D:\OneDrive\Documents\Desktop\report.md

Sort Answer Questions (Software Architecture)

1. Importance of Software Architecture in the Development Process

Software architecture is important because it provides a **high-level blueprint** of the system. It defines the system's structure, components, and interactions, helping developers understand how the system should be built. A good architecture improves **maintainability, scalability, performance, and security**. It also reduces development risks, guides technology choices, and ensures that both functional and non-functional requirements are met efficiently.

---

2. Role of Stakeholders in Architecture Design

Stakeholders play a critical role by providing **requirements, constraints, and expectations**. These include business managers, end users, developers, system administrators, and security teams. Their input helps architects balance business goals, technical feasibility, cost, performance, and security. Active stakeholder involvement ensures that the architecture aligns with real-world needs and organizational objectives.

---

3. Difference Between Architectural Patterns and Design Patterns

Architectural patterns define the **overall structure of a system**, such as MVC, layered architecture, or microservices. They address high-level concerns like scalability and deployment. Design patterns, on the other hand, solve **specific software design problems** within components, such as Singleton, Factory, or Observer. In short, architectural patterns operate at the **system level**, while design patterns work at the **code level**.

---

4. Concept of Technical Debt in Software Architecture

Technical debt refers to the **future cost** incurred when quick or sub-optimal architectural decisions are made to save time or resources. Examples include poor modularization or ignoring scalability. While technical debt may speed up initial development, it increases maintenance difficulty, reduces performance, and raises long-term costs if not managed properly.

---

5. Trade-offs Between Monolithic and Microservices Architectures

A **monolithic architecture** is simpler to develop, test, and deploy but becomes hard to scale and maintain as the system grows. A **microservices architecture** improves scalability, flexibility, and independent deployment but introduces complexity in communication, monitoring, and security. The choice depends on system size, team expertise, and scalability requirements.

---

6. Role of Scalability in Architectural Decisions

Scalability determines how well a system can handle **increasing workload or users**. Architectural decisions such as using load balancers, distributed systems, caching, or microservices directly affect scalability. Designing for scalability ensures long-term performance, availability, and user satisfaction as system demand grows.

---

7. Importance of Documentation in Software Architecture

Architecture documentation is important because it **communicates design decisions**, system structure, and constraints to developers and stakeholders. It helps with onboarding new team members, maintenance, and future system evolution. Proper documentation reduces misunderstandings and ensures consistency throughout the development lifecycle.

---

8. Concept of Separation of Concerns in Architecture

Separation of concerns means dividing a system into **distinct components**, each handling a specific responsibility. This improves modularity, readability, and maintainability. For example, separating user interface, business logic, and data access allows changes in one part without affecting others.

---

9. Role of Middleware in Distributed Systems

Middleware acts as a **communication layer** between different components or services in a distributed system. It handles tasks such as message passing, data translation, authentication, and transaction management. Middleware simplifies development by allowing systems to communicate without needing to manage low-level network details.

---

10. Impact of Security Considerations on Architectural Design

Security strongly influences architectural design decisions such as authentication mechanisms, access control, encryption, and network isolation. Architects must consider threats like data breaches and unauthorized access. Incorporating security early in architecture helps protect sensitive data, ensure compliance, and reduce vulnerabilities in the system.

long answer ->

1. Architecture vs. Implementation

Separating architectural decisions from implementation details is critical in enterprise software systems because architecture represents **long-term, high-impact decisions**, while implementation focuses on **short-term, changeable details**. Architectural decisions define system structure, technology boundaries, communication patterns, and quality attributes such as scalability and security.

This separation improves **long-term maintenance** by allowing implementation technologies, frameworks, or libraries to change without affecting the overall system structure. It also supports **system evolution**, as new requirements can be accommodated by modifying components internally rather than redesigning the entire architecture. Without this separation, systems become tightly coupled, difficult to maintain, and costly to evolve.

---

2. Monolithic vs. Microservices Trade-offs (E-commerce Startup Case)

For a mid-sized e-commerce startup with limited DevOps expertise, choosing a **monolithic architecture** is often more practical despite microservices being popular. A monolithic system is easier to **develop, deploy, test, and monitor**, requiring fewer infrastructure components and less operational complexity.

Microservices introduce challenges such as service discovery, container orchestration, distributed logging, and network security, which demand strong DevOps maturity. For a startup focused on fast delivery and stability, a monolithic architecture reduces risk, operational cost, and team overhead, while still allowing future migration to microservices if scalability demands increase.

---

3. Quality Attribute Conflicts: Performance vs. Security

In a real-time video streaming application, **performance and security often conflict**. Strong security measures such as encryption, authentication, and deep packet inspection add processing overhead, increasing latency and reducing throughput, which negatively impacts real-time performance.

