Question 1: Scroll-Triggered Animation Optimization (10 Marks)

To implement and optimize a complex scroll-triggered animation sequence, my approach would prioritize performance by using efficient techniques such as CSS animations and the requestAnimationframe API for JavaScript-based interactions. I would divide the animation sequence into smaller, manageable components and use lazy loading to trigger animations only when elements are within the viewport, minimizing unnecessary computations. SVG elements would be optimized by reducing file sizes and simplifying paths. I'd also use GPU-accelerated CSS properties like transform and opacity for smoother animations without consuming significant CPU power.

To ensure smooth performance across devices, I would profile the animation sequence using browser developer tools like Chrome DevTools. This would allow me to monitor frame rates, check for layout thrashing, and identify potential bottlenecks. Tools like Lighthouse would be used to benchmark the performance, and I would leverage the Performance panel to analyze the time spent on scripting, painting, and rendering. Bottlenecks such as janky scroll behavior or long-running JavaScript tasks would be addressed by optimizing the DOM structure and reducing reflows/repaints. Finally, I would perform real-time testing on both desktop and mobile devices to refine and ensure responsiveness.

Question 2: Scaling Node.js for Traffic Spikes (10 Marks)

To handle traffic spikes in a Node.js/Express application, the first step would be to diagnose bottlenecks by setting up monitoring tools like New Relic or Datadog to track response times, CPU usage, memory consumption, and database query performance. Tools like Apache JMeter or K6 could be used to simulate high traffic scenarios and identify potential bottlenecks. I would focus on profiling slow database queries, event loops, or external API calls that might be causing timeouts or slow responses.

For scaling, I'd implement both horizontal and vertical scaling strategies. Horizontally, I would introduce load balancing (using tools like Nginx) to distribute traffic across multiple instances of the application. Autoscaling groups would automatically spin up new instances when traffic surges occur. Vertically, I'd ensure the servers have adequate resources (CPU, RAM) for handling peak loads. Caching strategies, such as using Redis or a CDN for static assets, would reduce the load on the server. I'd also implement rate-limiting and request queuing to prioritize critical transactions during spikes, ensuring that the system remains resilient and responsive.

Question 3: Leveraging AI Tools for Complex API Integration (10 Marks)

When integrating AI-assisted coding tools like Chatgpt into the development workflow, I would start by using these tools to quickly generate code snippets and initial implementations for dealing with third-party APIs, especially in cases where documentation is sparse or incomplete. AI tools can be valuable for generating scaffolding code, suggesting request structures, or automating repetitive tasks. However, I'd apply a validation layer by testing the generated code rigorously in different scenarios, especially for edge cases and unexpected inputs.

For handling frequently changing APIs and inconsistent data structures, I'd rely on strong error handling and fallback mechanisms in the code, such as using Typescript to ensure type safety and predictable data structures. Additionally, I would implement unit tests and integration tests to catch API changes early and verify the reliability of the code. By leveraging version control and feature flags, I'd ensure that the use of AI tools does not introduce hidden technical debt. Regular code reviews would ensure that AI-generated code maintains quality and conforms to best practices. This approach would help balance the efficiency gains from AI tools with a focus on maintainability and long-term reliability.

Question 4: Modernizing a Legacy Node.js Monolithic Application (10 Marks)

To modernize a legacy Node.js monolithic application, I would adopt a phased refactoring strategy that balances ongoing feature development with incremental improvements. The first step would be to identify critical components to refactor, focusing on areas that have the highest impact, such as performance bottlenecks, security vulnerabilities, or sections that are heavily used by the application. I'd begin by decoupling these components, extracting them into individual services or modules that can be scaled and maintained independently.

Backward compatibility would be ensured by implementing APIs or interface adapters that allow the legacy system to communicate with the newly introduced services without breaking existing functionality. Each new module would be deployed using a CI/CD pipeline to automate testing and ensure smooth deployment without interruptions. I'd use tools like Docker to containerize services, which would allow for seamless scaling and improved modularity. Success would be measured by improvements in response times, ease of adding new features, and reductions in deployment times. To mitigate risks, I'd ensure comprehensive automated testing and incremental deployments, rolling back changes if necessary. Regular team syncs and documentation would keep everyone aligned during the transition.