1. System Architecture

This defines the fundamental structure and organization of a software system.[1] The choices made here impact scalability, maintainability, and deployment.[2]

Monolithic vs. Microservices

Feature	Monolithic Architecture	Microservices Architecture
Structure	A single, unified codebase where all components (UI, business logic, data access) are tightly coupled and run as a single service.	The application is broken down into a collection of small, independent services, each responsible for a specific business capability.
Deployment	The entire application must be redeployed for any change, no matter how small.	Services can be deployed independently, allowing for faster and more flexible updates.
Scalability	You must scale the entire application, even if only one small part is a bottleneck. This can be inefficient.	You can scale individual services based on their specific needs, leading to more efficient resource utilization.
Technology	Constrained to a single technology stack (e.g., all Java, all Python).	Each service can use the best technology stack for its specific job (polyglot architecture).
Complexity	Simple to develop and test initially. Becomes complex and difficult to manage as the application grows.	More complex to set up initially due to the distributed nature. Simpler to understand and manage individual services.

Fault Isolation	A failure in one component can bring down the entire application.	A failure in one service can be isolated and is less likely to impact the rest of the system (if designed correctly).

Modular Design

Modular design is a principle that applies to both monoliths and microservices. It involves breaking a system down into smaller, independent, and interchangeable modules.[3]

- **In a Monolith:** This means organizing the code into distinct packages or modules with well-defined boundaries (e.g., a "user management" module, a "payment" module).[4] This improves maintainability even if it's a single deployable unit.
- **In Microservices:** The modular boundary is the service itself.[5] Each microservice is a self-contained module.[6]

Key Architectural Qualities

- **Scalability:** The system's ability to handle a growing amount of work.[7] This is often achieved through **horizontal scaling** (adding more machines/services), which is a natural fit for microservices.[8]
- **Availability:** The percentage of time the system is operational and accessible. It's often expressed in "nines" (e.g., 99.9% is "three nines"). High availability is achieved through redundancy, failover mechanisms, and eliminating single points of failure.[9] Microservices can improve availability because a single service failure doesn't have to cause a total system outage.[10]
- **Consistency:** In distributed systems, this ensures that all nodes or services see the same data at the same time.[11] There is often a trade-off between consistency and availability, as defined by the **CAP Theorem** (Consistency, Availability, Partition Tolerance).[12] Systems may choose **Strong Consistency** (all replicas have the same data at all times) or **Eventual Consistency** (replicas will become consistent over time).[13]

---

2. Infrastructure

This refers to the core hardware and software components required to run the application.

- **Databases:** The heart of most applications, responsible for persistent data storage.[14] The choice between SQL (for structured data with transactional needs) and NoSQL (for unstructured data and high scalability) is a critical architectural decision.
- **Storage:** This includes block storage for server disks (like Amazon EBS), object storage for files, images, and backups (like Amazon S3 or Google Cloud Storage), and file storage for shared access.
- **CPU (Compute):** The processing power required to run your application logic. The amount of CPU needed depends on the computational intensity of your tasks.
- **Bandwidth (Network):** The amount of data that can be transferred to and from your servers.[15] This is crucial for user-facing applications and data-heavy internal communication.

Appropriate Scaling Methods

- **Vertical Scaling (Scaling Up):** Increasing the resources of a single server (more CPU, more RAM).[16] It's simple but has a hard limit and can be expensive.
- **Horizontal Scaling (Scaling Out):** Adding more servers to distribute the load.[17] This is the standard for modern, scalable applications and is enabled by load balancers.[18]
- **Auto-Scaling:** Automatically adjusting the number of servers based on real-time metrics like CPU utilization or request count.[19] This is a cost-effective way to handle variable traffic.

