Designing a Scalable URL Shortener on Microsoft Azure

Executive Summary

This document outlines the complete architecture for building a URL shortener service capable of handling 100 million URLs per day using Microsoft Azure cloud services. The design emphasizes scalability, reliability, and cost-effectiveness while leveraging Azure's managed services to minimize operational overhead.

Our architecture transforms a complex distributed systems challenge into a manageable solution by strategically combining Azure's platform services. Rather than building everything from scratch, we'll compose proven cloud components that handle the operational complexity while we focus on business logic.

Understanding the Problem Space

Before diving into Azure-specific solutions, we need to establish what we're building and why certain architectural decisions matter. A URL shortener might seem simple on the surface, but operating at scale introduces fascinating technical challenges that touch every aspect of distributed systems design.

Scale Requirements Analysis

When we say "100 million URLs per day," we're describing an average of 1,157 URLs per second. However, web traffic follows predictable patterns with significant peaks during business hours and major events. Planning for 5-10x average traffic during peak periods means our system must handle approximately 10,000-12,000 URL creation requests per second.

The more interesting challenge lies in the read-to-write ratio. Every URL we shorten potentially gets clicked hundreds or thousands of times. Popular social media links can receive millions of clicks within hours. Conservative estimates suggest a 100:1 read-to-write ratio, meaning our system must handle roughly 1 million redirect requests per second during peak periods.

Understanding this ratio fundamentally shapes our architecture. We'll optimize heavily for read performance while ensuring write operations remain fast and reliable.

Data Growth Projections

Storage planning requires careful calculation. Each URL record contains the original URL (averaging 200 characters), our generated short code (7 characters), timestamps, user metadata, and database indexes. Including overhead, each record consumes approximately 300 bytes.

With 100 million URLs daily, we're adding 30GB of data per day, or roughly 11TB annually. Factor in database overhead, indexes, replication, and backups, and we're planning for 25-30TB of annual growth. This projection helps us choose appropriate Azure database tiers and plan for long-term costs.

Azure Architecture Overview

Microsoft Azure provides an ideal platform for this challenge because it offers managed services that handle operational complexity while maintaining the flexibility to optimize for our specific requirements. Our architecture follows Azure's recommended patterns for high-availability, globally distributed applications.

The key insight behind our Azure approach is leveraging managed services wherever possible. Rather than managing virtual machines, operating systems, and database clusters, we'll use Platform-as-a-Service offerings that automatically handle scaling, patching, backup, and monitoring.

Regional Distribution Strategy

Azure's global infrastructure becomes a competitive advantage for our URL shortener. We'll deploy across multiple Azure regions to minimize latency for users worldwide while providing automatic failover capabilities.

Our primary regions will be East US, West Europe, and Southeast Asia, providing optimal coverage for major population centers. Each region contains a complete application stack with local caching and database read replicas. Azure Front Door intelligently routes users to their nearest healthy region.

Application Layer Architecture

Azure App Service Implementation

Azure App Service provides the perfect foundation for our URL shortening API. Think of App Service as a managed platform that eliminates server administration while providing enterprise-grade capabilities like automatic scaling, deployment slots, and integrated monitoring.

Our application design follows REST principles with two primary endpoints. The shortening endpoint accepts long URLs and returns short codes, while the redirect endpoint converts short codes back to original URLs. This separation allows us to optimize each operation independently.

```csharp
```

```
// Main controller implementing URL shortening logic
// Hosted on Azure App Service with automatic scaling
[ApiController]
[Route("api/[controller]")]
public class UrlController : ControllerBase
{
    private readonly IUrlService _urlService;
    private readonly IDistributedCache _distributedCache;
    private readonly ILogger<UrlController> _logger;
    private readonly IClickEventProcessor _clickProcessor;

    public UrlController(
        IUrlService urlService,
        IDistributedCache distributedCache,
        ILogger<UrlController> logger,
        IClickEventProcessor clickProcessor)
    {
        _urlService = urlService;
        _distributedCache = distributedCache;
        _logger = logger;
        _clickProcessor = clickProcessor;
    }

    /// <summary>
    /// Creates a short URL from a long URL
    /// Implements rate limiting and validation through Azure API Management
    /// </summary>
    [HttpPost("shorten")]
    public async Task<IActionResult> ShortenUrl([FromBody] ShortenUrlRequest request)
    {
        // Validate the incoming URL format and security
        if (!Uri.TryCreate(request.Url, UriKind.Absolute, out var uri))
        {
            _logger.LogWarning("Invalid URL format received: {Url}", request.Url);
            return BadRequest(new { error = "Invalid URL format" });
        }

        // Check against known malicious domains
        if (await _urlService.IsUrlMaliciousAsync(request.Url))
        {
            _logger.LogWarning("Malicious URL blocked: {Url}", request.Url);
            return BadRequest(new { error = "URL not allowed" });
        }

        try
        {
```

```
            // Generate short code using our base62 algorithm
            var shortCode = await _urlService.CreateShortUrlAsync(request.Url, request.CustomCode);

            _logger.LogInformation("Created short URL {ShortCode} for {OriginalUrl}",
                shortCode, request.Url);

            return Ok(new ShortenUrlResponse
            {
                ShortUrl = $"https://short.ly/{shortCode}",
                ShortCode = shortCode
            });
        }
        catch (DuplicateShortCodeException)
        {
            return Conflict(new { error = "Custom short code already exists" });
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to create short URL for {Url}", request.Url);
            return StatusCode(500, new { error = "Internal server error" });
        }
    }

    /// <summary>
    /// Redirects short URLs to original URLs
    /// Heavily cached for optimal performance
    /// </summary>
    [HttpGet("{shortCode}")]
    public async Task<IActionResult> RedirectUrl(string shortCode)
    {
        // Multi-level caching strategy for optimal performance

        // First, try Azure Cache for Redis (distributed cache)
        var cachedUrl = await _distributedCache.GetStringAsync($"url:{shortCode}");
        if (cachedUrl != null)
        {
            // Record click event asynchronously
            _ = _clickProcessor.RecordClickAsync(shortCode, Request);

            return Redirect(cachedUrl);
        }

        // Fallback to database lookup
        var urlRecord = await _urlService.GetUrlRecordAsync(shortCode);
        if (urlRecord == null)
        {
            _logger.LogInformation("Short code not found: {ShortCode}", shortCode);
```

```
            return NotFound();
        }

        // Check if URL has expired
        if (urlRecord.ExpiresAt.HasValue && urlRecord.ExpiresAt < DateTime.UtcNow)
        {
            _logger.LogInformation("Expired short code accessed: {ShortCode}", shortCode);
            return Gone(new { error = "This short URL has expired" });
        }

        // Cache the result for future requests
        await _distributedCache.SetStringAsync($"url:{shortCode}", urlRecord.OriginalUrl,
            new DistributedCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow = TimeSpan.FromHours(6),
                SlidingExpiration = TimeSpan.FromHours(1)
            });

        // Record click event for analytics
        _ = _clickProcessor.RecordClickAsync(shortCode, Request);

        return Redirect(urlRecord.OriginalUrl);
    }
}
```

Azure API Management Integration

Azure API Management sits in front of our App Service and handles cross-cutting concerns that would otherwise complicate our application code. Think of API Management as a sophisticated gateway that enforces policies, provides analytics, and manages API lifecycle without touching our core business logic.

