150 performance engineering keywords

---

1. **Average Response Time**
    - The arithmetic mean of all individual response times measured for requests processed by a system, typically computed in milliseconds.
    - It serves as a primary metric for gauging overall system performance under typical load conditions, helping to identify trends in latency over time.
2. **90th Percentile**
    - A statistical measure that indicates the response time below which 90% of all requests fall, discounting the worst 10% to highlight general performance.
    - It is used to assess user experience by emphasizing the performance experienced by most end-users while de-emphasizing extreme outliers.
3. **Throughput**
    - The total number of transactions, operations, or requests processed by a system per unit time (often expressed in requests per second or transactions per minute).
    - It reflects the system's capacity and efficiency, directly correlating with how well hardware and software resources are utilized under load.
4. **Hits/sec**
    - The number of discrete HTTP hits or network requests the server handles every second, typically used in web performance metrics.
    - It provides insight into the load and traffic intensity on web servers and is crucial for capacity planning and scaling strategies.
5. **Latency**
    - The delay between a request being initiated and the first byte of the response being received, often measured in milliseconds.
    - It is a critical parameter for evaluating the performance impact of network propagation, processing overhead, or bottlenecks within the application stack.

6. **Concurrent Users**
    - The count of users actively interacting with the system at the same time, making simultaneous requests during a given time window.
    - This metric is vital for capacity planning and load testing, indicating how the system scales under high-traffic, multi-user scenarios.
7. **Simultaneous Users**
    - A refined metric representing the number of users performing actions at exactly the same moment, providing a snapshot of peak concurrency.
    - Essential for understanding instantaneous load peaks that might stress the system's synchronous processing capabilities.
8. **Peak Load**
    - The maximum load (in terms of traffic, transactions, or resource usage) that a system experiences during a specific period.
    - Analyzing peak load helps in capacity planning and stress testing, ensuring that the system can handle extreme conditions without degradation.
9. **Baseline**
    - A reference set of performance metrics collected under normal operating conditions, used for future comparisons.
    - Establishing a baseline is critical in performance monitoring and regression testing, allowing teams to detect deviations introduced by new changes.
10. **Benchmark**
    - Standardized tests or sets of conditions used to measure and compare system performance against industry standards or previous releases.
    - They provide objective data points that help validate system improvements and identify performance regressions over time.
11. **Stateful**
    - Describes systems or applications that retain client or session state across multiple interactions, often using memory, databases, or distributed caches.

- Stateful design increases complexity in load balancing and scalability due to the need for session persistence and state synchronization.

12. **Stateless**
    - Architectures where each request is independent and contains all information needed to complete the processing, with no stored context between requests.
    - This design simplifies horizontal scaling and load distribution, as any server can handle any request without coordination overhead.
13. **Session**
    - A temporary, server-managed interaction period with unique identifiers (often via cookies or tokens), preserving client state across multiple requests.
    - Sessions manage authentication, user preferences, and transactional continuity, often backed by in-memory stores or databases.
14. **Cookie**
    - Small data files stored on the client-side by web browsers to maintain state, track user behavior, or store session identifiers.
    - Technically, cookies are transmitted via HTTP headers and require careful security (e.g., HttpOnly, Secure flags) to prevent exploits.
15. **Deployment**
    - The process of transferring code, configurations, and associated resources from a development environment into production.
    - It involves version control, CI/CD pipelines, and environment orchestration to ensure seamless rollouts with minimal downtime.
16. **Migration**
    - The movement or transformation of data, applications, or services from one platform, architecture, or environment to another.
    - Migration often involves compatibility testing, data integrity verification, and robust rollback strategies to maintain system continuity.

17. **Replication**
    - The process of duplicating data or systems across multiple nodes or geographic regions to enhance availability and fault tolerance.
    - In databases, replication strategies (master-slave, multi-master) require careful conflict resolution and consistency management.
18. **Upgrades**
    - The process of replacing or improving system components—hardware, software libraries, or entire platforms—to benefit from enhanced features and performance.
    - Upgrades require comprehensive testing, compatibility checks, and often, staged rollouts to avoid service disruption.
19. **Swapping**
    - The operation of moving inactive memory pages from RAM to disk-based swap space, allowing systems to free physical memory.
    - While swapping prevents out-of-memory errors, excessive swapping can severely impact performance due to slower disk I/O speeds compared to RAM.
20. **Paging**
    - A memory management technique that divides the system's virtual memory into fixed-size blocks (pages) which are mapped to physical memory frames.
    - Paging enables efficient use of memory but can lead to performance overhead due to page faults if memory is overcommitted.
21. **Context Switches**
    - The process by which an operating system suspends one process or thread and resumes another, thereby sharing CPU time among multiple tasks.
    - High rates of context switching may indicate contention or inefficient scheduling, impacting overall system responsiveness.
22. **Cache**
    - A fast, intermediary storage layer that holds a subset of data (or instructions) to improve read performance and reduce latency.
    - Used at various levels (CPU cache, application-level cache, distributed cache) to minimize expensive data retrieval operations from slower storage.

