Module 18: Week 20: Required Assignment
PPerformance Optimization and Scalability

1. Explain the process of model quantisation and discuss its benefits and potential drawbacks in AI model optimisation. (5 Marks)

2. Discuss the role of caching in scaling applications. Compare and contrast in-memory caching and distributed caching, providing examples of when each would be most appropriate. (5 Marks)

3. Explain the concept of horizontal scaling and vertical scaling. Provide examples of when you would choose one over the other in a real-world scenario. (5 Marks)

4. Describe the process of designing an effective load test for a web application. What key factors should be considered, and how would you determine the success criteria for the test? (5 Marks)

---

1. Explain the process of model quantisation and discuss its benefits and potential drawbacks in AI model optimisation. (5 Marks)

Model Quantisation:

Model quantisation is a model compression technique that reduces the numerical precision of a model's parameters (weights) and activations. This involves converting the standard 32-bit floating-point (FP32) representations to lower-precision formats like 16-bit floats (FP16), 8-bit integers (INT8), or even 4-bit integers. This process can be performed:

- **Post-Training Quantisation (PTQ):** The model is trained normally in FP32 and then converted to a lower precision. This is faster but can lead to a greater accuracy drop.
- **Quantisation-Aware Training (QAT):** During training, "fake" quantisation nodes simulate the effect of lower precision. This allows the model to adapt its weights to minimise the accuracy loss, typically yielding better results than PTQ.

Benefits:

- **Reduced Model Size & Memory Footprint:** An INT8 model is roughly **75% smaller** than its FP32 equivalent (8 bits vs. 32 bits per parameter). This is critical for deployment on devices with limited storage, such as smartphones and embedded IoT systems.
- **Faster Inference & Lower Latency:** Integer operations (INT8) are fundamentally faster and require less complex hardware than floating-point operations (FP32). This allows for more inferences per second and a smoother user experience, especially on edge devices with specialized AI accelerators (e.g.Google's Edge TPU) that are optimized for low-precision math.
- **Reduced Power Consumption:** Smaller models and simpler arithmetic operations consume significantly less energy, which is vital for battery-powered devices.

Drawbacks:

- **Potential Accuracy Loss:** The primary trade-off is a potential decrease in model accuracy. Reducing precision truncates information and can increase numerical error, which may degrade performance, particularly for tasks requiring high precision like medical image segmentation.
- **Hardware and Software Dependency:** Not all hardware can execute INT8 operations efficiently. Deploying a quantised model requires a supporting software stack (e.g., TensorFlow Lite,) to handle the conversion and execution, adding complexity to the deployment pipeline.
- **Engineering Overhead:** Implementing QAT requires modifying the training loop and can be technically challenging. Determining the optimal quantisation scheme (e.g., which layers to quantise) often requires experimentation and profiling.

---

2. Discuss the role of caching in scaling applications. Compare and contrast in-memory caching and distributed caching, providing examples of when each would be most appropriate. (5 Marks)

Role of Caching:
Caching is a fundamental technique for building scalable, high-performance applications. Its primary role is to **reduce latency** and **decrease the load on underlying data sources** (like databases or external APIs) by storing copies of frequently accessed data in fast, readily-accessible storage (typically RAM). This improves:

1. **User Experience:** Faster response times.
2. **Scalability:** The application can handle more requests per second, as many are served from the cache instead of hitting slower backend systems.
3. **Cost Efficiency:** Reduced database load can allow for smaller, less expensive database instances.

The most significant challenge in caching is **cache invalidation**—ensuring the cached data is updated or purged when the underlying data changes, to prevent users from seeing stale information.

Comparison:

Feature	In-Memory Caching	Distributed Caching
Architecture	Data stored in the RAM of a single application server.	Data distributed across multiple nodes in a dedicated cache cluster.
Best For	Small to medium-scale applications, or caching data local to a single user session (e.g., a user's shopping cart).	Large-scale, cloud-native applications with multiple application instances.
Pros	Extreme simplicity and blazing-fast access speed (no network calls).	Scalability, fault tolerance, and a consistent view of cached data for all application servers.