Rate limiting becomes particularly important at our scale. Without proper controls, a single misbehaving client could overwhelm our system. API Management implements multiple layers of rate limiting using built-in policies that require no custom code.

```xml
```

```
<!-- Azure API Management policies for rate limiting and security -->
<!-- These policies execute before requests reach our application -->
<policies>
  <inbound>
    <!-- Basic rate limiting by IP address -->
    <rate-limit-by-key
      calls="1000"
      renewal-period="3600"
      counter-key="@(context.Request.IpAddress)"
      increment-condition="@(context.Response.StatusCode >= 200 && context.Response.StatusCode < 300)" />

    <!-- Enhanced rate limiting for authenticated users -->
    <rate-limit-by-key
      calls="10000"
      renewal-period="3600"
      counter-key="@(context.Request.Headers.GetValueOrDefault('X-API-Key','anonymous'))" />

    <!-- Block known bad actors -->
    <ip-filter action="forbid">
      <address-range from="192.168.1.1" to="192.168.1.10" />
    </ip-filter>

    <!-- Validate request format -->
    <validate-jwt header-name="Authorization" failed-validation-httpcode="401">
      <openid-config url="https://login.microsoftonline.com/common/.well-known/openid_configuration" />
    </validate-jwt>
  </inbound>

  <backend>
    <!-- Forward to our App Service -->
    <forward-request />
  </backend>

  <outbound>
    <!-- Add security headers -->
    <set-header name="X-Content-Type-Options" exists-action="override">
      <value>nosniff</value>
    </set-header>
    <set-header name="X-Frame-Options" exists-action="override">
      <value>DENY</value>
    </set-header>
  </outbound>

  <on-error>
    <!-- Custom error handling -->
    <set-status code="429" reason="Rate limit exceeded" />
```

```
        <set-body>@{
            return new JObject(
                new JProperty("error", "Rate limit exceeded"),
                new JProperty("retryAfter", context.Variables["retryAfter"])
            ).ToString();
        }</set-body>
    </on-error>
</policies>
```

Database Architecture

Azure SQL Database Design

For our database requirements, Azure SQL Database provides the optimal balance of familiar SQL capabilities with cloud-native features. Unlike managing traditional SQL Server installations, Azure SQL Database automatically handles patching, backups, high availability, and intelligent performance tuning.

The choice of SQL over NoSQL databases might surprise some, but it's based on our specific requirements. We need ACID transactions for URL generation to prevent duplicate short codes, complex queries for analytics, and the ability to enforce referential integrity. Azure SQL Database delivers these capabilities while scaling to handle our projected load.

```sql
```

```
-- Azure SQL Database schema optimized for URL shortening workload
-- Includes cloud-specific optimizations and performance features

-- Main table storing URL mappings
CREATE TABLE urls (
    -- Identity column provides unique, sequential IDs for base62 encoding
    id BIGINT IDENTITY(1,1) PRIMARY KEY,

    -- Computed column automatically generates short codes from IDs
    -- PERSISTED means Azure stores this value for fast retrieval
    short_code AS (dbo.base62_encode(id)) PERSISTED,

    -- Original URL with reasonable length limit
    original_url NVARCHAR(2000) NOT NULL,

    -- UTC timestamps for consistency across regions
    created_at DATETIME2(3) DEFAULT SYSUTCDATETIME(),
    expires_at DATETIME2(3) NULL,

    -- User tracking for analytics and rate limiting
    created_by_user_id BIGINT NULL,
    created_by_ip_address NVARCHAR(45) NULL, -- Supports IPv6

    -- Soft delete for maintaining redirect functionality
    is_active BIT DEFAULT 1,

    -- Optional custom domain support
    custom_domain NVARCHAR(255) NULL,

    -- Performance optimization: clustered index on ID (default)
    -- Non-clustered indexes for common query patterns
    INDEX ix_short_code NONCLUSTERED (short_code)
        INCLUDE (original_url, is_active, expires_at),
    INDEX ix_user_created_at NONCLUSTERED (created_by_user_id, created_at),
    INDEX ix_expires_at NONCLUSTERED (expires_at)
        WHERE expires_at IS NOT NULL AND is_active = 1,

    -- Unique constraint ensures no duplicate short codes
    CONSTRAINT uq_short_code UNIQUE (short_code)

) WITH (DATA_COMPRESSION = PAGE); -- Azure SQL compression saves storage costs

-- Function to convert numeric IDs to base62 short codes
-- This approach guarantees uniqueness and predictable length
CREATE FUNCTION dbo.base62_encode(@num BIGINT)
RETURNS VARCHAR(10)
```

```
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @chars VARCHAR(62) = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
    DECLARE @result VARCHAR(10) = ''

    -- Handle edge case
    IF @num = 0 RETURN 'a'

    -- Convert to base62 representation
    WHILE @num > 0
    BEGIN
        SET @result = SUBSTRING(@chars, (@num % 62) + 1, 1) + @result
        SET @num = @num / 62
    END

    RETURN @result
END;

-- Separate table for user management and rate limiting
CREATE TABLE users (
    id BIGINT IDENTITY(1,1) PRIMARY KEY,
    email NVARCHAR(256) NOT NULL,
    api_key UNIQUEIDENTIFIER DEFAULT NEWID(),
    created_at DATETIME2(3) DEFAULT SYSUTCDATETIME(),
    is_active BIT DEFAULT 1,
    daily_url_limit INT DEFAULT 1000,

    INDEX ix_api_key NONCLUSTERED (api_key) INCLUDE (is_active, daily_url_limit),
    CONSTRAINT uq_email UNIQUE (email),
    CONSTRAINT uq_api_key UNIQUE (api_key)
);

-- View for active URLs with computed short URLs
CREATE VIEW active_urls
WITH SCHEMABINDING
AS
SELECT
    id,
    short_code,
    original_url,
    created_at,
    expires_at,
    created_by_user_id
FROM dbo.urls
WHERE is_active = 1
    AND (expires_at IS NULL OR expires_at > SYSUTCDATETIME());
```

```
-- Index on the view for optimal query performance
CREATE UNIQUE CLUSTERED INDEX ix_active_urls_short_code
ON active_urls (short_code);
```

Database Scaling Strategy

Azure SQL Database offers multiple scaling approaches that align perfectly with our growth trajectory. We'll start with the General Purpose tier, which provides excellent price-performance for most workloads, then migrate to Hyperscale as our data volume approaches the General Purpose limits.

The Hyperscale tier deserves special attention because it fundamentally changes how we think about database scaling. Unlike traditional databases that scale everything together, Hyperscale separates compute from storage and enables independent scaling of each component. Our 11TB annual growth becomes manageable because storage scales automatically while we adjust compute resources based on actual performance needs.