23. **JVM Stack**
    - A memory area allocated for each thread in a Java Virtual Machine (JVM) to store method call frames, local variables, and partial results.
    - The stack is managed via LIFO (Last In, First Out) order and is critical for execution context isolation during method calls.
24. **JVM Heap**
    - The runtime memory pool from which Java objects are allocated, subject to garbage collection.
    - Heap management—including sizing and garbage collector configuration—is key to maintaining optimal performance and avoiding memory leaks.
25. **Pods**
    - The smallest deployable units in Kubernetes, encapsulating one or more containers with shared storage and network namespaces.
    - Pods serve as the basic building blocks for deploying and scaling containerized applications within a cluster.
26. **Containers**
    - Isolated, lightweight execution environments that package application code and its dependencies together, enabling consistency across deployments.
    - They use kernel-level virtualization (often via Docker) and are orchestrated using container management systems like Kubernetes.
27. **Observability**
    - The measure of how well a system's internal state can be inferred from its external outputs (logs, metrics, events, traces).
    - High observability facilitates debugging, performance tuning, and real-time monitoring in complex distributed systems.
28. **Monitoring**
    - The systematic collection, processing, and analysis of performance and operational data from a system.
    - Employs tools for real-time alerting, dashboards, and anomaly detection to continuously gauge system health.

29. **Profiling**
    - The process of instrumenting an application to collect detailed metrics about resource consumption (CPU, memory, I/O) during runtime.
    - Profiling is used to uncover performance bottlenecks and optimize code execution through detailed metrics analysis.
30. **Scaling**
    - The adjustment of computing resources—either by adding more instances (horizontal scaling) or enhancing individual node capabilities (vertical scaling).
    - Effective scaling strategies ensure that a system can handle growth in demand while maintaining performance SLAs.
31. **Extrapolating**
    - Predicting future system behavior based on current and historical performance data using statistical or machine learning models.
    - This supports proactive capacity planning and helps forecast potential performance issues before they impact production.
32. **Visualization**
    - The graphical representation of performance data using charts, graphs, and dashboards to aid in analysis and decision-making.
    - Visualization tools provide an intuitive way to monitor trends, detect anomalies, and communicate complex performance data clearly.
33. **Distributed Systems**
    - Systems composed of multiple independent computers that work together to achieve a common goal, communicating via a network.
    - They require mechanisms for synchronization, fault tolerance, and data consistency to ensure reliable operation across nodes.
34. **Microservices**
    - An architectural style where a large application is divided into smaller, loosely coupled services, each handling a specific function.
    - Microservices allow for independent deployment, scaling, and technology heterogeneity but also introduce challenges in distributed communication and data consistency.

35. **Monolithic**
    - A unified software application that is built, deployed, and scaled as a single codebase, with tightly integrated components.
    - While easier to develop initially, monoliths can become difficult to maintain and scale as the application grows.
36. **Pacing**
    - The deliberate introduction of delays between consecutive test iterations or operations to simulate realistic user behavior.
    - In performance testing, pacing helps control the request rate, preventing artificial load spikes that don't reflect real-world usage.
37. **Think Time**
    - The simulated delay that represents the time a real user would spend interacting with an application between actions.
    - Incorporating think time in tests ensures more realistic workload simulations and more accurate measurement of system responsiveness.
38. **Load Balancers**
    - Network devices or software applications that distribute incoming requests across multiple servers to achieve optimal resource utilization.
    - Load balancers enhance fault tolerance, improve response times, and support scalability by preventing any single node from becoming a bottleneck.
39. **Fault Tolerance**
    - The ability of a system to continue operating in the event of component failures, using redundancy and error-handling mechanisms.
    - Achieved through techniques such as replication, graceful degradation, and failover to ensure uninterrupted service availability.
40. **High Availability**
    - A system design paradigm focused on ensuring continuous operational performance, typically targeting minimal downtime.
    - Implemented via redundant components, clustering, and geographically distributed architectures to withstand failures.

