TimeLineMVC Detailed Architecture Documentation

Current Implementation

1. Storage Implementation

Current Azure Storage Implementation

```
public interface IAzureStorageService
{
    Task<string> UploadAsync(IFormFile file);
    Task DeleteAsync(string blobName);
}

public class AzureStorageService : IAzureStorageService
{
    private readonly BlobServiceClient _blobServiceClient;
    private readonly string _containerName;

    public AzureStorageService(IConfiguration configuration)
    {
        _blobServiceClient = new BlobServiceClient(configuration["AzureStorage:ConnectionStr
        _containerName = configuration["AzureStorage:ContainerName"];
    }

    public async Task<string> UploadAsync(IFormFile file)
    {
        var containerClient = _blobServiceClient.GetBlobContainerClient(_containerName);
        var blobName = $"{Guid.NewGuid()}{Path.GetExtension(file.FileName)}";
        var blobClient = containerClient.GetBlobClient(blobName);

        await using var stream = file.OpenReadStream();
        await blobClient.UploadAsync(stream, new BlobHttpHeaders
        {
            ContentType = file.ContentType
        });

        return blobClient.Uri.ToString();
    }
}
```

AWS S3 Migration Path

```
public interface IS3StorageService
{
    Task<string> UploadAsync(IFormFile file);
```

```
    Task DeleteAsync(string key);
}

public class S3StorageService : IS3StorageService
{
    private readonly IAmazonS3 _s3Client;
    private readonly string _bucketName;

    public S3StorageService(IConfiguration configuration)
    {
        var credentials = new BasicAWSCredentials(
            configuration["AWS:AccessKey"],
            configuration["AWS:SecretKey"]);

        _s3Client = new AmazonS3Client(credentials, RegionEndpoint.USEast1);
        _bucketName = configuration["AWS:BucketName"];
    }

    public async Task<string> UploadAsync(IFormFile file)
    {
        var key = $"{Guid.NewGuid()}{Path.GetExtension(file.FileName)}";

        using var stream = file.OpenReadStream();
        var request = new PutObjectRequest
        {
            BucketName = _bucketName,
            Key = key,
            InputStream = stream,
            ContentType = file.ContentType,
            CannedACL = S3CannedACL.PublicRead
        };

        await _s3Client.PutObjectAsync(request);
        return $"https://{_bucketName}.s3.amazonaws.com/{key}";
    }
}
```

2. Timeline Optimization with Redis

Current Timeline Implementation

```
public class TimelineService
{
    private readonly IPostRepository _postRepository;

    public async Task<List<PostDto>> GetTimelineAsync(string userId, int page = 1, int pageS
```

```
    {
        // Current implementation - direct database query
        return await _postRepository.GetTimelinePostsAsync(userId, page, pageSize);
    }
}
```

Enhanced Timeline with Redis

```
public class EnhancedTimelineService
{
    private readonly IPostRepository _postRepository;
    private readonly IRedisConnectionFactory _redis;
    private const string TIMELINE_KEY = "timeline:{0}";
    private const int CACHE_EXPIRY_HOURS = 24;

    public async Task<List<PostDto>> GetTimelineAsync(string userId, int page = 1, int pageS
    {
        var timelineKey = string.Format(TIMELINE_KEY, userId);
        var db = _redis.GetDatabase();

        // Try to get timeline post IDs from Redis
        var cachedPostIds = await db.ListRangeAsync(timelineKey, (page - 1) * pageSize, page

        if (cachedPostIds.Length > 0)
        {
            // Fetch posts by IDs from database
            return await _postRepository.GetPostsByIdsAsync(
                cachedPostIds.Select(id => id.ToString()).ToList());
        }

        // If cache miss, rebuild timeline
        var posts = await _postRepository.GetTimelinePostsAsync(userId, page, pageSize);

        // Store post IDs in Redis
        var transaction = db.CreateTransaction();
        await transaction.ListRightPushAsync(timelineKey,
            posts.Select(p => (RedisValue)p.Id).ToArray());
        await transaction.KeyExpireAsync(timelineKey, TimeSpan.FromHours(CACHE_EXPIRY_HOURS)
        await transaction.ExecuteAsync();

        return posts;
    }
}
```

