Interview questions - Beyond REST

1️⃣How would you compare REST, gRPC, and GraphQL?Key Takeaways:

Feature	REST	gRPC	GraphQL
Communication Protocol	HTTP 1.1, HTTP/2	HTTP/2 (Binary)	HTTP (Text-based JSON)
Data Format	JSON, XML	Protocol Buffers (Binary)	JSON
Performance	Slower due to text-based data	High (efficient binary format, multiplexing)	Moderate (can be optimized, but parsing JSON adds overhead)
Flexibility	Fixed endpoints with predefined responses	Strongly typed contracts using `.proto` files	Highly flexible queries with field-level selection
Best for	Public APIs, simple CRUD applications	Microservices, real-time streaming, low-latency systems	Frontend-driven applications, APIs requiring flexible queries
Downsides	Over-fetching / under-fetching issues	Harder to debug due to binary serialization, steep learning curve	Complex caching, query optimization required
Streaming Support	No built-in support	Built-in bidirectional streaming	Requires WebSockets for real-time updates

- **Use REST** for simple, standardized APIs with broad adoption.
- **Use gRPC** for performance-sensitive systems requiring fast communication, especially microservices.
- **Use GraphQL** when clients need flexibility to query only required data.

2️⃣When would you use gRPC over REST?

✅ **When performance and efficiency matter:**

- gRPC uses **binary serialization (Protocol Buffers)**, which is much **faster** and more **efficient** than JSON.
- REST's text-based JSON is larger and slower to parse.

✅ **When you need real-time communication & streaming:**

- gRPC supports **bidirectional streaming** (e.g., continuous updates from a server).
- REST requires **polling**, which is inefficient for real-time applications.

✅ **For microservices communication:**

- gRPC allows direct **service-to-service calls** with **auto-generated client & server code**, making it easier to maintain.
- REST requires manual API calls with HTTP request parsing, which adds overhead.

✅ **For multi-language systems:**

- gRPC supports **automatic code generation** in multiple languages (Java, Python, Go, C++, etc.), making cross-language communication seamless.

✅ **For low-bandwidth environments:**

- gRPC's **compressed binary format** reduces data size, making it efficient for **mobile devices, IoT, and edge computing**.

🔴 **When NOT to use gRPC:**

- **If the API is public-facing** (gRPC is harder to use in web browsers, whereas REST is widely supported).
- **If debugging needs to be simple** (JSON is easier to inspect than Protocol Buffers).

3️⃣ What are the trade-offs of using GraphQL in a large-scale system?

✅ **Advantages:**
✔ **Flexible Queries:** Clients request exactly what they need, reducing over-fetching/under-fetching.
✔ **Efficient Data Fetching:** A single request can aggregate data from multiple sources.
✔ **Strongly Typed Schema:** Well-defined schema helps API evolution.

🔴 **Trade-offs & Challenges:**

❌ **Complex Caching:**

- REST APIs work well with HTTP caching (`Cache-Control`, CDNs).
- **GraphQL responses are dynamic**, making caching harder without additional tools like **DataLoader**.

❌ **Performance Overhead:**

- **Deeply nested queries** can result in expensive database joins.
- **Unoptimized queries** may overload databases.
- Requires **query cost analysis and rate limiting** to prevent abuse.

❌ **Security Concerns:**

- Allows **arbitrary queries**, which can lead to **DoS attacks** if not properly rate-limited.
- Needs **query complexity control** to avoid expensive operations.

❌ **Increased Backend Complexity:**

- Unlike REST (which maps 1:1 to resources), GraphQL **requires resolvers** for each query type, increasing backend development effort.

🚀 **Mitigation Strategies:**

- **Caching Solutions:** Use **Redis**, **CDNs**, and **Apollo Cache**.
- **Query Batching & Rate Limiting:** Limit nested queries, implement query complexity scoring.
- **GraphQL Federation:** Split schema into multiple services for **scalability**.

4️⃣ How does gRPC handle authentication & security?

✅ **Authentication in gRPC:**
1️⃣ **TLS Encryption (Transport Security)**

- gRPC uses **TLS 1.2+** for secure communication (similar to HTTPS).
- Ensures encryption of all requests and responses.

2️⃣ **Authentication Mechanisms:**

- **OAuth 2.0**: Use JWT tokens for authentication.
- **mTLS (Mutual TLS)**: Ensures **both client & server verify each other's identity**.
- **API Keys**: Lightweight authentication for internal services.

✅ **Security Features in gRPC:**
✔ **Token-based Authentication**: Pass JWT or API keys in metadata.
✔ **mTLS (Mutual Authentication)**: Used in microservices for extra security.
✔ **Authorization with RBAC**: Enforce **role-based access control** at the service level.
✔ **Rate Limiting**: Prevent **DoS attacks** by limiting the number of requests.
✔ **Logging & Monitoring**: Use tools like **gRPC interceptors** for monitoring requests.

🔴 **Potential Risks & Solutions:**

- **Man-in-the-middle attacks?** → Use **TLS encryption**.
- **Leaked API keys?** → Rotate and **store in a secure vault (e.g., AWS Secrets Manager)**.
- **DoS Attacks?** → Implement **rate limiting & request validation**.

5️⃣ How do you scale GraphQL APIs efficiently?

✅ **Scaling Strategies for GraphQL:**

1️⃣ **Use GraphQL Federation (Distributed GraphQL Services)**

- Split GraphQL schema into multiple **federated services** instead of a monolithic server.
- Tools: **Apollo Federation**, **Hasura Remote Schemas**.

2️⃣ **Implement Query Caching**

- Use **Apollo Cache**, **Redis**, or **CDN** to store frequently accessed responses.
- Example: **Persisted Queries** cache popular GraphQL queries.

3️⃣ **Optimize Database Queries**

- Use **DataLoader** to batch multiple queries into a **single request**, reducing database calls.
- Convert **deeply nested GraphQL queries** into efficient **SQL joins**.

4️⃣ **Enforce Query Complexity Limits**

- **Prevent expensive queries** by limiting **query depth & execution time**.
- Tools: **GraphQL Shield**, **Persisted Queries**.

5️⃣ **Use Server-Side Pagination & Rate Limiting**

- Implement **cursor-based pagination** instead of fetching large datasets at once.

Example:

```
query {
  users(first: 10, after: "cursor123") {
    edges {
      node { id, name, email }
    }
  }
}
```

-

6️⃣ **Horizontal Scaling (Load Balancers & Clusters)**

- Deploy GraphQL servers behind **load balancers (Nginx, AWS ALB)** to handle more requests.
- Use **containerized deployment** (Kubernetes, Docker) to autoscale GraphQL instances.

🚀 **Best Practices for GraphQL at Scale:**
✔ Use **Apollo Gateway** for federated architecture.
✔ **Cache query results** to reduce database load.

✔ **Rate-limit deep/nested queries** to prevent abuse.
✔ **Monitor query performance** with tools like **GraphQL Metrics** & **Datadog**.

Final Thoughts

✅ **REST is simple, gRPC is fast, GraphQL is flexible**.
✅ **gRPC is best for microservices, GraphQL for frontend-heavy applications**.
✅ **Security & performance tuning is critical** in both gRPC & GraphQL.
✅ **Scaling GraphQL requires caching, pagination, and federation**.