🚀 🚀 Key Technical Differences in Modern Architecture & Systems 🚀 🚀

---

1. State vs. Stateless

State

- **Definition**: A "stateful" system retains some information about a client session or process over time. This means the system can track ongoing interactions, store intermediate results, or maintain context about the user (e.g., items in a shopping cart).
- **Implementation**: Common mechanisms include server memory, session storage, databases, or caches that tie requests from the same client to previously stored context.
- **Pros**:
    - Allows rich, interactive experiences where the system remembers prior actions.
    - Can simplify logic on the client side, since the server holds the state.
- **Cons**:
    - Increased complexity in managing session data, especially across multiple servers.
    - Must handle session persistence and replication for load balancing or failover.
    - Scalability can be more challenging because state must be shared or migrated.

Stateless

- **Definition**: A "stateless" system does not persist client data between requests. Each request is treated independently, without knowledge of prior actions.
- **Implementation**: Typically, stateless architectures rely on client-side tokens (e.g., JWT—JSON Web Tokens) or other mechanisms so that the server can remain stateless.
- **Pros**:
    - Easier to scale horizontally because no session synchronization is required.
    - Simpler fault tolerance; any server can handle any request.
- **Cons**:
    - Requires careful design to handle context. The client or a central store must manage stateful information (if needed).
    - Certain use cases might become more complex (e.g., multi-step workflows).

---

2. Server vs. Serverless

Server

- **Definition**: Traditional "server-based" computing involves managing physical or virtual servers. You provision the machine, handle OS updates, patch vulnerabilities, and scale as needed by adding or upgrading servers.
- **Implementation**: Data centers, on-premise hardware, or virtual machines in the cloud (e.g., EC2 instances on AWS).
- **Pros**:
    - Complete control over the environment and configuration.
    - Predictable performance if sized correctly.
- **Cons**:
    - You are responsible for capacity planning, scaling, maintenance, and updates.
    - Costs may be higher if resources are underutilized.

Serverless

- **Definition**: "Serverless" doesn't mean there are no servers at all; rather, the underlying servers are fully managed by a cloud provider. You only pay for the actual compute time used (e.g., function execution time in AWS Lambda, Azure Functions, or Google Cloud Functions).
- **Implementation**: You deploy code in small functions. The cloud platform automatically spins up and shuts down environments as needed.
- **Pros**:
    - Zero maintenance burden on provisioning or managing servers.
    - Rapid auto-scaling based on demand.
    - Pay-per-use model can reduce costs significantly for intermittent workloads.
- **Cons**:
    - Limited execution time in some platforms (e.g., Lambda has a max timeout).
    - Cold starts can introduce latency.
    - Less control over runtime environment.

3. Agent vs. Agentless

Agent

- **Definition**: An agent-based approach requires installing a piece of software (an "agent") on the target system to collect metrics, logs, or perform actions.
- **Implementation**: Tools like Datadog Agent, New Relic Agent, or security scanning agents run as daemons or services on servers.
- **Pros**:
    - Deep visibility and control: Agents can gather detailed system metrics and perform actions locally.
    - Can operate even with intermittent network connectivity (storing data locally until connectivity is restored).
- **Cons**:
    - Requires installing and updating the agent on each host.
    - Can consume system resources and introduce complexity.

Agentless

- **Definition**: Agentless approaches gather information or perform actions remotely, typically using standard protocols (e.g., SSH, SNMP, WMI).
- **Implementation**: A centralized system connects to hosts without installing additional software, using existing system credentials and protocols.
- **Pros**:
    - Less overhead on target systems—no extra software to install.
    - Simpler deployment and maintenance.
- **Cons**:
    - May have less granular data or control compared to agents.
    - Network connectivity and protocol limits can reduce effectiveness.
    - Potential security concerns with remote credential management.

---

4. Concurrent vs. Simultaneous Users

Concurrent Users

- **Definition**: The number of users or sessions that are *active* within a certain timeframe, even if they are not performing actions at the exact same moment. For instance, if a user logs in and is idle, they might still be considered "concurrent."

- **Technical Note**: Concurrency often relates to resource usage in server applications, such as how many open connections the server can handle.

