Submitted by -Parth

Lecture Notes

Lecture-1: System Design Part -1

Sir's video really helped me understand how large apps and websites work behind the scenes. Although many of these topics were introduced during my 3rd year of B.Tech in DSAI, this session helped me go deeper and connect the dots better.

It began with the basics—client-server communication and how DNS acts like the internet's phonebook, translating domain names into IP addresses.

Things got more interesting with scaling. I used to think vertical scaling (upgrading a single server) was the go-to, but now I see why real-world systems rely on horizontal scaling with load balancers to handle growing traffic efficiently. Of course, as the instructor mentioned, this introduces its own overhead like every other tech decision.

The explanation of microservices was clear and easy to understand. Breaking a monolith into smaller, independent services—like login, orders, or payments—makes them easier to scale and maintain.

And the API Gateway acts like a smart router at the front door, directing requests to the right microservice.

What was new for me was the power of queue systems for asynchronous communication. Instead of tightly coupling services (e.g., payment waiting for email), queues allow them to pass tasks efficiently, improving fault tolerance and throughput.

I also learned about fan-out architecture, where one event (like a user placing an order) can trigger multiple downstream services—like sending an email, updating inventory, and logging activity—all at once.

This tied in perfectly with the publish-subscribe (pub/sub) model, where services can "subscribe" to events they're interested in. It’s a clean, decoupled way to design systems and opens up a lot of flexibility.

Overall, this was my first real exposure to System Design, and it made the whole subject feel much less intimidating. It showed me that building scalable systems isn't magic—it's just thoughtful architecture and clean communication between components.

Lecture-2: System Design Part -2

I learnt that system design is context-dependent—there is no fixed solution. The right architecture depends on traffic patterns, scalability, fault tolerance, and cost.

Sir compared 3 platforms:

- Netflix has predictable traffic and uses pre-scaling and CDN caching before content releases.

- YouTube handles unpredictable surges from viral content and live streams, requiring fast, reactive scaling.

- Hotstar during live sports is a hybrid case—match times are predictable, but in-game events cause sudden spikes. The "back button problem" shows how traffic on one service can overload another. To manage this, both the live-stream and home page services are pre-scaled, and auto-scaling is disabled to avoid risky scale-downs.

Traditional scaling methods (like using average CPU thresholds) are too slow. This led to a discussion on serverless architecture, especially AWS Lambda, which auto-scales on demand with no server management. It works well for unpredictable traffic but has downsides: cold starts, runtime limits, DDoS cost risks, and vendor lock-in (as using Lambda often leads to reliance on other AWS services like API Gateway, SQS, S3, CloudWatch, etc.).

The key takeaway is that System design must be based on actual usage patterns, and while serverless can be powerful, it requires careful consideration of trade-offs.

Lecture-3: Kafka Architecture and Working

I learnt that Apache Kafka, an open-source event streaming platform used for handling real-time, high-throughput data. It is widely adopted by companies like Uber, Zomato, and Discord to manage scenarios where data is generated faster than a traditional database can handle.

For example, inserting GPS updates or chat messages directly into a database can cause high latency and crashes due to excessive writes.

Kafka acts as a buffer between data producers and consumers. Data-producing apps (like a driver's app) send messages to Kafka, which stores them temporarily. Consumer apps (like analytics or maps) read the data at their own pace and perform bulk database inserts, reducing load and improving performance.

Kafka follows a producer-consumer architecture and organizes data into topics. Each topic can be split into partitions to allow parallel processing and improve scalability. This enables different consumers to read from different partitions simultaneously.

Kafka is not a database. It stores data temporarily and does not support complex queries. Its purpose is to move data efficiently between services.

The Key things learnt were using Kafka for high-speed data ingestion, while using normal databases for storage. Decoupling system components, and leveraging topics and partitions for better performance and scale.

Lecture-4: Master Role-Based Access Control Patterns

I understood the difference between authentication and authorization. Authentication is about verifying a user's identity, while authorization checks what actions the user is allowed to perform. While authentication is usually simple to implement, authorization can become complex as application requirements grow.

Sir explained several patterns of access control, each suited for different use cases and levels of complexity:

- RBAC (Role-Based Access Control)**:** Assigns one global role per user, such as admin or editor. Easy to implement but not flexible for multi-tenant systems.

- Fine-Grained RBAC**:** Assigns roles based on specific resources. For example, a user may be an admin of one blog but only a viewer of another.