41. **Failover**
    - The automatic switching to a standby system or redundant component when the primary one fails.
    - Designed to minimize service interruptions, failover mechanisms require state synchronization and constant health monitoring.
42. **Horizontal Scaling**
    - Increasing system capacity by adding more machines or nodes to a system or cluster.
    - This method enables distributed load and fault isolation, facilitating elastic scaling in cloud and data center environments.
43. **Vertical Scaling**
    - Enhancing a system's performance by upgrading existing hardware resources such as CPU, RAM, or storage.
    - While simpler to implement, vertical scaling is limited by hardware constraints and may lead to single points of failure.
44. **Load Testing**
    - A performance testing practice where a system is subjected to a predetermined load to evaluate its behavior under normal and peak conditions.
    - It identifies resource utilization thresholds and performance bottlenecks, enabling informed capacity planning.
45. **Stress Testing**
    - A type of performance evaluation where the system is pushed beyond its normal operational capacity to determine its breaking point.
    - It helps identify failure modes, recovery capabilities, and the resilience of the system under extreme conditions.
46. **Soak Testing**
    - Also known as endurance testing, it subjects a system to a high load over an extended period to detect memory leaks, resource exhaustion, or degradation.
    - This helps ensure long-term stability and performance consistency over prolonged operational periods.
47. **Smoke Testing**
    - A preliminary testing process that verifies whether the basic functionalities of an application are working after a new build or deployment.

- Often automated, it serves as a quick check before more detailed performance or regression tests are executed.

48. **Endurance Testing**
    - Evaluates how a system performs under a continuous load for a prolonged period, focusing on potential degradation.
    - It uncovers issues like resource leaks, slow memory buildup, or gradual decreases in throughput that might not be evident in shorter tests.
49. **Resilience Testing**
    - The practice of deliberately testing system recovery mechanisms by introducing failures to assess its ability to recover and maintain critical operations.
    - This process validates redundancy, automated recovery strategies, and the robustness of error-handling mechanisms.
50. **Regression Testing**
    - Re-running previously conducted tests after changes or updates to ensure that new code has not adversely affected existing functionality.
    - It is crucial for maintaining performance standards, especially after refactoring or integration of new features.
51. **API Gateway**
    - A centralized management layer that handles all client interactions with backend microservices, offering routing, authentication, and aggregation.
    - It abstracts complexity from clients while managing cross-cutting concerns such as rate limiting and logging.
52. **Middleware**
    - Software that functions as an intermediary between different systems or applications, providing services such as messaging, authentication, and data transformation.
    - It simplifies integration and communication between disparate systems and abstracts network complexities.
53. **Orchestration**
    - The automated coordination and management of complex service interactions and deployments, often using container orchestrators like Kubernetes.

- It ensures that distributed components are correctly configured, scaled, and maintained, reducing manual intervention and error rates.

54. **Automation**
    - The use of scripts, tools, and pipelines to perform tasks such as deployments, monitoring, and testing without manual input.
    - Automation enhances consistency, speeds up processes, and reduces the likelihood of human error in repetitive tasks.
55. **CI/CD**
    - Continuous Integration and Continuous Deployment pipelines that automate the testing and deployment of code changes in a streamlined manner.
    - They facilitate rapid iteration, quick rollback, and consistent quality assurance through automated builds, tests, and releases.
56. **Canary Deployment**
    - A strategy wherein a new release is initially rolled out to a small subset of users to monitor its performance and stability before full-scale deployment.
    - It minimizes risk and allows early detection of issues by comparing the new release's metrics against the baseline.
57. **Blue-Green Deployment**
    - A release management strategy that maintains two identical environments (blue and green), enabling a smooth transition by switching traffic from the old to the new environment seamlessly.
    - This approach minimizes downtime and simplifies rollback if the new environment fails to meet performance criteria.
58. **Tracing**
    - The detailed recording of the path and execution flow of a request as it traverses microservices and system components.
    - Distributed tracing allows engineers to pinpoint latency hotspots and pinpoint failures across complex call graphs.
59. **Metrics**
    - Quantitative measurements collected from various system components (e.g., CPU load, memory usage, I/O throughput) that indicate system performance.

