# System Design Documentation: Modern Microservices Architectures and Best Practices in C#/.NET Inspired by Industry Leaders

## Introduction

Modern cloud-native applications demand robust, scalable, and resilient systems capable of withstanding rapid growth, unexpected failures, and ever-changing business needs. As companies like Netflix, Uber, and Amazon have demonstrated, the right system design enables not just scale, but agility, maintainability, and global reach. This documentation synthesizes key system design concepts, unpacks real-world implementations by top technology organizations, and provides clear, actionable C#/.NET code snippets. Drawing insights from extensive industry case studies, this report aims to build foundational and advanced understanding for architects and engineers seeking to elevate systems to production-grade standards.

## Microservices Architecture Implementation in C#

### Principles and Benefits

Microservices architecture decomposes a complex application into independently deployable services, each encapsulating a distinct business capability. Compared to monolithic systems, microservices increase agility (small teams own services), enhance scalability (services scale as needed), and support resilience (failure is isolated)12.

**Key attributes:**

* **Service independence:** Each microservice has its own codebase and database, deployable independently.
* **Decentralized development:** Enables polyglot persistence and technology heterogeneity.
* **Scalability and resilience:** Services can be replicated or fail independently.
* **Isolation:** Promotes graceful degradation and robust error handling.

### C# Implementation Example

A typical C# microservice uses ASP.NET Core, with a dedicated data context, repository abstraction, and dependency injection:

|  |
| --- |
| // Product.cs (Entity) public class Product {  public int Id { get; set; }  public string Name { get; set; }  public decimal Price { get; set; } }  // ProductContext.cs (DbContext) public class ProductContext : DbContext {  public ProductContext(DbContextOptions<ProductContext> options) : base(options) { }  public DbSet<Product> Products { get; set; } }  // IProductRepository.cs (Repository Interface) public interface IProductRepository {  IEnumerable<Product> GetAll();  Product GetById(int id);  void Add(Product product);  void Update(Product product);  void Delete(int id);  void Save(); }  // ProductRepository.cs (Implementation) public class ProductRepository : IProductRepository {  private readonly ProductContext \_context;  public ProductRepository(ProductContext context) => \_context = context;   public IEnumerable<Product> GetAll() => \_context.Products.ToList();  public Product GetById(int id) => \_context.Products.Find(id);  public void Add(Product p) { \_context.Products.Add(p); Save(); }  public void Update(Product p) { \_context.Entry(p).State = EntityState.Modified; Save(); }  public void Delete(int id) {  var p = \_context.Products.Find(id);  if (p != null) \_context.Products.Remove(p);  Save();  }  public void Save() => \_context.SaveChanges(); } |

**Best practice:** Register the repository and context in Startup.cs for proper dependency injection3.

|  |
| --- |
| services.AddDbContext<ProductContext>(options => options.UseSqlServer(Configuration.GetConnectionString("ProductDB"))); services.AddScoped<IProductRepository, ProductRepository>(); |

**Real-world alignment:** Netflix, Uber, and Amazon structure their core systems on microservices, favoring domain-driven design and single-responsibility services2.

## API Gateway Patterns in .NET

### Overview and Benefits

An API Gateway serves as a centralized entry point for client requests in a microservices ecosystem, performing routing, protocol translation, authentication, logging, throttling, and aggregation45. It minimizes coupling between frontend clients and backend microservices, and allows unified cross-cutting concern management.

**Industry approach:** Netflix’s Zuul API Gateway handles routing, security, load balancing, rate limiting, and dynamic service discovery5. In .NET, Ocelot is a common choice for implementing API gateways.

### C# Implementation with Ocelot

1. **Install Ocelot:**

|  |
| --- |
| dotnet add package Ocelot |

1. **Basic ocelot.json Configuration:**

|  |
| --- |
| {  "Routes": [  {  "DownstreamPathTemplate": "/api/products",  "DownstreamScheme": "http",  "DownstreamHostAndPorts": [ { "Host": "localhost", "Port": 5001 } ],  "UpstreamPathTemplate": "/products",  "UpstreamHttpMethod": [ "GET" ]  }  ],  "GlobalConfiguration": { "BaseUrl": "http://localhost:5000" } } |

1. **Startup.cs Ocelot Setup:**

|  |
| --- |
| public void ConfigureServices(IServiceCollection services) {  services.AddOcelot(); } public void Configure(IApplicationBuilder app) {  app.UseOcelot().Wait(); } |

**Key features in production setups:**

* **JWT Authentication:** Secure APIs with bearer tokens.
* **Rate Limiting:** Prevent abuse or DDoS.
* **Caching:** Reduce load on backend services.
* **Load Balancing:** Distribute requests across service instances.
* **Aggregators:** Merge multiple service responses for a unified result6.