Simultaneous Users

- **Definition**: Users who are performing operations *at the exact same moment*. This is often a subset of concurrent users, as not all concurrent users are necessarily pressing "submit" at once.
- **Technical Note**: "Simultaneous" is particularly important in stress testing because it measures how the system handles truly concurrent transactions or requests.

---

5. Synchronous vs. Asynchronous

Synchronous

- **Definition**: In synchronous operations, the caller waits for a response or completion of a task before moving on.
- **Implementation**: Typical HTTP request/response patterns or function calls in many programming languages are synchronous by default.
- **Pros**:
    - Simpler flow control and logic (step-by-step).
    - Easy to understand and debug.
- **Cons**:
    - The calling thread or process is blocked until the response arrives, leading to potential performance bottlenecks.
    - Less efficient for I/O-heavy or long-running tasks.

Asynchronous

- **Definition**: The caller dispatches a task and can continue other work without waiting for the task to complete. A callback, promise, or event is used to handle the result when ready.
- **Implementation**: Event loops (Node.js), message queues (RabbitMQ, Kafka), or async libraries in various languages.
- **Pros**:
    - Can significantly improve throughput in I/O-bound scenarios.
    - More responsive systems under high load.

- **Cons**:
    - More complex programming model (callbacks, promises, concurrency considerations).
    - Harder to debug and trace.

---

6. Publisher vs. Subscriber

Publisher

- **Definition**: In a pub/sub (publish-subscribe) system, the "publisher" produces messages or events and sends them to a messaging channel.
- **Implementation**: Publishers do not typically know who the subscribers are. They just publish to a broker or topic (e.g., Kafka, Redis Pub/Sub, MQTT).
- **Pros**:
    - Decouples message producers from consumers.
    - Highly scalable when combined with a robust messaging platform.
- **Cons**:
    - Publishers must rely on the broker's availability and quality of service.

Subscriber

- **Definition**: A "subscriber" registers interest in certain channels or topics. When a message is published to that topic, the messaging system delivers it to all subscribed parties.
- **Implementation**: Subscribers may pull from the broker or receive push notifications.
- **Pros**:
    - Subscribers only receive messages of interest.
    - System easily extends with new subscribers without changing publisher code.
- **Cons**:
    - Must handle message processing and possible concurrency or ordering issues.
    - Reliability depends on message broker and subscriber's handling logic.

---

7. Monolithic vs. Microservices

Monolithic

- **Definition**: A single, large application in which all components (UI, business logic, data access) run in one process or as one deployable artifact.
- **Pros**:
  - Simpler initial development: everything in one codebase, one deployment pipeline.
  - Easier to do cross-cutting concerns (e.g., logging, security) in a single place.
- **Cons**:
  - Harder to scale individual components. You often must scale the entire app.
  - Can become very large and difficult to maintain over time (a "big ball of mud").
  - Deployments become riskier; a single bug can require redeploying the entire system.

Microservices

- **Definition**: An architecture style where the application is decomposed into smaller, independently deployable services, each responsible for a specific business capability.
- **Pros**:
  - Fine-grained scalability and fault isolation. One service can scale or fail independently.
  - Each service can use the most appropriate technology stack.
  - Smaller codebases for each service, potentially easier to maintain.
- **Cons**:
  - Operational complexity (service discovery, networking, observability).
  - Managing distributed transactions is more complex.
  - Potential for higher latency due to inter-service communication.

8. Vertical vs. Horizontal Scaling

Vertical Scaling

- **Definition**: Increasing the capacity of a single machine (or instance) by adding more CPU, RAM, or storage.
- **Pros**:
  - Can be simpler to implement because you upgrade one server.
  - Less complexity in terms of distributed systems or data partitioning.
- **Cons**:
  - There is usually an upper limit to how much you can scale vertically ("hardware ceiling").
  - Not always cost-effective at large scales.

Horizontal Scaling

- **Definition**: Adding more machines (or instances) to distribute workload. Typically used in clustered or distributed environments.
- **Pros**:
  - Virtually unlimited scaling potential if designed correctly.
  - Offers redundancy and high availability (multiple servers).
- **Cons**:
  - Requires load balancing, data partitioning, or replication.
  - More complex to manage and design for consistency in distributed systems.