- These metrics are often aggregated, processed, and visualized to monitor system health, troubleshoot issues, and support capacity planning.

60. **Logs**
    - Time-stamped records of system events, transactions, or errors generated by software applications and infrastructure components.
    - They serve as a primary source for debugging, forensic analysis, and tracking operational behavior in production environments.
61. **Distributed Tracing**
    - An advanced form of tracing that correlates logs and traces across multiple services in a distributed architecture.
    - It enables end-to-end monitoring of request flows, facilitating rapid isolation of performance issues or failures across interconnected components.
62. **Chaos Engineering**
    - The disciplined practice of deliberately injecting failures (e.g., network blackouts, service crashes) into a system to evaluate its resilience and robustness.
    - By simulating real-world failure scenarios, chaos engineering validates recovery strategies and reinforces system stability under unexpected conditions.
63. **Elasticity**
    - The property of a system that allows it to automatically adjust its resource allocation (scaling up or down) in response to real-time demand.
    - Elasticity is central to cloud-native architectures, ensuring cost-effectiveness while maintaining performance consistency during load fluctuations.
64. **Fault Injection**
    - A testing technique that purposefully introduces errors or failures into a system to verify its error-handling and recovery capabilities.
    - It is used to simulate uncommon failure modes and to validate that the system can gracefully handle and recover from unexpected disruptions.

65. **Immutable Infrastructure**
    - An approach in which servers or infrastructure components are never modified after deployment; any change is achieved by replacing the entire component.
    - This strategy minimizes configuration drift, simplifies rollback procedures, and ensures consistency across deployments.
66. **Service Discovery**
    - The automated process by which applications locate and communicate with service instances in dynamic, distributed environments.
    - It eliminates hard-coded endpoints and adapts to runtime changes through mechanisms like DNS-based resolution or dedicated service registries.
67. **Sharding**
    - The practice of horizontally partitioning data across multiple databases or nodes to distribute load and improve performance.
    - Each shard holds a subset of the overall data, which can lead to reduced latency and enhanced throughput for large-scale systems.
68. **Rate Limiting**
    - The enforcement of a maximum threshold on the number of operations (e.g., API calls) allowed over a fixed interval.
    - Rate limiting prevents overuse, protects backend resources, and maintains service quality during high-traffic scenarios.
69. **Circuit Breaker**
    - A fault-tolerance pattern that halts operations for a defined period when a particular service repeatedly fails, preventing system-wide cascading failures.
    - It monitors error rates and automatically “trips” to stop further calls, then gradually allows trial requests to check service recovery.
70. **Containerization**
    - The process of bundling an application and its dependencies into a standardized unit called a container, ensuring consistency across environments.

- Containers share the host OS kernel while remaining isolated from one another, optimizing resource usage and simplifying deployment workflows.

71. **Service Mesh**
    - An infrastructure layer that manages service-to-service communication in microservices architectures, providing features like traffic management, security, and observability.
    - It abstracts network complexities away from application code, enabling consistent policy enforcement and enhanced inter-service communication.
72. **Ingress Controller**
    - A Kubernetes component responsible for managing external access to services within a cluster, often handling HTTP/S traffic routing.
    - It performs tasks like SSL termination, load balancing, and URL-based routing based on Ingress configuration rules.
73. **Observability Pillars**
    - The three fundamental data sources—logs, metrics, and traces—that collectively provide full insight into system behavior.
    - Each pillar contributes uniquely to diagnosing issues, where logs capture detailed events, metrics offer quantitative snapshots, and traces show request flows.
74. **Garbage Collection (GC)**
    - An automated memory management process in high-level languages like Java, which reclaims memory from objects that are no longer referenced.
    - Proper GC tuning and profiling are critical to reducing pause times and ensuring minimal impact on application throughput.
75. **Thread Dump**
    - A snapshot of all threads and their execution state within a running application, providing detailed call stacks and synchronization status.
    - Analyzing thread dumps is invaluable for diagnosing deadlocks, performance bottlenecks, and contention issues in multi-threaded environments.

