Q1: An online shopping website is facing frequent downtime during sales.
How would you apply the SDLC phases to identify and permanently fix this issue?

To fix a website that keeps crashing during sales, I would start by looking at the data to see exactly where the system is choking. Once I know if it is a database issue or just too many people trying to click the same button, I would design a way for the site to automatically add more server power when traffic spikes. Before the next big sale, I would have the team run a massive test with thousands of fake users to make sure the fix actually holds up under pressure so it does not happen again.

Q2: A client wants to launch an MVP of an online food delivery app in 3 months.
Which SDLC model would you choose and why?

For a quick three month launch, I would definitely go with the Agile model. It is much better than trying to build the whole app at once because you focus on just the core stuff like ordering and payments first. You build it in small chunks, see if it works, and then keep adding to it. This way, you actually have something ready to show the world by the deadline instead of a half-finished project that tried to do too much.

Q3: After deployment, users report that the payment feature fails intermittently.
In which SDLC phase should this have been caught, and how would you prevent it next time?

This kind of payment glitch should have been caught during the testing phase before the app went live. The problem usually happens because the team only tested if payments work when the internet is perfect, but they did not test what happens when the connection is slow or the bank times out. To stop this from happening again, I would set up automated tests that specifically try to break the payment flow by mimicking bad network conditions.

Q4: A business requirement changes midway while developing a subscription-based platform.
How does the traditional SDLC handle this, and what are its limitations?

The old school waterfall method is really bad at handling changes once things are moving. In that system, you have to finish one big step before starting the next, so if a client changes their mind in the middle, it is a huge headache. You basically have to go all the way back to the start, which ends up costing way more money and making the whole project late because the system is just too stiff.

Q5: An online banking application must meet strict security and compliance requirements.
How would you incorporate security testing into the SDLC lifecycle?

For a banking app, you cannot leave security for the end. I would make it part of the daily work from day one. This means scanning the code for holes every single time a developer writes something new and thinking about how a hacker might get in while we are still drawing the plans. It is like building a house and checking the locks every day instead of just putting an alarm on at the very end.

Q6: Your team delivered a feature, but it doesn't match the customer's expectation. Which SDLC step likely failed, and how would you improve it?

When a team delivers something the customer does not like, it usually means the very first step of talking about the requirements failed. To fix this, I would not just rely on a list of words. I would show the customer rough sketches or a simple prototype early on so they can click around and see if they like the direction. It is much easier to fix a sketch than it is to fix a finished product.

Q7: A legacy e-commerce system needs to be migrated to a cloud-based architecture. How would SDLC guide this migration process?

Moving an old system to the cloud is like moving to a new house, and the SDLC acts as the checklist. You start by planning what actually needs to move and then design how it will fit in the new cloud environment. The most important part is the testing at the end where you make sure everything in the new cloud version works just as well as the old one before you officially switch over.