- ABAC (Attribute-Based Access Control)**:** Grants access based on user, resource, and environment attributes like team, role, or project tags.

- PBAC (Policy-Based Access Control)**:** Externalizes access rules into centralized policy files, making the system more maintainable and flexible.

- ReBAC (Relationship-Based Access Control)**:** Grants permissions based on relationships, such as folder hierarchies in Google Drive.

I also learned about Google's Zanzibar system, which was built to handle complex, large-scale authorization problems like ReBAC efficiently. The key

takeaway was to start with simple access control methods, plan for scale early if needed, and consider using mature solutions like Zanzibar-inspired tools instead of building complex systems from scratch.

Lecture-5: Kubernetes

I learned how Kubernetes addresses the complexities of deploying and managing applications at scale. Piyush sir began with explaining traditional deployment on physical servers, where manual setup made consistency difficult and scaling required hardware upgrades. AWS came and improved this by offering managed services and dynamic scaling, but environment mismatches still caused problems.

To solve that, containerization with tools like Docker was introduced. Containers package applications with all dependencies, making them lightweight, portable, and consistent across environments. However, managing large numbers of containers manually is not practical.

Google had already built an internal system called Borg to solve this at scale, and the team that built it, developed Kubernetes from scratch based on that experience.

Kubernetes is an open-source system developed by Google to automate the orchestration of containerized applications. Kubernetes manages deployment, scaling, self-healing, and monitoring using the desired state model—where I define what I want, and it ensures the system stays in that state.

I also learned about Kubernetes' architecture, which is split into two parts: the Control Plane(API Server, Scheduler, Controllers), which makes all decisions, and the Worker Nodes(Kubelet, Kube-proxy, Container Runtime), which run the actual containers. This design ensures reliable, scalable, and consistent application deployment across various environments.

I also learned about the Cloud Controller Manager, which allows Kubernetes to work across cloud providers.

Lecture-6: Scalable Notification Design System

I learned how to build a scalable, efficient, and user-friendly notification system. Piyush sir discussed the problem with synchronous notification sending, where the main application directly calls third-party services like SendGrid or Twilio. This approach slows down the application, increases cost, and doesn't scale well.

To solve this, the system shifts to asynchronous processing. Instead of sending notifications immediately, the app publishes events to a message queue like Kafka. This allows the main application to stay fast and offload the notification responsibility.

Next, a set of "smart consumers" pick up the events from the queue. These consumers perform initial filtering (e.g., transactional vs promotional), check user preferences, and determine the appropriate channels (email, SMS, in-app). They also generate the final message content.

To handle rate limits from providers, messages are placed into channel-specific queues, like email_queue or sms_queue, rather than being sent directly. Throttled dispatchers then process these queues, sending messages in a controlled manner to avoid getting blocked.

Advanced optimizations include:

- User presence detection: Only send SMS or email if the user is offline.
- Batching/Digesting: Combine similar messages into one to reduce noise and cost.
- Priority queues: Separate high-priority messages (like OTPs) from low-priority ones (like promotions), ensuring timely delivery of critical alerts.

Lecture-7: Host our Own Browser and OS

I learned how to run a full web browser or desktop operating system inside a Docker container on a remote server, and stream its interface directly to my local browser.

The main idea was to create a secure, isolated, and disposable environment for browsing or testing, which is useful for privacy, sandboxing, geo-unblocking, and temporary sessions.

Sir's setup used an AWS EC2 instance as the remote server, with Docker to run containerized GUI apps. The core tool is Kasm Workspaces, which provides ready-to-use Docker images for browsers and desktop environments with built-in VNC access.

After launching an EC2 instance, Sir has us install Docker, run a Kasm container (e.g., Vivaldi browser), and open port 6901 to access it via web browser. Once connected, I could use the remote browser as if it were local, but with the IP and location of the server.

I also learned about a SaaS business idea built around this concept—creating a user-friendly platform where users can launch secure browser or OS sessions with one click, without needing technical knowledge of AWS or Docker.

This could be monetized by charging per session, making it a valuable and scalable service.

Lecture-8: Build our Own VPN

I learned how to build my own VPN using OpenVPN and host it on AWS. The session began by explaining what a VPN is and why it's useful. Normally, when I browse the internet, my data travels directly through my Internet Service Provider (ISP), which means both my IP address and browsing activity can be tracked by websites, advertisers, and the ISP itself.