76. **Reverse Proxy**
    - A server that receives client requests and forwards them to one or more backend servers, often abstracting and load balancing services.
    - It can perform caching, SSL termination, and security filtering, thus improving overall system performance and security.
77. **Forward Proxy**
    - An intermediary server that processes client requests on behalf of external servers, often used for internet access control and content caching.
    - It provides anonymity, security filtering, and bandwidth optimization by caching frequently accessed resources.
78. **Virtualization**
    - The technique of creating virtual versions of hardware platforms, storage devices, or network resources to maximize utilization and flexibility.
    - Virtualization abstracts physical resources, enabling multiple virtual machines to run concurrently on a single physical host with isolated environments.
79. **Hypervisor**
    - Software that creates and manages virtual machines (VMs) by abstracting and partitioning physical hardware resources.
    - Examples include VMware ESXi, Microsoft Hyper-V, and KVM, which enable multi-tenancy and improved resource utilization within data centers.
80. **Infrastructure as Code (IaC)**
    - The management of infrastructure (networks, servers, storage) through code and configuration files rather than manual processes.
    - Tools like Terraform and AWS CloudFormation enforce version control, repeatability, and automated deployment of infrastructure components.
81. **Immutable Artifact**
    - A build output (such as a container image or compiled binary) that, once generated, is not altered.
    - This immutability ensures that deployments are consistent, traceable, and can be reliably rolled back if necessary.

82. **Rollback**
    - The controlled process of reverting a system or application to a previous stable version after identifying issues in the new release.
    - Rollbacks are a critical safety mechanism in deployments and rely on versioning and immutable artifacts for successful restoration.
83. **Hotfix**
    - A rapid and often urgent update deployed to address critical issues in a production environment without waiting for the regular release cycle.
    - Hotfixes aim to resolve security vulnerabilities or performance regressions swiftly, often with minimal testing in a controlled release pipeline.
84. **Zero Downtime Deployment**
    - Deployment techniques—such as rolling updates, blue-green deployment, or canary releases—that ensure uninterrupted service availability during updates.
    - This practice requires strategies to manage state, traffic redirection, and real-time health monitoring to avoid any service disruptions.
85. **Data Partitioning**
    - The segmentation of large datasets into discrete, manageable chunks (partitions or shards) to improve query efficiency and system scalability.
    - Data partitioning helps distribute I/O operations and processing loads, often using range, hash, or list partitioning methods in databases.
86. **Eventual Consistency**
    - A consistency model in distributed systems where updates propagate asynchronously, ensuring that all nodes will become consistent over time.
    - This model prioritizes availability and partition tolerance, accepting temporary discrepancies in favor of system scalability.
87. **Strong Consistency**
    - A data consistency model ensuring that once a data update is committed, all subsequent read operations reflect that change immediately across all nodes.

- Strong consistency is often enforced in traditional RDBMS or through distributed consensus protocols (e.g., Paxos, Raft), potentially at the expense of latency.

88. **Message Queue**
    - A communication middleware that enables asynchronous message passing between producers and consumers, decoupling processing through queuing.
    - Systems like RabbitMQ or Apache Kafka provide durable and reliable queuing mechanisms to handle high volumes of messages while ensuring delivery guarantees.
89. **Publish-Subscribe (Pub/Sub)**
    - An asynchronous messaging pattern where publishers broadcast messages to topics, and subscribers receive messages based on their interests.
    - This decouples the sender and receiver, allowing for highly scalable, real-time distribution of information across distributed systems.
90. **Leader Election**
    - A consensus process in distributed systems where nodes determine a single coordinator (leader) to manage tasks or resources.
    - Leader election algorithms (e.g., Bully Algorithm, Raft) are essential for coordination, preventing conflicts, and ensuring reliable system operations.
91. **Idempotency**
    - A property of operations where executing the same request multiple times produces the same result without side effects beyond the initial application.
    - Idempotency is crucial for reliable API design and handling retries in distributed systems to avoid unintended duplicate transactions.
92. **Tokenization**
    - The process of replacing sensitive data with non-sensitive placeholders (tokens) that can be mapped back only with authorized access.

- This technique is employed to secure data in transit or at rest, reducing exposure of personal or confidential information in applications.