**Real-world case:** Netflix’s Zuul and Amazon's API Gateway centralize all traffic routing, authentication, and monitoring, as echoed in Ocelot setups for .NET microservices75.

## Load Balancing Strategies with C#

### Patterns and Approaches

Load balancing is essential for high availability and scalability. Strategies include Round Robin, Least Connections, IP Hashing, and Weighted Distribution89.

|  |  |
| --- | --- |
| Strategy | Use Case |
| Round Robin | Homogeneous service instances |
| Least Connections | Varying request duration/processing times |
| Weighted | Different server capacities |
| IP Hash | Session stickiness needed |

**Implementation in C#:**

* **YARP (Yet Another Reverse Proxy):** Microsoft-supported, programmable reverse proxy and load balancer for ASP.NET Core10.
* **Kubernetes:** Handles balancing at the ingress/service level.
* **NGINX/Envoy:** Often used as platform-agnostic proxies/load balancers.

**YARP Example:**

|  |
| --- |
| builder.Services.AddReverseProxy()  .LoadFromConfig(builder.Configuration.GetSection("ReverseProxy")); |

*(Routes and clusters are defined in appsettings.json.)*

**Real-world analogy:** Netflix, Uber, and Amazon use sophisticated load balancers, such as AWS Elastic Load Balancer, Envoy, and custom solutions, to route hundreds of millions of requests efficiently11.

## Caching Techniques and CDN Integration

### Local Caching

**.NET Options:**

* **MemoryCache:** For same-process, short-lived application caching.
* **IDistributedCache (e.g., Redis, SQL):** For multi-node and high-availability scenarios1213.

|  |
| --- |
| services.AddStackExchangeRedisCache(options => {  options.Configuration = "localhost:6379"; }); |

**Usage:**

|  |
| --- |
| public class CacheService {  private readonly IDistributedCache \_cache;  public CacheService(IDistributedCache cache) { \_cache = cache; }  public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> getData) {  var cachedData = await \_cache.GetStringAsync(key);  if (cachedData != null)  return JsonConvert.DeserializeObject<T>(cachedData);   var value = await getData();  await \_cache.SetStringAsync(key, JsonConvert.SerializeObject(value),  new DistributedCacheEntryOptions { AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10) });  return value;  } } |

**Benefits:**

* Reduces database load
* Accelerates response times

### CDN Integration

CDNs cache static content at global edge nodes, reducing latency and improving load times. Azure CDN, AWS CloudFront, or Cloudflare are typical integrations14157.

**Key implementation steps:**

1. **Upload static assets (images, CSS, JS) to storage (e.g., Azure Blob Storage).**
2. **Link storage as CDN origin and reference CDN URLs in app/web config.**

**.NET tip:** Use PhysicalFileProvider in development to share assets, and automate asset deployment to CDN in production16.

**Industrial use:** Netflix deploys its “Open Connect” CDN to serve 95% of video content from edge caches within 100ms for most users, drastically cutting backbone traffic and delivery latency17.

## Database Sharding and Partitioning in C#

### Concepts

Sharding divides large datasets across multiple databases (shards), typically by region, tenant, or partition key187.

**Benefits:**

* Removes single-database performance bottlenecks
* Horizontal scalability
* Improved isolation/security for tenants

**Typical .NET/EF Core Implementation:**

* Use a shard key (e.g., tenantId or region) to select the shard/database.
* Example:

|  |
| --- |
| protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {  var shardConnectionString = ShardResolver.GetConnectionString(CurrentTenantId);  optionsBuilder.UseSqlServer(shardConnectionString); } |

* For more advanced routing, consider a DbContextFactory or repository pattern to instantiate contexts bound to the correct connection string at runtime.

**Hybrid Approach:** Combine multi-tenant (shared database) and sharding. Use EF Core's query filters to automatically constrain tenant data, and designate large tenants to their own shard/database19.

**Industry practice:** Netflix and Uber choose the right datastore per service (polyglot persistence), and partition/shard data intelligently for massive scale20.

## Event-Driven Architecture and Message Queues

### Principles

Event-driven architectures decouple publishers and subscribers through asynchronous messaging, enabling high scalability, reliable data propagation, and eventual consistency21722.