This can lead to privacy concerns, location tracking, and even content restrictions if certain sites are geo-blocked. A VPN solves this by creating an encrypted tunnel between my device and a remote server. My data travels securely through this tunnel, hiding my real IP address and replacing it with the VPN server's IP, which improves anonymity and helps bypass geo-restrictions.

Next, I followed the step-by-step process of setting up the VPN server. Sir showed us launching an EC2 instance on AWS, choosing the "OpenVPN Access Server" image from the AWS Marketplace. We selected a region different from our own for geo-unblocking benefits and chose an appropriate instance type. After configuring the key pair and launching the instance, We connected to it via SSH using the terminal. During the first login, an automatic OpenVPN setup script guided me through important configuration steps, such as accepting the license, choosing ports, enabling traffic routing, and creating an admin user and password.

Once the server was ready, we accessed the Admin Web UI using the public IP of the EC2 instance, logged in, and verified the VPN settings. Then we visited the Client UI page, downloaded the recommended OpenVPN client for my operating system, and connected using the credentials we had set. The client was pre-configured with my server settings, making connection easy. After successfully connecting, We verified that my IP had changed to reflect the server's location, confirming that all my traffic was now being routed securely through the VPN.

Finally, I learned about the importance of terminating the EC2 instance when not in use to avoid ongoing AWS charges. Overall, this video helped me understand how VPNs work, their advantages over commercial services, and how I can build and use my own secure, private, and customizable VPN infrastructure using only free tools and cloud services.

Lecture-9: Docker Management API

We learned how Docker orchestration works and how to implement a basic orchestration system using Node.js and the Docker Engine API. Docker orchestration refers to the automated management of container lifecycles, including tasks such as creating, scaling, and terminating containers.

Sir presented a practical use case of building a cloud-based playground where each user receives an isolated Docker container environment on demand.

We learned that the Docker CLI communicates with the Docker Daemon via a Unix socket (/var/run/docker.sock), and that applications can interact with Docker by sending HTTP requests or using libraries like dockerode. A Node.js server was set up using Express and dockerode to expose RESTful API endpoints such as GET /containers to list running containers and POST /containers to create new ones.

The server dynamically assigns an available host port to each new container, pulls the required image, creates the container with specific configurations (like enabling TTY and running a command such as /bin/bash), and then starts the container.

Sir also explained how to manage container sessions efficiently. By maintaining an in-memory mapping of active ports and their associated containers, the system can identify and remove containers once a user session ends.

Finally, he proposed a reverse proxy strategy to improve user experience, where traffic is routed through a single port using subdomains instead of exposing multiple host ports directly. This approach helps build a scalable, user-friendly cloud playground using Docker and Node.js.

Lecture-10: Build your Own Docker Hub

I learned how to build a secure, private Docker image registry from scratch, similar to Docker Hub. The process began by understanding what a Docker registry is—a centralized server to store and distribute Docker images—and why organizations use private registries for better security and control.

The goal was to deploy my own registry on an AWS EC2 instance and make it publicly accessible through a domain with HTTPS and authentication.

To implement this, I launched an Ubuntu-based EC2 instance and installed Docker. Then, using docker-compose, I configured and ran Docker's official registry:2 image to serve as the backend registry service.

I created a persistent storage directory for image layers and used Nginx as a reverse proxy to route incoming requests to the Docker Registry container.

I also set up a domain name pointing to the server and used Certbot with Nginx to enable HTTPS using a free SSL certificate from Let's Encrypt. To ensure that only authorized users could push and pull images, I implemented basic HTTP authentication using htpasswd and updated the Docker Compose file accordingly. Finally, I tested the setup by logging in from my local Docker client, tagging an image, and pushing it to my private registry.

Sir taught us the full architecture and deployment steps involved in self-hosting a secure Docker registry, including domain setup, reverse proxy configuration, SSL, and authentication.

It gave me hands-on knowledge about managing Docker images privately without relying on third-party platforms.

Lecture-11: How SSL Certificate Works?

I learned how SSL certificates ensure secure communication over the internet. It began by explaining the problem with HTTP, where data travels unencrypted and can be intercepted by hackers. To address this, encryption is necessary.

I first learned about symmetric encryption, where the same key is used for both encryption and decryption. However, the challenge is how to securely share that key with the server without exposing it to hackers. This is known as the key exchange problem.

