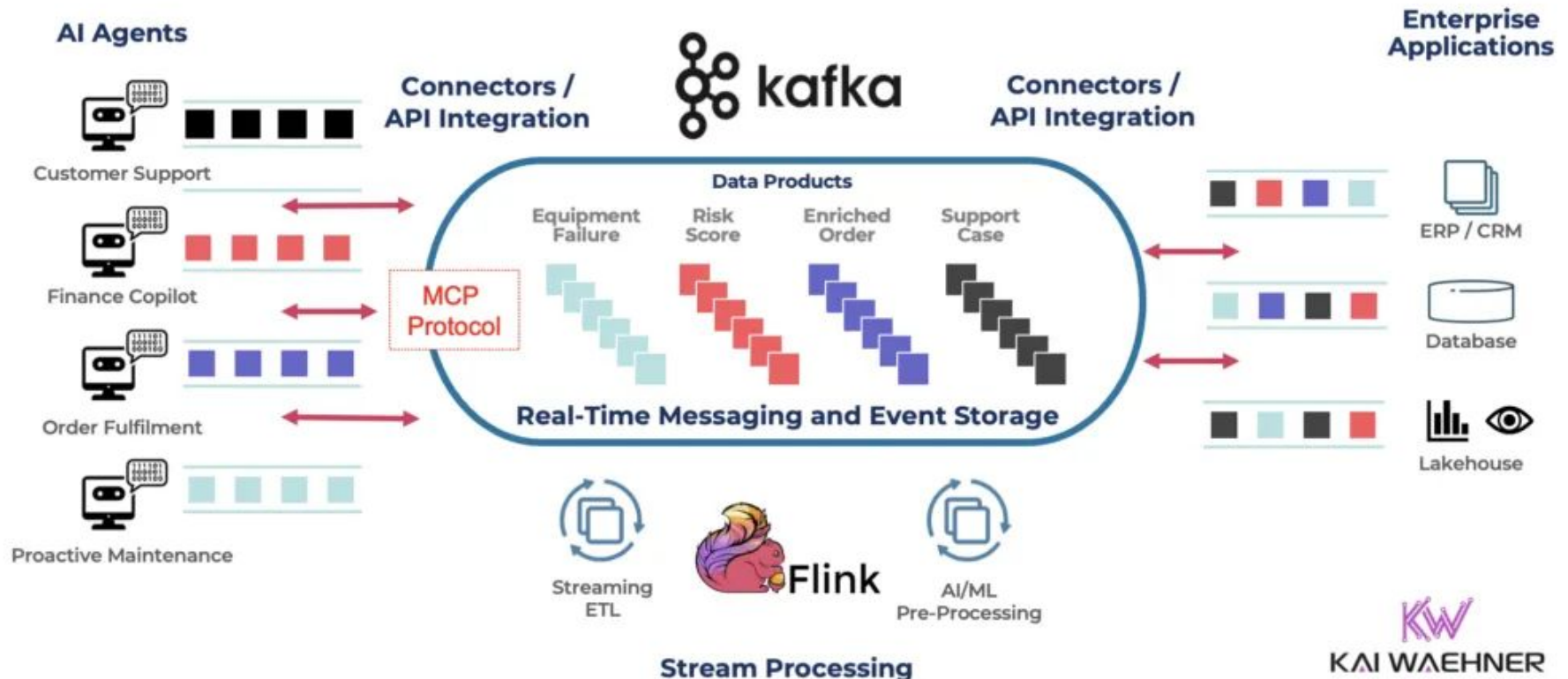


Distributed Microservices with kafka



Distributed Agentic AI

Agentic AI with Apache Kafka as Event Broker



HTTP: <https://en.wikipedia.org/wiki/HTTP>

HTTP stands for "Hypertext Transfer Protocol" and is the primary protocol used to transfer data across the web, essentially defining the rules for how web browsers and servers communicate with each other to display web pages and access online content.

- **Client-server model:**

HTTP operates on a client-server model where a client (like a web browser) sends a request to a server, which then responds with the requested data.

- **Stateless protocol:**

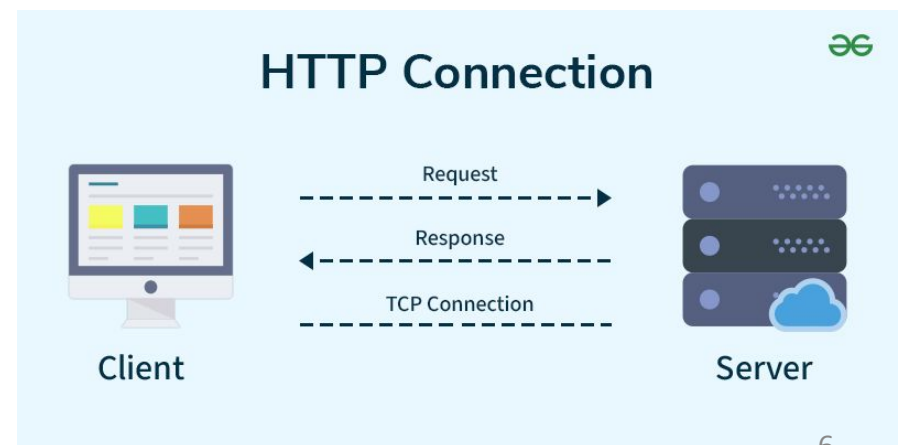
Each request from a client is treated independently, meaning the server doesn't keep track of previous interactions with that client.

- **Data transfer:**

HTTP is used to transfer various types of data like text, images, videos, and application data.

- **Request-response cycle:**

The basic interaction involves a client sending an HTTP request to a server, which then sends an HTTP response containing the requested data.



Rest Standards

- REST architecture is governed by HTTP methods, namely
- **GET**
- **POST**
- **PUT**
- **HEAD**
- **DELETE**
- **PATCH**
- **OPTIONS**

GET Method

- GET is used to request data from a specified resource.
- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests have length restrictions
- GET requests is only used to request data (not modify)

POST Method

- The data sent to the server with POST is stored in the request body of the HTTP request.
- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

PUT Method

- PUT is used to send data to a server to create/update a resource.
- The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.

DELETE Method

- The DELETE method deletes the specified resource.

Responding to various HTTP methods

Resource	POST (Create)	GET (Read)	PUT (Update)	DELETE (Delete)
/tasks	Create a new task	Return all tasks	Replace all tasks with new tasks.	Delete all tasks
/tasks/123	Create a task inside another task.	Return a specific task	Update this task if exist or create a new one.	Delete this task.

Well, this seems great..but how do applications come to know if their request succeeded or anything went wrong. (Specially for PUT and POST)?

HTTP - Response Codes

Server sends response code for every client request. This communicates to clients what happened to the their request (Pass, fail, error etc).

Response code for each type of request

Range (From client perspective)	Status code	Method	Comments.
1XX (100 - 199)	* - Informational	*	Request is accepted and is in progress
2xx (200 - 299) - Success	200 - OK	GET	Request is served.
	201 - Created	POST / PUT	Whenever a server create new resource on client's request
	202 - Accepted	*	Server accepted the requested but will be processed later.
	204 - No Content	PUT	Server performed the request but has no data to return.

HTTP - Response Codes

Range (From client perspective)	Status code	Method	Comments.
3XX - Redirection	301 - Permanently moved	GET	Server redirects client to the new URL of the same resource.
4xx (400 - 499) - Client Side Problem	400 - Bad Request.	*	Client has used either wrong URL or parameters, or provided insufficient data in the request.
	401 - Unauthorized.	*	Client is unauthorized to make this request. (May be client have passed wrong credentials)
	404 - Not Found	GET	The request resource cannot be found.
5xx (500 - 599) - Server Side Problem	500 - internal server error.	*	Server has encountered some exception while serving the request.

REST

What is REST?

- Representational State Transfer
- Style of Software Architecture for WWW
- Based on Client and Server
- Request and Response are built on transfer of “representation” or “resources”
- Uses HTTP protocol

Sites Using REST

- Amazon
- Yahoo,
- Google (search, OpenSocial)
- Flickr
- FaceBook
- MySpace
- LinkedIn
- IBM
- Microsoft
- Digg
- eBay
- Etc...

Resources

- Are just Concepts
- Located by URIs
- URIs tell client that there's a concept somewhere
- Client then asks for specific representation of the concept from the representations the server makes available
- Resource is any entity that you may want to expose and that can be named. Typically, Each entity in ER-Diagram can be thought of as a resource.
- E.g Webpage is a representation of a resource
contd..

REST guidelines (principles)

2 URI's per resource.

1. Listing (collection) - /v1/tasks
2. Detail - /v1/tasks/{id}

Use nouns instead of verbs to name resources. So, instead of saying /v1/getAllTasks, we should say /v1/tasks.

This means, we need to revise all our URIs to follow rest principles.

Prefer plurals over singular: /tasks over /task.

For associations: /resource/{id}/resource In our case, we may have /user/8908/tasks, to get all the tasks for a user.

RESTful Web Services

- Representational State Transfer (REST)
 - Roy Fielding, 2000 (doctoral dissertation)
 - Examination of the Internet as a stateless service of near-limitless expansion model with a simple but effective information delivery system
- Key concepts
 - Resources - source of information
 - Consistent access to all resources
 - As in interface and communication – Not content or function
 - Stateless protocol
 - Hypermedia – links in the information to other data (connectedness)

REST

- Concepts/Components
 - Representational
 - Information returned from a resource (represented using URLs)
 - Uniquely identifiable information
 - State
 - The accessing of information by the client (e.g., browser) is a state change
 - Viewing <http://www.mysite.com> changes what information is being accessed
 - Transfer
 - The act of changing state (URL) transfers information to the client

Definition

- REST is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.

- Roy Fielding in his Ph.D. dissertation in the year 2000

Example

- Consider a Book store which has enabled a Web Service for book ordering.
- It should enable customers to
 - Get a list of books
 - Get detailed information on each book
 - Submit a order to purchase it

contd..

Get books list

- Client uses following URL to get books list

<http://www.bookstore.com/books>

- The server returns

```
<?xml version="1.0"?>
<b:Books xmlns:p="http://www.Books-store.com"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <Book id="00345" xlink:href="http://www.Books-store.com/Books/00345"/>
<Book id="00346" xlink:href="http://www.Books-store.com/Books/00346"/>
<Book id="00347" xlink:href="http://www.Books-store.com/Books/00347"/>
<Book id="00348" xlink:href="http://www.Books-store.com/Books/00348"/>
</p:Books>
```

contd..

Get detailed information on each book

- The client then uses one of the book id and traverses to that resource.
- The client uses

<http://www.bookstore.com/books/00346>

The Server responds with

```
<?xml version="1.0"?>
<p:Book xmlns:b="http://www.Books-store.com"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <Book-ID>00345</Book-ID>
  <Name>Harry Potter</Name>
  <Description>This boos is written by JK Rowling</Description>
  <Specification xlink:href="http://www.Books-store.com/books/00346/readings"/>
    <UnitCost currency="USD">20.0</UnitCost>
    <Quantity>10</Quantity>
</b:Book>
```

contd..

