Summary of System Architecture and Trade-offs

LinkSphere is a secure, scalable, and modular social networking platform developed using a microservices architecture. The system is composed of several independently deployed services, each handling a specific domain of functionality, ensuring clear separation of concerns, easier scalability, and service isolation.

The architecture includes the following key components:

- **API Gateway**: Serves as the single entry point to the backend, handling routing, request validation, and token-based authentication using JWT.
- **User Service**: Handles user registration, authentication, and Google OAuth2 login integration.
- **Profile Service**: Manages user profile data including names, profile pictures, and bios.
- **Friend Service**: Manages sending and receiving friend requests and storing relationships.
- **Feed Service**: Aggregates posts and updates from a user's social circle.
- **Notification Service**: Listens to Kafka-based system events and manages user notifications using PostgreSQL for persistence and Redis for quick access to unread messages.
- **Frontend Client**: Built with React, it communicates with backend services through the API Gateway and provides an intuitive and responsive user interface.

Architectural Trade-offs:

- The choice of microservices improves modularity and horizontal scalability but increases the complexity of deployment, service coordination, and inter-service communication.
- Kafka enables asynchronous and decoupled communication between services, enhancing fault tolerance, but requires additional infrastructure management and reliable message handling.
- Redis caching improves performance for notification retrieval, though it introduces the need for careful synchronization with the persistent database.

- JWT-based stateless authentication offers simplicity and security, yet requires proper token lifecycle management and expiration handling.

Challenges Faced and Solutions Implemented

- **Authentication and Authorization**: Managing secure and centralized authentication across all services was a key challenge. This was solved by integrating OAuth2 using Google Sign-In and enforcing JWT verification at the API Gateway layer.

- **Reliable Inter-service Communication**: Ensuring decoupled and consistent data flow across services required a robust messaging system. Kafka was implemented to support asynchronous communication and handle events efficiently.

- **Efficient Notification Delivery**: Displaying unread notifications quickly while ensuring persistence was addressed by combining Redis (for caching) with PostgreSQL (for long-term storage).

- **API Consistency**: Maintaining uniformity in API design and error handling was achieved by adhering to RESTful standards and documenting all endpoints using the OpenAPI (Swagger) specification.

- **Monitoring and Observability**: Troubleshooting in a distributed system posed challenges. To overcome this, centralized logging using OpenSearch and system monitoring via Prometheus and Grafana were implemented to provide visibility into service health and metrics.

Performance Evaluation Results

The system was tested under various usage conditions to evaluate stability, responsiveness, and reliability. All core services performed as expected under concurrent access. The notification service showed significantly improved response times when Redis caching was enabled. Kafka-based asynchronous messaging provided reliable and scalable event processing. The platform's architecture successfully supports horizontal scaling and ensures each service can be developed, deployed, and scaled independently.