Next, the video introduced asymmetric encryption, which uses a public-private key pair. The server sends its public key to the client, which then encrypts a symmetric key and sends it back. The server uses its private key to decrypt it. This way, both sides share a symmetric key securely, which can then be used for fast, encrypted communication.

I also learned about a major risk—the imposter server problem. A hacker could pretend to be the server and send their own public key to the client. This would allow them to decrypt all data.

To solve this, the video explained the role of SSL certificates and Certificate Authorities (CAs). A CA verifies the identity of a website and issues a signed certificate. When my browser connects to a site, it checks this certificate using the CA's public key. If the certificate is valid, the browser trusts the server's public key.

Finally, I learned about self-signed certificates, which are useful for local testing but not trusted by browsers because they're not issued by a recognized CA. Sir clearly explained how encryption, identity verification, and trust work together to make HTTPS secure.

Lecture-12: Complete Git & Github Tutorial

I gained a strong understanding of Git as a distributed version control system. I learned that Git allows me to track, manage, and revert changes made to code over time. This is especially useful in software development, where keeping a clean history of changes is critical for collaboration, debugging, and maintaining code quality.

I learned how to install Git and configure it globally using git config --global, which sets my name and email so my commits are properly identified. I then understood how to initialize a Git repository using `git init`, which creates a hidden `.git` folder to begin tracking project changes.

The core Git workflow became clear to me: first, I modify files in my working directory, then use `git add` to stage them, and finally, use `git commit -m "message"` to record the changes permanently in the repository.

The concept of the Staging Area was particularly helpful—it allows me to selectively group related changes before committing them. This means I can commit only the files I want, even if I modify more.

I also explored important Git commands:

- `git status` helps me see which files are tracked, untracked, or staged.
- `git log` and `git log --oneline` allow me to view the commit history.
- `git diff` shows the line-by-line changes before committing.
- `git show` displays the full details of a specific commit.
- `git blame` helps identify who last changed each line in a file.

Another important lesson was how to undo mistakes. I learned the difference between `git reset` (which moves the HEAD pointer and removes commits—destructive) and `git revert` (which safely creates a new commit to reverse previous changes—non-destructive).

Understanding HEAD as the pointer to the latest commit helped me better visualize the flow of changes.

Finally, the video introduced me to the Git Graph extension for VS Code, which provides a visual interface for commits and branches. As a practice assignment, Sir encouraged us to create a repo, make several commits, and try using `reset`, `revert`, and `blame` to reinforce my learning.

Sir laid a solid foundation for confidently using Git in local development.

Lecture-13: Docker in One shot - Part 1

I understood the real meaning behind the phrase *"It works on my machine."* It describes a common problem in software development where an application works perfectly on one developer's system but fails on another. This usually happens due to differences in software versions, operating systems, or missing dependencies.

Docker addresses this problem by introducing the concept of containers. A container includes everything the application needs to run—such as the code, system libraries, dependencies, and environment configuration. This ensures that the application behaves the same on any machine, whether it's a developer's laptop, a test server, or a production server.

The video explained key Docker components like images (blueprints) and containers (running instances), and demonstrated important commands such as `docker run`, `docker ps`, `docker stop`, and `docker exec`. It showed how to map ports so that apps running inside containers can be accessed from the host system, and how to use environment variables when running containers.

I also learned how to write a `Dockerfile` to create a custom image for a Node.js app, how to build the image using `docker build`, and how to upload it to Docker Hub using `docker push`. Finally, the video introduced Docker Compose, a tool used to manage multi-container applications with a single configuration file.

Overall, the video helped me clearly understand how Docker simplifies development, avoids environment issues, and supports consistent deployment across systems.

Lecture-14: Docker for Open-Source Contributors

I learned advanced Docker concepts that are important for real-world and open-source development workflows. It began with Docker networking, explaining how containers connect to the internet using the default bridge network. Each container gets a private IP and can communicate through the host.

I also explored host network mode, where the container shares the host's network directly, making services accessible without port mapping, and none mode, where the container has no network access at all. I understood the benefits of custom networks, where containers can communicate with each other using container names—useful in multi-container setups like connecting a web app to a database.

Next, I learned about data persistence using volumes. Since containers are temporary and delete all data when stopped or removed, I now know how to use the `-v` flag to mount a host directory into a container. This way, data remains available even after the container is deleted.