Creating a RESTful service

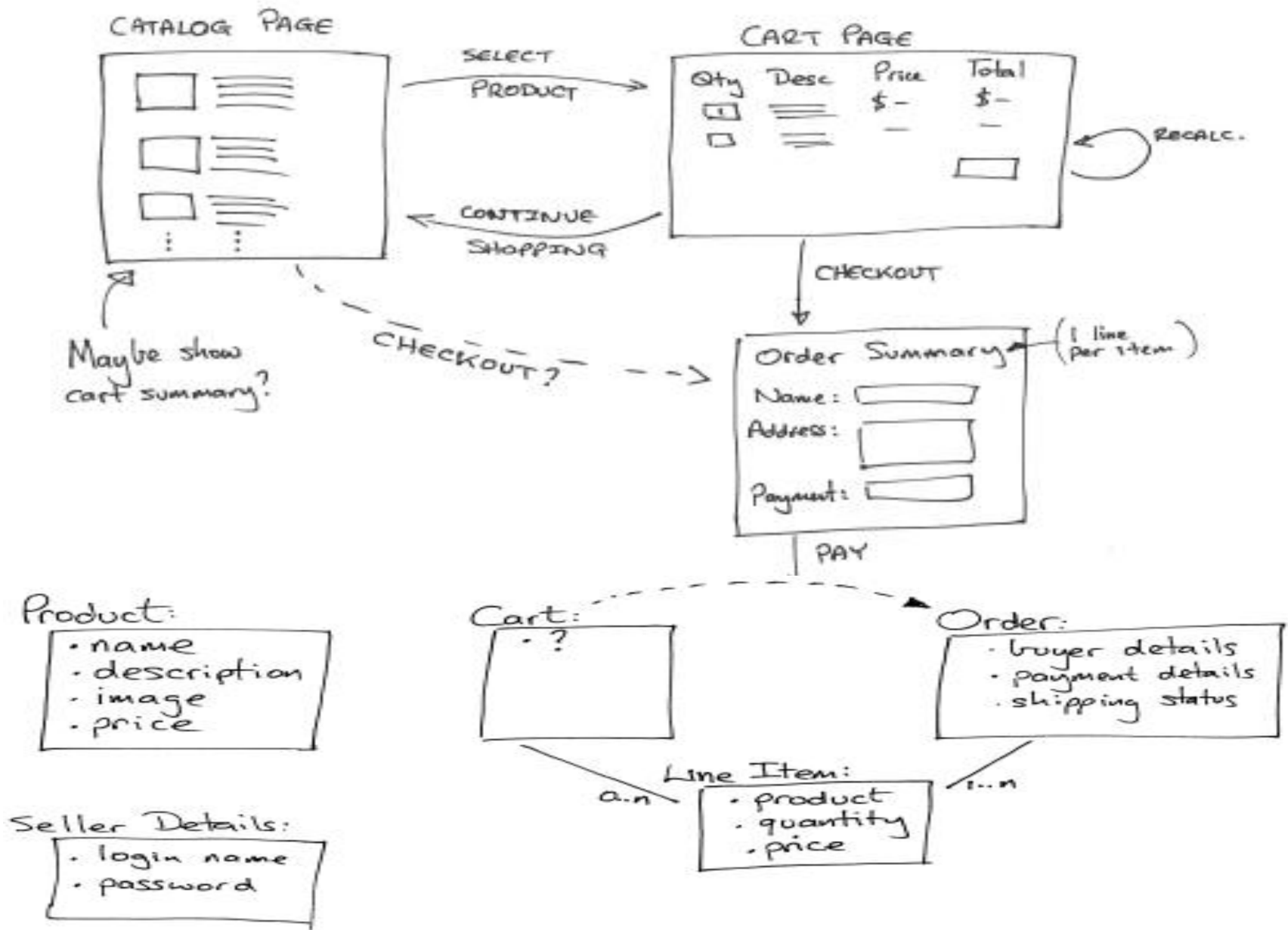
Steps

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



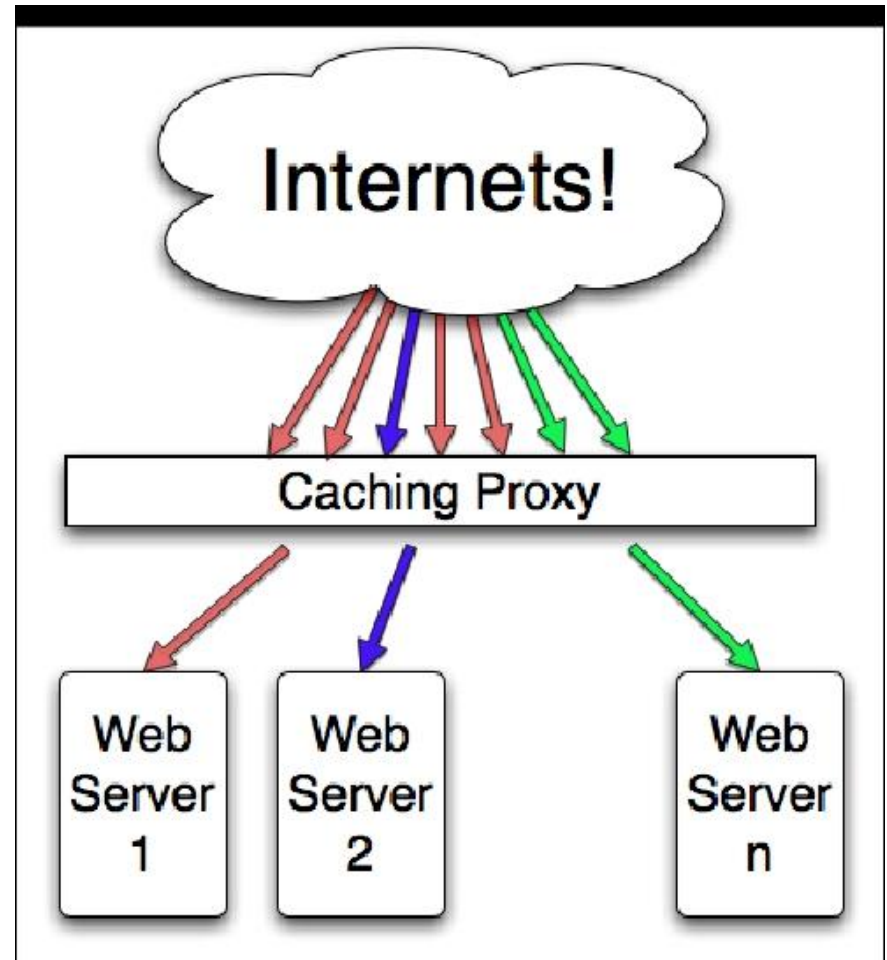
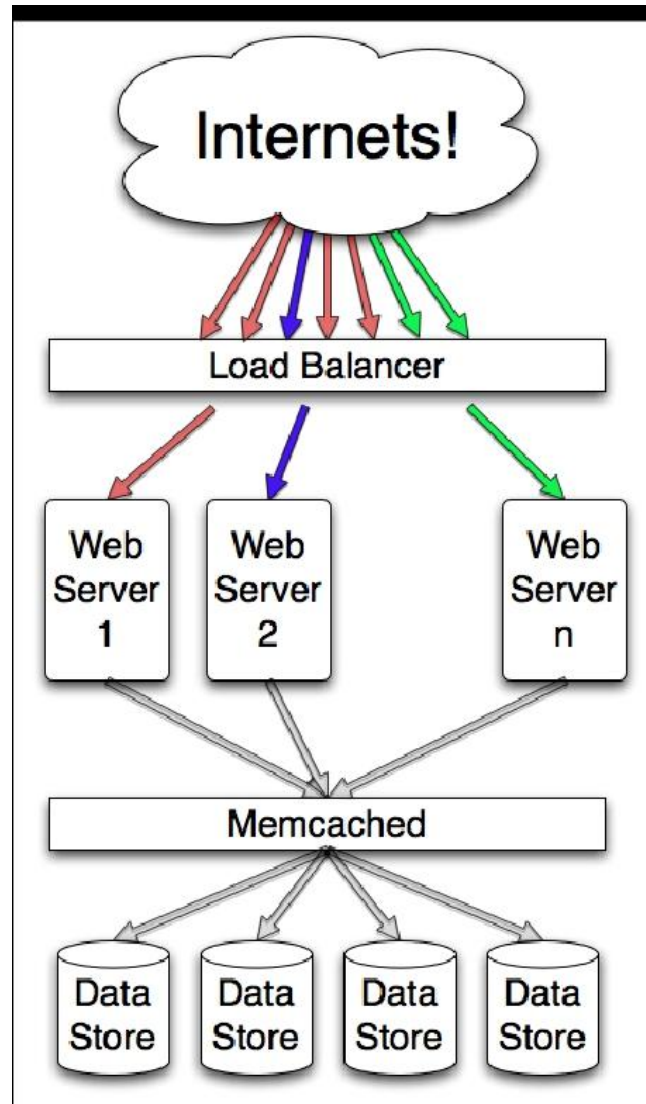
Down



When is it appropriate?

- The web services are completely stateless
- Caching infrastructure can be leveraged for performance
- The service producer and consumer have mutual understanding of context and content being passed e.g NetFlix Api
- Bandwidth is a constraint

Multi-level Caching



What else can we expect from proper REST APIs?

Pagination:

Return subset of all the data present in DB. User must be able to tell you get me next 20 records after 60.

This can be achieved by adding offset and limit parameters to our GET APIs.
ex. /v1/tasks?offset=60&limit=20

Multiple Representation Formats:

Server must return the data in more than one format (XML, JSON etc).

For client to specify what format the server should reply, ACCEPT http request header needs to be set. Ex. ACCEPT : Application/xml.

For server to specify what format is being used in the response format, Content-Type http response header needs to be set.
Ex. Content-Type : Application/xml.

What else can we expect from proper REST APIs?

Filtering based on parameters:

Filtering acts as a WHERE clause in SQL statement. To get all tasks whose completion date is 1st May 2013.

ex. /v1/tasks?completion-date=2013-05-01

Ordering:

This plays a role of OrderBy clause in SQL statement. To get all the tasks sorted by priority.

ex. /v1/tasks?order_by=priority - Sorts in ascending order of priority.

/v1/tasks?order_by=-priority - Sorts in descending order of priority.

Web Application Architecture {MVC}

Presentation Layer

Takes care of Content Representation. (HTML5, CSS3).

Business Layer

Computational Logic on data using language libraries.

Data Access Layer
(SQL or ORM)

Access data from DB using RAW SQL queries or an ORM



DB

Data Storage System (SQLite, MySQL, ORACLE).

References

- Roy Fielding, 2000 (doctoral dissertation)
 - <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- <http://java.sun.com/developer/technicalArticles/WebServices/restful/>
- http://en.wikipedia.org/wiki/Representational_State_Transfer
- <http://www.devx.com/DevX/Article/8155>
- http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Rest Api <https://restfulapi.net/resource-naming/>

What is NodeJS ?

Standard Google Definition :

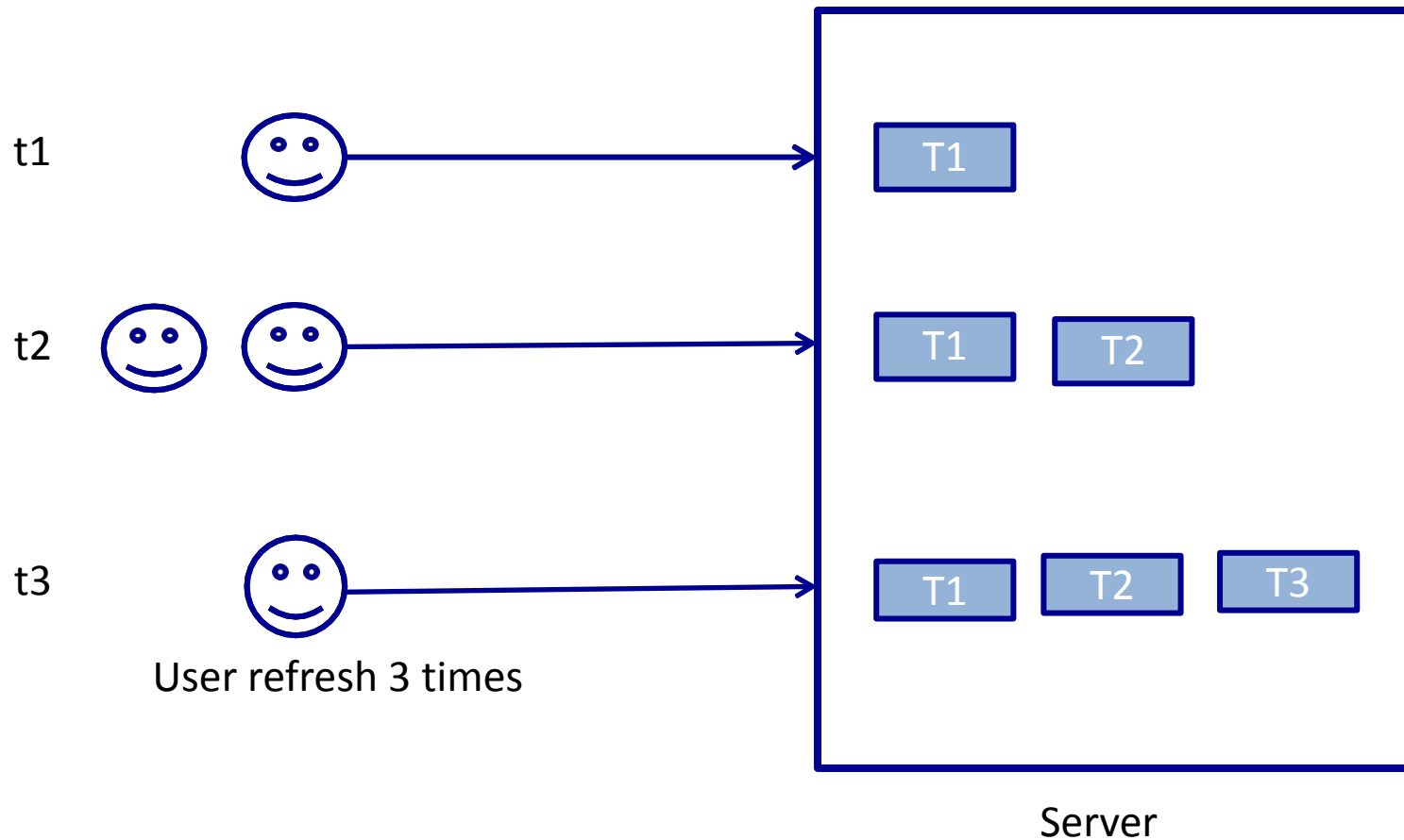
- Node.js[®] is a JavaScript runtime built on **Chrome's V8 JavaScript engine**.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

