Here are my notes from the Ajith’s Interview, capturing each question.

1. **Handling Complex Batch Operations:**
   * *Question:* "Describe your approach to managing a batch database update operation, particularly when dealing with high concurrency. How would you handle scenarios where data might be missing or improperly logged in a high-traffic environment?"
   * *Notes:* The client seemed focused on reliability and error handling, especially around parallel processing and retry mechanisms.
2. **API Design for Large Data Retrieval:**
   * *Question:* "If you were tasked with creating an API endpoint to handle large data requests, what considerations would you make to ensure efficient data delivery and avoid straining network performance?"
   * *Notes:* Emphasis on data chunking, pagination, and response compression. Discussed options like asynchronous streaming and caching.
3. **Robust Error Handling in APIs:**
   * *Question:* "How would you implement comprehensive error handling in API requests to ensure users receive clear, actionable feedback?"
   * *Notes:* The client asked about handling both client and server-side errors, possibly through standardized ProblemDetails responses and logging for detailed error tracking.
4. **Database Concurrency and Consistency:**
   * *Question:* "When performing large concurrent database transactions, how do you ensure data consistency and handle potential race conditions?"
   * *Notes:* They were interested in techniques like row versioning, isolation levels, and the use of retry policies to manage concurrent writes without losing data integrity.
5. **Asynchronous Programming Best Practices:**
   * *Question:* "What are your preferred best practices when implementing asynchronous code in .NET?"
   * *Notes:* The interviewer seemed focused on minimizing thread-blocking operations, proper task cancellation handling, and avoiding common pitfalls like Task.Result in asynchronous contexts.
6. **Error Handling in Multi-API Call Methods:**
   * *Question:* "Imagine you have a method that calls multiple APIs. How would you ensure each API call is handled gracefully in case of failures, and that the method returns a reliable result?"
   * *Notes:* They mentioned resilience strategies, such as using try-catch blocks around each API call, retry policies, and logging errors for failed requests.
7. **Scalable System Design:**
   * *Question:* "Describe your approach to designing a scalable and resilient system. What specific patterns and architectural principles would you apply?"
   * *Notes:* They wanted details on principles like microservices, load balancing, and the use of cloud-native features for horizontal scaling.
8. **Rate Limiting and Data Chunking:**
   * *Question:* "How would you implement rate limiting and data chunking in an API intended for high-traffic use to prevent server overload?"
   * *Notes:* Discussion included token bucket algorithms for rate limiting and breaking down large responses into smaller data packets to enhance performance.
9. **Notification System with SOLID Principles:**
   * *Question:* "How would you design a notification system (for Email, SMS, Push notifications) that adheres to SOLID principles?"
   * *Notes:* They wanted a clear separation of notification types and the application of dependency inversion to decouple the notification system from specific implementations.
10. **Filtering, Sorting, and Pagination in eCommerce API:**
    * *Question:* "In an eCommerce application with a large product catalog, how would you manage dynamic filtering, sorting, and pagination?"
    * *Notes:* Focus on making the API flexible enough to support various filtering types (e.g., exact match, starts with), sorting criteria, and efficient pagination to handle large datasets.
11. **Retry Mechanism for Unreliable External Services:**
    * *Question:* "If an external service is unreliable and prone to intermittent failures, how would you implement a retry mechanism that minimizes impact on the user experience?"
    * *Notes:* They highlighted interest in retry logic with exponential backoff, circuit breaker patterns, and configurable retry counts to maintain system resilience without overloading the external service.