The video also covered Docker build optimization, particularly how Docker uses a layer caching mechanism. I learned the importance of copying `package.json` and installing dependencies *before* copying the rest of the code in a `Dockerfile`, to avoid unnecessary rebuilds and make the process faster. I also learned to use `.dockerignore` and `WORKDIR` to keep builds clean and efficient.

Finally, the video introduced multi-stage builds, which help keep final Docker images small and secure by separating the build environment from the runtime environment. The first stage compiles or builds the app, while the second stage contains only what's needed to run it.

This approach prevents build-time tools from bloating the final image. Overall, this session gave me a practical understanding of networking, volume management, and build optimization in Docker.

Lecture-15: Redis Crash Course

I learned how Redis addresses a common performance issue in web applications—the repeated execution of expensive database queries. Typically, every time a user requests a page, the server retrieves data such as profiles, messages, or feeds by querying the database, which increases load and latency. Redis solves this problem through in-memory caching.

It stores frequently requested data in RAM, allowing the server to quickly retrieve it from Redis instead of the slower primary database. This significantly reduces response times and improves user experience. I also learned about cache misses and hits.

On a miss, data is fetched from the database and cached in Redis. On subsequent hits, Redis serves the data directly. Additionally, Redis supports setting expiry times for cached keys, ensuring that stale data is automatically removed and refreshed when needed.

The video then introduced Docker as a way to easily run Redis locally. Using the redis/redis-stack image and Docker CLI, I was able to run Redis and access its graphical interface, RedisInsight, for real-time data management. Through the CLI, I learned to use fundamental commands such as *SET*, *GET*, and *EXPIRE*, and understood that Redis stores data in various structured formats.

These include strings for basic key-value pairs, lists for ordered data (like queues), sets for unique elements, hashes for storing object-like data, and sorted sets for ranking items with scores. Each type has specific use cases depending on the data access pattern required.

Finally, I saw a practical Node.js implementation using the `ioredis` library. A route handler first checked Redis for cached API results. If available, the data was returned instantly; if not, it was fetched from the original API, cached in Redis with a TTL, and then returned to the user.

This pattern dramatically reduced the response time from multiple seconds to a few milliseconds. Overall, Sir gave us a clear understanding of Redis's role in enhancing performance, how to run and interact with it, and how to apply it effectively in a real-world project.

Lecture-16: WebSocket in Node.js

I learned the limitations of traditional HTTP for real-time applications and how WebSockets solve those problems. In HTTP, communication is one-way and short-lived—the client sends a request, the server responds, and then the connection closes. This makes it difficult to build features like live chat, where the server needs to push updates to clients without waiting for a request. The common workaround—HTTP polling—is inefficient, as it sends frequent empty requests, increasing server load and wasting resources.

WebSockets provide a better solution. They start as an HTTP request and then upgrade to a persistent, full-duplex connection, allowing the client and server to send messages back and forth freely without re-establishing connections. This enables real-time communication with low latency.

To implement this, I used Socket.IO, a library that simplifies working with WebSockets in Node.js. On the server side, I created an Express app and attached Socket.IO to an HTTP server. On the client side, I connected to the server using the `io()` function. We used `emit` to send custom events and to listen for them. For example, when a user sends a chat message, the client emits a `user-message`
event. The server receives it and broadcasts it to all clients using `io.emit`. All connected clients then listen for a `message` event and display it instantly.

This setup enabled me to build a basic real-time chat system where messages are sent and received instantly across clients. It demonstrated how WebSockets and Socket.IO can create smooth, real-time interactions without relying on inefficient techniques like polling.

Lecture-17: Creating a Discord Bot in Node.js

I learned how to create and deploy a basic Discord bot using Node.js and the `discord.js` library. I began by setting up the necessary environment through the Discord Developer Portal. This involved creating a new Discord application, converting it into a bot, assigning it the required permissions, and then inviting it to a test server I created.

Once the bot was added to the server, I moved on to the coding part. I initialized a Node.js project and installed the `discord.js` package. I learned how to use the bot token securely to log into Discord, and how to use **intents** to control what events the bot is allowed to listen to. For example, I enabled the ability to access message content and listen for new messages using the `messageCreate` event.