Evolution of web

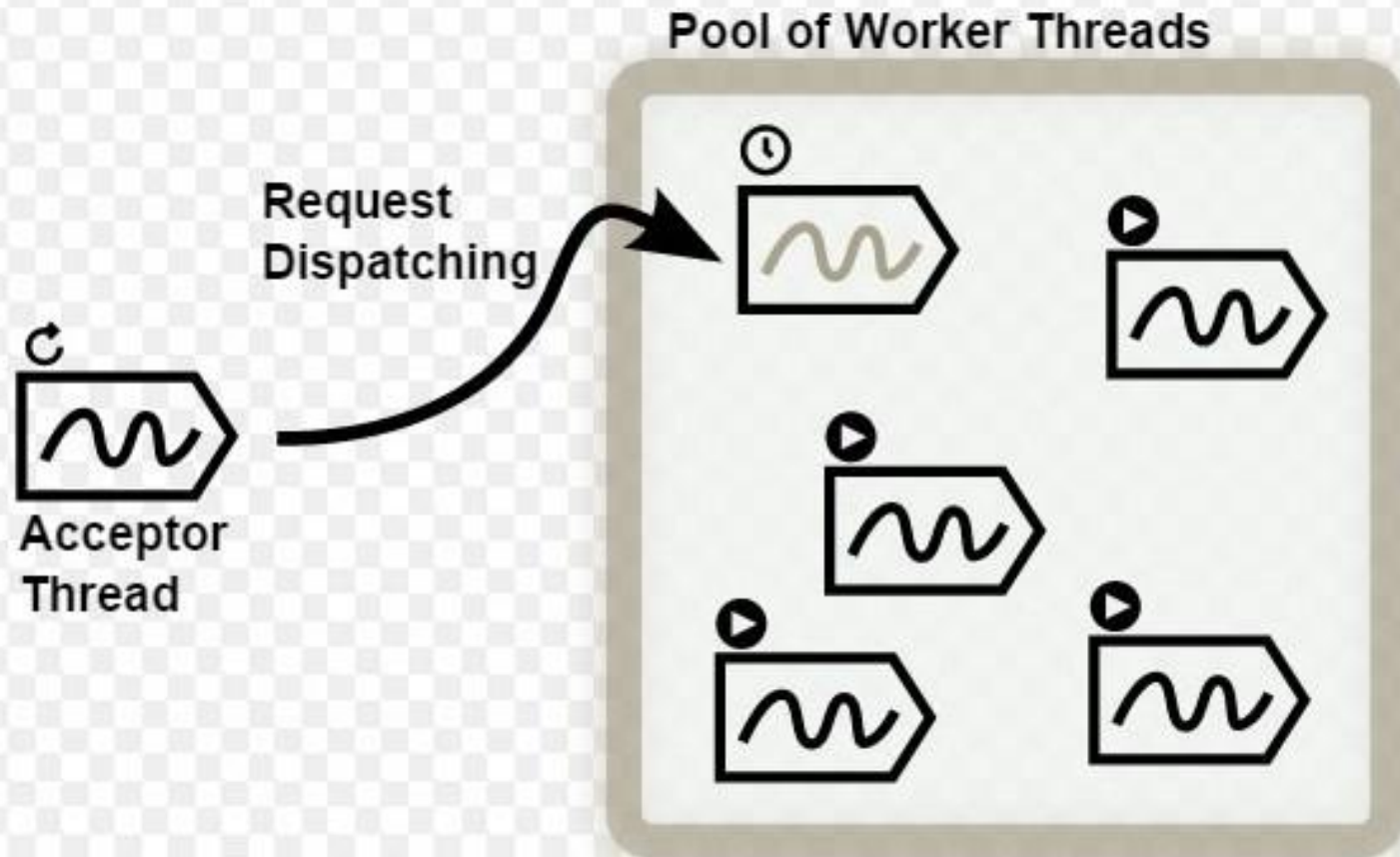
Web has evolved from

- Static websites (90's)
- Dynamic web applications (AJAX) (early 2000's)
- Real time web applications (notifications)