Architectural compromises include:

- **TLS termination at edge servers** to reduce encryption overhead
- **Token-based authentication** instead of repeated credential checks
- **Content Delivery Networks (CDNs)** to offload traffic securely

Patterns such as **edge computing** and **secure caching** help balance both attributes by maintaining security while optimizing performance.

---

4. Event-Driven Architecture Justification for IoT Platforms

An IoT platform handling millions of sensor events per minute benefits from an **event-driven architecture (EDA)** rather than request-response because of its asynchronous and decoupled nature.

Key advantages include:

1. **High scalability** – events can be processed independently and in parallel.
2. **Loose coupling** – producers and consumers do not depend on each other.
3. **Resilience** – message queues and brokers prevent data loss during failures.

EDA allows the system to efficiently process massive event streams without blocking or overwhelming services.

---

5. Database Technology Selection (Polyglot Persistence)

In a social media application, structured user data (profiles, authentication) fits well in a **relational database** due to strong consistency and transactional integrity. However, real-time activity feeds require high write throughput, horizontal scalability, and flexible schemas, which are better handled by **NoSQL databases**.

Using both databases enables **polyglot persistence**, where each database technology is chosen based on specific data access patterns and performance requirements, resulting in better scalability and responsiveness.

---

6. Caching Strategy Rationale in Financial Systems

Caching in a financial reporting system is more complex than in e-commerce because financial data requires **strong consistency and accuracy**. Unlike product listings, financial reports cannot tolerate stale or incorrect data.

Concerns influencing caching strategy include:

- **Data freshness and validity**
- **Regulatory compliance**
- **Cache invalidation timing**

Architects may use **read-through caches with strict expiration**, or cache only aggregated, non-critical data to minimize the risk of inconsistent financial reports.

---

7. API Gateway Necessity

In a microservices architecture with 50+ services, an **API Gateway** becomes essential to manage cross-cutting concerns such as authentication, routing, rate limiting, and monitoring in a centralized manner.

Without an API Gateway, each service must implement these concerns independently, leading to duplication and inconsistency. In contrast, for a system with only 3–4 services, an API Gateway may add unnecessary complexity and latency, making it an example of overengineering.

---

8. Architecture Documentation Value in Distributed Teams

Up-to-date architecture documentation is crucial when teams are distributed across time zones or organizations because it serves as a **single source of truth**. It reduces dependency on synchronous communication, minimizes misunderstandings, and ensures architectural consistency.

Documentation helps new teams onboard faster, supports parallel development, and prevents architectural drift caused by isolated decision-making across teams.

---

9. Failure Tolerance vs. Cost in Healthcare Systems

In a healthcare monitoring system, failure can lead to **life-threatening consequences**. Implementing redundant services across multiple availability zones ensures high availability, fault tolerance, and continuous monitoring.

Despite higher costs, redundancy is justified because:

- System downtime is unacceptable
- Patient safety is critical
- Regulatory standards demand high reliability

In such systems, **reliability outweighs cost considerations**.

---

10. Evolutionary Architecture vs. Big Design Upfront

Evolutionary architecture focuses on **incremental change and continuous adaptation**, unlike traditional big design upfront approaches that attempt to define the entire system in advance.

Evolutionary architecture is favored when:

- Business requirements change frequently
- Market uncertainty is high
- Rapid experimentation is needed

This approach allows systems to evolve alongside business needs, reducing risk and enabling faster innovation.

Fill in the Blanks – Software Architecture (Answers)

1. The **Model–View–Controller (MVC)** pattern separates an application's data, user interface, and control logic into three interconnected components.
2. **Microservices** is a software architecture style that structures an application as a collection of loosely coupled services.
3. The **Single Responsibility** principle states that a class should have only one reason to change.
4. **Adapter** is a pattern that allows incompatible interfaces to work together.
5. In client-server architecture, the **server** handles requests from clients.
6. The **Facade** pattern provides a unified interface to a set of interfaces in a subsystem.
7. **Layered** architecture organizes components into layers, each with specific responsibilities.
8. The **Singleton** pattern ensures a class has only one instance and provides global access to it.
9. **Maintainability** is the process of designing software to minimize the impact of changes.
10. The **Strategy** pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.
11. **REST (Representational State Transfer)** is an architectural style for distributed hypermedia systems.
12. The **Builder** pattern separates the construction of a complex object from its representation.
13. **Scalability** refers to the ability of a system to handle increasing amounts of work.

