# 25 System Designs in C# (Azure-focused)

Audience: 10–12+ yrs .NET full‑stack/architect. Focus on pragmatic, scalable designs with Azure services. Each section includes: **requirements**, **architecture**, **data model**, **key flows**, **scaling & reliability**, and **C# snippets** (domain, service/handler, infra).

## 0) Common Building Blocks (used across designs)

**Tech stack**

* **Runtime**: ASP.NET Core minimal APIs / Controllers, .NET 8, C# 12
* **Data**: Azure SQL, Cosmos DB, Redis (Azure Cache for Redis)
* **Messaging**: Azure Service Bus / Event Hubs / Kafka (Confluent on Azure)
* **Storage**: Azure Blob Storage
* **Auth**: Azure AD / B2C (JWT bearer)
* **Edge**: Azure Front Door + Azure CDN + API Management
* **Observability**: OpenTelemetry + Azure Monitor + Application Insights
* **CI/CD**: GitHub Actions → Azure (Bicep/Terraform)

**Cross-cutting patterns**

* **Clean Architecture + CQRS** (API → Application → Domain → Infrastructure)
* **Idempotency** for commands; **Outbox** + **Inbox** tables for exactly‑once message processing
* **Rate limiting** middleware; **Circuit breaker** (Polly) for downstream resiliency
* **Feature flags** (Azure App Configuration)

**Base abstractions (used in snippets)**

public interface ICommand { }

public interface IQuery<T> { }

public interface ICommandHandler<T> where T : ICommand { Task HandleAsync(T cmd, CancellationToken ct); }

public interface IQueryHandler<TQ,TR> where TQ : IQuery<TR> { Task<TR> HandleAsync(TQ q, CancellationToken ct); }

public record Result(bool Success, string? Error = null) {

public static Result Ok() => new(true);

public static Result Fail(string e) => new(false, e);

}

## 1) URL Shortener (TinyURL)

**Requirements**: Create short codes, redirect fast, track clicks, custom aliases, TTL.

**Architecture**:

* API (Write): POST /shorten → generate code → store mapping
* API (Read): GET /{code} → resolve → 301 redirect
* Read path served from **Redis** with fallback to **Cosmos DB** (key‑value).
* **Bloom filter** in Redis to avoid DB misses flood.
* **Background**: Click events to Event Hubs → Azure Data Explorer/Databricks for analytics.

**Data model** (Cosmos, partition key = code):

{ "code":"a1B9x", "url":"https://...", "createdAt":"...", "ttl": 31536000, "owner":"userId", "custom":true }

**C#** – code generation & resolve:

public interface IUrlRepo { Task StoreAsync(string code, string url, TimeSpan? ttl, CancellationToken ct); Task<string?> GetAsync(string code, CancellationToken ct);}

public class UrlService {

private readonly IUrlRepo \_repo; private readonly IDatabase \_redis; private const string KeyPrefix = "url:";

public UrlService(IUrlRepo repo, IConnectionMultiplexer mux) { \_repo = repo; \_redis = mux.GetDatabase(); }

public async Task<string> ShortenAsync(Uri url, TimeSpan? ttl, CancellationToken ct) {

var code = Base62.Create(7); // 62^7 ≈ 3.5e12

await \_repo.StoreAsync(code, url.ToString(), ttl, ct);

await \_redis.StringSetAsync(KeyPrefix+code, url.ToString(), ttl);

return code;

}

public async Task<Uri?> ResolveAsync(string code, CancellationToken ct) {

var cached = await \_redis.StringGetAsync(KeyPrefix+code);

if (cached.HasValue) return new Uri(cached!);

var url = await \_repo.GetAsync(code, ct);

if (url is null) return null;

await \_redis.StringSetAsync(KeyPrefix+code, url, TimeSpan.FromDays(30));

return new Uri(url);

}

}

**Scaling**: Read is cache‑first. Use **Front Door** anycast + **Functions** for edge redirects. Write‑heavy spikes: pre‑allocated code ranges per instance to avoid contention.