For our read-heavy workload, Azure SQL Database's read scale-out feature provides up to 4 read-only replicas. These replicas handle redirect requests while the primary database processes URL creation. This separation ensures that high redirect volume never impacts URL generation performance.

csharp

```
// Data access layer implementing read/write separation
// Leverages Azure SQL Database read replicas for optimal performance
public class UrlRepository : IUrlRepository
{
    private readonly string _writeConnectionString; // Points to primary database
    private readonly string _readConnectionString;  // Points to read replica
    private readonly ILogger<UrlRepository> _logger;

    public UrlRepository(IConfiguration configuration, ILogger<UrlRepository> logger)
    {
        // Connection strings configured to use appropriate endpoints
        _writeConnectionString = configuration.GetConnectionString("SqlDatabase");
        _readConnectionString = configuration.GetConnectionString("SqlDatabaseReadOnly");
        _logger = logger;
    }

    /// <summary>
    /// Creates a new URL mapping using the primary database
    /// Ensures strong consistency for immediate availability
    /// </summary>
    public async Task<string> CreateUrlAsync(string originalUrl, string userId = null)
    {
        using var connection = new SqlConnection(_writeConnectionString);
        await connection.OpenAsync();

        // Use transaction to ensure atomic URL creation
        using var transaction = connection.BeginTransaction();

        try
        {
            // Insert new URL record - ID auto-generates
            var insertCommand = new SqlCommand(@"
                INSERT INTO urls (original_url, created_by_user_id, created_by_ip_address)
                OUTPUT INSERTED.short_code
                VALUES (@originalUrl, @userId, @ipAddress)",
                connection, transaction);

            insertCommand.Parameters.AddWithValue("@originalUrl", originalUrl);
            insertCommand.Parameters.AddWithValue("@userId", (object)userId ?? DBNull.Value);
            insertCommand.Parameters.AddWithValue("@ipAddress", GetClientIpAddress());

            var shortCode = (string)await insertCommand.ExecuteScalarAsync();

            await transaction.CommitAsync();

            _logger.LogInformation("Created URL mapping: {ShortCode} -> {OriginalUrl}",
```

```
            shortCode, originalUrl);

        return shortCode;
    }
    catch (SqlException ex) when (ex.Number == 2627) // Unique constraint violation
    {
        await transaction.RollbackAsync();
        _logger.LogWarning("Duplicate short code generated, retrying...");
        throw new DuplicateShortCodeException("Generated short code already exists");
    }
    catch (Exception)
    {
        await transaction.RollbackAsync();
        throw;
    }
}

/// <summary>
/// Retrieves URL mapping using read replica
/// Optimized for high-throughput redirect operations
/// </summary>
public async Task<UrlRecord> GetUrlAsync(string shortCode)
{
    using var connection = new SqlConnection(_readConnectionString);

    var command = new SqlCommand(@"
        SELECT
            original_url,
            expires_at,
            is_active
        FROM active_urls
        WHERE short_code = @shortCode", connection);

    command.Parameters.AddWithValue("@shortCode", shortCode);

    await connection.OpenAsync();

    using var reader = await command.ExecuteReaderAsync();

    if (await reader.ReadAsync())
    {
        return new UrlRecord
        {
            OriginalUrl = reader.GetString("original_url"),
            ExpiresAt = reader.IsDBNull("expires_at") ? null : reader.GetDateTime("expires_at"),
            IsActive = reader.GetBoolean("is_active")
        };
```

```
        }

        return null;
    }
}
```

Caching Architecture

Azure Cache for Redis Implementation

Azure Cache for Redis forms the backbone of our performance optimization strategy. Given our read-heavy workload and the power law distribution of URL popularity, intelligent caching can reduce database load by 90% or more while dramatically improving response times.

Our caching strategy implements multiple layers with different characteristics. Local application caches provide microsecond response times for the most popular URLs, Azure Cache for Redis offers distributed caching across all application instances, and Azure Front Door caches complete redirect responses at edge locations worldwide.

csharp

```
// Multi-tiered caching implementation using Azure Cache for Redis
// Implements cache-aside pattern with automatic failover
public class DistributedUrlCache : IUrlCache
{
    private readonly IDatabase _primaryRedisCache;   // Primary region cache
    private readonly IDatabase _secondaryRedisCache;  // Backup region cache
    private readonly IMemoryCache _localCache;       // In-process cache
    private readonly ILogger<DistributedUrlCache> _logger;
    private readonly UrlCacheOptions _options;

    public DistributedUrlCache(
        IConnectionMultiplexer primaryRedis,
        IConnectionMultiplexer secondaryRedis,
        IMemoryCache localCache,
        IOptions<UrlCacheOptions> options,
        ILogger<DistributedUrlCache> logger)
    {
        _primaryRedisCache = primaryRedis.GetDatabase();
        _secondaryRedisCache = secondaryRedis.GetDatabase();
        _localCache = localCache;
        _options = options.Value;
        _logger = logger;
    }

    /// <summary>
    /// Retrieves URL from cache using fallback hierarchy
    /// Implements cache warming to optimize for future requests
    /// </summary>
    public async Task<string> GetUrlAsync(string shortCode)
    {
        // Level 1: Check local in-memory cache (fastest)
        if (_localCache.TryGetValue($"url:{shortCode}", out string cachedUrl))
        {
            _logger.LogDebug("Cache hit (local): {ShortCode}", shortCode);
            return cachedUrl;
        }

        try
        {
            // Level 2: Check primary Redis cache (distributed)
            var redisResult = await _primaryRedisCache.StringGetAsync($"url:{shortCode}");
            if (redisResult.HasValue)
            {
                _logger.LogDebug("Cache hit (Redis primary): {ShortCode}", shortCode);

                // Warm local cache for subsequent requests
```

```
            _localCache.Set($"url:{shortCode}", redisResult.ToString(),
                TimeSpan.FromMinutes(_options.LocalCacheTtlMinutes));

            return redisResult;
        }

        // Level 3: Check secondary Redis cache (failover region)
        redisResult = await _secondaryRedisCache.StringGetAsync($"url:{shortCode}");
        if (redisResult.HasValue)
        {
            _logger.LogDebug("Cache hit (Redis secondary): {ShortCode}", shortCode);

            // Warm both primary cache and local cache
            _ = _primaryRedisCache.StringSetAsync($"url:{shortCode}", redisResult,
                TimeSpan.FromHours(_options.RedisCacheTtlHours));

            _localCache.Set($"url:{shortCode}", redisResult.ToString(),
                TimeSpan.FromMinutes(_options.LocalCacheTtlMinutes));

            return redisResult;
        }

        _logger.LogDebug("Cache miss: {ShortCode}", shortCode);
        return null;
    }
    catch (RedisException ex)
    {
        _logger.LogWarning(ex, "Redis cache error for {ShortCode}, falling back to database",
            shortCode);
        return null; // Graceful degradation to database lookup
    }
}

/// <summary>
/// Stores URL in all cache layers
/// Uses fire-and-forget pattern to avoid blocking the response
/// </summary>
public async Task SetUrlAsync(string shortCode, string originalUrl)
{
    // Set local cache immediately
    _localCache.Set($"url:{shortCode}", originalUrl,
        TimeSpan.FromMinutes(_options.LocalCacheTtlMinutes));

    // Set distributed caches asynchronously (don't block response)
    var redisTasks = new[]
    {
        _primaryRedisCache.StringSetAsync($"url:{shortCode}", originalUrl,
```

```
                TimeSpan.FromHours(_options.RedisCacheTtlHours)),
            _secondaryRedisCache.StringSetAsync($"url:{shortCode}", originalUrl,
                TimeSpan.FromHours(_options.RedisCacheTtlHours))
        };

        // Fire and forget - don't await these operations
        _ = Task.WhenAll(redisTasks).ContinueWith(task =>
        {
            if (task.IsFaulted)
            {
                _logger.LogWarning(task.Exception,
                    "Failed to update Redis cache for {ShortCode}", shortCode);
            }
        });
    }

    /// <summary>
    /// Invalidates URL from all cache layers
    /// Used when URLs are disabled or expire
    /// </summary>
    public async Task InvalidateUrlAsync(string shortCode)
    {
        // Remove from local cache immediately
        _localCache.Remove($"url:{shortCode}");

        // Remove from distributed caches
        var deleteTasks = new[]
        {
            _primaryRedisCache.KeyDeleteAsync($"url:{shortCode}"),
            _secondaryRedisCache.KeyDeleteAsync($"url:{shortCode}")
        };

        await Task.WhenAll(deleteTasks);

        _logger.LogInformation("Invalidated cache for {ShortCode}", shortCode);
    }
}

// Configuration options for cache behavior
public class UrlCacheOptions
{
    public int LocalCacheTtlMinutes { get; set; } = 5;
    public int RedisCacheTtlHours { get; set; } = 6;
    public int MaxLocalCacheSize { get; set; } = 10000;
    public bool EnableCacheWarming { get; set; } = true;
}
```

