Summaries(https://github.com/nakshjoshi/per-proj-req)

1. **System Design - 1**

   In system design, the basic setup starts with a client, who is essentially the user making requests to a server. Servers are powerful machines that run round the clock with a public IP, often hosted anywhere in the world. To simplify access, DNS (Domain Name System) maps easy-to-remember domain names to these IP addresses. Sometimes, to handle large traffic or simulate multiple machines, we use Virtual Machines (VMs) which are like computers within computers. When too many users hit the server simultaneously and it lacks resources, it can get overloaded and crash. Even if systems are built to handle failures (fault-tolerant), they still have a breaking point. So, efficient use of resources is crucial—overdoing it might waste resources too.

   To manage heavy traffic, we scale our systems. Vertical scaling involves adding more power (CPU, RAM) to one machine, but this causes downtime since we have to reboot the system. Horizontal scaling is better for uptime because it adds multiple machines instead. However, DNS can only direct to one IP, so we put a Load Balancer (LB) in between. The LB smartly distributes incoming traffic to all available machines, checking their health before sending any request. Companies like Amazon offer this as Elastic Load Balancing. When building large applications, we use microservices—where each feature like login, payments, etc., runs as a separate service. These services have their own load balancers and scaling, and all are connected through an API Gateway, which also supports authentication for security. In AWS, it's something like: user request → API Gateway → specific microservice.

   For background tasks that take time, like batch processing, queue systems such as Amazon SQS are useful. These let the server accept requests and handle them later. Data gets queued, often in a format like CSV, and processed when ready. To protect against spam or DDoS attacks, rate limiting is used—it stops too many requests from one source. This is managed using models like Token Bucket or Leaky Bucket. For events that need to notify many services, the PUB-SUB model works well, especially with Amazon SNS. In fanout architecture, one trigger can notify multiple services like email, inventory, or shipping without waiting for replies. If a service fails, the message can either be requeued or sent to a Dead Letter Queue (DLQ).

   When the database becomes a bottleneck, read replicas are introduced. The main database handles writing/updating, while replicas deal with reading data, easing the load. For performance, caching tools like Redis store frequently used data in memory. The system first looks in the cache; if the data isn't there, it checks the database and then saves it for future use. To serve static content faster (like images or scripts), we use CDNs like CloudFront. These act as global delivery systems using Anycast, so the same IP can be accessed from different parts of the world, always routing the user to the nearest, fastest server. Altogether, these components help build scalable, fast, and reliable web systems.

2. **System Design – 2**

   In modern system design, scalability and cost control go hand-in-hand. Companies like Netflix, YouTube, and Hotstar plan differently based on how predictable their traffic is. Netflix pre-scales its servers and uses caching at CDNs to serve users faster during expected spikes. YouTube, while facing mostly unpredictable traffic, manages well during events. Hotstar deals with both movies and live events, and they pre-warm servers for live shows since auto-scaling is too slow in such cases. To reduce the headache of managing servers altogether, many opt for serverless computing using tools like AWS Lambda. Here, code runs only when triggered, scaling automatically and charging only for use. It's super cost-efficient and hands-off, but not without issues—like cold starts, timeout limits, potential DDoS costs, and cloud vendor lock-in.

When using auto-scaling horizontally, each new server must have the app code and all its dependencies. That's where the pain of "it worked on my machine" kicks in. To fix that, virtualization helps—developers can pack everything (OS, dependencies, code) into a VM image and deploy it on the cloud. But VMs are heavy, so containers became a better alternative. Containers ditch the OS layer, making them much lighter and faster to deploy. These aren't 1:1 for users—a single container can handle many, depending on the workload. To manage many containers across clusters, we need orchestration tools. Google internally uses Borg for this, but Kubernetes—its open-source counterpart—is now widely adopted across the tech industry for managing, scaling, and maintaining containers efficiently.