## 2) Scalable Notification System

**Requirements**: Multichannel (email/SMS/push), templates, retries, rate control, user preferences, auditing.

**Architecture**:

* Producer apps publish **NotificationRequested** to Service Bus (topic per channel).
* **Workers** per channel pull messages, apply throttling (Polly), send via providers (SendGrid/Twilio/FCM), write **NotificationSent/Failed** events.
* **Preference Service** (Cosmos) filters subscriptions.

**Data model (Preferences)**

{ "userId":"u1", "channels":{"email":true,"sms":false}, "quietHours":{"start":"22:00","end":"07:00","tz":"Asia/Kolkata"} }

**C#** – worker skeleton with outbox/idempotency:

public record NotificationRequested(string Id, string UserId, string Channel, string Template, Dictionary<string,string> Data);

public class EmailWorker : BackgroundService {

private readonly ServiceBusProcessor \_proc; private readonly IEmailSender \_email; private readonly IPreferenceStore \_prefs; private readonly IOutbox \_outbox;

public EmailWorker(ServiceBusClient client, IEmailSender email, IPreferenceStore prefs, IOutbox outbox) {

\_proc = client.CreateProcessor("notif", new ServiceBusProcessorOptions{ SubQueue = SubQueue.None });

\_email = email; \_prefs = prefs; \_outbox = outbox; }

protected override async Task ExecuteAsync(CancellationToken ct) => \_proc.ProcessMessageAsync += OnMsg;

private async Task OnMsg(ProcessMessageEventArgs arg) {

var evt = arg.Message.Body.ToObjectFromJson<NotificationRequested>();

if (await \_outbox.SeenAsync(evt.Id)) { await arg.CompleteMessageAsync(arg.Message); return; }

if(!await \_prefs.AllowedAsync(evt.UserId, evt.Channel)) { await arg.CompleteMessageAsync(arg.Message); return; }

var result = await \_email.SendAsync(evt.Template, evt.Data);

await \_outbox.StoreAsync(evt.Id);

await arg.CompleteMessageAsync(arg.Message);

}

}

**Scaling**: One topic, N subscriptions per channel/provider; add partitions; enable retry/TTL & DLQ. Use APIM + private endpoints for producer authentication.

## 3) News Feed (Facebook/Twitter‑style)

**Requirements**: Write posts, timelines, likes, fan‑out, ranking, pagination, cold storage.

**Architecture**:

* **Write path**: User posts → **Fan‑out** service computes follower set and writes to **Redis Streams** + **Cosmos** (UserTimeline). Very large accounts fall back to **fan‑out on read**.
* **Read path**: GET /feed → merge from Redis (recent) + Cosmos (older) → ranker (feature flags) → return cursors.
* **Engagement events** (likes, comments) to Event Hubs for ranking features.

**Data model**

* Post { postId, authorId, text, media[], ts } (Cosmos, partition by authorId)
* TimelineItem { userId, postId, authorId, ts } (Cosmos, partition by userId)

**C#** – fan‑out write:

public async Task FanOutAsync(PostCreated e, CancellationToken ct){

var followers = await \_graph.GetFollowersAsync(e.AuthorId, ct);

var items = followers.Select(f => new TimelineItem(f, e.PostId, e.AuthorId, e.Ts));

await \_cosmos.BulkInsertAsync(items, ct);

await \_redis.StreamAddAsync($"feed:{e.AuthorId}", new[] { new NameValueEntry("postId", e.PostId) });

}

**Scaling**: Hybrid approach (fan‑out write for small/medium, fan‑out read for celebrities). Use **Change Feed** on Cosmos to trigger backfills.

## 4) API Rate Limiter

**Requirements**: Limit per API key/IP; sliding window; headers X-RateLimit-\*; distributed.

**Architecture**: API gateway (APIM) for coarse limits; service‑level distributed limiter in Redis using **token bucket** or **fixed window with leaky bucket**.