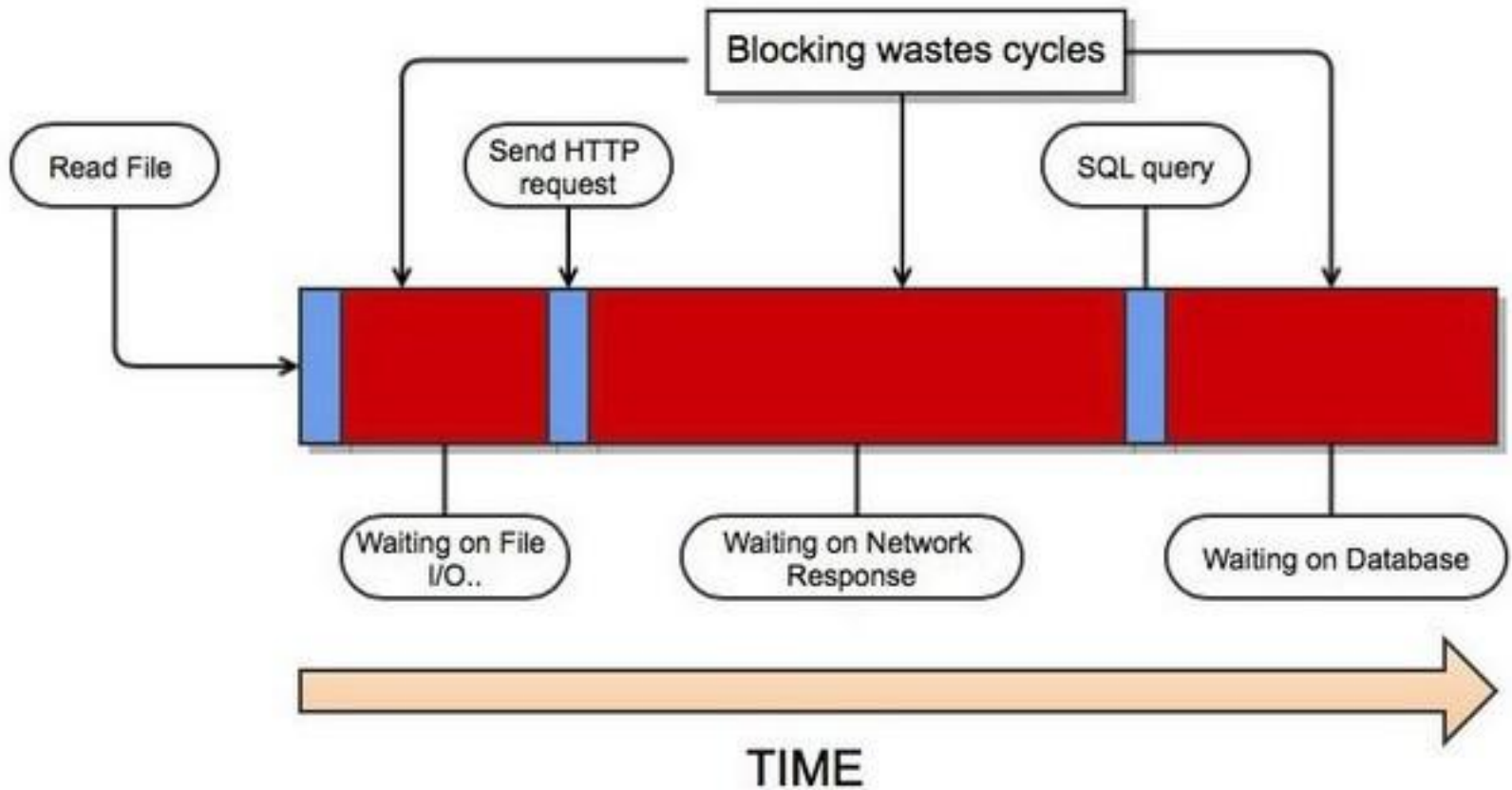
Traditional multi threaded server



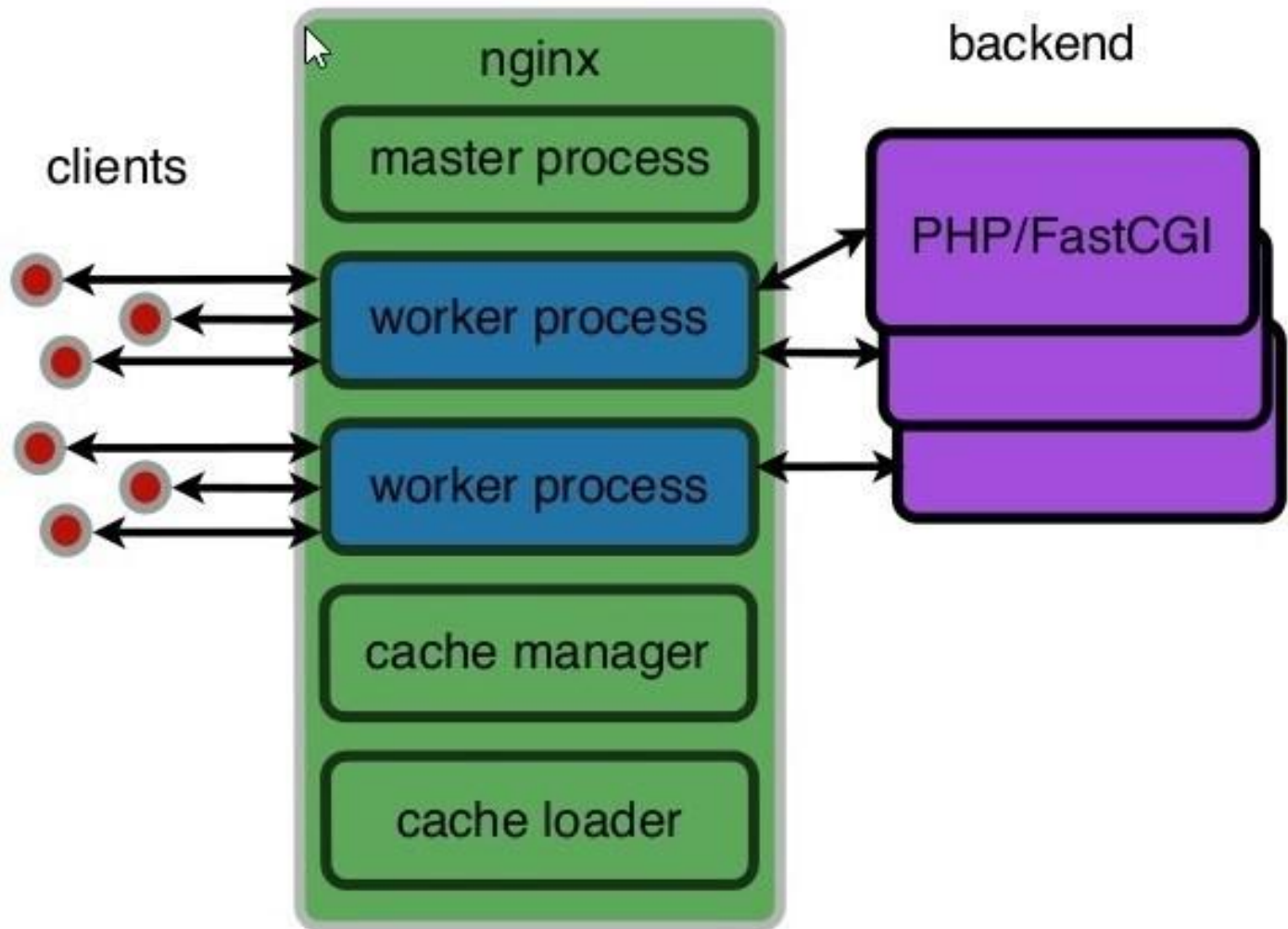
Web Server Architecture



Traditional (Blocking) Thread Model

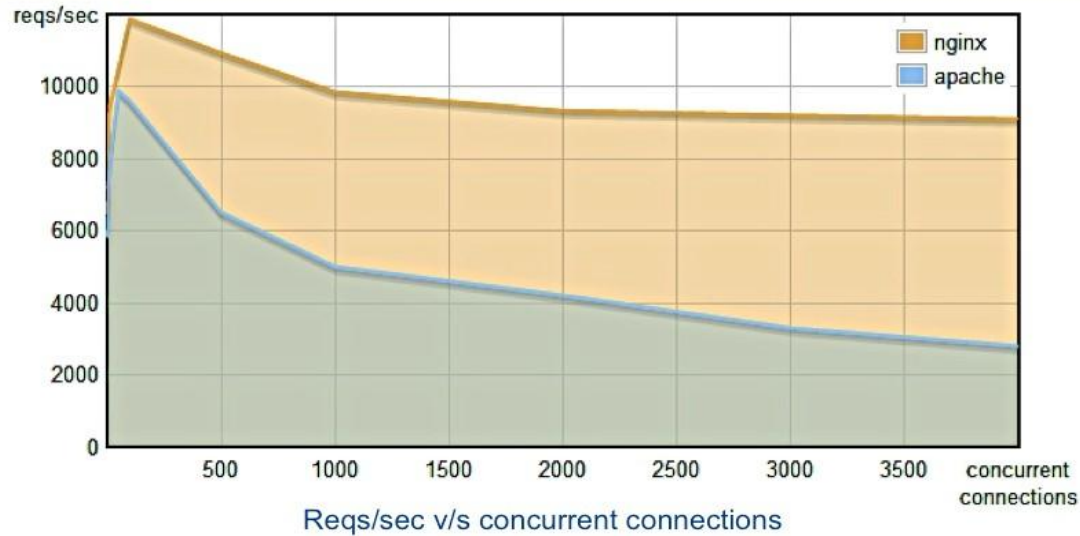


Nginx



Apache vs Nginx

Apache V/s Nginx: performance



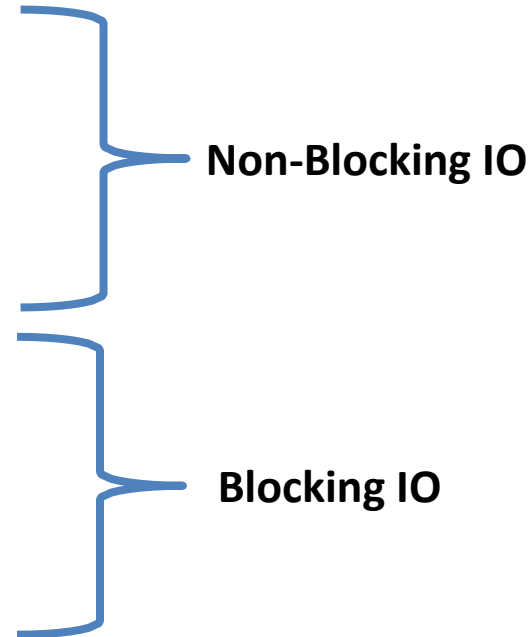
At ~4000 concurrent connections,
- Nginx can serve ~9000 reqs/sec
- Apache can serve ~3000 reqs/sec

Ref: <http://blog.webfaction.com/a-little-holiday-present>

Picture source : cloudfoundry, vmware

Cost of IO

- L1-cache 3 cycles
- L2-cache 14 cycles
- RAM 250 cycles
- Disk 41 000 000 cycles
- Network 240 000 000 cycles



Evented asynchronous platform

Javascript in browser

- Single Threaded.
- Asynchronous.
- Functional language. Native callback function support.

Asynchronous IO

Example

```
•db.getData('select * from users', function(err,  
  results) {  
    • console.log(results);  
  •});
```

Non-Blocking I/O

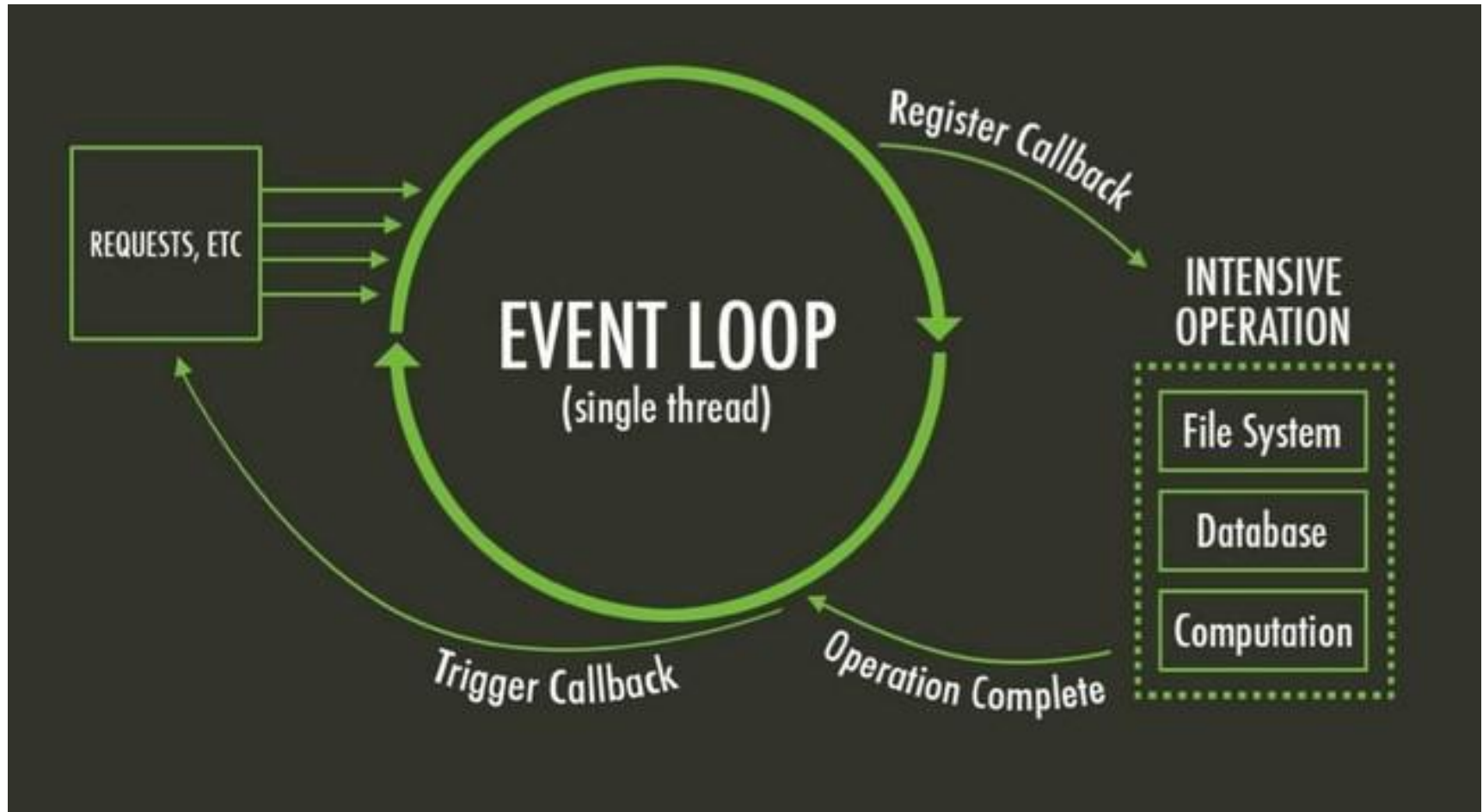
- Traditional I/O

```
var result = db.query("select x from table_x");  
doSomethingWith(result); //wait for result!  
doSomethingWithoutResult(); //execution is  
blocked!
```

- Non-traditional, Non-blocking I/O

```
db.query("select x from table_x",function (result){  
    doSomethingWith(result); //wait for result!  
});  
doSomethingWithoutResult(); //executes without any delay!
```


Event Loop Example

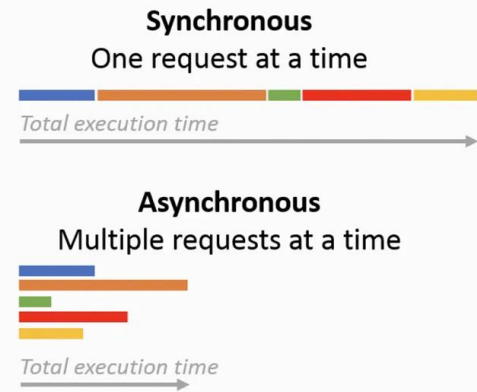


Non-blocking I/O

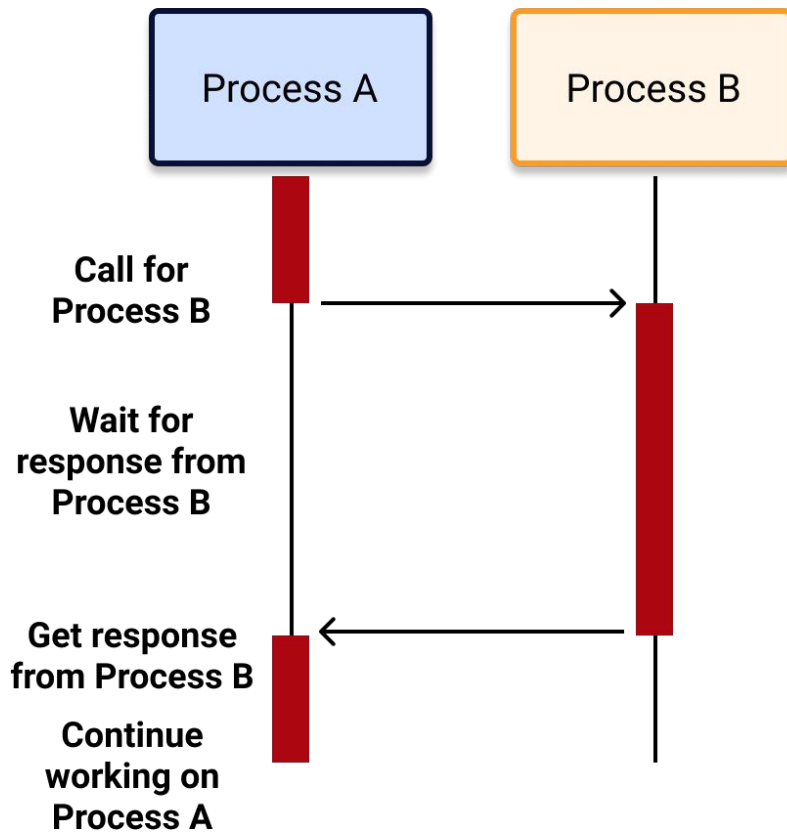
- Servers do nothing but I/O
 - Scripts waiting on I/O requests degrades performance
- To avoid blocking, Node makes use of the event driven nature of JS by attaching callbacks to I/O requests
- Scripts waiting on I/O waste no space because they get popped off the stack when their non- I/O related code finishes executing