3. **Kafka**

Real-time high-throughput data streaming means handling a large number of operations per second, like thousands of items moving through a system every moment. This kind of setup is important for systems like Zomato or Discord, where lots of user actions happen quickly. Normal databases like MongoDB and PostgreSQL can't handle such massive traffic—they crash or slow down under pressure. That's why we use something like Kafka. Kafka can handle really high throughput without failing, but it's not a replacement for a database since it has faster memory, smaller storage, and only keeps data temporarily. Also, you can't directly query data in Kafka like you can in a database. So, a common solution is to combine Kafka with a database. The flow goes something like: Data Producer → Kafka → Consumers (different services) → Database (bulk insert happens). Kafka stores data in topics, which are like logical divisions. Each topic has multiple partitions, but partitioning isn't done based on time—it depends on a specific field in the data. Kafka automatically balances the load between consumers and partitions using something called consumer groups. These groups make sure that partitions are evenly shared, first by dividing work among the groups and then among individual members within each group. This structure helps manage real-time data efficiently without overloading the system.

4. **Access Control Patterns**

Authorization is basically to identify what actions (like CRUD operations) a user can perform, and it's usually handled using JWT tokens or cookies. On the other hand, authentication is about giving permission to access a system in the first place. There are different types of authorization systems depending on the scale and use case. The most basic one is **RBAC** (Role-Based Access Control), which is used in small-scale setups. It assigns fixed roles like admin, editor, or viewer—for example, in a personal blog where editors can post and users can only comment. But RBAC doesn't work well for systems where each resource is owned or controlled differently.

Then there's **Fine-Grained RBAC**, where roles and permissions are decided at the resource level and not platform-wide. For every action, the system has to query the table to check the user's config, which adds complexity but more control. **ABAC** (Attribute-Based Access Control) is another type where access is decided by comparing the resource's attributes with the user's attributes—basically, matching who has what tag or property. Then you have **Policy-Based Access Control**, common in cloud systems like AWS, where you set clear policies for users, teams, and microservices to access different resources. **ReBAC** (Relation-Based Access Control) works on parent-child or nesting relationships—so access can be passed down or controlled based on hierarchical relations. A more advanced version is **Google Zanzibar**, which was a paper presented by Google explaining how they manage a consistent and scalable auth system using graph algorithms. A similar open-source project is **OpenFGA**, kind of inspired by Zanzibar and donated to CNCF, which also uses relation-based access control through graph structures.

5. **Kubernetes**

Bare metal deployment is like setting up your own server at home or renting it locally, installing your app on it, checking all dependencies, and then exposing its IP address to make it publicly available. The server is just a physical machine that's expected to stay on 24/7 and usually has a static public IP. The idea is that the code behavior on the server should be the same as on your local machine, so you make sure all dependencies and versions match. But when cloud platforms like AWS came in, they made things easier by offering cloud-native tools like static IPs, elastic load balancers, CDN (CloudFront), and auto-scaling groups. Still, sometimes developers faced the "it only works on my PC" problem.

To fix that, virtualization was used. It lets you choose your own OS version and dependencies, and then package everything into a system image that can be installed anywhere. But since it includes the OS too, the image size becomes huge. That's when Docker and containers came in—removing the OS from the image to make it lightweight while still keeping the same behavior across platforms. Managing containers at scale—like creating and destroying them based on traffic—was hard, so Google made a tool called Borg for container orchestration. Later, they open-sourced a version called Kubernetes and donated it to CNCF. Kubernetes helps with container management, like deploying, scaling, securing, and destroying them. It's cloud-agnostic because it's open-source, so you don't get locked into one cloud provider.(Ex: In AWS ECS, you just upload your Docker image directly).

Kubernetes has a specific architecture. It has a **control plane** (with API server, controller, etcd which is a key-value store, and a scheduler), and **working nodes** (with kubelet, kube-proxy, and the Container Runtime Interface). A developer writes a config and deploys API services; it first reaches the API server, which then goes to the controller, and stores data in etcd. The working node handles running containers and checks for safety and workloads. The pods—which are actual running containers—are created by the controller. Kubelet and CRI work together for scheduling and load balancing. There's also something called Firecracker VM, which is a lightweight VM used by AWS Lambda. Another component is CCM (Cloud Control Manager), which helps with managing cloud load balancers (like in EC2 or S3 buckets). It supports multiple cloud vendors like AWS, DigitalOcean, GCP, etc., and connects the Kubernetes API server to the cloud resources using the appropriate controller.