**C#** – middleware with Redis LUA (atomic):

public class RateLimitOptions { public int Limit { get; init; } = 100; public TimeSpan Window { get; init; } = TimeSpan.FromMinutes(1); }

public class RedisRateLimiter : IMiddleware {

private readonly IDatabase \_db; private readonly RateLimitOptions \_opt;

private const string Script = @"local key = KEYS[1] local now = tonumber(ARGV[1]) local window = tonumber(ARGV[2]) local limit = tonumber(ARGV[3]) redis.call('ZREMRANGEBYSCORE', key, 0, now - window) local count = redis.call('ZCARD', key) if count < limit then redis.call('ZADD', key, now, now) redis.call('PEXPIRE', key, window) return 1 else return 0 end";

public RedisRateLimiter(IConnectionMultiplexer mux, IOptions<RateLimitOptions> opt){ \_db = mux.GetDatabase(); \_opt = opt.Value; }

public async Task InvokeAsync(HttpContext ctx, RequestDelegate next){

var id = ctx.Request.Headers["X-Api-Key"].FirstOrDefault() ?? ctx.Connection.RemoteIpAddress?.ToString() ?? "anon";

var key = $"rl:{id}"; var now = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();

var ok = (int) (long) await \_db.ScriptEvaluateAsync(Script, new RedisKey[]{key}, new RedisValue[]{ now, (long)\_opt.Window.TotalMilliseconds, \_opt.Limit });

if (ok==1) { await next(ctx); }

else { ctx.Response.StatusCode = 429; await ctx.Response.WriteAsync("Too Many Requests"); }

}

}

**Scaling**: Keep keys small; shard Redis; add APIM rate‑limit policy as first shield.

## 5) Distributed Cache

**Requirements**: Read‑through, write‑through, TTL, cache stampede protection, versioning.

**Architecture**: Redis as L1; optional **CDN/Front Door** for static; **Cache‑Aside** pattern with **single‑flight** (mutex per key) to prevent thundering herd.

**C#** – cache aside helper:

public class CacheAside<T> {

private readonly IDatabase \_db; private static readonly ConcurrentDictionary<string, SemaphoreSlim> \_locks = new();

public CacheAside(IConnectionMultiplexer mux){ \_db = mux.GetDatabase(); }

public async Task<T?> GetOrSetAsync(string key, TimeSpan ttl, Func<Task<T?>> factory) {

var cached = await \_db.StringGetAsync(key);

if (cached.HasValue) return JsonSerializer.Deserialize<T>(cached!);

var gate = \_locks.GetOrAdd(key, \_ => new SemaphoreSlim(1,1));

await gate.WaitAsync();

try {

cached = await \_db.StringGetAsync(key);

if (cached.HasValue) return JsonSerializer.Deserialize<T>(cached!);

var value = await factory();

if (value is not null) await \_db.StringSetAsync(key, JsonSerializer.Serialize(value), ttl);

return value;

} finally { gate.Release(); \_locks.TryRemove(key, out \_); }

}

}

## 6) Video Streaming Platform (YouTube/Netflix‑like)

**Requirements**: Upload/transcode, adaptive bitrate (HLS/DASH), CDN, DRM, search, recommendations.

**Architecture**:

* Upload → **Blob Storage** → Event Grid → **Transcoding** (Azure Media Services / FFmpeg in AKS) → generate HLS renditions → store manifests in Blob → **Front Door + CDN** for delivery.
* Metadata in Cosmos/SQL; comments/likes events → Event Hubs → analytics & recommender.
* DRM (PlayReady/Widevine) via Media Services.

**C#** – FFmpeg job enqueue (simplified):

public record TranscodeRequest(string VideoId, string BlobUrl);

public class TranscodeController : ControllerBase {

private readonly ServiceBusSender \_sender;

public TranscodeController(ServiceBusClient c){ \_sender = c.CreateSender("transcode"); }

[HttpPost("/videos/{id}/transcode")] public async Task<IActionResult> Start(string id, [FromBody] string blobUrl){

await \_sender.SendMessageAsync(new ServiceBusMessage(BinaryData.FromObjectAsJson(new TranscodeRequest(id, blobUrl))));

return Accepted();

}

}

**Scaling**: Processing on AKS with **KEDA** scaling by queue depth. Delivery via **CDN**; protect blobs with SAS & origin private access.