---

9. Load Balancing vs. Auto Scaling

Load Balancing

- **Definition**: The process of distributing network or application traffic across multiple servers to ensure no single server becomes a bottleneck.
- **Implementation**: Commonly implemented via hardware appliances (F5) or software-based solutions (HAProxy, Nginx, AWS Elastic Load Balancer).
- **Pros**:
  - Improves application availability and responsiveness.
  - Can perform health checks and route traffic away from unhealthy instances.

- **Cons**:
    - Adds an extra layer in the architecture that must be managed.
    - Requires careful setup to handle session persistence if the application is stateful.

Auto Scaling

- **Definition**: Dynamically adjusting the number of computing resources (e.g., servers, containers) based on demand metrics (CPU usage, request throughput, etc.).
- **Implementation**: Cloud providers offer auto-scaling services (AWS Auto Scaling, Azure Autoscale, etc.) that spin up or shut down instances.
- **Pros**:
    - Cost-efficient as you only pay for resources when needed.
    - Helps handle sudden traffic spikes without manual intervention.
- **Cons**:
    - Requires good metrics and thresholds to avoid thrashing (constantly scaling up and down).
    - Scaling actions may introduce short delays.

---

10. Containerization vs. Virtualization

Containerization

- **Definition**: A lightweight method of packaging software and its dependencies into isolated environments (containers) on top of a single operating system kernel (e.g., Docker, Kubernetes).
- **Pros**:
    - Faster start-up times and lower overhead compared to VMs.
    - Consistent runtime environment across different deployment environments.
- **Cons**:
    - Isolation is at the process level, so there is a shared kernel among containers.
    - Less isolation than full virtual machines in certain security scenarios.

Virtualization

- **Definition**: Running multiple operating systems or instances (virtual machines) on a single physical host, each VM having its own OS kernel.
- **Implementation**: Hypervisors like VMware ESXi, Microsoft Hyper-V, or KVM.

- **Pros**:
    - Strong isolation, as each VM is a full OS instance.
    - Well-established technology for server consolidation.
- **Cons**:
    - Heavier on system resources (RAM, CPU) due to multiple OS kernels.
    - Slower boot times compared to containers.

---

11. Session Persistence vs. Sticky Session

Often used interchangeably, but there's a subtle difference in how they are discussed:

Session Persistence

- **Definition**: The concept that sessions should remain accessible across different requests. This can be implemented via a shared session store (like Redis or a database) so any server can retrieve session information.
- **Implementation**: Typically uses a central store, meaning any server in a load-balanced environment can fetch the session's state from this store.
- **Pros**:
    - True load balancing: requests can be served by any node without losing context.
    - Scalable as the session state is managed externally.
- **Cons**:
    - Requires an external session management system, adding complexity.
    - Potential performance or network overhead.

Sticky Session

- **Definition**: A load balancer "sticks" a client to a specific backend server for the duration of the session. Subsequent requests from the same client go to that same server.
- **Implementation**: Load balancers set cookies (like a "route" parameter) or rely on IP hashing.
- **Pros**:
    - Simplifies stateful scenarios—no external session store required.

- **Cons**:
    - Uneven load distribution if some sessions are much more demanding.
    - If a server fails, the user's session is lost (unless you have a fallback system).

---

12. In-Memory Caching vs. Distributed Caching

In-Memory Caching

- **Definition**: Storing frequently accessed data in RAM on a single server or instance (e.g., using a simple memory cache or frameworks like Ehcache in Java).
- **Pros**:
    - Extremely fast access times since data is in local memory.
    - Simple to implement for smaller scale or single-server scenarios.
- **Cons**:
    - Data is lost if the server restarts.
    - Not scalable across multiple servers without additional replication logic.

Distributed Caching

- **Definition**: A cache that spans multiple nodes in a cluster (e.g., Redis Cluster, Memcached, Hazelcast). Data can be shared or partitioned across multiple machines.
- **Pros**:
    - Can handle large data sets and high traffic volumes.
    - High availability and fault tolerance with replication across nodes.
- **Cons**:
    - Network overhead when accessing remote cache nodes.
    - Complexity in setup and configuration, especially with partitioning and replication.