6. **Scalable Notification System Design**

Designing a scalable notification system today is tricky because users now expect notifications across multiple channels—like email, SMS, WhatsApp, and in-app messages—and often want to choose how they get them. Especially in fintech apps where fast updates on transactions matter, notifications must be quick. But sending notifications synchronously for every event can slow down the backend and is not a great idea. That's why the better approach is asynchronous messaging using high-throughput systems like Kafka or SQS. In this design, event producers push messages into Kafka, which then distributes them to consumers like services for transactional or promotional messages. Based on message priority, different architectures like queue-based or bulk processing can be applied. Kafka topics help manage the message logic, like deciding which channel to use and how to apply and tackle rate limiting issues.

Now, one major issue is the rate limits imposed by external APIs—like Gmail, Outlook, WhatsApp, or SMS providers. Even if our system is fast, those services might slow us down or temporarily block us. To handle this, queues are used to avoid flooding the API and to manage retry logic. The system can also be made smarter: if a user is online, in-app messages are prioritized; for login OTPs or urgent alerts, a priority queue is used. Digest logic may be applied for batching low-priority notifications. Some systems also adopt a Pub/Sub model for better scalability and separation of concerns, and smart consumers are built to handle the most important notifications first.

7. **Host our Own Browser or OS in the Amazon EC2**

   Key Learning: How to host our own browser or even an operating system inside an Amazon EC2 machine using AWS cloud services. The main idea is to set up a local machine on the cloud that runs a Docker image with a browser UI, and then destroy it after use to save costs. So, the basic flow goes like this: we go to AWS, choose EC2, and configure it—like selecting the type of machine (e.g., T2-Large or T2-XLarge) depending on the CPU and RAM we need. Once it's up and running, we use something called KASM or KASM Workspaces to pull a Docker image of a browser, like Vivaldi, and follow the docs provided by KASM and AWS. We access the cloud machine's terminal, patch in our browser Docker image, and install it so it's usable remotely. Optionally, we can use tools like speedtest inside that machine to check the internet speed and get an idea of how fast our cloud-hosted browser setup actually is in terms of data in/out.

8. **Make Your Own VPN Service**

   A VPN, or Virtual Private Network, is basically a secure tunnel between your device and the internet that encrypts your data and hides your IP address. We use VPNs to protect our privacy, access geo-restricted content, or browse securely on public networks. Now, the general working is like this: the client is connected to its regular internet service provider (ISP), which normally sends requests directly to sites like Google. But with a VPN in between, all requests go through the VPN first. The VPN client encrypts the request and routes it to a remote VPN server in some other region. From there, the request goes to the ISP of that region and finally to the desired service. This extra hop adds some latency but gives benefits like security and IP masking.

   To make our own VPN using AWS, we go to Amazon EC2 and choose a different region like Virginia or somewhere in North America to get the benefit of location change. Then, we create a new EC2 instance and instead of using Ubuntu, we look for a specific OpenVPN Amazon Machine Image (AMI). After selecting it and setting a configuration like T2 Large, we launch the instance. Once it's running, we use tools like Git Bash or any SSH client to connect to the terminal of our cloud server. Then, we follow the steps given in OpenVPN's documentation—like pulling the required images and finishing the setup—to get our VPN up and running.

9. **Docker Management API**

   In Docker container management, instead of manually using terminal commands, we can automate everything using code. Like in AWS ECS, containers are spun up and destroyed automatically as users come and go—this whole orchestration can be replicated manually too. Say we have 3–4 users, we create 3–4 containers on demand, and once a user logs out, the container is killed and user data is wiped, so no storage is done. The idea is simple: for each user, there's a dedicated Docker container. We use the Docker Engine API to talk to Docker, and to handle this through code, we use Node.js with libraries like Dockerode and Express. We set up routes like GET /containers to list them or POST /containers to spin up new ones (maybe using Alpine), and assign ports using a port mapping logic (like from 8000–9000 range). A backend script handles free port discovery and tracks which port maps to which container to avoid clashes.

   So the final setup becomes: user logs in → backend finds a free port → starts a container mapped to that port → user connects through browser at localhost:port (like localhost:8081 or localhost:8082) → when the user logs out, the container gets destroyed. This setup helps make a scalable, isolated, and dynamic container environment, and also sort of achieves a mini load balancer behavior without using heavyweight tools.