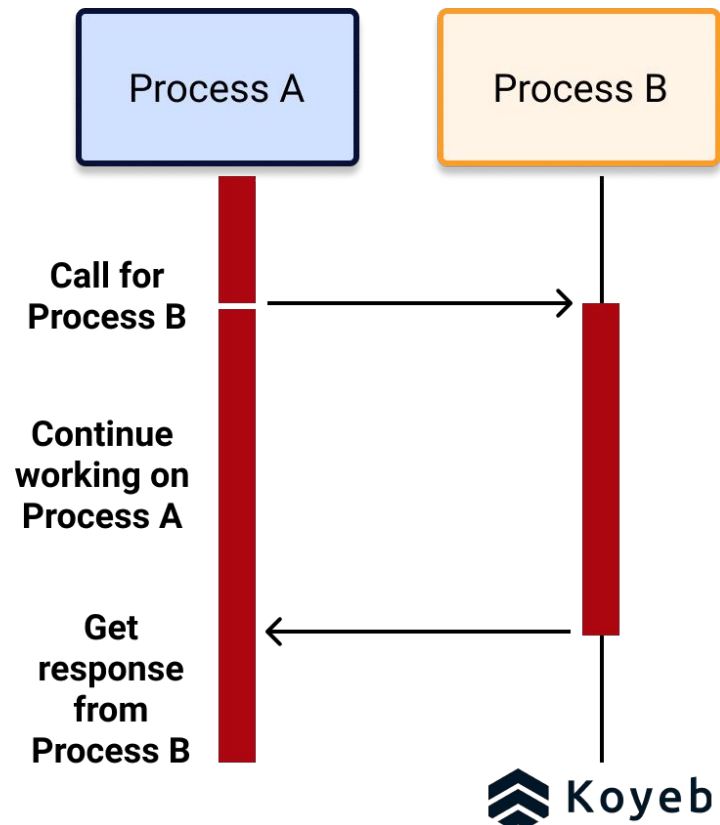
Synch vs Asynch



Synchronous Processing

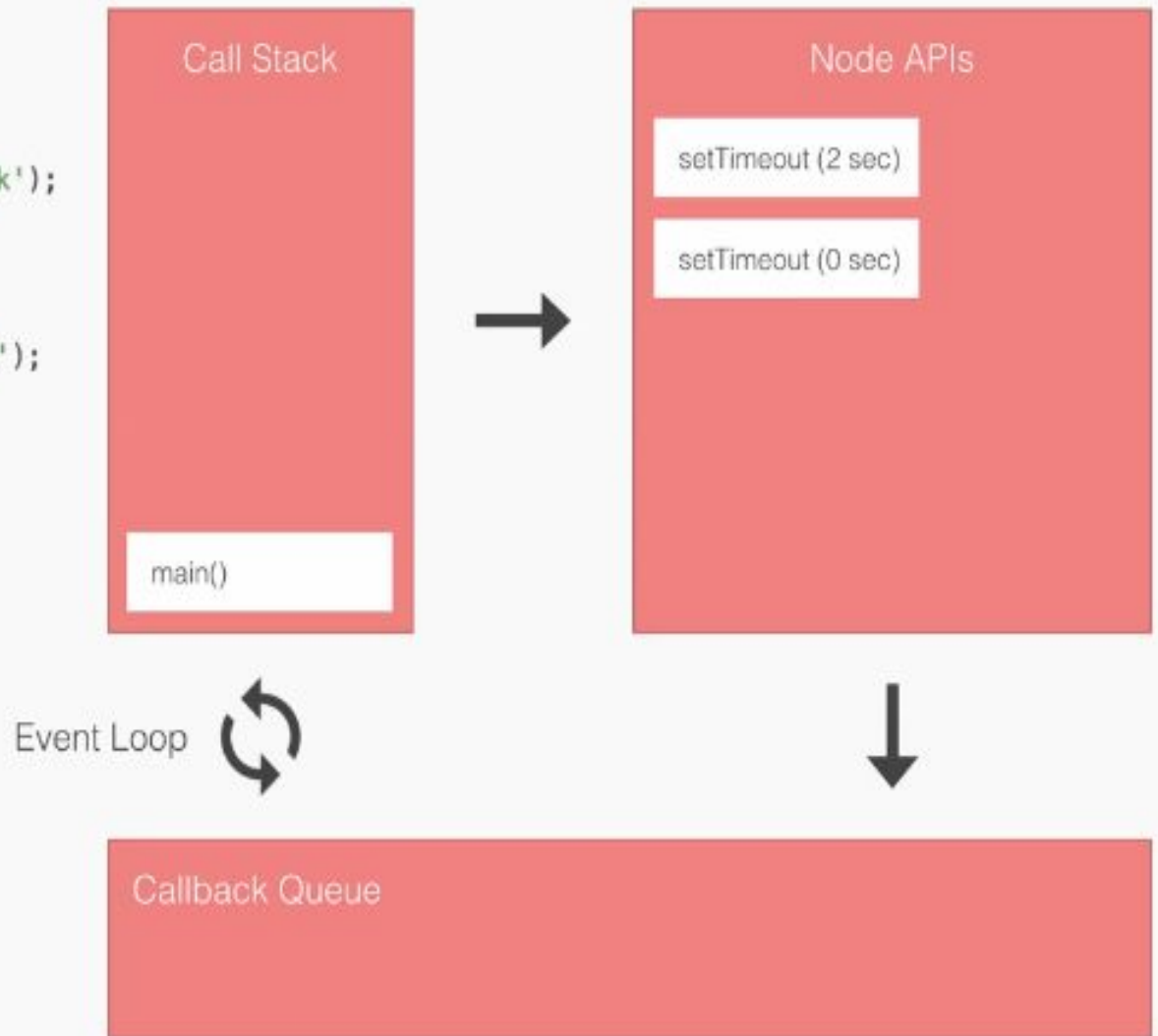


Asynchronous Processing



How JS Event Loop Works ?

```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



Installation

- Visit the Official Node Link to grab the installer for NodeJS:

<https://nodejs.org/en/> - (install v10.0.0 +)

- Check node version in terminal/cmd : `node -v`

NPM - Node Package Manager

- Adapt packages of code for your apps, or incorporate packages as they are.
- Download standalone tools you can use right away.
- Share code with any npm user, anywhere.
- Create Orgs (organizations) to coordinate package maintenance, coding, and developers.
- Form virtual teams by using Orgs.
- Manage multiple versions of code and code dependencies.
- Update applications easily when underlying code is updated.

Node.js – fs (Sync Read)

```
var fs = require('fs');
```

```
var in_data = fs.readFileSync('./fn_input.txt');
```

```
console.log('Sync input file content: ' + in_data);
```

```
console.log('Program Ended.');
```

Node.js – fs (Sync Write)

```
var fs = require('fs');  
var out_data = 'Output line 1.\r\nOutput line  
2.\r\nOutput last line.';  
  
fs.writeFileSync('./sync_output.txt', out_data);  
console.log('Sync output file content: ' +  
    out_data);  
  
console.log('Program Ended.');
```


Node.js – fs (Async Read)

```
var fs = require('fs');  
var in_data;  
  
fs.readFile('./fn_input.txt', function (err, data) {  
    if (err) return console.error(err);  
    in_data = data;  
    console.log('Async input file content: ' + in_data);  
});  
  
console.log('Program Ended.');
```

Node.js – fs (Async Write)

```
var fs = require('fs');  
var out_data = 'Output line 1.\r\nOutput line 2.\r\nOutput last  
line.';  
  
fs.writeFile('./async_output.txt', out_data, function (err) {  
    if (err) console.error(err);  
    console.log('Async output file content: ' + out_data);  
});  
  
console.log('Program Ended.');
```

Node.js – net (Network Server)

```
var net = require("net");
var server = net.createServer(function(connection) {
    console.log('Client connected.');
```

```
    connection.on('end', function() {
        console.log('Client disconnected.');
```

```
    });
    connection.write('Hello World!\n');
```

```
    connection.pipe(connection);
    // send data back to connection object which is client
});
server.listen(8080, function() {
    console.log('Server is listening.');
```

```
});
console.log('Server Program Ended.');
```

Node.js (Modules)

Module name	Description
buffer	buffer module can be used to create Buffer class.
console	console module is used to print information on stdout and stderr.
dns	dns module is used to do actual DNS lookup and underlying o/s name resolution functionalities.
domain	domain module provides way to handle multiple different I/O operations as a single group.
fs	fs module is used for File I/O.
net	net module provides servers and clients as streams. Acts as a network wrapper.
os	os module provides basic o/s related utility functions.
path	path module provides utilities for handling and transforming file paths.
process	process module is used to get information on current process.

Node.js – web module

- Web server is a software application which processes request using HTTP protocol and returns web pages as response to the clients.
- Web server usually delivers html documents along with images, style sheets and scripts.
- Most web server also support server side scripts using scripting language or redirect to application server which perform the specific task of getting data from database, perform complex logic etc. Web server then returns the output of the application server to client.
- Apache web server is one of the most common web server being used. It is an open source project.

Web server and Path

- The web server maps the path of a file using URL, Uniform Resource Locator. It can be a local file system or an external/internal program.
- A client makes a request using browser, URL: <http://www.abc.com/dir/index.html>
- The browser will make the request as:

```
GET /dir/index.html HTTP/1.1  
HOST www.abc.com
```

Creating Web Server using Node (http module)

- Create an HTTP server using `http.createServer` method. Pass it a function with parameters `request` and `response`.
- Write the sample implementation to return a requested page.
- Pass a port 8081 to `listen` method.

http_server.js