Cache Warming and Invalidation

One of the most sophisticated aspects of our caching strategy involves proactive cache warming based on traffic patterns. When Azure Monitor detects that a particular URL is receiving increased traffic, we can preemptively warm caches across all regions before the traffic spike hits.

This approach prevents the "thundering herd" problem where millions of users simultaneously request the same uncached URL, overwhelming our database. Instead, we detect trending URLs and ensure they're cached globally before they go viral.

Global Distribution Strategy

Azure Front Door Configuration

Azure Front Door represents the crown jewel of our global distribution strategy. More than just a content delivery network, Front Door acts as an intelligent global load balancer that routes traffic based on real-time health monitoring, latency measurements, and custom routing rules.

The key insight about Front Door is that it operates at the DNS and HTTP level simultaneously. DNS routing gets users to the optimal Azure region, while HTTP-level routing can redirect traffic instantly if a region becomes unhealthy. This dual-layer approach provides both performance and reliability benefits.

```json
```

```
{
  "frontDoorProfile": {
    "name": "url-shortener-global",
    "resourceGroup": "url-shortener-rg",
    "location": "Global",
    "sku": "Premium_AzureFrontDoor",
    "configuration": {
      "customDomains": [
        {
          "hostName": "short.ly",
          "certificateSource": "FrontDoor",
          "minimumTlsVersion": "1.2"
        }
      ],
      "originGroups": [
        {
          "name": "url-shortener-origins",
          "loadBalancingSettings": {
            "sampleSize": 4,
            "successfulSamplesRequired": 3,
            "additionalLatencyInMilliseconds": 50
          },
          "healthProbeSettings": {
            "probePath": "/health",
            "probeMethod": "GET",
            "probeProtocol": "Https",
            "intervalInSeconds": 30
          },
          "origins": [
            {
              "name": "eastus-origin",
              "hostName": "shortener-eastus.azurewebsites.net",
              "httpPort": 80,
              "httpsPort": 443,
              "originHostHeader": "shortener-eastus.azurewebsites.net",
              "priority": 1,
              "weight": 1000,
              "enabledState": "Enabled"
            },
            {
              "name": "westeurope-origin",
              "hostName": "shortener-westeurope.azurewebsites.net",
              "httpPort": 80,
              "httpsPort": 443,
              "originHostHeader": "shortener-westeurope.azurewebsites.net",
              "priority": 1,
```

```
        "weight": 1000,
        "enabledState": "Enabled"
      },
      {
        "name": "southeastasia-origin",
        "hostName": "shortener-southeastasia.azurewebsites.net",
        "httpPort": 80,
        "httpsPort": 443,
        "originHostHeader": "shortener-southeastasia.azurewebsites.net",
        "priority": 2,
        "weight": 500,
        "enabledState": "Enabled"
      }
    ]
  }
],
"routes": [
  {
    "name": "redirect-route",
    "customDomains": ["short.ly"],
    "originGroup": "url-shortener-origins",
    "supportedProtocols": ["Http", "Https"],
    "patternsToMatch": ["/*"],
    "forwardingProtocol": "HttpsOnly",
    "linkToDefaultDomain": "Enabled",
    "httpsRedirect": "Enabled",
    "cacheConfiguration": {
      "queryStringCachingBehavior": "IgnoreQueryString",
      "compressionSettings": {
        "contentTypesToCompress": [
          "application/json",
          "text/html",
          "text/plain"
        ],
        "isCompressionEnabled": true
      },
      "cacheDuration": "1.00:00:00"
    }
  }
],
"securityPolicies": [
  {
    "name": "security-policy",
    "wafPolicy": "/subscriptions/{subscription}/resourceGroups/url-shortener-rg/providers/Microsoft.Network/frontD
  }
]
}
```

```
    }
}
```

Regional Failover Strategy

Our multi-region deployment goes beyond simple load distribution. Each region operates independently with local database read replicas and Redis caches, ensuring that regional failures don't cascade globally.

The failover logic prioritizes user experience over perfect consistency. If the East US region fails, European users automatically route to West Europe without noticing any disruption. The only visible impact might be slightly increased latency for US East Coast users who temporarily route to alternative regions.

Azure Traffic Manager complements Front Door by providing DNS-level failover. While Front Door handles HTTP routing and caching, Traffic Manager ensures that DNS queries resolve to healthy regions even during major outages.

Analytics and Event Processing

Azure Event Hubs Integration

Our analytics architecture must handle the massive scale of click events while providing real-time insights and historical reporting. Azure Event Hubs provides the perfect foundation because it can ingest millions of events per second while maintaining order and durability guarantees.

