Conceptual and Analytical Questions

Q. Differentiate between the roles of a Data Scientist and a Machine Learning Engineer, focusing on key responsibilities and overlaps.

Data Scientists and Machine Learning Engineers both play crucial roles in machine learning projects, but their toolkit and responsibilities set them apart in day-to-day work. A Data Scientist is often found working with tools that help them ingest, clean, analyze, and visualize data. For instance, Python (with libraries such as Pandas and NumPy), R, SQL, and Jupyter Notebooks are staples for wrangling and exploring data. When it comes to generating reports or interactive visualizations, Tableau, Splunk are commonly used. For prototyping models and running experiments, frameworks like Scikit-learn and sometimes even TensorFlow are preferred. A big part of their job is understanding the data—spotting trends, building correlations, and **building prototypes** to demonstrate possible solutions to business problems.

The **Machine Learning Engineer**, working further downstream, chooses tools that allow them to bring these models into production reliably. Python is still important, but you'll see them use Docker and Kubernetes to containerize and deploy applications. For building scalable and maintainable workflows, ML engineers rely heavily on TensorFlow, PyTorch. On the backend, they work with cloud platforms like AWS SageMaker and sometimes employ CI/CD tools for automation and model management. Monitoring model behavior and maintaining performance typically involves tools such as MLflow, TorchServe, and TensorRT. They wrap up everything so that predictions run quickly and reliably for many users and data sources at a time.

You'll find Data Scientists thriving in industries like finance (risk analysis, fraud detection), healthcare (patient diagnosis from medical data), and retail (customer data mining and segmentation), powering dashboards and insights for decision makers. Machine Learning Engineers are more visible in tech-driven businesses such as autonomous vehicles (deploying image recognition models to embedded systems), SaaS companies (integrating recommendation engines with applications), and large-scale e-commerce (optimizing user experience with personalized search

or pricing). In practice, both roles often collaborate, but their respective toolkits reflect their differing priorities: Data Scientists build understanding; Machine Learning Engineers create scalable solutions ready for production deployment.

Q. Evaluate whether the CI/CD process is limited to automation and provide supporting arguments.

CI/CD is often associated with tools like Jenkins, GitHub Actions, or GitLab pipelines, but it is not limited to automation. Automation is only the visible part — the scripts that build, test, and deploy code. What makes CI/CD powerful is the way teams work around that automation.

For example, Continuous Integration is not just about running automated unit tests; it is about developers committing small, frequent changes instead of large, risky batches. Continuous Delivery is not just a deployment script; it requires practices like feature toggles, rollback strategies, and monitoring so that code can be released safely at any time. These cultural and process elements cannot be achieved with automation alone.

Imagine a team that automates everything but still merges code once a month. Despite having automation, that is not true CI/CD because feedback is slow and risk is high. In contrast, a team that commits and deploys daily, collaborates closely, and treats failures as learning opportunities is practising CI/CD, even though the same automation tools are used.

Therefore, CI/CD is best understood as automation plus culture. Automation is the backbone, but without the cultural shift towards collaboration, small iterative changes, and shared responsibility for quality, CI/CD cannot deliver its promised benefits.

Q. Compare monolithic CI/CD pipelines with microservices-based pipelines, including examples and team structure differences.

A **monolithic pipeline** builds, tests, and deploys the entire application as one unit. This is common in single-codebase applications, such as a traditional e-commerce platform where checkout, catalogue, and payments all deploy together. The

advantage is simplicity and a single source of truth, but the drawback is that any change requires rebuilding and redeploying the entire system, which slows feedback cycles and risks bottlenecks.

A **microservices pipeline**, on the other hand, gives each service its own independent build, test, and deploy cycle. For instance, in a microservices e-commerce system, payments, shipping, and recommendations each have their own pipeline. This enables faster iteration, isolated failures, and independent releases, but requires more complex tooling and careful observability.

Team structures also differ. Monolithic pipelines often rely on centralised teams with specialised roles, whereas microservices pipelines align with cross-functional "you build it, you run it" teams that own a service end-to-end.

Q. Challenge the statement: "HTTP POST is idempotent." Provide examples to support your stance

An operation is idempotent if repeating it produces the same end result as performing it once. HTTP methods such as GET, PUT, and DELETE are defined as idempotent under RFC 7231, but POST is not.

In practice, POST is intended for non-idempotent actions like creating resources. For example, sending the same `POST /orders` request twice would typically create two distinct orders, which clearly produces different results. This shows POST is not inherently idempotent.

However, application-level logic can simulate idempotency by using mechanisms such as idempotency keys. In this case, two identical POST requests with the same key may both return the same created order. Even so, the idempotent behaviour comes from the application design, not from the semantics of POST itself.

Therefore, the claim that "HTTP POST is idempotent" is incorrect by default. POST is intentionally non-idempotent, and only through additional safeguards can it behave idempotently in practice.