```
var http = require('http');
var fs = require('fs');
var url = require('url');
http.createServer(function (request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log('Request for ' + pathname + ' received.');
```

fs.readFile(pathname.substr(1), function (err, data) {

 if (err) { console.log(err.stack);

 response.writeHead(404, {'Content-Type' : 'text/html'}); // HTTP status:

404 : NOT FOUND

 } else { response.writeHead(200, {'Content-Type' : 'text/html'}); // HTTP status:

200 : OK

 response.write(data.toString());

 }

 response.end(); // send the response body

});

}).listen(8081);

console.log('Server running at <http://127.0.0.1:8081/test.html>');

console.log('Server Program Ended.');

Express

- Express is a minimal and flexible Node.js web application **framework**. Simply . . .
- Routing
- Middlewares
- Templating
- Error Handling
- Guide Section in <https://expressjs.com/>

Agentic Behavior & Agent-to-Agent (A2A) Interactions

Quick recap (from last class)

Agenda we covered:

1. What is an Agent?
2. How Agents work
3. Core patterns
4. Memory that matters
5. Guardrails & safety
6. Observability & evaluation
7. When **not** to use an agent
8. Mini case study

What "agentic" really means

Simple idea: a loop of

Plan → Do → Check → Adjust/Act (think PDCA).

It's a continuous improvement cycle first developed by W. Edwards Deming, widely used in engineering, management, learning, and problem-solving.

- Plan – Decide what you want to do and how.
- Do – Execute the plan (try it out).
- Check – Review the results (did it work as expected?).
- Adjust (or Act) – Make changes to improve, then repeat.

Example: Article Summarization

- Plan: Outline key points you want in the summary.
- Do: Fetch and summarize the article.
- Check: Is the summary concise and accurate?
- Adjust: Shorten text, add references.
→ Repeat loop until the summary is sharp.



Behavior 1 — Reason + Act (ReAct)

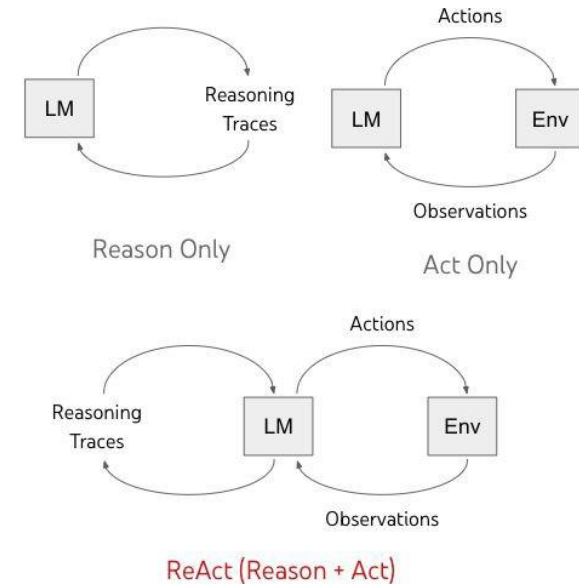
LLMs **think** (reason) and **use tools** (act) in steps.

Think of why GPT suddenly decides to browse the web even if you don't have an explicit instruction to do that

Why it works: reasoning guides which tool to call; tools bring real data back.

How LLMs Think & Act

- **Reasoning (Think):** LLMs break down a problem step by step.
- **Acting (Do):** They decide which **tool** to use (e.g., web search, database, calculator, API).
- **Loop:** Reasoning guides which tool to call → Tool brings back real data → LLM uses it to refine the answer.



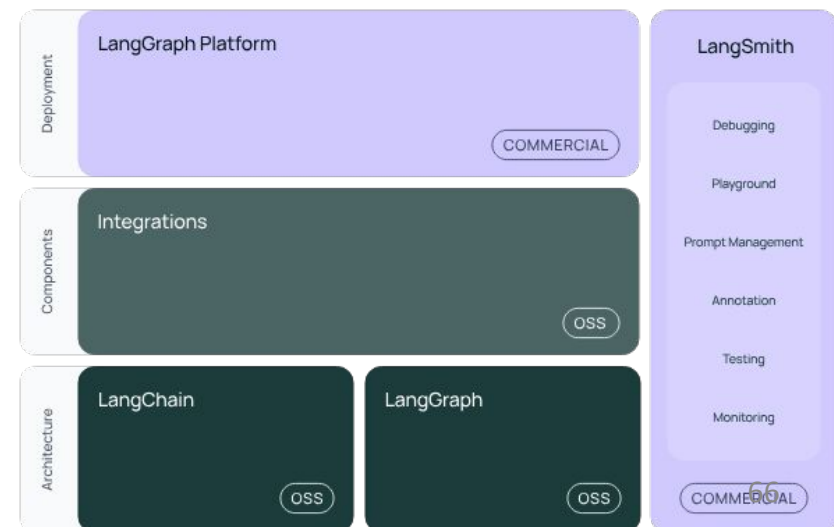
LangChain:

<https://python.langchain.com/docs/introduction/>

LangChain is a framework for developing applications powered by large language models (LLMs).

LangChain simplifies every stage of the LLM application lifecycle:

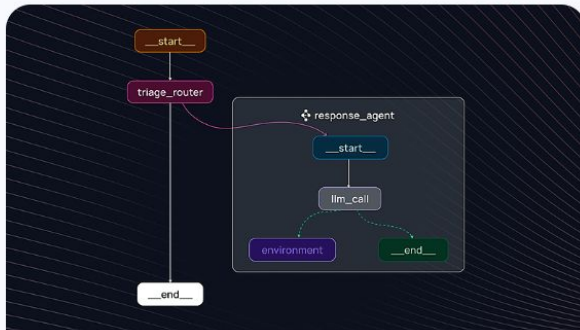
- Development: Build your applications using LangChain's open-source **components** and **third-party integrations**. Use **LangGraph** to build stateful agents with first-class streaming and human-in-the-loop support.
- Productionization: Use **LangSmith** to inspect, monitor and evaluate your applications, so that you can continuously optimize and deploy with confidence.
- Deployment: Turn your LangGraph applications into production-ready APIs and Assistants with **LangGraph Platform**.



LangChain Academy

Level up with LangChain Academy:

<https://academy.langchain.com/>



Project: Building Ambient Agents with LangGraph

📖 Course

Build your own ambient agent to manage your email. You'll learn the fundamentals of LangGraph as you build an email assistant from scratch, and use...



Foundation: Introduction to LangSmith

📖 Course

Learn the essentials of agent evaluations with LangSmith — our platform for agent development. Continuously improve your agents with LangSmith'...



Project: Deep Research with LangGraph

📖 Course

Build your own deep research agent to handle research tasks. Learn how to use LangGraph to build a multi-agent system, then use LangSmith to...

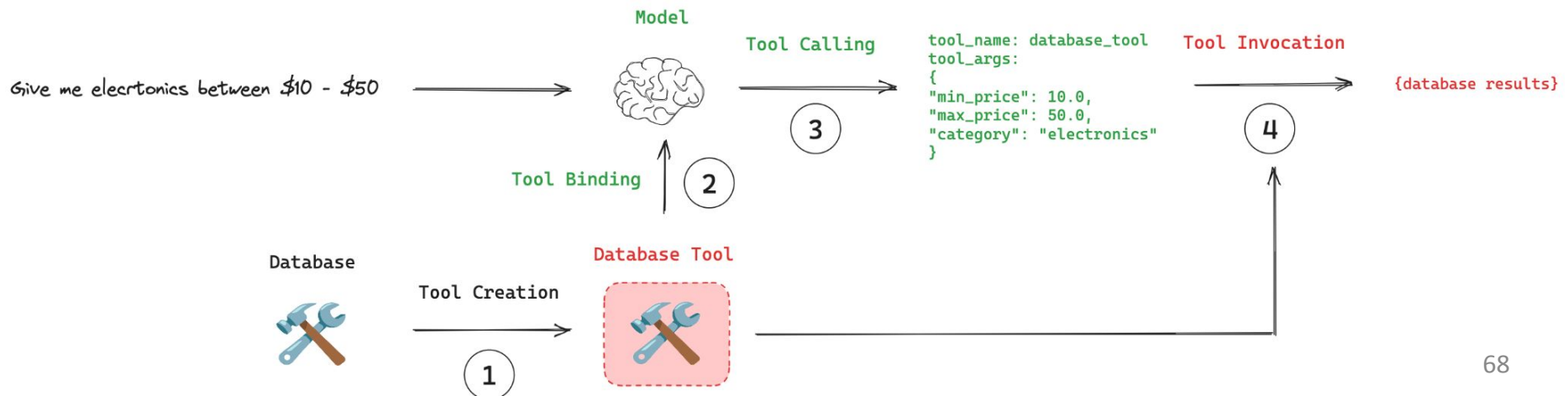


Tool Calling

https://python.langchain.com/docs/concepts/tool_calling/

Key concepts

1. Tool Creation: Use the `@tool` decorator to create a `tool`. A tool is an association between a function and its schema.
2. Tool Binding: The tool needs to be connected to a model that supports tool calling. This gives the model awareness of the tool and the associated input schema required by the tool.
3. Tool Calling: When appropriate, the model can decide to call a tool and ensure its response conforms to the tool's input schema.
4. Tool Execution: The tool can be executed using the arguments provided by the model.



ReAct (Defining a method for tool calling)

```
@tool
def get_datetime(tz: str = "UTC") -> dict:
    """Return the current date/time for an IANA timezone (e.g., 'Asia/Kolkata', 'UTC')."""
    try:
        now = datetime.now(zoneinfo.ZoneInfo(tz))
    except Exception:
        # Fallback to UTC if bad timezone
        tz = "UTC"
        now = datetime.now(zoneinfo.ZoneInfo("UTC"))
    return {
        "tz": tz,
        "iso": now.isoformat(timespec="seconds"),
        "date": now.strftime("%Y-%m-%d"),
        "weekday": now.strftime("%A"),
        "time": now.strftime("%H:%M"),
    }
```

```
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0) #for deterministic output
llm_with_tools = llm.bind_tools([get_datetime])
```

Conversation where the model MUST use the tool **for** the date (**not** guess)

```
messages = [
    SystemMessage(content="Always call get_datetime for any date/time info."),
    HumanMessage(content=(
        "Create a short note that includes today's date in both Asia/Kolkata and
        America/Los_Angeles, " "then add one friendly sentence."
    )),
]
ai = llm_with_tools.invoke(messages)
tool_calls = ai.additional_kwargs.get("tool_calls", []) or []

# If the model only called one timezone, we can nudge it with another turn:
if len(tool_calls) < 2:
    # Ask it explicitly to also fetch the second timezone (teaches multi-call)
    messages += [ai, HumanMessage(content="Also get the time in America/Los_Angeles.")]
    ai = llm_with_tools.invoke(messages)
    tool_calls = ai.additional_kwargs.get("tool_calls", []) or []
```

```
#Execution
tool_results = []
for call in tool_calls:
    args = json.loads(call["function"]["arguments"] or "{}")
    tool_results.append(ToolMessage(
        tool_call_id=call["id"],
        name=call["function"]["name"],
        content=json.dumps(get_datetime.invoke(args))
    ))

final = llm_with_tools.invoke(messages + [ai] + tool_results)
print(final.content)
```

Tool Calling: https://python.langchain.com/docs/concepts/tool_calling/

Output



Note:

- Today's date in Asia/Kolkata is August 29, 2025 (Friday) at 02:13.
- Today's date in America/Los_Angeles is August 28, 2025 (Thursday) at 13:43.

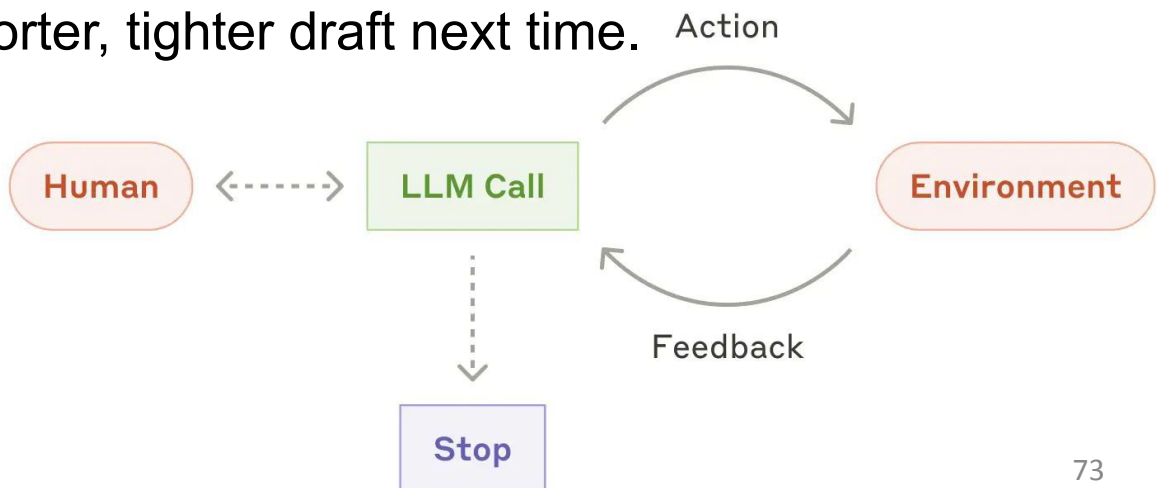
Have a wonderful day filled with joy and laughter!

Behavior 2 — Memory & Reflection

- Agents don't just act — they remember past experiences, evaluate what worked, and self-critique before giving the next reply.
- This creates a feedback loop:
Store → Evaluate → Improve → Repeat
- Memory + Reflection makes agents smarter over time → not just reactive, but adaptive and improving.

Example: Writing Assistant

- **Store:** Keeps track that the last draft was too long.
- **Evaluate:** “Feedback said it needed to be concise.”
- **Improve:** Produces a shorter, tighter draft next time.



Memory & Reflection — pick the right store (what to use, when)

Agents need memory to **recall**, **reflect**, and **improve**.

Different vector stores are suited for different use cases:

Semantic / RAG (similarity search): Core idea: Store text → retrieve by meaning, not keywords.

- FAISS (local, Python lib): best for small datasets, single machine, fast prototyping.
- Chroma (local/server): good for quick dev projects; minimal ops.
- PGVector (Postgres extension): Mix vectors + SQL filters/joins together; Great when metadata matters. Example: Search documents by both content + date tags.
- Qdrant / Milvus (self-hosted): Scalable, high-QPS vector search on your own infra.
- Weaviate / Pinecone (managed): production scale without ops; pay-as-you-go. Example: SaaS chatbot serving thousands of customers

FAISS

Step 1: Query → Embedding

Query:

```
"Explain REST APIs"
```

Embedding (toy example, 3-dim vector):

```
query_embedding = [0.1, -0.2, 0.9]
```

Step 2: Stored Document Embedding

Document:

```
"REST allows communication via HTTP  
methods"
```

Embedding:

```
doc_embedding = [0.12, -0.18, 0.85]
```


Step 3: Compute Similarity (dot product / cosine)

Using distance operator \leftrightarrow (in FAISS or PGVector, it means “how far apart are these vectors”):

```
similarity = dot(query_embedding,  
doc_embedding)
```

```
# = (0.1*0.12) + (-0.2*-0.18) +  
(0.9*0.85)
```

```
# = 0.012 + 0.036 + 0.765 = 0.813
```

High score → very similar 

Step 4: FAISS Retrieval

```
import faiss
```

```
import numpy as np
```

```
# Index with L2 similarity  
index = faiss.IndexFlatL2(3)
```

```
# Add doc vector
```

```
docs = np.array([[0.12, -0.18,  
0.85]], dtype='float32')  
index.add(docs)
```