10. Creating our Own Docker Hub

A Docker Hub is basically a place where we can push Docker images so others—like teammates or org members—can pull and use them without worrying about setup mismatches. This solves the classic "it only runs on my machine" issue.

To build your own private Docker Hub, first launch an EC2 instance on AWS (Ubuntu & t2.large type works well). After connecting to it via SSH, install Docker and Docker Compose. Then set up a proper folder structure to store registry data. We will write a docker-compose.yml file that defines the registry container config, making it easier to manage and start/stop services.

Next, you configure Nginx to act as a reverse proxy—it forwards requests from your domain to the Docker registry running on the EC2 instance. You also set up HTTPS using Let's Encrypt to secure the connection. For added protection, basic authentication is implemented so only allowed users can access your registry. Once everything is ready, run your Docker registry using Docker Compose. From your local machine, you can now log in to this private registry, tag images, and push them. And that's how you host your own secure, private Docker Hub using AWS.

11. SSL (Secured Socket Layer)

In a basic client-server setup (like one using NGINX), the client sends a GET request and the server replies. But in between, someone could intercept the data—this is the "man-in-the-middle" problem. That's why we need SSL (Secure Socket Layer), which works using encryption. Now, there are two types of encryption: symmetric (same key for encryption and decryption) and asymmetric (public and private key pair). In symmetric encryption, the client encrypts the data and sends both the encrypted data and key to the server, but the problem is a hacker can still grab both in transit. So we use **asymmetric encryption**, where the server sends a **public key** to the client. The client uses that public key to encrypt a **symmetric key** it generates and sends that encrypted symmetric key to the server. The server then uses its **private key** to decrypt that and get the symmetric key, which is then used to actually send and receive the data securely.

Imp Point*: Asymmetric encryption is slower and heavy for large data, so symmetric encryption is used after that initial secure handshake between client and the server

But even this system can be hacked. A hacker could act as a fake server and send their own public key to the client. The client ends up encrypting its symmetric key with the hacker's key, allowing the hacker to decrypt the data. To stop this, **SSL Certificates** are used. A trusted **SSL provider** (certificate authority) gives the server a certificate, which is like a verified ID. This certificate contains the server's public key and a signature generated using the SSL provider's private key. The server then sends this to the client. The client/browser checks the certificate's reference ID and contacts the SSL provider to verify if it's genuine using their public key. If everything checks out, the browser knows the server is legit and secure communication begins.

12. Git / GitHub

Key Learnings (Detailed Notes Find Here: https://github.com/nakshjoshi/per-proj-req/blob/main/Git-GitHub.pdf):

- Types of VCS (version control systems) like Git, Apache Subversion, Piper
- Initializing Git Repo in a folder or Project
- Tracking Files using Git and implementing version control (staging, committing)
- Reverting Back to Previous Versions
- Branching Concepts
- Pushing Code to a Remote Repo

13. Docker Part-1

Docker helps us solve the classic “it works on my machine” issue by making sure our app runs exactly the same everywhere—on our system, someone else’s, or on a remote server. It’s like a shipping container that wraps our entire service with all the tools, configs, and code needed to run it. We define an **Image** as the recipe that has everything our app needs, and we spin up **Containers** from that image—these are lightweight, isolated environments running just our app, without interfering with anything else.

We use the **Docker CLI** to interact with Docker (like using docker run), and the background work—creating, running, and managing containers—is handled by the **Docker Daemon**. To expose parts of our service (say, a web app) to the outside world, we use **Port Mapping**, which connects a port from our machine to one inside the container (ex: 8080:80 lets us access our app via port 8080 on the browser). For passing in secret settings like DB credentials, we use **Environment Variables**

We write a **Dockerfile** to automate image creation. It might say: start with Ubuntu, install Node.js, copy our code—and we build this with docker build to get a reusable image. If we want to share or reuse our setup, we use **Docker Hub**, which works like YouTube for app images: we can push our image to the cloud or pull one made by others. Lastly, when our service includes multiple parts (like a frontend, backend, and database), **Docker Compose** lets us define and run all containers together using one config file (docker-compose.yml), so we can bring up or tear down the entire stack with just docker compose up and down.

