**Explain handling millions of API calls per day in Interview C#**

**1. Scalability and Architecture**

* **Horizontal Scaling:**
  + Use load balancers to distribute API traffic across multiple server instances.
  + Deploy to cloud platforms like Azure or AWS that support auto-scaling based on demand.
* **Microservices Architecture:**
  + Break the application into smaller, independently deployable services.
  + Each service should handle a specific task, making the system easier to scale and maintain.

**2. API Gateway**

* Use an API Gateway (e.g., Azure API Management, AWS API Gateway) to handle:
  + Rate limiting and throttling.
  + Authentication and authorization.
  + Caching of frequently requested data.
  + Routing requests to appropriate microservices.

**3. Optimizing the Code**

* **Asynchronous Programming:**
  + Use async and await to handle I/O-bound operations, such as database access and external API calls, without blocking threads.

public async Task<IActionResult> GetDataAsync()

{

var result = await dataService.FetchDataAsync();

return Ok(result);

}

 **Efficient Serialization:**

* Use high-performance libraries like System.Text.Json instead of Newtonsoft.Json for JSON serialization/deserialization.

 **Connection Pooling:**

* Use connection pooling for database or HTTP connections to reduce overhead.
* Example with HttpClient

private static readonly HttpClient client = new HttpClient();

**4. Database Optimization**

* **Indexing:**
  + Use proper indexes on frequently queried database fields to speed up lookups.
* **Sharding:**
  + Split data across multiple databases based on criteria like user ID.
* **Caching:**
  + Implement caching using in-memory stores like Redis or Memcached to reduce database load.

**5. Load Testing**

* Use tools like Apache JMeter, k6, or Azure Load Testing to simulate millions of API calls and identify bottlenecks.

**6. Logging and Monitoring**

* Implement logging using libraries like Serilog or NLog to track API performance and errors.
* Use monitoring tools like Application Insights (Azure), CloudWatch (AWS), or Prometheus to observe system behavior in real-time.

**7. Rate Limiting and Throttling**

* Protect your API from abuse by implementing rate limiting and throttling.
* Example using middleware:

public void Configure(IApplicationBuilder app)

{

app.UseRateLimiter();

app.UseEndpoints(endpoints => { endpoints.MapControllers(); });

}

**8. Security**

* Use HTTPS to encrypt communication.
* Implement token-based authentication (e.g., OAuth, JWT).
* Prevent overloading with distributed denial-of-service (DDoS) protection.

**9. Queueing and Message Handling**

* Use message queues (e.g., Azure Service Bus, RabbitMQ) to handle high-throughput workloads asynchronously.
* Example:
  + API receives a request and enqueues it for background processing.
  + A worker service processes the queue in the background.

**10. Caching**

* Use caching to reduce the load on the database and backend services.
* Implement caching at various levels:
  + **In-Memory Caching:** Use MemoryCache for local caching in small-scale deployments.
  + **Distributed Caching:** Use Redis for shared caching across multiple servers.
* **Sample Implementation:**
* **Scenario:** A simple ASP.NET Core API handling high traffic with caching and async processing.

public class WeatherController : ControllerBase

{

private readonly IDistributedCache \_cache;

public WeatherController(IDistributedCache cache)

{

\_cache = cache;

}

[HttpGet("weather/{city}")]

public async Task<IActionResult> GetWeather(string city)

{

string cacheKey = $"weather\_{city}";

string cachedWeather = await \_cache.GetStringAsync(cacheKey);

if (!string.IsNullOrEmpty(cachedWeather))

{

return Ok(cachedWeather);

}

// Simulate an external API call

string weatherData = await GetWeatherFromApiAsync(city);

// Cache the data

await \_cache.SetStringAsync(cacheKey, weatherData, new DistributedCacheEntryOptions

{

AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)

});

return Ok(weatherData);

}

private async Task<string> GetWeatherFromApiAsync(string city)

{

await Task.Delay(100); // Simulating API latency

return $"Weather data for {city}";

}

}

**11. Tools and Technologies**

* **Load Balancing:** Azure Traffic Manager, AWS Elastic Load Balancer.
* **Caching:** Redis, Azure Cache for Redis.
* **Queueing:** RabbitMQ, Azure Service Bus, Kafka.
* **Monitoring:** Application Insights, ELK Stack, Prometheus, Grafana.

**Key Takeaways**

1. **Design for Scalability:** Use a distributed architecture.
2. **Optimize Performance:** Use async programming, caching, and connection pooling.
3. **Ensure Reliability:** Implement rate limiting, queueing, and proper monitoring.
4. **Test Extensively:** Load test to identify bottlenecks.

By combining these techniques, you can handle millions of API calls per day in a robust and scalable manner.