The key architectural decision involves separating click tracking from redirect functionality. Redirect responses must be fast and reliable, so we never block them waiting for analytics processing. Instead, we publish click events asynchronously and let Event Hubs handle delivery guarantees.

```csharp
```

```
// Click event processing system using Azure Event Hubs
// Separates analytics from core redirect functionality for optimal performance
public class ClickEventProcessor : IClickEventProcessor
{
    private readonly EventHubProducerClient _eventHubClient;
    private readonly ILogger<ClickEventProcessor> _logger;
    private readonly IMemoryCache _ipGeoCache;
    private readonly ClickEventOptions _options;

    public ClickEventProcessor(
        EventHubProducerClient eventHubClient,
        ILogger<ClickEventProcessor> logger,
        IMemoryCache ipGeoCache,
        IOptions<ClickEventOptions> options)
    {
        _eventHubClient = eventHubClient;
        _logger = logger;
        _ipGeoCache = ipGeoCache;
        _options = options.Value;
    }

    /// <summary>
    /// Records click event asynchronously without blocking redirect response
    /// Enriches event data with geographic and device information
    /// </summary>
    public async Task RecordClickAsync(string shortCode, HttpRequest request)
    {
        try
        {
            // Extract client information from request headers
            var clientIp = GetClientIpAddress(request);
            var userAgent = request.Headers.UserAgent.ToString();
            var referer = request.Headers.Referer?.ToString();

            // Enrich with geographic data (cached to avoid API limits)
            var geoData = await GetGeoDataAsync(clientIp);

            // Create structured click event
            var clickEvent = new ClickEvent
            {
                EventId = Guid.NewGuid(),
                ShortCode = shortCode,
                Timestamp = DateTimeOffset.UtcNow,

                // Client information
                IpAddress = clientIp,
```

```
            UserAgent = userAgent,
            Referer = referer,

            // Geographic data
            Country = geoData?.Country,
            Region = geoData?.Region,
            City = geoData?.City,
            Latitude = geoData?.Latitude,
            Longitude = geoData?.Longitude,

            // Device information parsed from User-Agent
            DeviceType = ParseDeviceType(userAgent),
            Browser = ParseBrowser(userAgent),
            OperatingSystem = ParseOperatingSystem(userAgent),

            // Request metadata
            AcceptLanguage = request.Headers.AcceptLanguage.ToString(),
            RequestSize = request.ContentLength ?? 0
        };

        // Serialize event data
        var eventData = new EventData(JsonSerializer.SerializeToUtf8Bytes(clickEvent));

        // Use short code as partition key for ordered processing per URL
        eventData.PartitionKey = shortCode;

        // Add custom properties for downstream filtering
        eventData.Properties.Add("EventType", "UrlClick");
        eventData.Properties.Add("Version", "1.0");

        // Send to Event Hubs (fire and forget for performance)
        await _eventHubClient.SendAsync(new[] { eventData });

        _logger.LogDebug("Recorded click event for {ShortCode} from {IpAddress}",
            shortCode, clientIp);
    }
    catch (Exception ex)
    {
        // Never let analytics failures impact redirect functionality
        _logger.LogWarning(ex,
            "Failed to record click event for {ShortCode}, continuing...", shortCode);
    }
}

/// <summary>
/// Extracts client IP address handling Azure Front Door forwarded headers
/// </summary>
```

```
    private string GetClientIpAddress(HttpRequest request)
    {
        // Azure Front Door adds X-Forwarded-For header with client IP
        var forwardedFor = request.Headers["X-Forwarded-For"].FirstOrDefault();
        if (!string.IsNullOrEmpty(forwardedFor))
        {
            // Take first IP in case of multiple proxies
            return forwardedFor.Split(',')[0].Trim();
        }

        // Fallback to direct connection IP
        return request.HttpContext.Connection.RemoteIpAddress?.ToString() ?? "unknown";
    }

    /// <summary>
    /// Retrieves geographic data for IP address with caching
    /// </summary>
    private async Task<GeoData> GetGeoDataAsync(string ipAddress)
    {
        // Check cache first to avoid repeated API calls
        var cacheKey = $"geo:{ipAddress}";
        if (_ipGeoCache.TryGetValue(cacheKey, out GeoData cachedGeoData))
        {
            return cachedGeoData;
        }

        try
        {
            // Call geolocation service (implement with your preferred provider)
            var geoData = await _geoLocationService.GetGeoDataAsync(ipAddress);

            // Cache for future requests (cache for 1 hour)
            _ipGeoCache.Set(cacheKey, geoData, TimeSpan.FromHours(1));

            return geoData;
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Failed to get geo data for IP {IpAddress}", ipAddress);
            return null;
        }
    }
}

// Structured click event data model
public class ClickEvent
{
```

```
    public Guid EventId { get; set; }
    public string ShortCode { get; set; }
    public DateTimeOffset Timestamp { get; set; }
    public string IpAddress { get; set; }
    public string UserAgent { get; set; }
    public string Referer { get; set; }
    public string Country { get; set; }
    public string Region { get; set; }
    public string City { get; set; }
    public double? Latitude { get; set; }
    public double? Longitude { get; set; }
    public string DeviceType { get; set; }
    public string Browser { get; set; }
    public string OperatingSystem { get; set; }
    public string AcceptLanguage { get; set; }
    public long RequestSize { get; set; }
}
```

Stream Analytics Processing

Azure Stream Analytics consumes our click events from Event Hubs and performs real-time aggregations while routing data to multiple destinations. This approach enables both immediate insights and long-term analytics without building complex data pipelines ourselves.

Stream Analytics queries use SQL-like syntax, making them accessible to business analysts while providing the performance characteristics needed for real-time processing. The service automatically scales based on event volume and provides exactly-once processing guarantees.

```sql
```

```
-- Azure Stream Analytics query for real-time click aggregation
-- Processes millions of events per second with automatic scaling

-- Real-time click counts by short code (1-minute tumbling windows)
WITH ClickCounts AS (
    SELECT
        ShortCode,
        COUNT(*) as ClickCount,
        System.Timestamp() as WindowEnd
    FROM ClickEvents TIMESTAMP BY Timestamp
    GROUP BY ShortCode, TumblingWindow(minute, 1)
)

-- Geographic distribution (5-minute windows)
, GeoDistribution AS (
    SELECT
        ShortCode,
        Country,
        COUNT(*) as Clicks,
        System.Timestamp() as WindowEnd
    FROM ClickEvents TIMESTAMP BY Timestamp
    WHERE Country IS NOT NULL
    GROUP BY ShortCode, Country, TumblingWindow(minute, 5)
)

-- Device type analysis (hourly windows)
, DeviceAnalysis AS (
    SELECT
        ShortCode,
        DeviceType,
        Browser,
        COUNT(*) as Clicks,
        System.Timestamp() as WindowEnd
    FROM ClickEvents TIMESTAMP BY Timestamp
    WHERE DeviceType IS NOT NULL
    GROUP BY ShortCode, DeviceType, Browser, TumblingWindow(hour, 1)
)

-- Hot URLs detection (URLs with >1000 clicks in 5 minutes)
, HotUrls AS (
    SELECT
        ShortCode,
        COUNT(*) as RecentClicks,
        System.Timestamp() as DetectedAt
    FROM ClickEvents TIMESTAMP BY Timestamp
    GROUP BY ShortCode, TumblingWindow(minute, 5)
```

```
    HAVING COUNT(*) > 1000
)

-- Output to Azure Cosmos DB for real-time dashboards
SELECT
    ShortCode,
    ClickCount,
    WindowEnd,
    'click_count' as RecordType
INTO CosmosDBOutput
FROM ClickCounts

-- Output to Azure Synapse for historical analysis
UNION ALL
SELECT
    ShortCode + '|' + Country as PartitionKey,
    CAST(Clicks as bigint) as ClickCount,
    WindowEnd,
    'geo_distribution' as RecordType
FROM GeoDistribution

-- Output to Event Grid for alerting on hot URLs
SELECT
    ShortCode,
    RecentClicks,
    DetectedAt
INTO EventGridOutput
FROM HotUrls;
```

Security Architecture

Azure Key Vault Integration

Security at our scale requires systematic approaches rather than ad-hoc implementations. Azure Key Vault centralizes all secrets management while providing audit logging and access controls that integrate seamlessly with Azure Active Directory.