**Core components:**

* **Event Producers:** Trigger changes/events.
* **Message Broker:** Distributes events (RabbitMQ, Kafka, Azure Service Bus).
* **Event Consumers:** React to events, update state, or trigger workflows.

### C# Implementation Example

**Event publishing using MassTransit + RabbitMQ:**

|  |
| --- |
| public record OrderCreatedEvent { public Guid OrderId { get; init; } } |

**Producer:**

|  |
| --- |
| await bus.Publish(new OrderCreatedEvent { OrderId = orderId }); |

**Consumer:**

|  |
| --- |
| public class OrderCreatedConsumer : IConsumer<OrderCreatedEvent> {  public Task Consume(ConsumeContext<OrderCreatedEvent> context) {  // Handle event  return Task.CompletedTask;  } } |

**Best Practices:**

* Use well-defined, versioned integration events.
* Ensure idempotency in consumers to avoid data duplication.
* Monitor event bus health and message delivery.

**Real-world:** Netflix and Uber leverage event buses (Kafka, custom, or cloud-native services) to orchestrate distributed processes, pipeline data (e.g., logging, ETL), and enable consistent business workflows220.

## Circuit Breaker and Resilience Patterns in C#

### Why Use Circuit Breakers

In distributed systems, repeatedly invoking failing services can lead to cascading failures and wasted resources. Circuit breakers detect persistent failures and “open the circuit” to temporarily block calls, allowing dependent services time to recover237.

### Polly: The De Facto Resilience Library for .NET

Install and configure Polly policies as middleware for HttpClient:

|  |
| --- |
| // Add Polly with CircuitBreaker and Retry to an HttpClient builder.Services.AddHttpClient("MyService")  .AddPolicyHandler(Policy.Handle<HttpRequestException>()  .CircuitBreakerAsync(handledEventsAllowedBeforeBreaking: 3, durationOfBreak: TimeSpan.FromSeconds(30)))  .AddPolicyHandler(Policy.Handle<HttpRequestException>()  .WaitAndRetryAsync(3, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)))); |

**Policy configuration:**

|  |
| --- |
| IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy() =>  HttpPolicyExtensions.HandleTransientHttpError()  .CircuitBreakerAsync(3, TimeSpan.FromSeconds(30),  onBreak: (exception, ts) => \_logger.LogWarning("Circuit break: {Exception}", exception),  onReset: () => \_logger.LogInformation("Circuit reset"),  onHalfOpen: () => \_logger.LogInformation("Half-open state."));  IAsyncPolicy<HttpResponseMessage> GetRetryPolicy() =>  HttpPolicyExtensions.HandleTransientHttpError()  .WaitAndRetryAsync(3, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))); |

**Graceful failures:** Use Polly’s Fallback policy to provide default responses or cached data if upstream is unavailable.

**Real-world:** Netflix pioneered circuit breaker patterns with Hystrix (since retired; patterns adopted in .NET via Polly)23.

## Data Replication and Consistency Models

Systems at scale need both data redundancy for durability and careful consistency models to reconcile distributed updates7.

|  |  |  |
| --- | --- | --- |
| Model | Guarantee | Example |
| Strong Consistency | Reads reflect latest write | Amazon S3 since 2020, newer RDBMS, Cosmos DB strong consistency |
| Eventual Consistency | All copies converge, but temporary staleness allowed | DynamoDB, S3 (pre-2020), Netflix data plane |

**Amazon S3 Example:**

* **11 9’s durability**: Data is replicated across 3+ Availability Zones.
* **Checksums** and automatic healing repair corrupt replicas.
* **Versioning** protects objects from accidental/partial deletion.
* **Cross-region replication** for geo-resilience and regulatory compliance2415.

**C# Example:**

|  |
| --- |
| var s3Client = new AmazonS3Client(); var putRequest = new PutObjectRequest { BucketName = "my-bucket", Key = "file.txt", FilePath = @"C:\file.txt" }; await s3Client.PutObjectAsync(putRequest); |

**Note:** S3's consistency model offers guaranteed read-after-write for PUTS and DELETEs in all AWS regions since 2020.

## Service Discovery and Registry

As microservices scale out, their locations (IP, port) change frequently due to orchestrator rescheduling or auto-scaling2572.

**Pattern:** Service discovery assigns human-friendly names to services (“OrderService”), which are resolved to current endpoints by a service registry (e.g., Consul, Eureka, DNS).