---

13. CAP Theorem

Definition

- **CAP** stands for **Consistency**, **Availability**, and **Partition Tolerance**. Eric Brewer's theorem states that in a distributed system, you can only fully guarantee two of these three properties at any time.
- **Properties**:
    1. **Consistency**: Every read receives the most recent write or an error.
    2. **Availability**: Every request receives a response (without a guarantee that it is the latest data).
    3. **Partition Tolerance**: The system continues to operate despite arbitrary message loss or failures in part of the system.
- **Implications**:
    - Systems must make trade-offs when network partitions occur.
    - Consistency + Availability is not fully possible if a network partition occurs; either you wait (sacrifice availability) or serve stale data (sacrifice consistency).
- **Practical Examples**:
    - **CP systems** (e.g., some database clusters) choose consistency over availability in case of a partition.
    - **AP systems** (e.g., DynamoDB in certain configurations) choose availability over strong consistency.
    - **CA** is generally only possible in non-distributed or partition-free scenarios (which is often unrealistic).

---

14. Push vs. Pull Communication

Push

- **Definition**: The server or producer proactively sends (pushes) data to the client or consumer when new data is available.
- **Implementation**: WebSockets, Server-Sent Events, MQTT push notifications, or email alerts.
- **Pros**:
    - Real-time updates without clients constantly polling.
    - Efficient for event-driven scenarios.

- **Cons**:
    - Requires a persistent or semi-persistent connection (which can be more complex to manage).
    - Can be more challenging to scale depending on the technology used.

Pull

- **Definition**: The client or consumer requests (pulls) data periodically or on-demand from the server.
- **Implementation**: Regular HTTP GET requests, polling intervals, or scheduled tasks.
- **Pros**:
    - Simpler implementation for many scenarios.
    - Clients control when data is fetched, potentially reducing server load at random times.
- **Cons**:
    - Data can be stale if the polling interval is too long.
    - Increased overhead if polling too frequently for real-time needs.

---

15. ACID vs. BASE

ACID

- **Definition**: A set of properties guaranteeing reliable database transactions: **Atomicity**, **Consistency**, **Isolation**, and **Durability**.
- **Use Cases**: Traditional relational databases (MySQL, PostgreSQL, Oracle) commonly implement ACID for financial transactions, banking, or scenarios requiring strong consistency.
- **Pros**:
    - Ensures data correctness and integrity.
    - Simplifies application logic, as you can rely on strict transaction guarantees.
- **Cons**:
    - Performance overhead for ensuring strict consistency.
    - Less tolerant of network partitions in distributed systems.

BASE

- **Definition**: A looser model often used in distributed systems: **Basically Available**, **Soft state**, **Eventually consistent**.
- **Use Cases**: NoSQL databases like Cassandra, DynamoDB often implement eventual consistency for high availability and partition tolerance.
- **Pros**:
    - High scalability and fault tolerance in distributed environments.
    - Flexible data models for large-scale or globally distributed systems.
- **Cons**:
    - Application logic must handle data that might be temporarily inconsistent.
    - More complex to design for consistency across multiple nodes.

---

16. Polling vs. Streaming

Polling

- **Definition**: A client or consumer periodically requests new data from a source. Similar concept to "pull" communication.
- **Implementation**: Regular intervals to check for updates (e.g., an API endpoint every 5 seconds).
- **Pros**:
    - Straightforward and easy to implement.
    - Robust to short-lived connection issues.
- **Cons**:
    - Inefficient if updates are infrequent (unnecessary queries).
    - Delays in receiving data if the polling interval is long.

Streaming

- **Definition**: Continuous flow of data between producer and consumer. The producer pushes data as it becomes available, and the consumer processes it in near real-time.
- **Implementation**: Technologies like Apache Kafka, Flink, Spark Streaming, or WebSocket-based real-time streams.
- **Pros**:
    - Near real-time updates, high throughput for data pipelines.

    - Scalable solutions for handling large volumes of data in motion.
- **Cons**:
    - More complex ecosystem with message brokers or stream processors.
    - Potentially higher resource usage if not managed properly.

---

17. Symmetric vs. Asymmetric Encryption

Symmetric Encryption

- **Definition**: The same key is used for both encryption and decryption.
- **Implementation**: Algorithms like AES, DES, 3DES. Key must be shared securely between parties.
- **Pros**:
    - Generally faster than asymmetric encryption.
    - Good for bulk data encryption.
- **Cons**:
    - Key exchange is a challenge (securely distributing the key to both parties).
    - If the key is compromised, both encryption and decryption are compromised.