## 7) E‑commerce Checkout Flow

**Requirements**: Cart, pricing, promotions, inventory hold, payment, order creation, idempotency, SAGA for failures.

**Architecture**: Microservices: **Cart**, **Pricing**, **Inventory**, **Payment**, **Order** orchestrated by **Checkout Orchestrator** (Durable Functions or custom SAGA). Outbox for events.

**Data model**: Order (SQL) with OrderLines; Inventory (SQL) with reservedQty.

**C#** – Orchestrator (Durable Functions style pseudo):

[Function("CheckoutSaga")]

public static async Task Run([OrchestrationTrigger] TaskOrchestrationContext ctx){

var cmd = ctx.GetInput<CheckoutCommand>();

try {

var price = await ctx.CallActivityAsync<Money>("Price", cmd);

await ctx.CallActivityAsync("ReserveInventory", cmd);

var paymentId = await ctx.CallActivityAsync<string>("ChargePayment", new Charge(cmd.UserId, price));

var orderId = await ctx.CallActivityAsync<string>("CreateOrder", new CreateOrder(cmd, paymentId));

await ctx.CallActivityAsync("ConfirmInventory", new Confirm(cmd.Items));

} catch (Exception) {

await ctx.CallActivityAsync("ReleaseInventory", cmd.Items);

await ctx.CallActivityAsync("RefundIfAny", cmd.UserId);

throw;

}

}

**Scaling**: Stateless services; use **idempotency keys** for payment; **compensations** on failure; inventory reservation with expiry (Redis TTL).

## 8) Chat Application (WhatsApp/Slack)

**Requirements**: 1:1, group chats, typing indicators, presence, message ordering, delivery receipts, E2E (optional), media.

**Architecture**:

* WebSockets (SignalR Service) for realtime; messages through **Service Bus / Kafka**; storage in Cosmos (partition by conversationId).
* **Sequence numbers** per conversation for ordering; **ack** receipts.

**C#** – SignalR hub:

public class ChatHub : Hub {

private readonly IMessageStore \_store; private readonly IBus \_bus;

public ChatHub(IMessageStore s, IBus b){ \_store = s; \_bus = b; }

public async Task SendMessage(string convoId, string text){

var msg = new ChatMessage(convoId, Context.UserIdentifier!, text, DateTimeOffset.UtcNow);

await \_store.AppendAsync(msg);

await \_bus.PublishAsync(new MessageCreated(msg));

await Clients.Group(convoId).SendAsync("message", msg);

}

public Task Join(string convoId) => Groups.AddToGroupAsync(Context.ConnectionId, convoId);

}

**Scaling**: Use **Azure SignalR Service** (serverless scale), sticky groups, backpressure by queueing to Kafka.

## 9) Ride‑Hailing (Uber/Ola)

**Requirements**: Driver location updates, matching, ETA, surge pricing, trip state machine, payments.

**Architecture**:

* **Location**: drivers publish GPS to **Event Hubs**; **Geo‑index** (Azure Cosmos + spatial) or Elastic;
* **Matcher**: partitioned by city/zone; nearest‑neighbor within radius;
* **Trip Service** with state machine; **Pricing** (surge) uses demand/supply metrics.