Feature	In-Memory Caching	Distributed Caching
Cons	**Not shared** across servers, leading to **stale or duplicated data**. A server restart clears the cache.	Introduces network latency. More complex to set up and manage (e.g., cluster configuration).
Examples	**ASP.NET MemoryCache,** Ideal for a single-instance blog server caching its homepage.	**Redis Cluster, Memcached with a client-side distribution layer, AWS ElastiCache.** Essential for a microservices architecture where a "product service" needs to share product details with all instances of the "web API service."

---

3. Explain the concept of horizontal scaling and vertical scaling. Provide examples of when you would choose one over the other in a real-world scenario. (5 Marks)

Concepts:

- **Vertical Scaling (Scaling Up/Down):** Adding more power (CPU, RAM, SSD) to an existing machine. This is like upgrading to a more powerful car engine.
- **Horizontal Scaling (Scaling Out/In):** Adding more machines (instances/nodes) to a pool of resources, typically behind a load balancer. This is like adding more cars to a train.

A critical architectural implication is that **horizontal scaling is vastly simplified for stateless applications** (e.g., web servers). If a server doesn't store user-specific data locally, any request can be routed to any server. Stateful systems (like traditional databases) are much harder to scale horizontally.

When to Choose:

- **Choose Vertical Scaling:**
  - **Scenario:** a monolithic application or a stateful component like a SQL database (e.g., PostgreSQL, MySQL) that isn't easily sharded.
  - **Example:** database server is running at 90% CPU utilization. The quickest and simplest solution is to upgrade it from 8 vCPUs to 16 vCPUs. This avoids the immense complexity of database sharding.
- **Choose Horizontal Scaling:**
  - **Scenario:** You need to scale a stateless component or require high availability and fault tolerance.
  - **Example:** web application frontend is experiencing traffic spikes during a holiday sale. Configure your cloud platform's auto-scaling group to spin up 10 additional identical web servers behind a load balancer to handle the load. If one server fails, the load balancer stops sending it traffic, and the system continues running seamlessly.

---

4. Describe the process of designing an effective load test for a web application. What key factors should be considered, and how would you determine the success criteria for the test? (5 Marks)

Design Process:
Designing a load test is a methodical process:

1. **Define Objectives & Goals:** Determine what we want to find out. Is it to find the breaking point? Validate performance under expected peak load? Check stability over a long period?
2. **Identify Key Scenarios & Workload Model:** Pinpoint the critical user journeys (e.g., "User logs in, browses products, adds item to cart, checks out"). Model the expected traffic mix and user think times.
3. **Configure Test Environment:** Setup a **staging environment** that mirrors production as closely as possible (hardware, software, network, database size). Using production data may require anonymization.
4. **Implement Test Scripts:** Use tools like **k6, JMeter** to script the identified scenarios. Parameterize inputs (e.g., user credentials, product IDs) to avoid unrealistic caching.
5. **Execute Tests Gradually:** Start with a baseline test, then ramp up load through stages (e.g., 50, 100, 200 concurrent users) to see how performance degrades. Include **soak tests** (endurance tests) to find memory leaks and **stress tests** to find the system's breaking point.
6. **Analyze Results & Report:** Compare results against success criteria. Identify bottlenecks (e.g., slow database queries, high CPU on a service, memory leaks).

Key Factors to Consider:

- **Concurrent Users / Throughput:** Virtual User Count and Requests Per Second (RPS).
- **Latency/Response Times:** P95 (95th percentile) and P99 times are more important than averages, as they show the experience for the slowest users.
- **Error Rate:** The percentage of failed requests (e.g., HTTP 5xx errors, timeouts).
- **System Resources:** CPU, Memory, Network I/O, and Disk I/O utilization on all servers (application, database, cache).
- **Database Metrics:** Number of connections, query throughput, and slow queries.

Success Criteria:
Success is defined by meeting **Service Level Objectives (SLOs)**:

- **Performance:** P95 response time for all critical transactions is under [target] ms (e.g., < 2 seconds).
- **Reliability:** Error rate is < 0.1% (i.e., 99.9% of requests are successful).
- **Stability:** System resources (CPU, Memory) remain under a safe threshold (e.g., < 80%) during the test, and no failures occur under expected load. The system should also recover gracefully when load decreases.