14. Docket Part – 2

In Docker networking, by default, all containers are placed on a Bridge network, which acts like a virtual switch. Containers on the same bridge can talk to each other, but we need port mapping if we want to access them from the host. We can inspect it using docker network inspect bridge. The Host network skips this altogether and directly uses the host’s networking—no port mapping needed here (--network=host). There's also the None network, which completely disables networking for full isolation (--network=none). For better control, we can make our own custom bridge network where containers can talk by name. Just create it (docker network create -d bridge mynet) and use it when running containers.

Now, since container storage vanishes when the container stops, we use volume mounting to make data persistent. Mount a folder from our system like this: ‘-v /host/path:/container/path ’ and the data will stay even if the container is gone. Alternatively, we can use Docker-managed volumes using docker volume create. While building images, Docker caches each instruction as a separate layer, so unchanged layers are reused. To speed things up, put stable instructions (like FROM, RUN apt-get) early and changing ones (like COPY) at the end. For neat and smaller images, we can also do multi-stage builds by splitting build and runtime steps using multiple FROM lines. And to avoid dumping junk into our image (like .git/ or node_modules), we add a .dockerignore file. Lastly, using WORKDIRECTORY sets a working folder for the next commands in the Dockerfile, so we don’t have to keep writing full paths again and again.

15. Redis

Redis is basically an open-source, in-memory data store that helps us avoid hitting the database again and again for the same stuff. Since it stores data in RAM, it's super fast—but temporary. The main idea is to add Redis as a cache layer between our server and the actual database. So, when the server needs something, it first checks Redis—if it’s there, it returns the data immediately. If not, it queries

the database, stores the result in Redis, and then returns it. This reduces redundant queries, cuts down cost, and makes our response times much faster. A good example is caching stuff like unread message counts—we can just update Redis instead of recalculating from the database every time.

We can set up Redis using Linux, Windows, macOS, or Docker (Docker is usually preferred for quick setup). We interact with it via redis-cli or through a GUI like RedisInsight (port 8001), while the Redis server runs on 6379. Redis supports different data types: **Strings** for basic key-value pairs (with commands like SET, GET, and even INCR for counters), **Lists** for ordered items (using LPUSH, LPOP, etc.), **Sets** for unique unordered strings (with commands like SADD, SMEMBERS), **Hashes** to store object-like data (perfect for things like user profiles), and **Streams** for real-time data logging. There's also **Geospatial** support, where we can store coordinates and run location-based queries (GEOADD, GEORADIUS). A very common real-world use case is caching API responses: before hitting the DB, check Redis—if not found, fetch from DB, cache it with TTL, and return. Speeds things up big time and saves resources.

16. Web-Scokets

So basically, in normal cases, the client sends a request to the server, the server replies back with a response, and that's it—the connection ends. This is a one-way (unidirectional) communication model, which works fine for simple apps but not for things like chat apps, video calls, or voice chats where we need continuous back-and-forth. The simplest way to make a chat app work is by **polling** the server at regular intervals to check for updates. But that ends up hitting the server again and again unnecessarily, increasing the number of requests, responses, and overall cost—basically overkilling the system.

That's where **WebSockets** come in. The client initially makes a regular HTTP request to the server but includes an Upgrade header asking the server to switch to a WebSocket connection. Once accepted, this connection stays open—it doesn't auto-close like normal HTTP. It only closes if the client or server intentionally ends it. This gives us a **bidirectional**, **full-duplex**, and **lossless** channel, which is perfect for real-time communication. According to documentation, this whole switch happens through an HTTP header upgrade that enables WebSocket mode, letting both sides send and receive data freely without restarting the connection every time.

Key Learning Beside this: Implementation in Node.js

17. Creating a Discord Bot

To create a Discord bot, the bot basically listens to messages and responds back. First, we make our own server on Discord—say for a club or community—and give it a name. Usually, the default **General channel** is enough to test things. Then in **User Settings > Advanced**, we turn on **Developer Mode**. After that, we head to the **Discord Developer Portal**, go to **Applications**, and create a new application. We give it a name, then go to the **Bot** section inside that application, create a bot, name it, and make sure to give it all necessary admin permissions.