3. Background Job Processing Improvements

Current Hangfire Implementation

```
public class ImageProcessingJob
{
    private readonly IStorageService _storageService;

    public async Task ProcessImage(int postId, string originalImageUrl)
    {
        // Basic implementation
        await ProcessAndResizeImage(originalImageUrl);
    }
}
```

Enhanced Background Processing

```
public class EnhancedImageProcessingJob
{
    private readonly IStorageService _storageService;
    private readonly IRedisCache _cache;
    private readonly ILogger<EnhancedImageProcessingJob> _logger;

    public async Task ProcessImage(int postId, string originalImageUrl)
    {
        try
        {
            // Process multiple image sizes concurrently
            var processingTasks = new[]
            {
                ProcessImageVariant(originalImageUrl, 1200, "large"),
                ProcessImageVariant(originalImageUrl, 800, "medium"),
                ProcessImageVariant(originalImageUrl, 400, "small"),
                ProcessImageVariant(originalImageUrl, 150, "thumbnail")
            };

            var results = await Task.WhenAll(processingTasks);

            // Cache image URLs
            await _cache.HashSetAsync(
                $"post:{postId}:images",
                results.ToDictionary(r => r.size, r => r.url)
            );
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error processing image for post {PostId}", postId);
```

```
            // Retry logic or dead letter queue
            throw;
        }
    }
}
```

Advanced Timeline System Design

1. Fan-out Write Optimization (Twitter/Facebook Inspired)

```
public class TimelineFanoutService
{
    private readonly IRedisConnectionFactory _redis;
    private readonly IFollowerService _followerService;

    public async Task FanoutPost(Post post)
    {
        // Get followers (with pagination for large follower counts)
        var followers = await _followerService.GetFollowerIdsAsync(post.UserId);

        // Batch followers into groups for parallel processing
        var batches = followers.Chunk(1000);

        foreach (var batch in batches)
        {
            // Process each batch in parallel
            var tasks = batch.Select(followerId =>
                AddPostToTimeline(followerId, post.Id));

            await Task.WhenAll(tasks);
        }
    }

    private async Task AddPostToTimeline(string followerId, long postId)
    {
        var db = _redis.GetDatabase();
        var timelineKey = $"timeline:{followerId}";

        // Add to Redis sorted set with timestamp as score
        await db.SortedSetAddAsync(
            timelineKey,
            postId.ToString(),
            DateTime.UtcNow.Ticks);

        // Trim timeline to prevent excessive memory usage
        await db.SortedSetRemoveRangeByRankAsync(
```

```
            timelineKey,
            0,
            -5001); // Keep last 5000 posts
    }
}
```

2. Read-time Timeline Aggregation

```
public class TimelineAggregationService
{
    private readonly IRedisConnectionFactory _redis;
    private readonly IPostRepository _postRepository;

    public async Task<List<PostDto>> GetAggregatedTimeline(
        string userId,
        int page = 1,
        int pageSize = 20)
    {
        var db = _redis.GetDatabase();

        // Get timeline entries from multiple sources
        var timelineKey = $"timeline:{userId}";
        var advertisementKey = $"ads:{userId}";
        var suggestedPostsKey = $"suggested:{userId}";

        // Merge multiple sorted sets with weights
        var mergedResults = await db.SortedSetRangeByRankAsync(
            new RedisKey[] { timelineKey, advertisementKey, suggestedPostsKey },
            new double[] { 1.0, 0.2, 0.3 }, // Weights for different content types
            Aggregate.Sum,
            (page - 1) * pageSize,
            page * pageSize - 1,
            Order.Descending);

        // Fetch full post data
        var posts = await _postRepository.GetPostsByIdsAsync(
            mergedResults.Select(r => r.ToString()).ToList());

        // Apply personalization and ranking
        return await RankAndPersonalizePosts(posts, userId);
    }
}
```