93. **Redundancy**
    - The duplication of critical components or functions to increase system reliability, ensuring that failure of one element does not lead to system collapse.
    - Redundancy is implemented via clustering, replication, or failover mechanisms, enhancing fault tolerance and continuous availability.
94. **Elastic Load Balancer (ELB)**
    - A cloud-based load balancing solution that dynamically distributes incoming application traffic across multiple backend instances.
    - ELBs, such as those offered by AWS, integrate health checks and auto-scaling to adapt to varying load conditions in real time.
95. **Health Check**
    - Automated probes or tests that determine whether a system component, such as a server or microservice, is functioning correctly.
    - Health checks are integral to load balancers and orchestration systems, triggering automatic recovery or rerouting when failures are detected.
96. **Middleware Caching**
    - Caching implemented at the middleware layer to store and rapidly serve frequently requested data, reducing downstream processing.
    - It minimizes latency by offloading repetitive data retrieval operations and thereby alleviates the load on databases or backend services.
97. **Rate Throttling**
    - Controlling the flow of requests to a service by enforcing a maximum number of allowed operations per time unit.
    - This protects the system from overload during traffic bursts, ensuring equitable resource usage and system stability.

98. **Namespace**
    - A logical partition within systems like Kubernetes that isolates resources (pods, services, etc.) to enforce organizational boundaries and manage access control.
    - Namespaces facilitate multi-tenancy and prevent resource conflicts by segregating workloads within the same cluster.
99. **Autoscaling**
    - The dynamic adjustment of computing resources based on real-time monitoring metrics, such as CPU or memory utilization.
    - Autoscaling helps maintain performance targets while optimizing cost efficiency by ensuring that resources match the current demand.

100. **Cold Start**

- The initialization latency experienced when a service, container, or serverless function is invoked for the first time, requiring loading of dependencies and configurations.
- Critical in ephemeral environments where the delay can significantly impact user experience, especially under sporadic request patterns.

101. **Warm Start**

- A scenario where pre-initialized resources or containers remain available to handle incoming requests quickly, reducing startup latency.
- Achieved by keeping instances "warm" (persistently allocated) to avoid the overhead of reinitialization, ensuring faster response times.

102. **API Rate Limiting**

- A mechanism that restricts the number of API calls a client can make in a specific timeframe to protect backend services.
- Helps prevent abuse, mitigates the risk of server overload, and maintains overall system performance by enforcing strict quotas.

103. **Horizontal Pod Autoscaler (HPA)**

- A Kubernetes feature that automatically adjusts the number of pod replicas in a deployment based on real-time resource metrics (e.g., CPU, memory).
- Ensures that applications dynamically scale to meet load variations, maintaining performance and efficiency.

104. **StatefulSet**

- A Kubernetes controller designed for managing stateful applications that require stable network identities and persistent storage.
- Guarantees ordered deployment, scaling, and updates, which is essential for maintaining data consistency in systems like databases.

105. **DaemonSet**

- A Kubernetes configuration ensuring that a copy of a specific pod runs on all (or selected) nodes throughout the cluster.
- Commonly used for deploying system-level services (e.g., logging, monitoring, security agents) consistently across nodes.

106. **CronJob**

- A scheduled task in Kubernetes configured to run at specific times or intervals using cron syntax.
- Automates repetitive operations such as backups, report generation, or clean-up tasks, ensuring regular maintenance without manual intervention.

107. **Disaster Recovery (DR)**

- A comprehensive set of policies and procedures to restore critical systems and data after catastrophic failures or significant disruptions.
- Involves strategies like data backups, offsite replication, and predefined failover processes to minimize downtime and data loss.

108. **Rolling Update**

- A deployment strategy that incrementally replaces old application instances with new ones, ensuring that some instances remain operational throughout the process.
- Minimizes service disruption and allows for continuous monitoring of performance, with the ability to rollback if issues arise.

109. **Distributed Lock**

- A synchronization mechanism that ensures only one process or node can access a shared resource or execute a critical section at any given time in a distributed system.
- Typically implemented using tools like ZooKeeper or Redis, distributed locks prevent race conditions and maintain data consistency.

110. **Hot Path**

- The section of code or execution path that is most frequently invoked and critically influences overall system performance.
- Optimizing the hot path (through algorithm improvements or hardware acceleration) can significantly reduce latency and boost throughput.

111. **Warm Path**

- A data processing route optimized for near-real-time analytics, where timely processing is important but not as critical as strict low-latency requirements.
- Typically leverages in-memory processing and batch techniques to balance speed with resource efficiency.

112. **Cold Path**

- The part of a data processing pipeline designed for non-real-time, batch processing where latency is less critical than cost efficiency and throughput.
- Often used for long-term analytics, data warehousing, or offline reporting using frameworks like Hadoop.