```
# Search query vector
```

```
query = np.array([[0.1, -0.2,  
0.9]], dtype='float32')  
D, I = index.search(query, k=1)
```

```
print("Closest doc:", I,  
"Distance:", D)
```

Output → Matches our doc:

```
Closest doc: [0]
```

```
Distance: [0.0008]
```

PGVector

metadata in **PGVector** usually refers to **extra fields stored alongside embeddings** that you can filter or join on using SQL. Examples of Metadata for PGVector: Document Metadata, title, author, source_url, Time Metadata created_at, updated_at, year

With PGVector you can do:

```
SELECT content
FROM documents
WHERE topic = 'AI' AND created_at > '2024-01-01'
ORDER BY embedding <-> query_embedding
LIMIT 5;
```

Query: "Explain REST APIs"

query_embedding → [0.07, -0.22, 0.91, ...]

Database finds chunks with embeddings close to it, e.g.: "REST allows communication via HTTP methods"

This combines **semantic similarity** with **structured filters** — super powerful for **metadata-heavy RAG**.

When to use a Vector DB

Semantic Recall

- You need to find text by **meaning**, not just exact keywords.
- *Example:* Query: “cheap flights” → Matches “low-cost airfare”.

Re-ranking / Hybrid Search

- Combine **keyword search (lexical)** + **semantic search (vectors)** for better accuracy.
- *Example:* Search for “Python data analysis” → Lexical hits + semantic matches from tutorials.

Metadata Filters

- Filter results with structured fields like author, date, tags.
- *Example:* “Find research papers about AI published after 2022.”

Graph / Entity memory (relationships)

Graph / Entity Memory (relationships)

- **Tools:** Neo4j, AWS Neptune, TigerGraph
- **RDF triple stores (GraphDB (ontotext)):** when ontologies & reasoning are needed.
- **Use When:**
 - You need to traverse multi-hop relationships: *“User → Project → Constraint”*
 - Answer questions like *“Who connects to what, and why?”*
- **Example:**
 - *Find all developers working on projects that depend on Database X.*

Full-text / Hybrid Search

- **Tools:** Elasticsearch, OpenSearch
- **What they offer:**
 - Keyword + BM25 ranking, Aggregations & filters (facets), Hybrid search (KNN + lexical)
- **Use When:**
 - You need log analytics, dashboards, or combining keyword + semantic search.

Key Idea:

- **Graph DBs** = best when **relationships matter**.
- **Full-text / Hybrid Search** = best when **keywords, filters, and aggregations** are

RDF Triple Stores (GraphDB)

RDF = Resource Description Framework

- Stores knowledge as **triples**:
(**Subject** → **Predicate** → **Object**)
 - Example:
 - (Alice → worksOn → ProjectX), (ProjectX → dependsOn → DatabaseY)
- ♦ **Triple Store**
 - A database optimized for storing and querying these triples.
 - You use **SPARQL** (query language for RDF) to ask complex questions.
- ♦ **Ontologies & Reasoning**
 - **Ontology** = a formal schema of concepts + relationships (like a “knowledge map”).
 - **Reasoning** = the engine can infer new facts from existing triples.
 - If (Alice → worksOn → ProjectX) and (ProjectX → dependsOn → DatabaseY), it can infer (Alice → indirectlyUses → DatabaseY).

Examples of RDF Triple Store Use

1. **Healthcare Ontology**
 - (Patient123 → diagnosedWith → Diabetes)
 - (Diabetes → treatedBy → Metformin)
 - Reasoning: The system can suggest *“Patient123 may need Metformin.”*
2. **Knowledge Graph (Research)**
 - (PaperA → cites → PaperB)
 - (PaperB → authoredBy → Dr.Smith)
 - Query: *“Which authors are indirectly connected to PaperA?”*

Session / Short-term Memory

Session / Short-term Memory = “focus on what’s relevant now, drop the rest.”

- How: In-memory buffers + rolling summaries (framework-level memory objects).
- Use When: Only the last N turns matter → keep tokens/turns low.
- Example: A chatbot that only needs to remember the last 5 exchanges in a support conversation.

Episodic / Run Logs

- How: SQLite / Postgres (append-only tables); ClickHouse / Timescale for high-volume.
- Use When: You need replay, audits, or evaluation traces.
- Example: Store logs of every agent step so you can debug “why did it give that answer?”

Shared State (Agent-to-Agent (A2A) Blackboard): central bulletin board where agents leave notes for each other → enables coordination, modularity, and parallel work.

- How: Redis: Hashes (state), Streams (events), Pub/Sub (handoffs).
 - Postgres: durable tables per agent.
 - Kafka / NATS: event buses for coordination.
- Use When: Multiple agents coordinate, need TTL (time-to-live) + fast reads/writes.
Example: Agent A fetches data → puts it on the blackboard, Agent B reads it → applies reasoning → passes to Agent C. Agent C → picks up results → delivers final output

Key Idea:

- Session memory = short-term focus.
- Episodic logs = accountability & replay.
- Shared state = teamwork across agents.

Quick Rule of Thumb for Choosing a Store

- Need meaning similarity → Vector DB
Find semantically similar chunks (RAG, embeddings).
- Need who-relates-to-whom → Graph DB
Traverse relationships (multi-hop reasoning, ontologies).
- Need filters / aggregations / logs → Elastic / OpenSearch
Keyword + BM25 (Best Match 25, ranking algorithm) + analytics dashboards.
- Need shared fast state → Redis
Ephemeral memory, TTL, pub/sub between agents.
- Need durable structured data → Postgres
Transactions, tables, strong schema.
(+ PGVector if you also want RAG with metadata filters.)
- Need raw blobs → S3 / GCS (Google Cloud Storage) + index elsewhere
Store files, PDFs, videos; keep embeddings/metadata in another DB.

Key Idea:

Each store has a sweet spot → pick based on what kind of memory the

Behavior #3 — Delegation

Break work across **specialized agents** (Example: planner, researcher, coder).

A2A — what is Agent-to-Agent?

Agents message each other to solve tasks, sometimes with a supervisor or a group chat manager deciding who speaks next.

A2A = agents collaborating like a team.

They pass messages, share results, and build on each other's work.

A2A — ping-pong (to & fro context passing)

What Happens:

1. **Agent A** → sends a message with context.
2. **Agent B** → replies, adds reasoning or output.
3. The **chat history/summary** is shared back so both agents see the updated state.
4. Repeat this back-and-forth until a **stop rule** is met (e.g., task complete or max turns).

Example: Debugging a Function

- **Agent A (Coder):** “Here’s the function, but it throws an error.”
- **Agent B (Reviewer):** “The issue is a missing import. Add `import os`.”
- **Agent A:** Updates code, posts new version.
- **Agent B:** Runs tests again, confirms fix.
- Stop when Reviewer approves

Key Idea

Ping-pong = agents **pass context back and forth**, each improving or correcting, until the goal is reached.

Minimal Example

A2A Ping-Pong (Fibonacci Task) : Goal: *“Write Python to compute the 10th Fibonacci number.”*

Turn 1 — Analyst (A): “Plan: Use recursion or iteration. Keep it simple with a loop.”

→ Sends plan to Coder (B).

Turn 2 — Coder (B): “Wrote a loop version. Code runs but missing print statement.”

→ Returns code sketch + status.

Turn 3 — Analyst (A): “Context summary: last turn = loop works, missing print.

TODO: add print.” → Suggests fix.

Turn 4 — Coder (B): “Updated code. Now prints the 10th Fibonacci number = 55.”

→ Reports success.

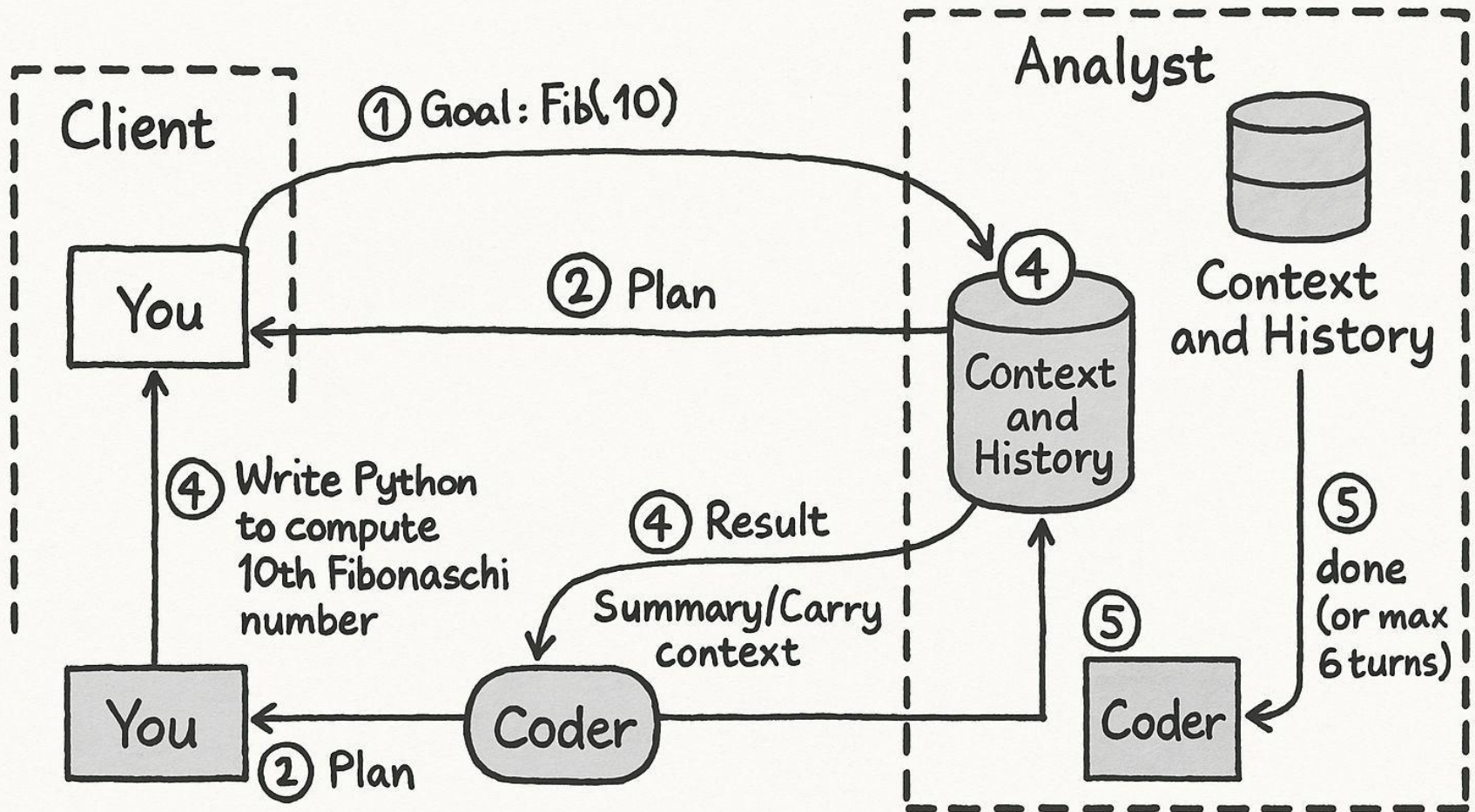
Carry Context Each Turn: Summarize what changed + open TODOs → keeps loop efficient.

Stop Rule:

- End when Coder outputs final working code
- Or max 6 turns to prevent infinite ping-pong.

Ping-Pong A2A = goal → back-and-forth refinement → stop when done.

A2A — minimal “ping-pong”



A2A with graphs — LangGraph supervisor pattern

- Agents are graph nodes (Planner, Researcher, Coder, Reviewer).
- A **Supervisor LLM** is a routing node: it reads current state and decides the next node.
- Edges encode possible next hops; the graph executes step-by-step until a stop rule.

What state does the next agent see?

You choose the state model. Two common patterns:

1) Pass messages (chat-style state) State: `messages[]` (running conversation) + optional `scratchpad`.

- Next agent sees: the latest summarized transcript (or last N turns) plus any role tags.
- Pros: Natural, debuggable, model-agnostic.
- Cons: Token growth; you must summarize/trim.

2) Pass tool args / structured fields (form-style state)

- State: typed fields the next agent needs (e.g., `papers[]`, `summaries[]`, `endpoint_spec`).
- Next agent sees: only the fields needed; not the whole chat.
- Pros: Compact, deterministic, great for long pipelines.
- Cons: Less conversational flexibility; you must design the schema.