The key insight about Key Vault is that it eliminates secrets from our application code entirely. Connection strings, API keys, and certificates are retrieved dynamically at runtime, which means our source code never contains sensitive information and our deployment pipelines remain secure.

```
csharp
```

```
// Secure configuration management using Azure Key Vault
// Eliminates hard-coded secrets and provides centralized secret rotation
public class SecureConfigurationService : IConfigurationService
{
    private readonly SecretClient _keyVaultClient;
    private readonly ILogger<SecureConfigurationService> _logger;
    private readonly IMemoryCache _configCache;

    public SecureConfigurationService(
        SecretClient keyVaultClient,
        ILogger<SecureConfigurationService> logger,
        IMemoryCache configCache)
    {
        _keyVaultClient = keyVaultClient;
        _logger = logger;
        _configCache = configCache;
    }

    /// <summary>
    /// Retrieves database connection string from Key Vault with caching
    /// Supports automatic secret rotation without application restarts
    /// </summary>
    public async Task<string> GetDatabaseConnectionStringAsync()
    {
        return await GetSecretWithCacheAsync("sql-connection-string");
    }

    /// <summary>
    /// Retrieves Redis connection string for distributed caching
    /// </summary>
    public async Task<string> GetRedisConnectionStringAsync()
    {
        return await GetSecretWithCacheAsync("redis-connection-string");
    }

    /// <summary>
    /// Retrieves Event Hubs connection string for analytics
    /// </summary>
    public async Task<string> GetEventHubConnectionStringAsync()
    {
        return await GetSecretWithCacheAsync("eventhub-connection-string");
    }

    /// <summary>
    /// Retrieves third-party API keys (geolocation, security scanning, etc.)
    /// </summary>
```

```
    public async Task<string> GetApiKeyAsync(string serviceName)
    {
        return await GetSecretWithCacheAsync($"{serviceName}-api-key");
    }

    /// <summary>
    /// Generic secret retrieval with local caching for performance
    /// Cache TTL is intentionally short to support secret rotation
    /// </summary>
    private async Task<string> GetSecretWithCacheAsync(string secretName)
    {
        var cacheKey = $"secret:{secretName}";

        // Check cache first (short TTL for security)
        if (_configCache.TryGetValue(cacheKey, out string cachedSecret))
        {
            return cachedSecret;
        }

        try
        {
            // Retrieve from Key Vault
            var secret = await _keyVaultClient.GetSecretAsync(secretName);
            var secretValue = secret.Value.Value;

            // Cache for 5 minutes to balance performance and security
            _configCache.Set(cacheKey, secretValue, TimeSpan.FromMinutes(5));

            _logger.LogDebug("Retrieved secret {SecretName} from Key Vault", secretName);
            return secretValue;
        }
        catch (RequestFailedException ex) when (ex.Status == 404)
        {
            _logger.LogError("Secret {SecretName} not found in Key Vault", secretName);
            throw new ConfigurationException($"Required secret {secretName} not found");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to retrieve secret {SecretName}", secretName);
            throw;
        }
    }
}

// Startup configuration for Azure Key Vault integration
public class Startup
{
```

```
    public void ConfigureServices(IServiceCollection services)
    {
        // Configure Azure Key Vault client with managed identity
        services.AddSingleton<SecretClient>(serviceProvider =>
        {
            var configuration = serviceProvider.GetRequiredService<IConfiguration>();
            var keyVaultUri = configuration["KeyVault:Uri"];

            // Use managed identity for authentication (no credentials in code)
            var credential = new DefaultAzureCredential();

            return new SecretClient(new Uri(keyVaultUri), credential);
        });

        // Register configuration service
        services.AddSingleton<IConfigurationService, SecureConfigurationService>();

        // Configure database context with Key Vault connection string
        services.AddDbContext<UrlShortenerContext>((serviceProvider, options) =>
        {
            var configService = serviceProvider.GetRequiredService<IConfigurationService>();
            var connectionString = configService.GetDatabaseConnectionStringAsync().Result;
            options.UseSqlServer(connectionString);
        });
    }
}
```

Web Application Firewall Configuration

Azure Web Application Firewall, integrated with Front Door, provides our first line of defense against web-based attacks. WAF rules automatically block common attack patterns like SQL injection, cross-site scripting, and bot traffic while allowing legitimate requests through without latency impact.

The beauty of Azure WAF lies in its integration with Microsoft's threat intelligence network. Rules are automatically updated based on global attack patterns, which means our application benefits from protection against zero-day attacks without manual intervention.

```
json
```

```
{
  "webApplicationFirewallPolicy": {
    "name": "url-shortener-waf",
    "resourceGroup": "url-shortener-rg",
    "location": "Global",
    "policySettings": {
      "enabledState": "Enabled",
      "mode": "Prevention",
      "requestBodyCheck": true,
      "maxRequestBodySizeInKb": 128,
      "fileUploadLimitInMb": 10
    },
    "managedRules": {
      "managedRuleSets": [
        {
          "ruleSetType": "Microsoft_DefaultRuleSet",
          "ruleSetVersion": "2.0",
          "ruleGroupOverrides": [
            {
              "ruleGroupName": "REQUEST-920-PROTOCOL-ENFORCEMENT",
              "rules": [
                {
                  "ruleId": "920300",
                  "enabledState": "Disabled",
                  "action": "Block"
                }
              ]
            }
          ]
        },
        {
          "ruleSetType": "Microsoft_BotManagerRuleSet",
          "ruleSetVersion": "1.0"
        }
      ]
    },
    "customRules": [
      {
        "name": "RateLimitRule",
        "priority": 1,
        "enabledState": "Enabled",
        "ruleType": "RateLimitRule",
        "rateLimitDurationInMinutes": 1,
        "rateLimitThreshold": 100,
        "matchConditions": [
          {
```

```
            "matchVariable": "RemoteAddr",
            "operator": "IPMatch",
            "matchValue": ["0.0.0.0/0"]
          }
        ],
        "action": "Block"
      },
      {
        "name": "GeoBlockRule",
        "priority": 2,
        "enabledState": "Enabled",
        "ruleType": "MatchRule",
        "matchConditions": [
          {
            "matchVariable": "RemoteAddr",
            "operator": "GeoMatch",
            "matchValue": ["CN", "RU", "KP"]
          }
        ],
        "action": "Block"
      }
    ]
  }
}
```

Monitoring and Observability

Azure Monitor Integration

Comprehensive monitoring becomes critical at our scale because small problems can cascade into major outages within minutes. Azure Monitor provides a unified observability platform that correlates metrics, logs, and traces across our entire application stack.