To make the bot respond to messages, I wrote logic to detect incoming text and reply with a simple message, while ensuring it doesn't reply to its own messages to prevent an infinite loop. I also learned how modern bots now rely on slash commands, like `/ping`, which are more structured and user-friendly. I used a separate script to register such commands with Discord's API and listened for interaction events to handle user inputs.

The video concluded with a project idea—extending the bot to create short URLs using a MongoDB database. This gave me a sense of how bots can interact with external APIs and perform real-world tasks. Overall, this tutorial provided a clear and practical foundation for building interactive Discord bots.

Lecture-18: Cookies in Node.js

I learned how authentication tokens are securely managed in web applications, particularly using cookies and authorization headers. Initially, I understood the problems with stateful authentication using databases—mainly high latency and increased costs—since each request needs a database call to verify the session.

To solve this, stateless authentication using JWT (JSON Web Tokens) was introduced. The challenge, however, was how to transport the token between client and server.

I explored two main methods. First, with cookies, I learned that once a server sends a JWT as a cookie, the browser stores it and automatically sends it back with every request. This method is simple and secure, especially when using `HttpOnly` and `Secure` flags. However, cookies are browser-specific and prone to CSRF if not handled correctly.

The second method used authorization headers. Here, the server sends the token in the response body, and the client must store it (usually in local storage) and manually send it with each request in the `Authorization: Bearer <token>` header.

This method is more flexible and works with mobile apps and APIs too, but it requires more effort from the developer and is vulnerable to XSS if not secured properly.

Overall, I gained a clear understanding of when and why to use cookies versus headers, depending on the type of application and security considerations.

Lecture-19: Server-Side Scripting with EJS and Node.js

I learned about Server-Side Rendering (SSR) and how to implement it using EJS with Node.js and Express. SSR is a method where the server prepares a complete HTML page, including dynamic data, and sends it to the browser.

This allows the page to load with all the required content without relying heavily on client-side JavaScript.

The video explained that writing HTML directly in route handlers using string concatenation is not efficient and can lead to unstructured code. To handle this more effectively, EJS is used as a templating engine that separates HTML structure from server-side logic.

I set it up by installing the `ejs` package, configuring the view engine, and defining the location of the `views` folder.

Using the `res.render()` method, I was able to send EJS templates to the client and pass dynamic data from the server into them. Inside these templates, I used `<%= %>` to display values and `<% %>` for writing JavaScript logic like loops and conditions. This approach helped keep the code organized.

As part of the demonstration, I built a basic form for a URL shortener. The form accepted a URL from the user, and on submission, the server created a short ID and returned the same page with the generated short URL displayed. This process was handled entirely on the server using EJS and Express.

Lecture-20: Model Context Protocol(MCP) Server

II learned about the Model-Context Protocol (MCP), a standard created by Anthropic to help Large Language Models (LLMs) like Claude or GPT access external data and tools in a structured way. LLMs are powerful, but they have limitations — they cannot access real-time information, and they have a limited context window. MCP solves these problems by defining a protocol through which AI models can fetch just the right amount of information from external sources, at the right time.

MCP consists of three parts: the MCP Host (such as a code editor or AI interface), the MCP Server (a program that provides external data or functionality), and the MCP Client (which connects the two). I learned that the server can expose "tools" — like `get_weather_by_city` — and the host can call these tools when needed. For example, if a user asks "What's the weather in Patiala?", the AI model uses MCP to find and call the appropriate server tool, which fetches data from a weather API and sends it back. The model then uses that data to generate a response.

There are two ways the server and host can communicate: STDIO, which is used locally via standard input/output (like a terminal), and SSE (Server-Sent Events), which is used over the internet via HTTP. STDIO is easier to use for local setups, while SSE is more flexible for remote tools.

I also learned how to build a basic MCP server in Node.js using the `@mcp/core` package. I created a tool, defined its input using the Zod library, and wrote a function to return mock weather data. Finally, I connected this server to an MCP-compatible app like Cursor IDE.

This video helped me understand how MCP can bridge the gap between LLMs and the external world in a clean and efficient way. It allows developers to enhance AI applications by securely and modularly connecting them to real-time tools and data sources.

Langchain

Source :- Building AI Agents with LangChain, LangGraph, and LangSmith by Piyush Garg sir.

Summary

Piyush Sir's video helped me understand how to build more advanced AI applications by using three important tools: LangChain, LangGraph, and LangSmith. I learned that relying on simple LLM prompts is not enough for complex tasks. Real-world applications often need multiple steps, context memory, decision-making, and the use of external tools. These requirements are addressed well by the LangChain ecosystem.