---

3. Security

Security is not an afterthought; it must be designed into the system from the beginning.[20]

- **JWT (JSON Web Token):** A standard for creating access tokens. When a user logs in, the server generates a JWT containing user details (the "payload") and signs it with a secret key.[21] The token is sent to the client, which includes it in the header of subsequent requests.[22] The server can then verify the token's signature to authenticate the user without needing to look up session data in a database, making it ideal for stateless, distributed systems.[23]
- **IP Whitelisting:** A security measure where you create a list of approved IP addresses that are allowed to access your system or specific endpoints.[24] This is commonly used

to restrict access to sensitive internal admin panels or APIs to only office or VPN IP addresses.[25]

- **RSA Encryption:** An **asymmetric encryption** algorithm that uses a key pair: a **public key** (which can be shared with anyone) and a **private key** (which must be kept secret).[26]
    - **Encryption:** Data encrypted with the public key can *only* be decrypted with the corresponding private key.[27] This is used to securely transmit data. A client can encrypt data with a server's public key, knowing that only the server can read it.
    - **Digital Signatures:** Data signed with the private key can be verified by anyone with the public key. This is used to verify the identity of the sender (e.g., JWT signatures).
- **PII (Personally Identifiable Information) Handling:** PII is any data that can be used to identify a specific individual (e.g., name, address, email, phone number, national ID).[28] Handling PII requires strict security measures:
    - **Encryption at Rest:** Encrypting the data when it is stored in the database or on disk.[29]
    - **Encryption in Transit:** Using TLS/SSL to encrypt data as it travels over the network.[30]
    - **Access Control:** Limiting who can access PII data based on their role.[31]
- **Data Protection Compliance:** Adhering to legal frameworks for data protection, such as:
    - **GDPR (General Data Protection Regulation):** EU law governing data protection and privacy for all individual citizens of the European Union and the European Economic Area.[32]
    - **CCPA (California Consumer Privacy Act):** A state statute intended to enhance privacy rights and consumer protection for residents of California.[33]

---

4. Observability

Observability is about understanding the internal state of your system by observing its external outputs.[34] It is often described by three pillars: logs, metrics, and traces.[35]

- **Tools:**

    - **Prometheus:** A powerful open-source tool for collecting and storing time-series data (**metrics**).[36] It periodically "scrapes" metrics from configured endpoints on your services.[37]
    - **Grafana:** An open-source visualization tool.[38] It connects to data sources like Prometheus to create dashboards, graphs, and alerts, allowing you to visualize your system's health and performance.[39]
    - **OpenTelemetry:** An open-source observability framework that provides a standardized way to generate and collect telemetry data (metrics, logs, and traces) from your applications.[40] It aims to unify how you instrument your code.
- **Trace/Span IDs for Debugging:** In a microservices architecture, a single user request might travel through dozens of services. Distributed tracing is essential for debugging.[41]
    - **Trace ID:** A unique identifier assigned to an entire request as it enters the system.[42] This ID is passed along to every service that the request touches.
    - **Span ID:** A unique identifier for a single unit of work or operation within a service (e.g., a database call or an API call to another service).[43]
    - By logging the `Trace ID` and `Span ID` in every service, you can aggregate all the logs for a single request and visualize the entire call graph, making it easy to pinpoint which service is causing an error or latency.[44]

---

5. Architecture Migration

Moving from a monolith to a microservices architecture is a complex process that must be done carefully.[45]

Strategies for Transitioning from Monolith to Microservices

- **Strangler Fig Pattern:** This is the most common and recommended approach. Instead of a "big bang" rewrite, you gradually "strangle" the monolith by replacing its functionality piece by piece with new microservices.[46]
    - An **API Gateway** or proxy is placed in front of the monolith.[47]
    - You identify a module to extract (e.g., user profiles).[48]
    - You build a new microservice for user profiles.

    - You update the API Gateway to route all calls for user profiles to the new microservice instead of the monolith.
    - Repeat this process for other modules until the monolith is either gone or has shrunk to a manageable core.
- **Handling Database Calls:** This is often the hardest part.
    - **Shared Database:** Initially, the new microservice might still read from and write to the monolith's database. This is a temporary anti-pattern but can ease the transition.
    - **Database per Service:** The ideal state is for each microservice to own its own data. This requires carefully migrating data from the monolith's database to a new database for the service and refactoring the monolith to call the new service's API for that data instead of accessing the table directly.[49]

---

6. System Estimations (Back-of-the-Envelope Calculations)

These are quick, approximate calculations to estimate system requirements. They are crucial in system design interviews to demonstrate your ability to reason about scale.