**Consul Example (C# setup):**

|  |
| --- |
| var consulClient = new ConsulClient(x => x.Address = new Uri("http://localhost:8500")); var registration = new AgentServiceRegistration {   ID = "orderservice-1", Name = "OrderService",  Address = "127.0.0.1", Port = 5001,  Check = new AgentServiceCheck { HTTP = "http://127.0.0.1:5001/health", Interval = TimeSpan.FromSeconds(10) } }; await consulClient.Agent.ServiceRegister(registration); |

**.NET 8+**: Use Microsoft.Extensions.ServiceDiscovery for config/DNS-based resolution26.

**Industrial usage:** Netflix, Uber, Amazon, and Zalando automate service registration/discovery to enable seamless scaling, rolling deployments, and self-healing behaviors2.

## Distributed Logging, Monitoring, and Observability

### The Need

In distributed systems, tracing, searching, and correlating requests across many microservices is critical for debugging, SLA compliance, and operational awareness27287.

### Implementation

* **Logging libraries:** Serilog (with sinks for files, Seq, Elasticsearch), NLog, or Microsoft.Extensions.Logging.
* **Structured Logging:** Enables rich, queryable logs (e.g., in JSON via Serilog).
* **Tracing:** OpenTelemetry tracks requests across services, supporting context propagation and distributed traces.
* **Aggregation:** Log shippers forward logs to ELK Stack (Elasticsearch, Logstash, Kibana), Seq, or cloud-native backends.

**Serilog/Seq Setup:**

|  |
| --- |
| Log.Logger = new LoggerConfiguration()  .MinimumLevel.Debug()  .Enrich.FromLogContext()  .WriteTo.Seq("http://localhost:5341")  .CreateLogger(); builder.Host.UseSerilog(); |

**Trace correlation:**

* Use middleware to push correlation/request IDs into log context.
* Integrate OpenTelemetry for distributed traces across HttpClient, ASP.NET Core, and message buses.

**Industry example:** Netflix uses custom telemetry platforms; Amazon and Uber gather distributed traces for every request to enable rapid diagnosis and measure SLAs/latency bottlenecks2729.

## Security and Authentication Strategies

### Principles

* **Centralized authentication and authorization** (preferably with OAuth 2.0/OpenID Connect).
* **API Gateway is the single point for authentication, rate limiting, and threat protection.**
* **Per-service authorization** using JWT tokens and claims3074.

**IdentityServer4 Integration Example:**

|  |
| --- |
| services.AddIdentityServer()  .AddSigningCredential(new X509Certificate2("signing.pfx"))  .AddInMemoryApiResources(ApiResources.GetAll())  .AddAspNetIdentity<ApplicationUser>(); |

**Securing an ASP.NET Core API with JWT:**

|  |
| --- |
| services.AddAuthentication("Bearer")  .AddJwtBearer("Bearer", options =>  {  options.Authority = "https://localhost:5000"; // IdentityServer4 address  options.TokenValidationParameters = new TokenValidationParameters { ValidateAudience = false };  }); |

**API Gateway (e.g., Ocelot) secures access, validates tokens, and forwards only authenticated requests to microservices.**

**Other key practices:**

* Encrypt sensitive communication (HTTPS).
* Apply least-privilege via IAM/roles.
* Protect secrets using vaults or secret managers.

## High Availability and Failover Mechanisms

### Core Patterns

**Design for failure:** All components must tolerate underlying infrastructure failures, including power/network losses, instance termination, or software crashes.

**Industry-standard practices:**

* **Redundancy:** Run services in active-active mode across data centers or AZs.
* **Health Checks:** Use liveness/readiness probes to monitor service state; orchestrators handle restart/replace logic.
* **Load Balancing:** Distribute load and reroute on instance failure.
* **Failover Clusters:** Automatically shift master roles when a critical node fails (e.g., leader election in databases).
* **Data replication:** Replicate across regions/zones (e.g., Amazon S3 copies data across 3+ AZs for 11 9’s durability)317.

**Production-proven systems:** Amazon, Netflix, and Uber all implement auto-failover across multiple AZs and replicate data for truly global high availability.

## CDN Usage for Static Assets in .NET

CDN integration is nearly universal for static assets in modern applications16.

**.NET Integration:**

* Use Azure CDN or AWS CloudFront in front of storage accounts/Blob storage containing static files.
* Reference CDN endpoints in your application or web config.
* Use Azure DevOps/GitHub Actions/Terraform for automated deployment & cache purging after new releases.

**Advantages:**

* Fast, localized delivery.
* Lower load on application servers.
* Efficient cache invalidation for rapid updates/deployment.