**C#** – Trip state machine (Stateless library):

public enum TripState { Requested, DriverAssigned, InProgress, Completed, Cancelled }

public enum Trigger { AssignDriver, Start, Complete, Cancel }

var sm = new StateMachine<TripState, Trigger>(TripState.Requested);

sm.Configure(TripState.Requested)

.Permit(Trigger.AssignDriver, TripState.DriverAssigned)

.PermitReentry(Trigger.Cancel)

.OnExit(() => Publish("DriverAssigned"));

sm.Configure(TripState.DriverAssigned)

.Permit(Trigger.Start, TripState.InProgress);

sm.Configure(TripState.InProgress)

.Permit(Trigger.Complete, TripState.Completed);

**Scaling**: Shard by city; keep hot data in Redis; long‑poll to websocket fallback for low‑end devices.

## 10) Payment System

**Requirements**: Create charge, hold, capture, refund; PCI concerns; webhooks; reconciliation.

**Architecture**:

* Tokenized card via payment gateways (Stripe/Razorpay) – never store PAN.
* **Payment Intent** model; outbox to publish PaymentSucceeded/Failed.
* **Webhook receiver** validates signatures, updates state idempotently.

**C#** – Idempotent intents:

public class PaymentService {

private readonly IPaymentProvider \_provider; private readonly IPaymentStore \_store;

public async Task<Result> CreateOrGetAsync(string idempotencyKey, Money amount){

var existing = await \_store.GetByKeyAsync(idempotencyKey);

if (existing is not null) return existing.Status == "succeeded" ? Result.Ok() : Result.Fail(existing.Error!);

var resp = await \_provider.CreateIntentAsync(idempotencyKey, amount);

await \_store.SaveAsync(resp.Intent);

return resp.Success ? Result.Ok() : Result.Fail(resp.Error!);

}

}

## 11) File Storage/Sharing (Dropbox/Drive)

**Requirements**: Upload/download, versioning, sharing, sync client, conflict resolution, dedupe.

**Architecture**: Blob Storage with **hierarchical namespace** (ADLS Gen2). Metadata in SQL (files, versions, ACLs). Chunked uploads; **delta sync** via ChangeFeed. Share links via SAS.

**C#** – chunk upload controller:

[HttpPost("/files/{id}/chunks/{n}")]

public async Task<IActionResult> UploadChunk(string id, int n){

using var stream = Request.Body;

var blockId = Convert.ToBase64String(Encoding.UTF8.GetBytes(n.ToString("D6")));

var blob = \_blob.GetBlockBlobClient($"files/{id}");

await blob.StageBlockAsync(blockId, stream);

await \_repo.RecordChunkAsync(id, n);

return Accepted();

}

## 12) Search Engine (Indexing)

**Requirements**: Crawl, parse, index, query, ranking.

**Architecture**: Crawl → Parse → **Indexer** (Azure Cognitive Search or Elastic). Store raw docs in Blob. Queries served by Cognitive Search; custom ranking via skills.

**C#** – indexing batch:

var batch = IndexDocumentsBatch.Upload(docs);

await \_searchClient.IndexDocumentsAsync(batch);

## 13) Logging & Metrics (like Splunk)

**Requirements**: Ingest logs/events, parse, query, alerts, retention tiers.

**Architecture**: Apps emit OTLP → **Azure Monitor / Log Analytics**. Self‑hosted: Ingest via Kafka → Fluent Bit → ClickHouse. Alerts via Action Groups.

**C#** – OpenTelemetry setup:

builder.Services.AddOpenTelemetry().WithTracing(t => t

.AddAspNetCoreInstrumentation()

.AddHttpClientInstrumentation()

.AddOtlpExporter());

## 14) Real‑time Collaborative Docs (Google Docs)

**Requirements**: Multi‑user editing, CRDT/OT, presence, cursors, persistence.

**Architecture**: SignalR for realtime; **CRDT (Yjs/Automerge) hosted in Node or .NET)**; snapshots in Blob; ops journal in Cosmos.

**C#** – skeleton for OT operation apply:

public class DocSession {

private readonly IOpStore \_store;

public Task ApplyAsync(string docId, Operation op){ /\* transform against concurrent ops, persist, broadcast \*/ return Task.CompletedTask; }

}

## 15) Online Ticket Booking (BookMyShow)

**Requirements**: Showtimes, seat maps, locks, payment, anti‑oversell.

**Architecture**:

* **Seat Lock Service** (Redis) with TTL;
* **Reservation** confirms seats transactionally in SQL;
* **SAGA** with payment;
* **Read model** (projection) for seat availability cached.