Asymmetric Encryption

- **Definition**: Uses a key pair: a public key for encryption and a private key for decryption.
- **Implementation**: Algorithms like RSA, ECC (Elliptic Curve Cryptography).
- **Pros**:
    - Simplifies key distribution—public key can be shared openly, while the private key is kept secret.
    - Can be used for digital signatures (authentication, integrity).
- **Cons**:
    - Slower than symmetric encryption, especially for large data.
    - Requires more computational resources and longer key lengths for equivalent security.

---

18. Batch Processing vs. Stream Processing

Batch Processing

- **Definition**: Accumulating data over a period and processing it in batches. Typically, the process runs on a schedule (daily, hourly) or once the dataset is complete.
- **Implementation**: Systems like Apache Hadoop (MapReduce), traditional ETL jobs, or data warehousing processes.
- **Pros**:
    - Good for large-scale, comprehensive data transformations and analytics.
    - Easy to maintain historical processing and reprocess data.
- **Cons**:
    - Results are not real-time. There is an inherent delay between data generation and processing.
    - Can be resource-intensive if not carefully optimized.

Stream Processing

- **Definition**: Continuous and incremental processing of data as it flows in. Data is processed event by event or in micro-batches.
- **Implementation**: Apache Kafka + Kafka Streams, Apache Flink, Apache Spark Streaming, or real-time data pipelines.
- **Pros**:
    - Low latency; near real-time insights.
    - Can handle unbounded, continuous data streams.
- **Cons**:
    - More complex architecture, requiring robust, fault-tolerant streaming frameworks.
    - Potentially higher operational overhead to keep the stream pipeline stable.

---

19. API Gateway vs. Load Balancer

API Gateway

- **Definition**: An API gateway acts as an entry point for client requests to backend services, providing request routing, authentication, rate limiting, logging, and monitoring.
- **Implementation**: AWS API Gateway, Kong, NGINX API Gateway, Apigee.

- **Pros**:
    - Manages authentication, security, and API rate limits.
    - Can provide transformation (e.g., converting REST to GraphQL).
    - Helps implement microservices easily by abstracting backend services.
- **Cons**:
    - Adds latency due to additional processing.
    - Can become a single point of failure if not scaled properly.

Load Balancer

- **Definition**: A load balancer distributes incoming network traffic across multiple backend servers to optimize resource use and avoid overloading any single server.
- **Implementation**: AWS ELB, HAProxy, NGINX, F5.
- **Pros**:
    - Improves availability and fault tolerance by redirecting traffic.
    - Can handle SSL termination, session persistence, and failover.
- **Cons**:
    - Does not provide API-specific functionalities (e.g., rate limiting, authentication).
    - Requires a separate API gateway or middleware for API management.

---

20. Data Lake vs. Data Warehouse

Data Lake

- **Definition**: A storage system that holds raw, unstructured, semi-structured, and structured data in its native format until it is needed.
- **Implementation**: AWS S3, Azure Data Lake, Hadoop HDFS.
- **Pros**:
    - Handles any data type (text, images, videos, logs, etc.).
    - Cheap storage for large-scale data analytics.
    - Supports AI/ML workloads with raw data access.
- **Cons**:
    - Data is not optimized for fast querying (requires additional processing).

- Risk of "data swamp" if not properly managed.

Data Warehouse

- **Definition**: A structured database optimized for fast SQL-based queries and analytics.
- **Implementation**: Amazon Redshift, Snowflake, Google BigQuery.
- **Pros**:
  - Optimized for analytical processing (OLAP).
  - Structured, cleaned, and indexed data for fast access.
- **Cons**:
  - Expensive for large-scale raw data storage.
  - Requires ETL (Extract, Transform, Load) to process data before ingestion.

---

21. Strong Consistency vs. Eventual Consistency

Strong Consistency

- **Definition**: Guarantees that once a write operation completes, all future read operations return the latest data.
- **Implementation**: Relational databases (MySQL, PostgreSQL), CP systems in CAP theorem.
- **Pros**:
  - Ensures accurate and up-to-date data.
  - No need for application-side reconciliation.
- **Cons**:
  - High latency in distributed systems.
  - Difficult to scale globally.

Eventual Consistency

- **Definition**: Allows temporary inconsistencies; all nodes eventually reach consistency as updates propagate asynchronously.
- **Implementation**: NoSQL databases (DynamoDB, Cassandra), AP systems in CAP theorem.
- **Pros**:
  - Faster performance with low-latency writes.
  - Better suited for distributed, globally available applications.