**Best Practices:**

* Leverage long cache durations for versioned/build assets.
* Ensure HTTPS usage and leverage free SSL certificates provided by the CDN service.
* Monitor CDN health and performance using integrated vendor dashboards17.

## Case Study: Netflix System Design

### Microservices Implementation

Netflix broke up monolithic architecture into thousands of microservices, enabling rapid development, independent deployments, and global scale25.

**Key patterns:**

* API Gateway (Zuul) for request routing and cross-cutting concerns.
* Eureka for service discovery and registry.
* Ribbon for intelligent client-side load balancing.
* Hystrix (now retired) for system resilience and circuit breaking.
* Cloud-native scaling with AWS for infrastructure.
* Atlas and ELK for monitoring/logging.
* Open Connect CDN to cache 95%+ of traffic at the edge.

**C# Inspiration:** Adopt Ocelot (API Gateway), Polly (circuit breaking), and Consul or Kubernetes Service Discovery to mimic Netflix's architecture in .NET432.

## Case Study: Uber System Design

### ETA Computation at Web Scale

Uber processes up to **500,000 ETA requests per second**, requiring:

* Geo-sharded, distributed microservices.
* **Consul/Kubernetes** for dynamic discovery and scaling.
* **Kafka** for high-throughput event streaming between components.
* Real-time and batch processing (Lambda Architecture) for data integration2033.
* ML-enhanced route planning using Graph Neural Networks, LSTM.
* Adaptive load balancing and caching with Redis and Memcached.

**Techniques to emulate in .NET:**

* Horizontal sharding (by region/zone or customer).
* Leveraging streaming/EventBus libraries (MassTransit, Kafka, RabbitMQ).
* Deploy with orchestrators (Kubernetes, Docker Swarm) for auto-scaling and self-healing.
* Employ distributed caches (Redis) and pattern-based routes for fast lookup and resilience.

## Case Study: Amazon S3 Durability and Scalability

Amazon S3 epitomizes cloud-scale storage:

* Achieves **99.999999999% durability** via redundant, cross-AZ, and optionally, cross-region replication.
* Uses checksums and automatic healing to keep data consistent.
* Supports versioning, lifecycle management, and automatic data migration between storage tiers.
* Integrated with IAM, encryption, and transparent access logging.
* Strong read-after-write consistency as of 2020, simplifying client-side logic1524.

**C#/.NET Integration:**

* Use AWS SDK for S3 object upload/download and configuration.
* Configure lifecycle, versioning, and CRR through API.

**Design lesson:** Replicate S3’s high availability with redundant storage, routine health checks, and robust monitoring in your designs.

## Conclusion: Summing Up System Design Excellence

Top technology firms have codified a blueprint for building systems that are **scalable, resilient, observable, and secure**. The patterns outlined here-microservices, API gateways, event-driven architectures, database sharding, distributed caching, service discovery, resilience via circuit breaking, and rigorous observability-are not academic; they form the operational backbone for the platforms we use every day.

**In C# and .NET, these patterns are not just possible-they are first-class citizens** thanks to the robust framework support, comprehensive tooling, and rich community libraries. By drawing on real-world inspirations from Netflix, Uber, Amazon, and others, we ensure our own systems are ready for growth, failure, and success at any scale.

## Further Reading and Ongoing Learning

Dive deeper into each pattern and technique using resources from Microsoft Learn, the DEV Community, GeeksforGeeks, official documentation (e.g., Ocelot, Polly, AWS SDK, Consul), and architecture blogs by major cloud providers. Experiment with the code snippets, deploy real microservices to containers or orchestrators, and measure your system with production-grade logging and observability.

**Remember:** Architecture is a journey; the best systems are forged iteratively, with learning from both success and failure at every scale.

# References (47)

21. *Implementing event-based communication between microservices ...*. <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications>

30. *Authentication and Authorization with IdentityServer4 in .NET Core*. <https://www.freecodespot.com/blog/secure-dot-net-core-using-identity-server/>

28. *C# Tracing with Serilog and SerilogTracing - Structured Blog*. <https://datalust.co/blog/dotnet-tracing-with-serilog>

20. *DeepETA: How Uber Predicts Arrival Times Using Deep Learning*. <https://www.uber.com/blog/deepeta-how-uber-predicts-arrival-times/>

22. *cxalem/eventual-consistency-example - GitHub*. <https://github.com/cxalem/eventual-consistency-example>

23. *Implementing the Circuit Breaker pattern - .NET* . <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-circuit-breaker-pattern>