**C#** – seat lock:

public async Task<bool> TryLockSeat(string showId, string seat){

var key = $"seat:{showId}:{seat}"; return await \_redis.StringSetAsync(key, "1", TimeSpan.FromMinutes(5), When.NotExists);

}

## 16) Job Scheduling (Distributed Cron)

**Requirements**: Time‑based jobs, retries, exactly‑once, backoff, time zones.

**Architecture**: Quartz.NET clustered with SQL; or custom: enqueue due jobs to Service Bus using **Scheduled Messages**; workers execute; store **Lease** in Redis to avoid duplicate runners.

**C#** – Service Bus scheduled:

await sender.ScheduleMessageAsync(new ServiceBusMessage(payload), DateTimeOffset.UtcNow.AddMinutes(10));

## 17) Ad‑Serving (Google Ads)

**Requirements**: Targeting, auctions (second‑price), low latency, pacing, fraud detection.

**Architecture**:

* Ad request → **Edge** compute (Azure Functions at Edge) → fetch eligible ads from Redis; run auction; return creative URL via CDN.
* **Impression/Click** events to Event Hubs; pacing controllers adjust budgets.

**C#** – auction:

public Ad SelectWinner(IEnumerable<AdBid> bids) {

var ordered = bids.OrderByDescending(b => b.Bid);

var winner = ordered.First();

var price = ordered.Skip(1).FirstOrDefault()?.Bid ?? winner.FloorPrice; // second price

winner.PayPrice = price; return winner;

}

## 18) Monitoring & Alerting (Prometheus‑style)

**Requirements**: Scrape metrics, TSDB, alert rules, dashboards.

**Architecture**: Use **Azure Monitor** as managed; self‑host: Prometheus + Grafana + Alertmanager on AKS. For .NET export **/metrics** (OpenMetrics).

**C#** – prometheus-net endpoint:

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

## 19) Content Delivery Network (CDN)

**Requirements**: Edge caching, invalidation, origin shielding, signed URLs, compression.

**Architecture**: Azure CDN (Front Door Standard/Premium). Origins: Blob/Static Web Apps. **Rules Engine** for caching headers, gzip/brotli, geo‑filters. Signed URLs via HMAC.

**C#** – sign URL:

public string Sign(string path, TimeSpan ttl){

var exp = DateTimeOffset.UtcNow.Add(ttl).ToUnixTimeSeconds();

var sig = Convert.ToHexString(HMACSHA256.HashData(Key, Encoding.UTF8.GetBytes($"{path}{exp}")));

return $"{path}?exp={exp}&sig={sig}";

}

## 20) IoT Device Data Ingestion

**Requirements**: Millions of devices, telemetry ingestion, device twin, commands, cold/hot path.

**Architecture**: Azure **IoT Hub** → routes to **Event Hubs**; hot path to Stream Analytics/Functions; cold path to ADLS; device twins manage config.

**C#** – device send telemetry:

var devClient = DeviceClient.CreateFromConnectionString(cs, TransportType.Mqtt);

await devClient.SendEventAsync(new Message(Encoding.UTF8.GetBytes(JsonSerializer.Serialize(payload))));

## 21) Real‑time Gaming Backend

**Requirements**: Matchmaking, rooms, state sync, authoritative server, anti‑cheat, leaderboards.

**Architecture**:

* SignalR/UDP for realtime; matchmaking queue in Redis; game state authoritative in server process; snapshots to Redis; leaderboards in Redis Sorted Sets.