```

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0) parser =
StrOutputParser()

# Prompts
researcher_prompt =
    ChatPromptTemplate.from_messages([
        ("system",
            "You are a precise researcher. Extract 3-5 concise facts relevant to the
            user's goal. "
            "Use short bullets only. No preamble, no summary."), ("human",
            "Goal: {goal}\n\nReturn only bullet facts.")
    ])

writer_prompt = ChatPromptTemplate.from_messages([
    ("system",
        "You are a clear, concise technical writer. Using the provided facts, "
        "write a 3-4 sentence summary for a technical audience. End with
        one caution."),
    ("human", "Goal: {goal}\n\nFacts:\n{facts}\n\nWrite the final
        summary.")
    ])

```

Output

=== TRACE (who said what) ===

01 [22:04:37] [USER] (goal)

Goal: summarize agent-to-agent (A2A) interactions

02 [22:04:37] [SUPERVISOR] (route)

Routing to researcher (facts first).

03 [22:04:38] [RESEARCHER] (facts)

- A2A interactions involve direct communication between autonomous agents without human intervention.
- These interactions can enhance efficiency in tasks such as data sharing, decision-making, and resource allocation.
- A2A communication protocols often utilize standards like MQTT or CoAP for lightweight messaging.
- Security in A2A interactions is critical, often implemented through encryption and authentication mechanisms.
- A2A systems are commonly used in IoT applications, robotics, and distributed AI networks.

Output (Continued)

04 [22:04:38] [SUPERVISOR] (route)

Facts gathered; routing to writer.

05 [22:04:40] [WRITER] (final)

Agent-to-agent (A2A) interactions facilitate direct communication between autonomous agents, enhancing efficiency **in** tasks such as data sharing,

decision-making, and resource allocation.

These interactions typically employ lightweight messaging protocols like MQTT or CoAP,

while prioritizing security through encryption and authentication mechanisms.

A2A systems are prevalent **in** IoT applications, robotics, and distributed AI networks. Caution should be

exercised regarding potential vulnerabilities **in** security implementations, as they can expose systems to risks.

Output (Continued)

2025-08-15T10:12:01.433Z [supervisor] in:

messages=[Human:"Goal: summarize X"]

2025-08-15T10:12:01.434Z [supervisor] decide:

rationale="No facts yet; route to researcher to gather key points."

goto=researcher

2025-08-15T10:12:01.612Z [researcher] prompt:

"Provide 3-5 concise, verifiable facts about agent-to-agent (A2A) LLM systems: typical topologies, pros/cons, and when to use them. Keep bullets tight."

2025-08-15T10:12:02.128Z [researcher] out:

- "A2A coordinates multiple specialized agents (planner, coder, critic) via message passing."
- "Common topologies: supervisor (router), peer-to-peer, and hierarchical graphs."
- "Pros: specialization, modularity, clearer control; Cons: latency, cost, loop risk."
- "Guardrails and a shared context/state store reduce drift and repeated work."
- "Frameworks such as LangGraph and AutoGen provide routing, memory, and tooling." update={"messages": ["FACTS:
 - A2A coordinates specialized agents via messages.
 - Topologies: supervisor, peer, hierarchical.
 - Pros: modularity/specialization; cons: latency/cost/loops.
 - Guardrails + shared context mitigate risks.
 - LangGraph/AutoGen are common frameworks."]}

Output (Continued)

```
2025-08-15T10:12:02.129Z    [supervisor]  in:
    messages=[..., "FACTS: ..."]
2025-08-15T10:12:02.129Z    [supervisor]  decide: rationale="Facts
    gathered; hand off to writer for synthesis."
    goto=writer

2025-08-15T10:12:02.471Z    [writer]  prompt:
    "Write a 3-4 sentence summary of 'agent-to-agent (A2A) interactions' for a
    technical audience, using the provided facts. End with one caution."
2025-08-15T10:12:02.997Z    [writer]  out:
    "Agent-to-agent (A2A) systems coordinate multiple specialized LLM agents through
    message passing to solve complex tasks. Common topologies include a supervisor that
    routes turns, peer-to-peer swarms, and hierarchical graphs. This design improves
    modularity and lets each agent focus on planning, coding, or critique, but it
    increases latency/cost and can loop without guardrails and a shared context store.
    Always add stop rules and tracing."
    update={"messages": ["Final: Agent-to-agent (A2A) systems coordinate multiple
    specialized LLM agents... Always add stop rules and tracing."]}

2025-08-15T10:12:02.998Z    [engine]  done:
    returned_state_keys=["messages"]
```


Output (Final Summary)

```
{
  "messages": [
    "Goal: summarize A2A behaviour",
    "FACTS:\n• A2A coordinates specialized agents via messages.\n• Topologies:

supervisor, peer,
hierarchical.\n
  • Pros: modularity/specialization;

cons: latency/cost/loops. Guardrails + shared context mitigate risks.\n LangGraph/AutoGen are common frameworks.",
    "Final: Agent-to-agent (A2A) systems coordinate multiple specialized LLM agents
    through message passing to solve complex tasks.

Common topologies include a supervisor that routes turns, peer-to-peer swarms, and hierarchical graphs.

This design improves modularity and lets each agent focus on planning, coding, or critique,
but it increases latency and cost and can loop without guardrails and a shared
context store. Always add stop rules and tracing."
  ]
}
```

Guardrails for agent behavior & A2A chats

Add programmable guardrails between app code and the LLM.

Similar to a single agent situation: topic control, jailbreak detection, PII redaction.

Topic control / policy gates

- Allowlist: “Only answer about course topics: HTML, REST, FastAPI...”
- Denylist: Block “malware creation,” “grading policy leaks,” etc.

Jailbreak & prompt-injection detection

- Classifier/regex on inputs: flag “ignore previous instructions,” “system override,” etc.
- If flagged → route to safe reply or human review.

PII detection & redaction

- Detect emails, phone #s, SSNs; replace with [REDACTED].
- Example: Before an agent posts to group chat, scrub PII.

A2A specifics

- Supervisor uses guardrails to decide who speaks next (route only to allowed nodes).
- Ping-pong chats: run both directions through guardrails (A→B and B→A).
- Shared state/blackboard: scrub PII on write; enforce TTL, size limits, and field schema.

Guardrails sits between your app and the LLM.



```
DENY = {"passwords", "credit card"}  
def allow(user_msg: str) -> bool:  
    return not any(bad in user_msg.lower() for bad in DENY)  
  
msg = "Can you store my credit card?"  
print("Blocked" if not allow(msg) else "Allowed")
```

Evaluate A2A behavior

1) Traces (who spoke, when, why)

- Log per hop: {ts, speaker, input_hash, output_hash, tools_used, reason}
- Reason strings: short “why I acted” notes (route decisions, tool choices).
- Example row:
12:01:04 | Supervisor -> Coder | reason="needs code for /summarize"

2) Success criteria

- Task finished? Stop rule met (status=done, artifact produced, tests pass)
- Tool errors? Rate, types, retries, backoffs
- Loops avoided? Max turns not exceeded; no oscillation ($A \leftrightarrow B \rightarrow A \leftrightarrow B$ with no state change)
- Latency & cost: per-agent, per-tool, per-episode budgets

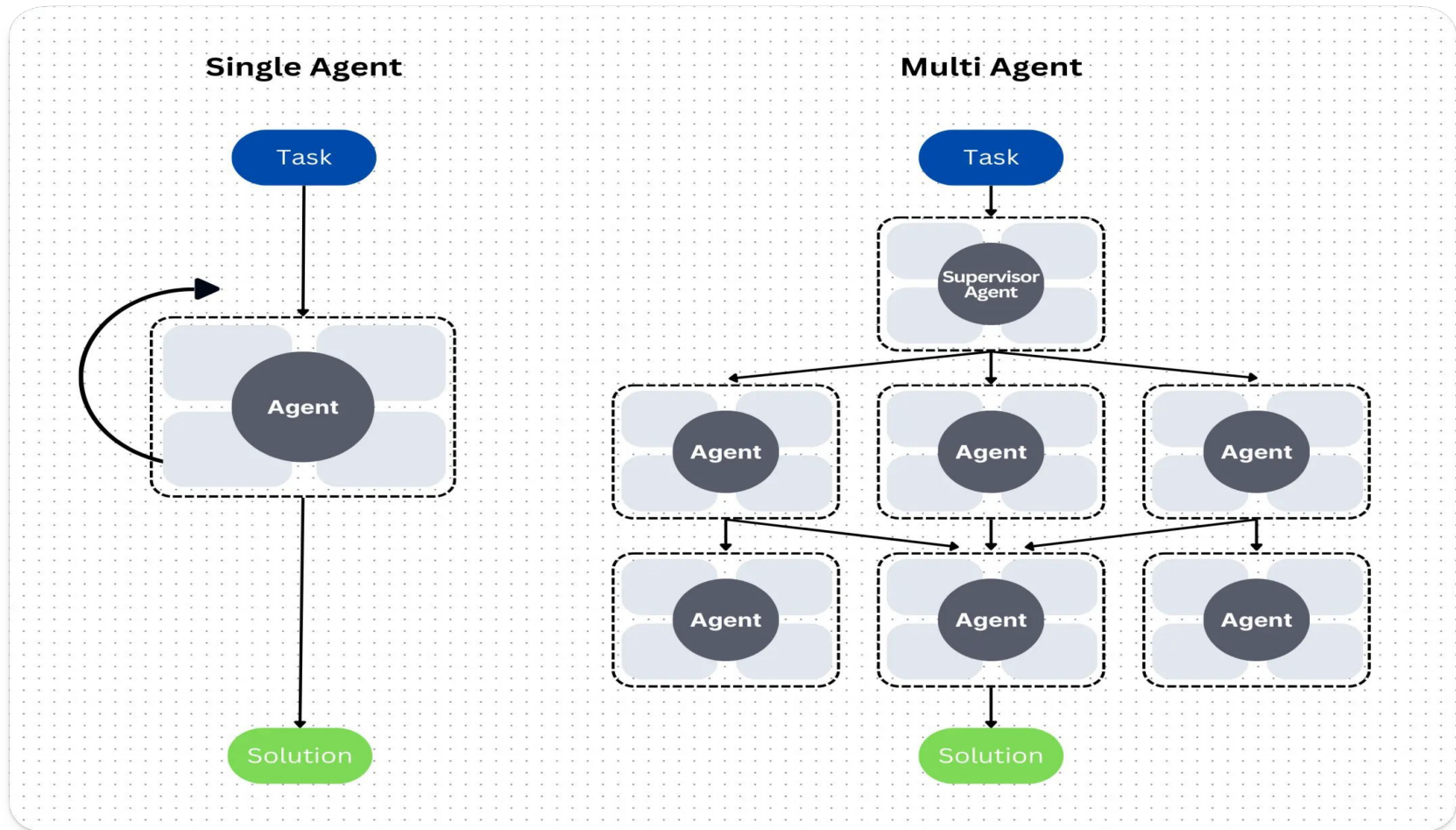
3) Unit tests for agents (role fixtures, a fixture = predefined setup or data used for repeatable tests)

- **check: pass fixtures through each role?**
- Planner: given a goal fixture, returns ordered steps with acceptance criteria
- Researcher: given a query fixture, returns sources + citations (schema-validated)
- Coder: given a spec fixture, returns runnable code + tests
- Reviewer: given outputs, returns pass/fail + actionable diffs
- Tip: run these headless (no other agents), then in integration with the graph.

4) Spot-check ReAct (reasoning \leftrightarrow action alignment)

- **Check: did reasoning justify the action?**
- Red flags: “I’ll browse” with no freshness need; calling Python for trivial math; hallucinated sources.
- Pass example:
Reason: “Stock price requested for today \rightarrow knowledge may be stale \rightarrow browse web.”

the multi-agent system preferred over the single-agent system



One agent, powered by an LLM, analyzes survey responses for sentiment and nuances, detecting elements like frustration or satisfaction.

Another agent, specialized in crafting dynamic queries, generates follow-up questions aligned with the detected sentiment, user profile, and product feature.

For instance, if frustration about a product feature is identified, the follow-up might delve deeper into the specific pain point that applies to the user profile.