Section 1:

1. Software Architecture

- **Monolithic vs. Microservices:**
  - Monolithic: Single, large application. Simple to develop initially, but difficult to scale and maintain.
  - Microservices: Collection of small, independent services. Easier to scale and maintain, but more complex to manage.
  - For a scalable analytics platform, microservices are preferable due to their ability to scale individual components based on demand.
- **Architecture Design:**
  - Use a layered architecture.
  - UI (React): Communicates with the backend via RESTful APIs.
  - Backend (Node.js/Express): Handles API requests, data processing, and database interactions.
  - ML Server: Exposes ML models as APIs, receives data from the backend, and returns predictions.

2. Database Design

- **MongoDB:**
  - `userActivities`: {userId, timestamp, activityType, details}
  - `anomalies`: {anomalyId, userId, timestamp, modelId, severity, details}
- MongoDB for flexible, unstructured data (anomaly details, user activities).

3. API Design

- **Anomaly Data Endpoint:**
  - Method: GET
  - Path: `/api/anomalies`
  - Request Parameters: `startDate` (YYYY-MM-DD), `endDate` (YYYY-MM-DD)
  - Response Format: JSON array of anomaly objects.
- **API Best Practices:**
  - Use RESTful principles.
  - Implement versioning.
  - Provide clear documentation.
  - Use authentication and authorization.
  - Handle errors gracefully.
  - Use standard HTTP status codes.

Section 3: Scenario-Based Questions

1. Deployment Strategy

- **Cloud Services:**
  - AWS: EC2 for backend/ML servers, ECS/EKS for container orchestration, RDS/DynamoDB for databases, API Gateway for API management, S3 for storage.
  - Azure: Virtual Machines, Azure Kubernetes Service (AKS), Azure SQL Database/Cosmos DB, Azure API Management, Azure Blob Storage.
  - GCP: Compute Engine, Google Kubernetes Engine (GKE), Cloud SQL/Cloud Spanner, Cloud Endpoints, Cloud Storage.
- **Scaling:**
  - Use auto-scaling groups for servers.
  - Use managed database services with scaling options.
  - Implement load balancing.
- **Downtime:**
  - Implement redundancy and failover mechanisms.
  - Use health checks and monitoring.

2. Performance Optimization

- **Backend:**
  - Optimize database queries.
  - Implement efficient data retrieval and processing.
  - Use asynchronous operations.
  - Implement pagination for large datasets.
- **Frontend:**
  - Optimize rendering.
  - Use lazy loading.
  - Minimize network requests.
- **Caching:**
  - Use Redis or Memcached for caching frequently accessed data.
  - Use browser caching.