113. **Service Level Agreement (SLA)**

- A formal contract that specifies performance, uptime, and responsiveness targets agreed upon between a service provider and its clients.
- Defines measurable objectives and incorporates remediation or penalty clauses for non-compliance, ensuring accountability and reliability.

114. **Service Level Indicator (SLI)**

- A specific, quantifiable metric (such as response time, error rate, or availability) used to evaluate a service's performance against its SLA.
- Provides the necessary data to assess whether the service is meeting its agreed-upon performance standards.

115. **Service Level Objective (SLO)**

- A clearly defined target value or range for an SLI, setting the acceptable performance threshold for a service over a period.
- Used as a performance benchmark to trigger alerts and corrective actions when the service deviates from expected behavior.

116. **Synthetic Monitoring**

- The use of automated, scripted transactions to simulate user interactions and continuously test service performance from various locations.
- Enables proactive identification of issues by emulating realistic usage scenarios under controlled conditions.

117. **Real User Monitoring (RUM)**

- The collection and analysis of performance data directly from real users interacting with the application in production.
- Provides a detailed understanding of end-user experience, capturing variations across different geographies and device types.

118. **Application Performance Monitoring (APM)**

- A suite of tools and practices that continuously track, analyze, and optimize the performance of an application's code, infrastructure, and transactions.
- Combines metrics, traces, and logs to provide deep diagnostics, helping engineers quickly identify and resolve performance bottlenecks.

119. **Memory Leak**

- A situation in which an application fails to release memory that is no longer needed, resulting in gradual depletion of available memory over time.
- Can lead to severe performance degradation or system crashes, necessitating careful profiling and code analysis to identify and correct the leak.

120. Resource Utilization

- The measurement of how efficiently system resources (CPU, memory, disk I/O, network bandwidth) are being consumed under operational load.
- Monitoring resource utilization is key to identifying inefficiencies and making informed decisions on scaling or optimization.

121. Thread Pool

- A collection of pre-instantiated threads that are used to execute multiple tasks concurrently, reducing the overhead of thread creation.
- Improves performance in multi-threaded applications by managing resource allocation and ensuring timely processing of queued tasks.

122. Bottleneck Analysis

- The systematic examination of system components to identify limiting factors that restrict overall performance.
- Helps pinpoint hardware or software constraints so targeted optimizations can be implemented to improve throughput and efficiency.

123. Lock Contention

- A scenario in which multiple threads or processes simultaneously compete to acquire the same synchronization lock, resulting in delays.
- High lock contention often signals the need for improved concurrency control or refactoring to reduce critical section scope.

124. Deadlock

- A state where two or more processes are each waiting indefinitely for the other to release a resource, resulting in a standstill.
- Preventing deadlocks involves designing systems with careful resource management, employing timeouts, and applying deadlock detection algorithms.

125. **CPU Utilization**

- The percentage of the CPU's processing capacity that is actively used by running tasks and processes at any given time.
- High CPU utilization may indicate intensive computation or inefficient processes, necessitating performance tuning or hardware upgrades.

126. **Memory Utilization**

- The ratio of memory currently in use to the total available memory, including allocations in the JVM heap, caches, and buffers.
- Monitoring memory utilization is crucial for detecting leaks, planning for capacity increases, and optimizing overall system performance.

127. **Garbage Collection Tuning**

- The process of configuring and optimizing garbage collection parameters (heap size, GC algorithms, pause thresholds) in managed runtimes such as the JVM.
- Effective tuning minimizes GC pause times and enhances throughput by ensuring timely reclamation of unused memory.

128. **Exception Handling**

- The mechanism for capturing and managing errors during runtime to prevent unexpected application termination and allow graceful recovery.
- Robust exception handling contributes to system stability and provides detailed diagnostic information for debugging performance issues.

129. **Instrumentation**

- The integration of monitoring code or agents (such as APM tools) into an application to capture detailed performance metrics and operational data.
- Instrumentation enables fine-grained analysis of latency, throughput, and resource consumption, facilitating proactive optimization.

130. **Profiling Overhead**

- The additional CPU, memory, or I/O cost incurred by the tools and processes used for monitoring and profiling application performance.
- It is essential to minimize profiling overhead to ensure that measurement tools do not significantly impact the performance being evaluated.

131. **Performance Budget**

- A pre-established limit on critical performance metrics (e.g., load time, resource usage) that guides development and optimization efforts.
- Serves as a quantitative constraint ensuring that new features or code changes do not exceed established performance thresholds.