First, LangChain made it easier to connect with different LLMs and provided helpful tools like document loaders and vector store integrations. It helped manage inputs, outputs, and tool wrappers in a consistent way.

Next, LangGraph allowed me to design the agent's logic as a graph. Each node in the graph performs a task, and the system moves between them based on the AI's decisions. This structure helps the agent manage complex tasks in a flexible and repeatable way, using a shared state that stores messages and results.

Finally, LangSmith provided full traceability. It allowed me to see each step of the agent's actions, including LLM calls and tool executions. This was very helpful for debugging and understanding the agent's reasoning.

Sir showed a simple calculator agent that solved a multi-step arithmetic task. Watching the LLM and tools work together in a loop, and seeing every step visually in LangSmith, gave me a clear understanding of how these tools combine to build intelligent and maintainable AI agents in JavaScript.

Pydantic AI

Source :- Complete Pydantic Course in Hindi by Chai and Code

Summary

This Video helped me clearly understand how to use Pydantic in Python for data validation and type enforcement. Python is a dynamically typed language, which can often lead to unexpected data issues. Pydantic solves this by allowing us to define data models with type hints, and it automatically checks and converts input data to match those types.

At the beginning, I learned how to set up Pydantic using `pip` or `uv`, and how it is often used with FastAPI and tools like `uvicorn`, `pydantic-settings`, and `python-dotenv`. Then, I saw how to create basic models by inheriting from `BaseModel`, defining fields with type hints, and how Pydantic performs type coercion (e.g., turning `"123"` into `123`).

The video also showed how to use the `Field()` class for advanced field validation, like setting minimum or maximum lengths, marking fields as required, or adding constraints such as salary being greater than a certain value. I understood how to make models more flexible and descriptive with these options.

Next, I explored custom validators using decorators like `@field_validator` for individual fields and `@model_validator` for validating multiple fields together. I also learned how to create computed fields using `@computed_field`, which calculate values based on other fields in the model.

A very useful part was learning about nested models and self-referencing models, especially in situations like a comment having replies (which are also comments). I now know how to use forward references and `model_rebuild()` to make this work.

Finally, the video explained how to serialize data using `model_dump()` and `model_dump_json()`, and how to customize serialization for types like

`datetime` using `ConfigDict`. The integration with **FastAPI** was also very informative—Pydantic models are used to define request bodies, perform automatic validation, and even manage application settings through dependency injection.

Overall, this tutorial gave me a strong foundation in Pydantic and showed me how it helps write cleaner, safer, and more reliable Python applications, especially in API development with FastAPI.

Apache Airflow

Source :- Apache Airflow for Big Data by TG117 Hindi

Summary

This video helped me understand the core purpose and working of Apache Airflow, which is a tool designed to orchestrate workflows, not to process data directly. I learned that Airflow is ideal for managing and scheduling data pipelines, especially in batch processing scenarios. Its main role is to automate a series of tasks—like extracting, transforming, and loading data—by defining them in a Directed Acyclic Graph (DAG).

One key takeaway was that Airflow supports different types of triggers: scheduled, event-based, and manual. This flexibility allows pipelines to run automatically at set times, in response to external events like a new file in S3, or when a user manually triggers them.

The video also clarified common misunderstandings. For example, Airflow is not a real-time streaming tool, not a data processor like Spark, and not an ETL tool with a visual interface. It simply controls and monitors the execution of tasks, which are usually defined in external systems or scripts.

Through a live demo, I saw how the Airflow UI offers excellent visibility and control. Views like the Graph, Calendar, Gantt, and Logs help track task status and performance. Since Airflow pipelines are written in Python, I can use full programming logic to build complex workflows.

Finally, I learned about the architecture of Airflow:

- The Web Server provides the user interface.
- The Scheduler checks when tasks should run
- The Executor decides how to run them.
- The Workers actually perform the tasks.
- The Metadata Database stores all system states, logs, and configurations.

Browser Use(Agentic Browser)

Source :- Browser AI Agent using any LLM Model + Playwright + Browser-Use + Web-UI by Naveen Automationlabs

Summary

This video gave me a clear and step-by-step understanding of how to set up a browser-based AI agent that can perform real tasks in a web browser using natural language instructions. The agent uses three main open-source tools—Browser-Use, Playwright, and Web-UI—along with an LLM like Google Gemini to understand and act on user prompts.