Our monitoring strategy focuses on both technical metrics (response times, error rates, resource utilization) and business metrics (URLs created, clicks processed, popular domains). This dual approach ensures we detect problems before they impact users while providing insights that drive business decisions.

csharp

```
// Comprehensive monitoring and observability implementation
// Integrates with Azure Monitor for unified insights across the application stack
public class ApplicationTelemetry
{
    private readonly TelemetryClient _telemetryClient;
    private readonly ILogger<ApplicationTelemetry> _logger;
    private readonly DiagnosticSource _diagnosticSource;

    public ApplicationTelemetry(
        TelemetryClient telemetryClient,
        ILogger<ApplicationTelemetry> logger,
        DiagnosticSource diagnosticSource)
    {
        _telemetryClient = telemetryClient;
        _logger = logger;
        _diagnosticSource = diagnosticSource;
    }

    /// <summary>
    /// Tracks URL creation metrics and performance
    /// Provides insights into creation patterns and system health
    /// </summary>
    public void TrackUrlCreation(string shortCode, string originalUrl, TimeSpan duration)
    {
        // Custom metric for Azure Monitor dashboards
        _telemetryClient.TrackMetric("Urls.Created", 1, new Dictionary<string, string>
        {
            ["ShortCode"] = shortCode,
            ["Domain"] = ExtractDomain(originalUrl),
            ["Duration"] = duration.TotalMilliseconds.ToString()
        });

        // Performance tracking
        _telemetryClient.TrackDependency("Database", "CreateUrl",
            DateTime.UtcNow.Subtract(duration), duration, true);

        // Structured logging for detailed analysis
        _logger.LogInformation("URL created: {ShortCode} for {Domain} in {Duration}ms",
            shortCode, ExtractDomain(originalUrl), duration.TotalMilliseconds);

        // Business intelligence event
        _telemetryClient.TrackEvent("UrlCreated", new Dictionary<string, string>
        {
            ["ShortCode"] = shortCode,
            ["OriginalUrl"] = originalUrl,
            ["Timestamp"] = DateTime.UtcNow.ToString("O")
```

```
        });
    }

    /// <summary>
    /// Tracks redirect performance and cache effectiveness
    /// Critical for understanding user experience and system optimization
    /// </summary>
    public void TrackRedirect(string shortCode, bool cacheHit, TimeSpan duration)
    {
        // Cache performance metrics
        _telemetryClient.TrackMetric("Cache.HitRate", cacheHit ? 1 : 0,
            new Dictionary<string, string>
            {
                ["ShortCode"] = shortCode,
                ["CacheLevel"] = cacheHit ? "Hit" : "Miss"
            });

        // Response time tracking by cache status
        _telemetryClient.TrackMetric("Redirects.ResponseTime", duration.TotalMilliseconds,
            new Dictionary<string, string>
            {
                ["CacheStatus"] = cacheHit ? "Hit" : "Miss",
                ["ShortCode"] = shortCode
            });

        // Structured logging for troubleshooting
        _logger.LogInformation("Redirect processed: {ShortCode}, Cache: {CacheHit}, Duration: {Duration}ms",
            shortCode, cacheHit, duration.TotalMilliseconds);
    }

    /// <summary>
    /// Tracks system errors with context for rapid troubleshooting
    /// Integrates with Azure Monitor alerting for immediate notification
    /// </summary>
    public void TrackError(Exception exception, string operation, Dictionary<string, string> context = null)
    {
        // Exception tracking with full context
        _telemetryClient.TrackException(exception, context ?? new Dictionary<string, string>(),
            new Dictionary<string, double>
            {
                ["Severity"] = GetSeverityScore(exception),
                ["Timestamp"] = DateTimeOffset.UtcNow.ToUnixTimeSeconds()
            });

        // Structured error logging
        _logger.LogError(exception, "Operation {Operation} failed with context: {@Context}",
            operation, context);
```

```
        // Custom metric for error rate monitoring
        _telemetryClient.TrackMetric("Errors.Count", 1, new Dictionary<string, string>
        {
            ["Operation"] = operation,
            ["ExceptionType"] = exception.GetType().Name,
            ["ErrorCode"] = GetErrorCode(exception)
        });
    }

    /// <summary>
    /// Tracks custom business metrics for decision making
    /// Provides insights beyond technical performance
    /// </summary>
    public void TrackBusinessMetric(string metricName, double value, Dictionary<string, string> properties = null)
    {
        _telemetryClient.TrackMetric($"Business.{metricName}", value, properties);

        _logger.LogInformation("Business metric {MetricName}: {Value} with properties {@Properties}",
            metricName, value, properties);
    }

    /// <summary>
    /// Creates correlation context for distributed tracing
    /// Enables end-to-end request tracking across services
    /// </summary>
    public IDisposable StartOperation(string operationName, string shortCode = null)
    {
        var operation = _telemetryClient.StartOperation<RequestTelemetry>(operationName);

        if (!string.IsNullOrEmpty(shortCode))
        {
            operation.Telemetry.Properties["ShortCode"] = shortCode;
        }

        return operation;
    }

    private string ExtractDomain(string url)
    {
        try
        {
            return new Uri(url).Host;
        }
        catch
        {
            return "unknown";
```

```
        }
    }

    private double GetSeverityScore(Exception exception)
    {
        return exception switch
        {
            ArgumentException => 1.0,
            InvalidOperationException => 2.0,
            SqlException => 3.0,
            _ => 2.0
        };
    }

    private string GetErrorCode(Exception exception)
    {
        return exception switch
        {
            SqlException sqlEx => $"SQL{sqlEx.Number}",
            HttpRequestException httpEx => $"HTTP{httpEx.Data["StatusCode"]}",
            _ => exception.GetType().Name
        };
    }
}
```

Azure Monitor Workbooks and Alerting

Azure Monitor Workbooks provide interactive dashboards that combine metrics, logs, and traces into comprehensive views of system health. Our monitoring strategy includes multiple workbook templates that focus on different aspects of the system.