24. *Data protection in Amazon S3 - Amazon Simple Storage Service*. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html>

25. *Service Discovery in Microservices With .NET and Consul*. <https://www.milanjovanovic.tech/blog/service-discovery-in-microservices-with-net-and-consul>

26. *Service discovery in .NET - .NET* . <https://learn.microsoft.com/en-us/dotnet/core/extensions/service-discovery>

27. *Mastering Distributed Tracing with Serilog and Seq in .NET*. <https://dev.to/iamcymentho/mastering-distributed-tracing-with-serilog-and-seq-in-net-fp>

29. *Resiliency and high availability in microservices - .NET*. <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/resilient-high-availability-microservices>

31. *Design Patterns for High Availability - GeeksforGeeks*. <https://www.geeksforgeeks.org/system-design/design-patterns-for-high-availability/>

32. *Service Discovery using Consul - C# Corner*. <https://www.c-sharpcorner.com/article/service-discovery-using-consul/>

1. *Microservices using ASP.NET Core - Dot Net Tutorials*. <https://dotnettutorials.net/lesson/microservices-using-asp-net-core/>

2. *Kinda Technical* . <https://kindatechnical.com/microservices-architecture/lesson-97-case-study-netflix.html>

3. *Repository Design Pattern in C# - Dot Net Tutorials*. <https://dotnettutorials.net/lesson/repository-design-pattern-csharp/>

4. *Implementing API Gateways with Ocelot - .NET* . <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/implement-api-gateways-with-ocelot>

5. *An Essential Guide to API Gateway: A Netflix Case Study*. <https://xcelore.com/blog/an-essential-guide-to-api-gateway-a-netflix-case-study/>

6. *API Gateway Pattern: Mastering C# Microservices Architecture*. <https://softwarepatternslexicon.com/patterns-c-sharp/8/2/>

7. *How Netflix Scales to 270 Million Users with Java and Microservices*. <https://www.codemotion.com/magazine/microservices/how-netflix-scales-to-270-million-users-with-java-and-microservices/>

8. *Configure ASP.NET Core to work with proxy servers and load balancers ...*. <https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/proxy-load-balancer?view=aspnetcore-9.0>

9. *Load Balancing and Traffic Management for .NET Core Microservices*. <https://dev.to/soham_galande/load-balancing-and-traffic-management-for-net-core-microservices-27j7>

10. *Simple Load Balancer in .NET Core with YARP - C# Corner*. <https://www.c-sharpcorner.com/article/simple-load-balancer-in-net-core-with-yarp/>

11. *How Uber Computes ETA at Half a Million Requests per Second*. <https://www.geeksforgeeks.org/system-design/how-uber-computes-eta-at-half-a-million-requests-per-second/>

12. *Redis Distributed Cache in C#.NET with Examples - TheCodeBuzz*. <https://thecodebuzz.com/redis-distributed-cache-asp-net-core-csharp-redis-examples/>

13. *Distributed caching in ASP.NET Core* . <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-9.0>

14. *BSC Analytics*. <https://bscanalytics.com/insights/deploying-static-content-via-azure-cdn-with-storage-account-integration>

15. *Amazon S3: The Ultimate Guide to Scalable and Secure Cloud Storage*. <https://dev.to/borhanitrash/amazon-s3-the-ultimate-guide-to-scalable-and-secure-cloud-storage-1hhj>

16. *Sharing static assets between asp.net core applications (or via Azure ...*. <https://stackoverflow.com/questions/40736169/sharing-static-assets-between-asp-net-core-applications-or-via-azure-cdn>

17. *Azure CDN for Static Assets, APIs, and Front Door* . <https://last9.io/blog/azure-cdn-for-static-assets-apis-and-front-door/>

18. *How to use database sharding with EF Core and C#"*. <https://stackoverflow.com/questions/61632457/how-to-use-database-sharding-with-ef-core-and-c>

19. *Part6: Using sharding to build multi-tenant apps using ASP.NET Core and ...*. <https://www.thereformedprogrammer.net/part6-using-sharding-to-build-multi-tenant-apps-using-asp-net-core-and-ef-core/>

33. *Advanced Computational Techniques in Uber’s ETA Calculation: A ...*. <https://www.researchgate.net/profile/Ahmad-Abdur-Rahman-Khan/publication/383752088_Understanding_Uber's_ETA_Calculation_A_Computational_Perspective/links/66db4e6dfa5e11512ca3def6/Understanding-Ubers-ETA-Calculation-A-Computational-Perspective.pdf>