132. **Latency Budget**

- The total permissible delay allocated across the various components or stages involved in processing a request.
- Distributing a latency budget across services helps identify and optimize individual components contributing to overall response time.

133. **Cache Invalidation**

- The process of removing or refreshing stale data from a cache to ensure that the information served remains consistent with the underlying data source.
- Effective cache invalidation strategies (time-based, event-driven) are critical for maintaining data integrity while benefiting from the speed of caching.

134. **Cache Hit Ratio**

- The proportion of cache accesses that result in a successful data retrieval, compared to total cache lookups.
- A high cache hit ratio indicates efficient caching mechanisms, which reduce the need for slower, repeated backend data fetches.

135. **Thundering Herd**

- A phenomenon where a large number of processes or requests simultaneously attempt to access a shared resource, overwhelming the system.
- Mitigation strategies include randomized backoff, request queuing, and rate limiting to distribute the load more evenly.

136. **Hot Code Path**

- The segment of an application's code that is executed most frequently and thus has a significant impact on performance.
- Optimizations in the hot code path (via refactoring or algorithmic improvements) can dramatically reduce response times and resource usage.

137. **Asynchronous Processing**

- A programming paradigm that allows tasks to be executed independently of the main execution thread, avoiding blocking operations.
- Enhances application responsiveness by leveraging callbacks, futures, or event loops to process tasks concurrently.

138. **I/O Wait**

- The duration during which a process is idle while waiting for input/output operations (such as disk reads/writes or network transfers) to complete.
- High I/O wait times are indicative of bottlenecks in storage or network subsystems, often prompting hardware upgrades or optimization of I/O patterns.

139. **Back Pressure**

- A control mechanism that signals data producers to slow down when the downstream consumers are overwhelmed, thereby preventing overload.
- Vital in streaming and high-throughput systems, back pressure ensures smooth operation and prevents resource exhaustion.

140. **Service Dependency**

- The inter-relationship between various services where the performance of one system component directly impacts others.
- Mapping and managing these dependencies is crucial for diagnosing performance issues and ensuring that cascading failures are prevented.

141. **Transaction Tracing**

- The end-to-end tracking of a business transaction across various services and system components, capturing detailed timing and context.
- Provides granular visibility into the complete lifecycle of a transaction, enabling pinpoint identification of performance bottlenecks.

142. **Tracing Span**

- A discrete unit or segment within a distributed trace that encapsulates the start, end, and metadata of a single operation.
- Aggregated spans form a complete trace, allowing in-depth analysis of latency and operational context across distributed systems.

143. **Burst Traffic Management**

- Techniques to handle sudden, short-term spikes in traffic without compromising system stability or performance.
- Involves autoscaling, buffering, and rate limiting to absorb transient surges while maintaining quality of service.

144. **Concurrency Control**

- Mechanisms such as locks, semaphores, or optimistic concurrency used to manage simultaneous access to shared resources in multi-threaded or distributed environments.
- Ensures data integrity and orderly execution when many processes or threads operate concurrently under high load.

145. **SLO Violation**

- An occurrence wherein a service fails to meet its defined Service Level Objective, signaling degraded performance or reliability.
- Triggers alerts and detailed investigations to determine root causes and implement corrective measures promptly.

146. **Distributed Caching**

- The spread of cache storage across multiple nodes or servers to provide faster data retrieval and improved redundancy.
- Systems like Redis or Memcached are used in distributed caching to reduce load on primary data stores and lower response times.

147. **Connection Pooling**

- A resource management technique that maintains a pool of active connections (to databases or network services) for reuse in multiple requests.
- Reduces the overhead of establishing new connections, thereby significantly enhancing overall request processing speed.

148. **JVM Tuning**

- The process of configuring Java Virtual Machine parameters (e.g., heap size, garbage collector settings, thread configurations) to optimize application performance.
- Critical for balancing throughput, memory efficiency, and latency in Java applications under varying workloads.

149. Code Optimization

- The systematic refinement of software code to improve execution speed, reduce resource consumption, and lower latency.
- Involves profiling, algorithmic improvements, and low-level enhancements (e.g., loop unrolling, inlining) to achieve measurable performance gains.

150. Performance Regression

- A decline in system performance or efficiency introduced by new code changes, upgrades, or configuration modifications compared to previous benchmarks.
- Continuous performance regression testing is essential to quickly detect and remediate any degradations before they impact end users.