Next, we have to add the bot to our Discord server. For that, we go to the **OAuth2 > URL Generator**, select the "bot" scope and check "administrator" under permissions. Then we copy the generated URL, paste it into the browser, pick our server, and the bot gets added there. To make the bot actually interact with users, we use **Discord.js** with **Node.js**. So, in VS Code (or any IDE), we do npm install discord.js. Then in our main index.js file, we import Discord, set up intents (especially for reading messages), paste the bot token, and write the basic code to respond to messages. It follows the standard flow as shown in the Discord.js documentation.

18. Cookies in Node.js

Cookies and Node.js mainly come into play when we talk about authentication—both **stateful** and **stateless**. In **stateful authentication**, we use things like JWT tokens or session IDs that are stored in the user's memory. But the catch is, if the user refreshes the page, the session can be lost, and they get logged out. So here's what normally happens: the client sends a request with their username and password. The server checks those credentials in the database, and if all's good, it generates a JWT token containing stuff like the user's ID, name, and email. This token is then sent back in the form of a **cookie** using res.cookie. The client's browser stores that cookie.

In the next request, the browser automatically includes this cookie, so the server can grab it, verify the token, and confirm the user's identity. Cookies are domain-specific and can travel back and forth between server and client. We can even set options like domain and an expiry time inside res.cookie. But one important point: **cookies only work inside browsers**. If we want cross-platform functionality—say on an Android app or a non-browser client—then we use **JSON Web Tokens via headers** for auth instead of relying on cookies.

19. SSR (Server-Side Rendering)

Server-Side Rendering (SSR) in the Node.js context means the server generates the HTML instead of the browser doing it. For this, we use a templating engine called EJS (Embedded JavaScript), which lets us put JavaScript code inside HTML. We install it using npm install ejs. In the project, we create a views folder where all the .ejs template files live, and we organize these HTML files based on routes like pre notes, views, or URLs. Then, in Express routes, we use res.render("index") or similar commands to render these EJS templates and send the final HTML to the client.

20. MCP (Model Context Protocol)

MCP stands for Model Context Protocol. The "model" is the LLM we're using, "context" is the external info we feed into the model, and "protocol" means the rules we follow. This idea comes from Anthropix, the parent company of Claude. MCP uses standard protocols like HTTP and TCP.

The big challenge with LLMs is that training is expensive and they have a limited context window, so it's tough to keep them updated efficiently.

MCP acts like a USB-C port for AI—it's a standard way to connect the model to different data sources and tools. Basically, the user interacts(or act as a MCP Client) with an MCP client, which connects to multiple MCP servers. Each server links to some external service like Google Search, YouTube, or a local database, feeding fresh context to the AI.

MCP uses standard input-output 'STDIO'(like terminal commands with echo), acting as a bridge between AI and real-time external data. The same MCP server can be used across different models or IDEs.

On the client side (MCP Pad), tools like IDEs or Claude or ChatGPT handle input/output, manage workflows, and connect the AI to the context. On the server side (MCP Server), it's lightweight and shell-based, running task-specific programs that pull data from local databases or remote APIs like Google Drive. MCP servers can generate prompts, route API responses, manage AI resources, and build reusable prompt templates and workflows.

However, since standard input-output can't work in cloud deployments, MCP uses SSE (server-sent events) transport protocol to run on the cloud.

Summaries of topics for Govt. SLM Project

(https://github.com/nakshjoshi/per-proj-req)

1. **LangChain(https://youtu.be/DcNxg61kSFc?si=52o-MRcshKFiINHu , https://youtu.be/nAmC7SoVLd8?si=_DibbvQNk1RxS7xM and little inputs from IBM documentations):**

   LangChain is not a library --> it is a complete framework that allows us to create LLM-based applications with ease, and it has support for both JavaScript and Python (although Python has better docs).We can Imagine it as something that drives the already created LLMs such as GPT-3.5, GPT-4, Gemini, or HuggingFace available models. The concept behind LangChain is to provide adequate context to LLMs. Now the problem is this: LLMs are token-based, so we can't pump them a whole document or image or whatever large thing every single time the user submits a question. That's why we need LangChain .

