These slides provide a comprehensive guide to performance evaluation, particularly focusing on cloud computing systems and services. The presentation covers general concepts, specific metrics for cloud environments, testing techniques, a structured procedure for evaluation, and practical advice for setting up experiments and interpreting results, especially in the context of auto-scaling.

1. General Concepts of Performance Evaluation (Slides 3-4) Computer systems performance evaluation assesses how efficiently a system uses its resources (CPU, memory, storage) to achieve desired outputs[cite: 3]. It helps identify improvement areas, optimize resource use, and ensure systems meet performance requirements[cite: 3]. Four main techniques are analytical modeling, Petri-net modeling, simulation modeling, and empirical or testbed analysis[cite: 3].

For cloud systems and services, performance evaluation assesses the effective functioning of cloud resources, helping to identify bottlenecks and enhance overall system performance[cite: 4]. This typically involves empirical evaluation through monitoring metrics, conducting performance tests, and analyzing results[cite: 4]. Key areas of focus include availability, response times, scalability, and resource utilization[cite: 4]. The benefits of such evaluation are optimizing resource utilization, ensuring business continuity, improving scalability and flexibility, and meeting Service Level Agreements (SLAs)[cite: 4].

2. Key Metrics and Testing Techniques (Slides 5-6)

- **Key Metrics to Monitor (Slide 5):**
    - **Availability:** Percentage of time a service is operational and accessible[cite: 5].
    - **Response Time:** Time taken for a system to respond to a request; latency is a key component[cite: 6].
    - **Throughput:** Amount of data or work processed within a given timeframe[cite: 7].
    - **Resource Utilization:** Measures like CPU utilization, memory usage, and disk I/O, indicating resource efficiency[cite: 8].
    - **Error Rate:** Frequency of errors or failures[cite: 9].
    - **Scalability:** System's ability to handle increasing workloads without performance degradation[cite: 10].
- **Performance Testing Techniques (Slide 6):**
    - **Load Testing:** Simulates typical workloads to assess performance under normal conditions[cite: 11].
    - **Stress Testing:** Exposes the system to extreme loads to identify weaknesses and capacity limits[cite: 11].
    - **Endurance Testing:** Tests ability to maintain performance over extended periods[cite: 11].
    - **Spike Testing:** Simulates sudden bursts of activity to evaluate handling of unpredictable load spikes[cite: 11].

3. Procedure for Cloud Service Performance Evaluation (Slides 7-13) A general procedure for measurement-based cloud service performance evaluation involves seven steps[cite: 13]:

1. Specify the purpose and scope of the evaluation (e.g., identify bottlenecks, verify SLAs, optimize resources, ensure business continuity, improve scalability)[cite: 15, 16, 17]. The perspective can be user-oriented, service provider-oriented, or both[cite: 18]. Project-specific examples are provided for different evaluation goals[cite: 20, 21, 22, 23, 24].

2. Identify the features/aspects of the Cloud services to be evaluated. Complex services have many components (application logic, web server, data store, authentication, CDN); a study might focus on all, few, or one[cite: 25]. This helps determine the workload and metrics[cite: 25]. Examples include focusing on Lambda functions for image processing or web application logic for a multi-tier app, assuming other layers are not bottlenecks[cite: 27].
3. Determine the performance metrics to be analyzed (Slide 14). These are categorized as:
    - **User-oriented (client-side):** Availability, Response Time, Error Rate[cite: 28].
    - **System-oriented (server-side):** Throughput, Resource Utilization, Scalability, Availability[cite: 28]. The specific units for metrics like throughput vary by component (e.g., transactions/sec for DB, requests/sec for applications)[cite: 29].
4. Select an appropriate tool for evaluating performance (Slides 21-23). The choice depends on the features to be evaluated (API, web server, DB, disk) and the workload (type, intensity, duration of requests)[cite: 36]. Many tools are available for web application testing (e.g., Jmeter, Locust, ApacheBench), allowing definition of test duration, request types, and collection of metrics like throughput and response time[cite: 37]. Considerations include the need to target APIs directly (for Lambdas) versus full-stack web requests, and the variety of requests needed to simulate realistic usage[cite: 38, 39].
5. Design performance evaluation experiments (Slides 24-28). This is driven by the purpose, scope, features, and metrics. Experiments are characterized by:
    - **Workload Generated:** Type of requests (CPU, Disk, Memory, or Network intensive) and intensity (requests per second)[cite: 40]. For an image processing app example, requests should vary by image size or complexity[cite: 42]. Intensity should be chosen considering practical limits like AWS free tier or learning lab constraints to avoid account banning[cite: 43, 44].
    - **Duration:** Long enough to observe the system and collect sufficient data points (e.g., 10-30 minutes if data points are generated every minute)[cite: 41].
    - **Number of Runs:** Multiple runs are needed for reliable conclusions (theory: at least 20; practice: 3-5 given budget/time constraints)[cite: 41].
    - **Workload Shape (Slide 26):** Experiments often use a workload pattern with phases like Warm-up (WU), Ramp-up (RU), Steady (S), and Ramp-down (RD) to test scaling[cite: 45]. Some tools automate load increase, others require multiple runs with increasing load values[cite: 46]. It's advised not to let the system cool down between consecutive runs[cite: 46]. Workload specification varies by tool; simple tools like ab might always request the same page, while tools like Jmeter offer more control but can be harder to configure[cite: 47, 48, 49].
    - **Reproducing User Behavior (Slide 28):** Can be done by selecting representative requests with assigned probabilities or using tools like Selenium WebDriver[cite: 50].
6. Set up an experimental environment (Slides 29-30). This involves ensuring the application is bug-free, setting up a workload generator (preferably within the cloud provider's perimeter to avoid external network variability and potential bans, though this impacts user-oriented measurements)[cite: 51, 52], configuring monitoring tools like AWS CloudWatch, and preparing workload generator configuration files, avoiding overly intensive workloads[cite: 53].
7. Run performance evaluation experiments and analyze data (Slide 31). Collect data points for selected metrics, download them, average values across multiple runs, and plot the averages[cite: 54, 55].

4. Specific Considerations for Auto-scaling Evaluation (Slides 32-36)

- **Setting Threshold Values for Autoscaling (Slides 32-33):** This is often heuristic. A common rule of thumb is 66% of maximum capacity[cite: 57]. For CPU utilization, 66%-70% is considered a safe

level[cite: 57]. Network traffic thresholds can be set similarly based on instance type bandwidth limits[cite: 57]. To demonstrate scaling, thresholds should be set relative to the maximum load generated by the specific test (e.g., if max CPU load is 50%, a 35% threshold might be appropriate) [cite: 58].

- **Testing Availability (Slide 34):** Involves generating random failures by killing services or stopping VMs/containers[cite: 59].
- **Common Mistakes (Slides 35-36):**
    - **Using CPU stress tools incorrectly (Slide 35):** If a CPU stress tool on one VM triggers scaling, but the stress ends before the new VM contributes or if the tool doesn't run on new instances, the average CPU load will be misleading. The stress tool must run on each instance throughout its participation in the experiment[cite: 60]. This setup may only effectively reproduce warm-up and ramp-up phases[cite: 61].
    - **Inadequate request mix (Slide 36):** For applications with diverse functionalities (e.g., image/text posts), testing only with simple GET requests is insufficient; a mix of relevant request types (uploads of varying sizes) is needed[cite: 62]. For applications with a light front-end but heavy database interaction, evaluating only the front-end's scalability is meaningless; a mix of database R/W requests must be generated[cite: 62].