The key insight about effective monitoring is that different audiences need different views of the same data. Operations teams need real-time system health metrics, product managers need business intelligence about URL usage patterns, and security teams need threat detection dashboards.

```json
```

```
{
  "alertRules": [
    {
      "name": "High Error Rate Alert",
      "description": "Triggers when error rate exceeds 1% for 5 minutes",
      "condition": {
        "allOf": [
          {
            "metricName": "Errors.Count",
            "operator": "GreaterThan",
            "threshold": 10,
            "timeAggregation": "Total",
            "windowSize": "PT5M"
          }
        ]
      },
      "actions": [
        {
          "actionGroupId": "/subscriptions/{subscription}/resourceGroups/url-shortener-rg/providers/Microsoft.Insights/ac
          "severity": "1"
        }
      ]
    },
    {
      "name": "Cache Hit Rate Low",
      "description": "Alerts when cache hit rate drops below 85%",
      "condition": {
        "allOf": [
          {
            "metricName": "Cache.HitRate",
            "operator": "LessThan",
            "threshold": 0.85,
            "timeAggregation": "Average",
            "windowSize": "PT10M"
          }
        ]
      },
      "actions": [
        {
          "actionGroupId": "/subscriptions/{subscription}/resourceGroups/url-shortener-rg/providers/Microsoft.Insights/ac
          "severity": "2"
        }
      ]
    },
    {
      "name": "Database Connection Pool Exhaustion",
```

```
      "description": "Warns when database connections are running low",
      "condition": {
        "allOf": [
          {
            "metricName": "Database.ActiveConnections",
            "operator": "GreaterThan",
            "threshold": 80,
            "timeAggregation": "Average",
            "windowSize": "PT5M"
          }
        ]
      }
    }
  ]
}
```

Disaster Recovery and Business Continuity

Azure Site Recovery Implementation

Disaster recovery planning at our scale requires automation and regular testing. Azure Site Recovery provides continuous replication of our entire infrastructure to secondary regions, enabling rapid failover with minimal data loss.

The key principle behind our disaster recovery strategy is that recovery procedures must be tested regularly and automated completely. Manual disaster recovery procedures fail under pressure, while automated procedures can be tested monthly without impacting production operations.

Our recovery time objective (RTO) is 15 minutes, meaning we can restore service within 15 minutes of a disaster declaration. Our recovery point objective (RPO) is 5 minutes, meaning we accept at most 5 minutes of data loss during a major failure. These objectives drive our replication and backup strategies.

```json
```

```
{
  "disasterRecoveryPlan": {
    "name": "url-shortener-dr-plan",
    "primaryRegion": "East US",
    "secondaryRegion": "West US 2",
    "components": {
      "appServices": {
        "replicationMethod": "Azure Site Recovery",
        "recoveryPointRetention": "24 hours",
        "replicationFrequency": "15 minutes"
      },
      "sqlDatabase": {
        "replicationMethod": "Active Geo-Replication",
        "secondaryReplicas": [
          {
            "region": "West US 2",
            "readableSecondary": true
          },
          {
            "region": "West Europe",
            "readableSecondary": true
          }
        ]
      },
      "redisCache": {
        "replicationMethod": "Geo-Replication",
        "secondaryRegions": ["West US 2", "West Europe"]
      },
      "keyVault": {
        "replicationMethod": "Cross-Region Backup",
        "backupFrequency": "Daily"
      }
    },
    "failoverProcedures": {
      "automatic": {
        "enabled": true,
        "healthCheckEndpoint": "/health",
        "failureThreshold": 3,
        "checkIntervalSeconds": 30
      },
      "manual": {
        "approvalRequired": true,
        "approvers": ["ops-team@company.com"],
        "maxApprovalTimeMinutes": 10
      }
    },
```

```
    "testingSchedule": {
      "frequency": "Monthly",
      "testType": "Non-disruptive",
      "automatedValidation": true
    }
  }
}
```

Cost Optimization Strategy

Azure Cost Management

Operating at scale requires careful cost management to maintain profitability while providing excellent service. Azure's consumption-based pricing model aligns costs with actual usage, but requires active management to prevent unexpected expenses.

Our cost optimization strategy combines reserved capacity for predictable workloads with consumption-based pricing for variable traffic. Azure SQL Database reserved capacity can reduce database costs by 40% for our baseline load, while App Service auto-scaling handles traffic spikes cost-effectively.

csharp

```
// Cost optimization monitoring and automated scaling
// Balances performance requirements with cost efficiency
public class CostOptimizationService
{
    private readonly AzureResourceManager _resourceManager;
    private readonly ILogger<CostOptimizationService> _logger;
    private readonly CostOptimizationOptions _options;

    /// <summary>
    /// Monitors resource utilization and recommends optimizations
    /// Runs daily to identify cost-saving opportunities
    /// </summary>
    public async Task OptimizeResourcesAsync()
    {
        // Analyze database utilization and recommend tier changes
        await OptimizeDatabaseTiersAsync();

        // Review App Service scaling patterns
        await OptimizeAppServicePlansAsync();

        // Analyze storage costs and lifecycle policies
        await OptimizeStorageAsync();

        // Review cache utilization
        await OptimizeCacheResourcesAsync();
    }

    private async Task OptimizeDatabaseTiersAsync()
    {
        var databases = await _resourceManager.GetSqlDatabasesAsync();

        foreach (var database in databases)
        {
            var metrics = await GetDatabaseMetricsAsync(database.Id);

            // Recommend downsizing if utilization is consistently low
            if (metrics.AverageCpuPercent < 30 && metrics.AverageDataIoPercent < 20)
            {
                _logger.LogInformation("Database {DatabaseName} may be over-provisioned. " +
                    "CPU: {CpuPercent}%, Data IO: {DataIoPercent}%",
                    database.Name, metrics.AverageCpuPercent, metrics.AverageDataIoPercent);

                await CreateCostOptimizationRecommendationAsync(
                    "Database", database.Name, "Downsize",
                    $"Reduce tier to save estimated ${CalculateSavings(database.CurrentTier, database.RecommendedTier)} pe
            }
```

```
        }
    }

    private async Task OptimizeAppServicePlansAsync()
    {
        var appServicePlans = await _resourceManager.GetAppServicePlansAsync();

        foreach (var plan in appServicePlans)
        {
            var metrics = await GetAppServiceMetricsAsync(plan.Id);

            // Recommend scaling down if usage is consistently low
            if (metrics.AverageCpuPercent < 40 && metrics.AverageMemoryPercent < 50)
            {
                _logger.LogInformation("App Service Plan {PlanName} may be over-provisioned",
                    plan.Name);
            }
        }
    }
}
```

Conclusion

This comprehensive Azure architecture provides a robust, scalable foundation for a URL shortener service capable of handling 100 million URLs per day while maintaining excellent performance and reliability. By leveraging Azure's managed services, we achieve enterprise-grade capabilities while minimizing operational overhead.

The key architectural decisions - using Azure SQL Database for strong consistency, implementing multi-layer caching with Azure Cache for Redis, leveraging Azure Front Door for global distribution, and building comprehensive monitoring with Azure Monitor - work together to create a system that scales automatically while remaining cost-effective.

Most importantly, this architecture grows with business needs. We can start with basic functionality and add advanced features like machine learning-powered analytics, advanced security controls, and global expansion as requirements evolve. Azure's service ecosystem provides a clear upgrade path without requiring architectural rewrites.

The design principles demonstrated here - separation of concerns, defense in depth, automation over manual processes, and monitoring everything - apply beyond URL shorteners to any high-scale web application. Understanding these patterns provides a foundation for architecting complex systems that serve millions of users reliably and efficiently.