   So how it is done is: we chunk the entire text, and whenever the user queries something, we select the most appropriate chunk depending on what they are asking and pass it along with the question to the LLM. Previously, we needed to do this ourselves—code our own chunner to chunk the data, embed it into a vector with embedding models, and then query a vector DB. But LangChain does all of that internally for you. Businesses wants AI to work as to operate on their own data, can be adjusted to their staff or services, are less expensive than simply sending everything to the model, and allow input/output in any format, not merely text, and the best is to use multiple LLMs at the same time.

   Now LangChain allows us to plug in various models and tools. It supports Google Search, YouTube, Wikipedia, or even your local/private databases. It modularizes your code, making it reusable and much more scalable. The fundamental idea in LangChain is the Chain. At its most basic level, a Chain = LLM + Prompt. For instance, if you are working with OpenAI, you simply import it using from langchain.llms import OpenAI. You can also control the temperature — with 0 being conservative (less creative, more fact/pre-data oriented) and 1 being creative but risky also.

   In the case of embeddings, the notion is: we translate every paragraph or chunk of our content in doc into a numerical vector that represents its meaning. You do this with models such as Word2Vec, Doc2Vec, or BERT. Those vectors are then indexed in a vector database such as FAISS, Chroma, or Pinecone. If the user is asking a question, we also translate that question into a vector and compare it with all the vectors stored and identify the most relevant ones. Depending on cosine similarity or another measure of distance, we select the top N/K most similar chunks and pass them along with the query to the LLM — in this way, we pass useful information rather than complete data.

   System Desing of a basic LangChain App:

   1. User uploads a PDF, which can be stored in the cloud (like AWS S3).
   2. We then load the document using a document loader.

3. Next, chunk the PDF or doc into manageable chunks.
4. Send each chunk to an embedding model to get numeric vectors.
5. Store those vectors in a vector DB(Pinecone, FAISS, Chroma).
6. When the user asks a query, convert it into an embedding too.
7. Match that query embedding with the ones stored to determine the nearest chunks.
8. Merge those relevant chunks with the original user query and send it to the LLM for final processing.

Chains: LangChain also enables Sequential Chains, which support when you're required to execute several steps sequentially. Such as if the output of one LLM needs to become the input for another prompt or step, then Sequential Chains aid in that gracefully.

2. **Pydantic AI(Source: https://www.youtube.com/watch?v=XIdQ6gO3Anc&t=131s , https://www.geeksforgeeks.org/introduction-to-python-pydantic-library/)**

Pydantic is a Python library mainly used for data validation and type checking. If our app handles user inputs or external data, and we want to make sure everything coming in is clean and structured. So we basically define a data model using Python classes that inherit from BaseModel, and Pydantic automatically takes care of parsing and validating data based on your type hints. So whether we're dealing with JSON data, API inputs, or even configuration files, it'll make sure the data matches the expected format, and if not, it throws clear and detailed error messages.

The main goal behind Pydantic is to bring that **FastAPI-style** developer experience to GenAI app building. It's fast (even faster if you enable optional Cython), supports custom validators using @field_validator, lets us set default values for fields, and works like a regular Python class once initialized.

Some of the most common use cases include API data validation, managing configs safely, and ensuring consistency in data processing pipelines. At the core of this setup, especially in AI workflows, it helps in creating structured input/output models and plays a key role when combining agents like **Agent = LLM + Tools**.

3. **Apache Airflow (https://www.datacamp.com/tutorial/getting-started-with-apache-airflow):**

Apache Airflow is an open-source tool designed to help run and manage data pipelines in production. It uses Python, so you can write your workflows as code and then schedule and monitor them easily. At its core, Airflow revolves around the idea of tasks and dependencies, which together form DAGs (Directed Acyclic Graphs). These DAGs are structured workflows without loops—each task depends on another in a one-way direction, like in typical ETL (Extract → Transform → Load) processes. Loops are avoided because they can cause the pipeline to run endlessly. The arrows in a DAG show this flow clearly.