Example: Estimating Bandwidth and Storage for a Simple Image Service

Assumptions:

- 1 million Daily Active Users (DAU).
- Each user uploads 1 image per day on average.
- Each user views 20 images per day on average.
- Average image size: 200 KB.

Calculations:

- **Storage Estimation (per day):**
    - 1,000,000 users/day * 1 image/user * 200 KB/image = 200,000,000 KB/day = 200 GB/day
    - **Storage for 1 year:** 200 GB/day * 365 days = 73,000 GB = 73 TB
- **Bandwidth Estimation (Egress/Outbound Traffic per day):**
    - 1,000,000 users/day * 20 images/user * 200 KB/image = 4,000,000,000 KB/day = 4,000 GB/day
    - **Bandwidth per second (average):** 4,000 GB / (24 hours * 3600 sec/hour) ≈ 46 MB/s
    - You would need to design for peak traffic, which might be 2-3x the average, so roughly ~100-150 MB/s.
- **Cost Estimation (using AWS S3 and CloudFront as an example):**
    - **S3 Storage Cost:** Look up the current price for S3 Standard (e.g., ~$0.023 per GB/month).[50] For 73 TB, this would be a significant cost.

    - **CDN/Bandwidth Cost:** Look up AWS CloudFront data transfer out pricing (e.g., ~$0.085 per GB).[51] `4,000 GB/day * 30 days * $0.085/GB = $10,200/month`.

These rough numbers help you make informed decisions about technology choices (e.g., the need for a CDN is immediately obvious) and infrastructure capacity.

---

7. Database Scaling

As your application grows, the database often becomes the bottleneck.[52]

- **Replication:**
    - This involves creating copies (replicas) of your database. A common pattern is **Primary-Replica (or Master-Slave) Replication**.[53]
    - All **writes** go to the primary database.
    - The primary database then replicates the changes to one or more replica databases.[54]
    - All **reads** are directed to the replicas.[55]
    - This is an excellent way to scale for **read-heavy loads**, as you can add more replicas to handle increased read traffic. It also provides high availability; if the primary fails, a replica can be promoted to become the new primary.[56]
- **Sharding (Partitioning):**
    - Replication helps with read loads but not write loads, as all writes still go to one primary server.[57] Sharding solves this by partitioning the data across multiple databases (shards).[58]
    - Each shard contains a subset of the data.[59] For example, you could shard by `user_id`, where users 1-1,000,000 are on Shard A, users 1,000,001-2,000,000 are on Shard B, and so on.[60]
    - When a request comes in, the application logic (or a proxy layer) uses a **shard key** (e.g., `user_id`) to determine which shard contains the relevant data.
    - Sharding allows for massive write scalability but adds significant complexity to the application.[61]

---

8. Interview Prep Focus Areas

For system design interviews, focus on demonstrating a structured thought process.

- **Load Balancers:** Understand their role in distributing traffic, enabling horizontal scaling, and improving availability. Know the difference between Layer 4 (TCP) and Layer 7 (HTTP) load balancing.
- **API Gateways:** Understand that they are more than just load balancers. They act as a single entry point for clients, handling concerns like authentication (JWT validation), rate limiting, routing to different microservices, and response aggregation.[62]
- **Real-World Problem Solving:**
    1. **Clarify Requirements:** Always start by asking questions to understand the functional and non-functional requirements (e.g., "How many users?", "What is the expected latency?", "Is consistency critical?").
    2. **Make High-Level Decisions:** Start with a simple architecture (e.g., a load balancer, a few web servers, a database).
    3. **Identify Bottlenecks:** As you discuss scale, identify potential bottlenecks (e.g., "The database will be read-heavy").
    4. **Introduce Solutions:** Propose solutions for these bottlenecks (e.g., "We can add a caching layer," "We can use database replication").
    5. **Be Pragmatic:** Discuss trade-offs. No solution is perfect. Explaining *why* you chose one approach over another (e.g., "I chose eventual consistency to improve availability") shows a deep understanding.
    6. **Do Back-of-the-Envelope Calculations:** Use estimations to justify your design choices.