Future Scalability Enhancements

1. Distributed Cache Architecture

```
public class DistributedCacheConfig
{
    public static IServiceCollection AddDistributedCaching(
        this IServiceCollection services,
        IConfiguration configuration)
    {
        // Add Redis cluster configuration
        services.AddStackExchangeRedisCache(options =>
        {
            options.Configuration = configuration.GetConnectionString("Redis");
            options.InstanceName = "Timeline_";
        });

        // Add Redis sentinel support for high availability
        services.AddSingleton<IRedisConnectionFactory>(sp =>
        {
            var sentinelConfig = new ConfigurationOptions
            {
                CommandMap = CommandMap.Sentinel,
                EndPoints = { "sentinel1:26379", "sentinel2:26379" },
                ServiceName = "mymaster",
                AbortOnConnectFail = false
            };

            return new RedisConnectionFactory(sentinelConfig);
        });

        return services;
    }
}
```

2. Write-Through Cache Pattern

```
public class WriteThoughCacheService
{
    private readonly IRedisCache _cache;
    private readonly IPostRepository _repository;

    public async Task<bool> UpdatePost(Post post)
    {
        // Begin transaction
        using var transaction = await _repository.BeginTransactionAsync();
        try
```

```
        {
            // Update database
            await _repository.UpdatePostAsync(post);

            // Update cache atomically
            await _cache.HashSetAsync(
                $"post:{post.Id}",
                new Dictionary<string, string>
                {
                    ["content"] = post.Content,
                    ["lastModified"] = DateTime.UtcNow.ToString("O")
                });

            // Commit transaction
            await transaction.CommitAsync();
            return true;
        }
        catch
        {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```

Performance Considerations

1. Redis Memory Optimization

```
public class RedisMemoryOptimizer
{
    private readonly IRedisConnectionFactory _redis;

    public async Task OptimizeTimelineStorage()
    {
        var db = _redis.GetDatabase();

        // Set expiry for inactive timelines
        var inactiveUsers = await GetInactiveUsers();
        foreach (var userId in inactiveUsers)
        {
            await db.KeyExpireAsync(
                $"timeline:{userId}",
                TimeSpan.FromDays(30));
        }
```

```
        // Trim old entries
        var allTimelines = await GetAllTimelineKeys();
        foreach (var timeline in allTimelines)
        {
            await db.SortedSetRemoveRangeByScoreAsync(
                timeline,
                0,
                DateTime.UtcNow.AddDays(-90).Ticks);
        }
    }
}
```

2. Monitoring and Metrics

```
public class PerformanceMetricsService
{
    private readonly IMetricsClient _metrics;

    public async Task TrackTimelineMetrics(string userId, TimeSpan fetchDuration)
    {
        await _metrics.GaugeAsync(
            "timeline.fetch.duration",
            fetchDuration.TotalMilliseconds,
            new Dictionary<string, string>
            {
                ["user_id"] = userId
            });

        await _metrics.IncrementCounterAsync(
            "timeline.fetch.count",
            1,
            new Dictionary<string, string>
            {
                ["user_id"] = userId
            });
    }
}
```

Conclusion

This implementation provides a solid foundation for a social media timeline system while allowing for future scaling. The use of Redis for caching and timeline storage, combined with efficient background processing and storage abstraction, creates a robust architecture that can handle growth in user base and content volume.

The system is designed to be maintainable and extensible, with clear separation

of concerns and well-defined interfaces. The storage abstraction allows for easy migration between cloud providers, while the Redis implementation provides efficient timeline delivery inspired by proven social media platforms.

Future improvements can be implemented gradually as needed, without requiring a complete system redesign. The architecture supports both vertical and horizontal scaling, with considerations for high availability and fault tolerance built into the design.

Would you like me to expand on any particular aspect or add more implementation details for specific components?