Airflow's architecture includes key components like the Scheduler (schedules DAGs based on the time and start date you set), Executor (allocates tasks to resources), Metadata Database (stores run logs, connections, variables, etc.), and the Webserver UI (a dashboard to track everything visually). When setting it up, you configure things like your home directory and metadata DB. Some good practices in Airflow include keeping tasks modular (do one thing per task), making them deterministic (same input = same result every time), and ensuring idempotency (re-running doesn't create duplicates). Airflow is best used as an orchestration tool, especially when you just need to lightly transform data and focus more on managing complex workflows.

4. **Browser Use (Agentic Browser)**

   Agentic Browser Use is basically about letting AI agents (like ChatGPT, Gemini, or other assistants) use a browser just like humans—searching, clicking, reading, and taking actions on real websites in real-time to complete tasks. At its core, it's powered by a mixture of a brain (the LLM) and tools. This is useful because normal LLMs can't access up-to-date info—they're trained on past data. With an agentic browser, we can fetch current news, automate form filling, pull data from long PDFs or webpages, and even use it like a smart web scraper or crawler. Instead of loading massive documents into prompts, the agent just browses and fetches only the required parts.

   Behind the scenes, it uses tools like Playwright or Puppeteer for headless browsing, LangChain for planning and using tools, retrievers for pulling content, and OpenAI GPT/Gemini models for decision-making and actions.

   There are some limitations though—it struggles with JavaScript-heavy sites, can be slow since it loads pages, and can't bypass logins or CAPTCHAs. It also relies heavily on how well the prompt guides it.

   To set it up, you need Python 3.11+, pip, and internet access. Install browser-use, optionally with memory, and then install Chromium via Playwright. You also need to set your OpenAI API key in a .env file. For custom behavior, you can tweak settings using BrowserConfig and BrowserContextConfig, which control browser behavior and page-level settings like timeouts or screen size.

5. **N8N Automations (https://www.youtube.com/playlist?list=PLlET0GsrLUL59YbxstZE71WszP3pVnZfI)**

   n8n is an open-source workflow automation tool that helps us connect APIs, services, and apps to automate repetitive tasks and build logic/data pipelines—basically a self-hosted alternative to Zapier but with more control. What makes it different is its visual drag-and-drop interface, ability to run on your own server, support for custom JavaScript and expressions, and 400+ ready-to-use nodes like Google Sheets, Telegram, Airtable, HTTP, etc.

   We can also schedule workflows or trigger them using webhooks and API calls, making it flexible for real-world use cases like fetching weather data, interacting with databases, or performing automated Slack alerts.

   The whole system runs on nodes, and there are mainly three types: **Trigger nodes** (which start the workflow), **Regular nodes** (do stuff like sending HTTP requests or adding rows to Google Sheets), and **Logic nodes** (add conditions like if-else, switches, merging data).

   Each node handles data in JSON format—receives items as key-value pairs, processes them, and then passes them forward to the next node. A typical flow looks like: trigger → processing → logic → final node.

6. **Firecracker Virtual Machine (https://firecracker-microvm.github.io/)**

   Firecracker is an open-source virtualization tech made mainly to spin up secure, lightweight microVMs for container and function-based services. It's built for multi-tenancy, meaning we can run workloads from different users on the same hardware while keeping them isolated and safe. Using Linux's KVM under the hood, Firecracker can boot apps super fast(in around 125 ms) and can launch around 150 microVMs every second. These microVMs are super lightweight, using less than 5MB of memory, which lets us run thousands on a single server without performance issues.

Woking of the FireCracker VM:

Firecracker sits on top of a bare-metal server running Linux and uses KVM for virtualization. Firecracker itself operates in user space and spins up microVMs that each contain a minimal guest OS and the app it needs to run. These VMs are highly customizable in terms of CPU and memory. On the security front, it uses two layers: a virtualization barrier and a jailer that uses Linux features like namespaces and chroot for tighter isolation between each MicroVM, guarding the users data .

Firecracker exposes a REST API so external systems like AWS Lambda can manage microVMs. Its architecture separates the control plane and data plane, kinda how Kubernetes works.

Just for my notes and analogy:

1. Firecracker MicroVMs = Pods with VM-level isolation
2. Firecracker REST API = Docker Daemon API or K8s Control Plane
3. Kernel + rootfs in Firecracker = Docker image or Pod container image