14. The **Observer** pattern defines a one-to-many dependency between objects.
15. **MVC (Model–View–Controller)** architecture divides an application into model, view, and controller components.
16. The **Open–Closed** principle states that software entities should be open for extension but closed for modification.
17. The **Proxy** pattern provides a surrogate or placeholder for another object.
18. **Maintainability** is a measure of how well components can be understood and modified.
19. The **Decorator** pattern allows adding behavior to individual objects dynamically.
20. **Event-driven** architecture uses events to trigger and communicate between services.
21. The **Dependency Inversion** principle suggests that high-level modules should not depend on low-level modules.
22. The **Adapter** pattern converts the interface of a class into another interface clients expect.
23. **Interoperability** is the ability of different systems to work together.
24. The **Composite** pattern composes objects into tree structures to represent part-whole hierarchies.
25. **Hybrid** architecture combines both synchronous and asynchronous communication patterns.
26. The **Interface Segregation** principle advises that many client-specific interfaces are better than one general-purpose interface.
27. The **Iterator** pattern provides a way to access elements of an aggregate object sequentially.
28. **Reliability** is the probability that a system will perform its intended function under stated conditions.
29. The **Template Method** pattern defines a skeleton of an algorithm in a method.
30. **Microservices** architecture uses independent, self-contained units that communicate via APIs.
31. The **Law of Demeter** principle states that a component should not know about the internal details of other components.
32. The **Memento** pattern captures and restores an object’s internal state.
33. **Fault tolerance** is the ability of a system to continue operating properly in the event of failure.

34. The **Visitor** pattern separates an algorithm from the object structure on which it operates.
35. **Cloud-native** architecture emphasizes horizontal scaling and stateless components.
36. The **KISS (Keep It Simple, Stupid)** principle suggests keeping software designs as simple as possible.
37. The **Visitor** pattern represents an operation to be performed on elements of an object structure.
38. **Security** is the ability of a system to protect data and resources from unauthorized access.
39. The **Factory Method** pattern creates objects without specifying their concrete classes.
40. **Mediator** architecture uses a central hub to manage communication between components.
41. The **Low Coupling, High Cohesion** principle recommends that classes should be loosely coupled and highly cohesive.
42. The **Mutex (Mutual Exclusion)** pattern ensures only one object can access a resource at a time.
43. **Testability** is the ease with which a system can be tested.
44. The **Observer** pattern defines a dependency between objects so when one changes state, all are notified.
45. **Domain-driven** architecture organizes code around business capabilities.
46. The **Dependency Inversion** principle suggests that a class should depend on abstractions, not concretions.
47. The **Facade** pattern provides a simplified interface to a complex subsystem.
48. **Agility** is the ability to make changes quickly and reliably.
49. The **State** pattern allows an object to alter its behavior when its internal state changes.
50. **Repository** architecture style uses a shared data store accessible by multiple components.

Multiple Choice Questions (Software Architecture)