- **Cons**:
    - Requires handling temporary inconsistencies at the application level.
    - Reads may return stale data.

---

22. Volatile vs. Non-Volatile Storage

Volatile Storage

- **Definition**: Memory that loses data when power is turned off.
- **Implementation**: RAM (Random Access Memory), CPU caches.
- **Pros**:
    - Extremely fast access times.
    - Ideal for temporary data storage (caching, buffering).
- **Cons**:
    - Loses all data on power loss.
    - Limited in size and costlier than disk storage.

Non-Volatile Storage

- **Definition**: Storage that retains data even when power is lost.
- **Implementation**: SSDs, HDDs, NVMe drives, flash memory.
- **Pros**:
    - Permanent data storage.
    - Suitable for databases, files, and logs.
- **Cons**:
    - Slower than RAM.
    - Limited write endurance in SSDs.

---

23. Immutable Infrastructure vs. Mutable Infrastructure

Immutable Infrastructure

- **Definition**: Infrastructure where components (servers, containers) are never modified after deployment; instead, they are replaced.

- **Implementation**: Kubernetes pods, Terraform, AWS Lambda.
- **Pros**:
    - Eliminates configuration drift and patching issues.
    - Improves deployment reliability with versioned artifacts.
- **Cons**:
    - Requires rebuilding and redeploying for any update.
    - More storage overhead due to maintaining multiple versions.

Mutable Infrastructure

- **Definition**: Infrastructure that allows modifications after deployment (e.g., manually updating running servers).
- **Implementation**: Traditional VMs, on-prem servers.
- **Pros**:
    - Allows incremental updates without redeployment.
    - Easier for troubleshooting and hotfixes.
- **Cons**:
    - Susceptible to configuration drift.
    - Harder to track changes over time.

---

24. Blue-Green Deployment vs. Canary Deployment

Blue-Green Deployment

- **Definition**: Two identical environments (blue and green). One is live while the other is updated, and traffic is switched instantly.
- **Implementation**: AWS Elastic Beanstalk, Kubernetes rolling updates.
- **Pros**:
    - Zero-downtime deployments.
    - Easy rollback if issues arise.
- **Cons**:
    - Requires maintaining two full environments, increasing costs.
    - Switching traffic instantly can expose all users to issues if a bug exists.

Canary Deployment

- **Definition**: Gradually roll out a new version to a small percentage of users before full deployment.
- **Implementation**: Feature flags, Istio, Kubernetes progressive deployments.
- **Pros**:
    - Reduces risk by exposing only a fraction of users to new changes.
    - Allows performance monitoring before full rollout.
- **Cons**:
    - Requires additional monitoring and traffic routing logic.
    - More complex than blue-green deployment.

---

25. JWT vs. OAuth

JWT (JSON Web Token)

- **Definition**: A self-contained token that includes claims (user data, roles) and is signed for authenticity.
- **Implementation**: Used in authentication flows (e.g., API authorization).
- **Pros**:
    - Stateless authentication (no need to query a database).
    - Can be used across different domains and services.
- **Cons**:
    - Larger token size due to embedded claims.
    - Requires careful security measures (e.g., expiration handling).

OAuth

- **Definition**: An authorization protocol that allows third-party access without sharing credentials (e.g., logging in via Google/Facebook).
- **Implementation**: OAuth 2.0, OpenID Connect.
- **Pros**:
    - Secure delegation of access to APIs.
    - Supports multi-service authentication.

- **Cons**:
    - More complex flow than JWT alone.
    - Requires token exchange and management.

---

26. IPv4 vs. IPv6

IPv4

- **Definition**: The older, widely used internet protocol with 32-bit addresses.
- **Pros**:
    - Well-supported across all networks and devices.
    - Simpler to configure manually.
- **Cons**:
    - Limited address space (4.3 billion addresses).
    - Requires NAT (Network Address Translation) to extend usability.

IPv6

- **Definition**: The newer protocol with 128-bit addresses, designed to replace IPv4.
- **Pros**:
    - Vast address space (≈340 undecillion addresses).
    - Improved security and network efficiency.
- **Cons**:
    - Not universally supported in legacy systems.
    - More complex address notation.