**C#** – leaderboard:

await \_redis.SortedSetIncrementAsync("lb:season1", playerId, scoreDelta);

var top10 = await \_redis.SortedSetRangeByRankWithScoresAsync("lb:season1", 0, 9, Order.Descending);

## 22) Workflow Orchestration (Airflow‑style)

**Requirements**: DAGs, retries, backfills, UI, sensors/operators.

**Architecture**: Use **Azure Data Factory** for managed; or **DAG engine** with Temporal.io / Dapr Workflows / Durable Functions. Store DAGs in Git; executors on AKS.

**C#** – Durable Functions DAG:

[Function("Pipeline")]

public static async Task Run([OrchestrationTrigger] TaskOrchestrationContext ctx){

await ctx.CallActivityAsync("Extract");

await ctx.CallActivityAsync("Transform");

await ctx.CallActivityAsync("Load");

}

## 23) Recommendation Engine (Netflix/Amazon)

**Requirements**: Offline training, online serving, feature store, AB tests.

**Architecture**:

* Offline: Databricks/Spark trains ALS/Deep models; export embeddings to **Cosmos/Redis**.
* Online: **Candidate generation** from similar items; **re‑rank** using features; AB via Front Door rules.

**C#** – cosine similarity (online quick KNN):

public static double Cosine(double[] a, double[] b){

double dot=0,na=0,nb=0; for(int i=0;i<a.Length;i++){ dot+=a[i]\*b[i]; na+=a[i]\*a[i]; nb+=b[i]\*b[i]; }

return dot / (Math.Sqrt(na)\*Math.Sqrt(nb));

}

## 24) Fraud Detection System

**Requirements**: Real‑time scoring, rules + ML, case management, feedback loop.

**Architecture**:

* Events → Kafka → **Streaming scorer** (Flink/Spark Structured Streaming) calling model; threshold → block/hold; write to **Cases** (SQL) for manual review; feedback to model training.

**C#** – rules engine snippet:

public bool IsSuspicious(Transaction t) =>

t.Amount > 100000 || (t.Country!=t.User.HomeCountry && t.NightTime);

## 25) Sharded Database & Consistent Hashing

**Requirements**: Horizontal scaling, rebalancing, minimal movement, HA.

**Architecture**: Consistent hash ring of N shards (Azure SQL elastic pools / Cosmos logical shards). **Router** maps key→shard; **Rebalancer** adds virtual nodes.

**C#** – consistent hashing ring:

public class HashRing<TNode> {

private readonly SortedDictionary<uint,TNode> \_ring = new();

public HashRing(IEnumerable<TNode> nodes, int vnodes=100){ foreach(var n in nodes) AddNode(n, vnodes); }

public void AddNode(TNode node, int vnodes){ for(int i=0;i<vnodes;i++) \_ring[Hash($"{node}-{i}")] = node; }

public TNode GetNode(string key){ var h=Hash(key); if(!\_ring.TryGetValue(h, out var node)){ var kv=\_ring.FirstOrDefault(k=>k.Key>h); node = kv.Value==null?\_ring.First().Value:kv.Value; } return node; }

private static uint Hash(string s){ using var md5 = MD5.Create(); return BitConverter.ToUInt32(md5.ComputeHash(Encoding.UTF8.GetBytes(s)),0); }

}

**Scaling**: Add **virtual nodes** to smooth distribution; migrate with dual‑writes + backfill.

# Reference Architecture Diagram (text)

[Client] → [Front Door/CDN] → [APIM] → [ASP.NET Core APIs] → [Services]

↘︎ Observability (OTel → AppInsights)

Services → [Cosmos/SQL/Redis/Blob] | Async → [Service Bus/Event Hubs/Kafka]

# Security & Compliance Notes

* Use **Managed Identities**; private endpoints; WAF; token validation; PII tokenization; encryption at rest (CMK if needed).

# Testing Strategy

* Contract tests (HTTP + Async), chaos testing (fault injection with Polly), performance tests (k6), canaries + feature flags.

# How to Use

* Treat each section as a blueprint. Copy snippets into Clean Architecture solution: Api, Application, Domain, Infrastructure. Replace stub interfaces with concrete Azure SDK implementations.