1. Which pattern separates an application’s data, user interface, and control logic?

A. Layered B. MVC C. Microservices D. Client-Server **Answer: B**

---

2. Which architecture style structures an application as a collection of loosely coupled services?

A. Monolithic B. Layered C. Microservices D. Repository **Answer: C**

---

3. Which principle states that a class should have only one reason to change?

A. Open–Closed B. Single Responsibility C. Dependency Inversion D. Interface Segregation **Answer: B**

---

4. Which pattern allows incompatible interfaces to work together?

A. Facade B. Proxy C. Adapter D. Decorator **Answer: C**

---

5. In client-server architecture, which component handles client requests?

A. Client B. Middleware C. Database D. Server **Answer: D**

---

6. Which pattern provides a unified interface to a subsystem?

A. Adapter B. Facade C. Composite D. Bridge **Answer: B**

---

7. Which architecture organizes components into layers?

A. Event-driven B. Layered C. Microservices D. Pipe-and-filter **Answer: B**

---

8. Which pattern ensures a class has only one instance?

A. Factory B. Builder C. Singleton D. Prototype **Answer: C**

---

9. Which term refers to designing software to minimize the impact of change?

A. Reliability B. Maintainability C. Scalability D. Portability **Answer: B**

---

10. Which pattern defines interchangeable algorithms?

A. Strategy B. Observer C. State D. Command **Answer: A**

---

11. Which architectural style is used for distributed hypermedia systems?

A. SOAP B. RPC C. REST D. GraphQL **Answer: C**

---

12. Which pattern separates object construction from representation?

A. Factory B. Prototype C. Builder D. Abstract Factory **Answer: C**

---

13. Which term describes a system’s ability to handle increased load?

A. Reliability B. Availability C. Scalability D. Maintainability **Answer: C**

---

14. Which pattern defines a one-to-many dependency between objects?

A. Strategy B. Decorator C. Observer D. Proxy **Answer: C**

---

15. Which architecture divides an application into Model, View, and Controller?

A. MVVM B. Layered C. MVC D. Client-Server **Answer: C**

---

16. Which principle states software should be open for extension but closed for modification?

A. LSP B. ISP C. OCP D. SRP **Answer: C**

---

17. Which pattern acts as a placeholder for another object?

A. Adapter B. Proxy C. Facade D. Bridge **Answer: B**

---

18. Which quality attribute measures ease of understanding and modification?

A. Testability B. Maintainability C. Scalability D. Portability **Answer: B**

---

19. Which pattern adds behavior to objects dynamically?

A. Decorator B. Observer C. Composite D. Adapter **Answer: A**

---

20. Which architecture uses events for communication?

A. Layered B. Client-Server C. Event-driven D. Repository **Answer: C**

---

21. Which principle states high-level modules should not depend on low-level modules?

A. SRP B. OCP C. DIP D. ISP **Answer: C**

---

22. Which pattern converts one interface into another expected by clients?

A. Facade B. Proxy C. Adapter D. Strategy **Answer: C**

---

23. Which term refers to systems working together?

A. Portability B. Interoperability C. Scalability D. Availability **Answer: B**

---

24. Which pattern represents part-whole hierarchies?

A. Composite B. Decorator C. Iterator D. Visitor **Answer: A**

---

25. Which architecture combines synchronous and asynchronous communication?

A. Layered B. Event-driven C. Hybrid D. Repository **Answer: C**

---

26. Which principle suggests many small interfaces over one large interface?

A. DIP B. ISP C. SRP D. OCP **Answer: B**

---

27. Which pattern allows sequential access to elements?

A. Iterator B. Observer C. Command D. Proxy **Answer: A**

---

28. Which quality attribute measures probability of correct operation?

A. Availability B. Reliability C. Security D. Maintainability **Answer: B**

---

29. Which pattern defines an algorithm skeleton?

A. Strategy B. Visitor C. Template Method D. State **Answer: C**

---

30. Which architecture uses independent services communicating via APIs?

A. Layered B. Monolithic C. Microservices D. Client-Server **Answer: C**

---

31. Which principle says a component should not know internal details of others?

A. ISP B. DIP C. Law of Demeter D. SRP **Answer: C**

---

32. Which pattern stores and restores object state?

A. Observer B. Memento C. State D. Strategy **Answer: B**

---

33. Which term describes continued operation despite failures?

A. Reliability B. Availability C. Fault Tolerance D. Scalability **Answer: C**

---

34. Which pattern separates algorithms from object structure?

A. Visitor B. Decorator C. Adapter D. Command **Answer: A**

---

35. Which architecture emphasizes stateless services and horizontal scaling?

A. Monolithic B. SOA C. Cloud-native D. Repository **Answer: C**

---

36. Which principle promotes simplicity in design?

A. DRY B. KISS C. SOLID D. YAGNI **Answer: B**

---

37. Which pattern performs operations on object structures?

A. Iterator B. Observer C. Visitor D. Strategy **Answer: C**

---

38. Which quality attribute protects data from unauthorized access?

A. Reliability B. Security C. Scalability D. Portability **Answer: B**

---

39. Which pattern creates objects without specifying concrete classes?

A. Builder B. Factory Method C. Prototype D. Singleton **Answer: B**

---

40. Which architecture uses a central hub for communication?

A. Broker B. Mediator C. Client-Server D. Repository **Answer: B**

---

41. Which principle emphasizes loose coupling and high cohesion?

A. SRP B. KISS C. Low Coupling–High Cohesion D. DIP **Answer: C**

---

42. Which pattern ensures exclusive access to a resource?

A. Proxy B. Mutex C. Singleton D. Monitor **Answer: B**

---

43. Which quality attribute measures ease of testing?

A. Maintainability B. Testability C. Reliability D. Scalability **Answer: B**

---

44. Which pattern notifies dependent objects of state changes?

A. State B. Observer C. Strategy D. Command **Answer: B**

---

45. Which architecture organizes software around business domains?

A. Layered B. MVC C. Domain-driven D. Event-driven **Answer: C**

---

46. Which principle says depend on abstractions, not concretions?

A. SRP B. ISP C. DIP D. OCP **Answer: C**

---

47. Which pattern simplifies a complex subsystem interface?

A. Adapter B. Proxy C. Facade D. Decorator **Answer: C**

---

48. Which term describes ability to change quickly and reliably?

A. Maintainability B. Agility C. Scalability D. Reliability **Answer: B**

---

49. Which pattern allows behavior change based on internal state?

A. Strategy B. State C. Observer D. Command **Answer: B**

---

50. Which architecture uses a shared data store?

A. Microservices B. Event-driven C. Repository D. Client-Server **Answer: C**