I learned how Browser-Use acts as the control layer between the LLM and the browser, while Playwright handles the actual browser automation (like clicking, typing, and navigating). The Web-UI provides a simple interface where I can write prompts, configure the LLM API, and view results, logs, and even screen recordings of the agent's activity.

The setup process involved installing Python packages, setting up Playwright and uv, cloning the Web-UI project, and running a local server. Once configured, I was able to access the interface through my browser and connect it to an LLM using an API key.

The agent was then able to carry out various complex tasks, such as:

- Searching for information on Google.
- Completing an e-commerce checkout process.
- Filling out and submitting a registration form using unique data.

What impressed me most was that no manual coding was required—the AI understood the prompt and translated it into automated browser actions. After execution, I could review the results through summaries, logs, and recordings provided by the UI.

N8N Automations

Source :- Automate Tasks with Pre-Built AI Agents | n8n Tutorial

Summary

This video helped me understand how to use n8n, a powerful open-source workflow automation tool that allows the creation of AI-powered agents and task pipelines. I learned how to self-host n8n on a VPS (Virtual Private Server) using Hostinger, where the installation process was made easy by selecting n8n as a pre-installed application during server setup. After launching the server, I accessed the n8n dashboard via its IP address and registered my instance using a free license key sent to my email.

The tutorial first showed how to use a pre-built AI Agent Chat template available within the n8n UI. I imported it into my workspace and connected it to external services by entering my OpenAI API key and a free SerpApi key (for Google search). With this setup, the AI agent was able to process prompts like *"Search for the best Nike shoes under 10k"*, fetch real-time results using SerpApi, and then format a user-friendly response using OpenAI.

The second part of the video focused on building a custom workflow from scratch—an agent that sends a daily computer science joke via email. I followed the process step by step:

1. I added a Schedule Trigger node set to run every day at midnight.

2. Connected it to an OpenAI node that generates a joke using a natural language prompt.

3. Finally, I linked it to a Send Email node, which picked the joke as email content and required SMTP credentials from a service like Mailgun or SendGrid.

I also learned how data flows between nodes, how each step can be tested individually, and how workflows can be activated or deactivated with a simple toggle. The interface made it clear how to track execution and debug if needed.

Firecracker VM

Source :- How AWS's Firecracker virtual machines work by Amazon Science

Summary

Firecracker is an open-source Virtual Machine Monitor (VMM) developed by AWS. It is designed to run thousands of lightweight virtual machines called microVMs on a single physical server. These microVMs are used in AWS services like Lambda and Fargate to run short, secure, and isolated workloads.

AWS needed a way to run many small serverless functions from different users on the same machine, with strong security and low resource usage. Traditional VMs like QEMU were too heavy, and containers didn't provide strong enough isolation. Firecracker solves this by combining the security of VMs with the speed and efficiency of containers.

Firecracker is built on top of KVM, which handles hardware-level virtualization. It removes unnecessary components like BIOS, GUI, and PCI devices to reduce size and speed up boot time. It supports only essential I/O via virtio-net (for network) and virtio-block (for disk). It boots a Linux kernel directly, skipping the usual bootloader process.

Firecracker can start a microVM in about 125 milliseconds, even when launching 50 in parallel. In comparison, QEMU slows down with parallel launches. Firecracker performs well with serial I/O tasks like small reads and writes. However, it has some limits with parallel I/O, which the team is working to improve.

Lambda functions can be as small as 128MB. AWS runs them on powerful machines like `m5.metal`, which have many cores and large memory. Firecracker allows AWS to pack thousands of isolated functions onto one server without slowing down or compromising security.

To avoid issues with software updates, AWS does not keep long-running machines. Instead, all microVMs are immutable and short-lived—no VM lives

longer than a day. They are created from fixed images, run the job, and are then removed. This improves reliability and security.

Firecracker is under active development. Future updates aim to improve parallel I/O speed, resource scheduling, and security using formal methods. Features like snapshots (for faster startup) and memory ballooning (for dynamic RAM control) are being added.

AWS has split Firecracker's core parts into a set of reusable Rust libraries under the `rust-vmm` project. This allows others to build custom virtual machines with the same secure and lightweight principles. It's a growing ecosystem that may shape the future of cloud computing and secure workload isolation.