# **Comprehensive C# Class Library for Python-C# Integration via gRPC**

I'll create a detailed C# class library for integrating with the Python semantic layer using gRPC, focusing on efficiency, scalability, and performance optimization.

## **1. Project Structure and Organization**

namespace ProgressPlay.NLP.SemanticLayer

{

// Interfaces

// Models

// Configuration

// Client Implementation

// Extensions

// Monitoring

}

## **2. Interfaces and Models**

namespace ProgressPlay.NLP.SemanticLayer.Interfaces

{

/// <summary>

/// Represents a client for the semantic layer service.

/// </summary>

public interface ISemanticLayerClient

{

/// <summary>

/// Translates NLP entities to SQL query.

/// </summary>

Task<SqlTranslationResult> TranslateToSqlAsync(string rawQuery, QueryEntities entities, CancellationToken cancellationToken = default);

/// <summary>

/// Validates a SQL query against security policies.

/// </summary>

Task<QueryValidationResult> ValidateQueryAsync(string sql, string dataset, CancellationToken cancellationToken = default);

/// <summary>

/// Gets the health status of the semantic layer service.

/// </summary>

Task<bool> IsHealthyAsync(CancellationToken cancellationToken = default);

}

/// <summary>

/// Factory for creating semantic layer clients.

/// </summary>

public interface ISemanticLayerClientFactory

{

/// <summary>

/// Creates a new instance of semantic layer client.

/// </summary>

ISemanticLayerClient CreateClient(string instanceName = "Default");

}

}

namespace ProgressPlay.NLP.SemanticLayer.Models

{

/// <summary>

/// Represents the result of translating NLP entities to SQL.

/// </summary>

public class SqlTranslationResult

{

/// <summary>

/// The generated SQL query.

/// </summary>

public string Sql { get; set; }

/// <summary>

/// The target dataset or table for the query.

/// </summary>

public string Dataset { get; set; }

/// <summary>

/// The metrics included in the query.

/// </summary>

public List<string> Metrics { get; set; } = new List<string>();

/// <summary>

/// The dimensions included in the query.

/// </summary>

public List<string> Dimensions { get; set; } = new List<string>();

/// <summary>

/// The filters applied in the query.

/// </summary>

public List<string> Filters { get; set; } = new List<string>();

/// <summary>

/// The estimated execution cost of the query (1-10).

/// </summary>

public int EstimatedExecutionCost { get; set; }

}

/// <summary>

/// Represents the result of validating a SQL query.

/// </summary>

public class QueryValidationResult

{

/// <summary>

/// Indicates whether the query is valid.

/// </summary>

public bool IsValid { get; set; }

/// <summary>

/// The reason for validation failure, if any.

/// </summary>

public string ValidationMessage { get; set; }

/// <summary>

/// The optimized SQL query, if available.

/// </summary>

public string OptimizedSql { get; set; }

}

/// <summary>

/// Exception thrown when a semantic layer operation fails.

/// </summary>

public class SemanticLayerException : Exception

{

public SemanticLayerException(string message) : base(message) { }

public SemanticLayerException(string message, Exception innerException)

: base(message, innerException) { }

}

}

## **3. Configuration Classes**

namespace ProgressPlay.NLP.SemanticLayer.Configuration

{

/// <summary>

/// Configuration options for the semantic layer client.

/// </summary>

public class SemanticLayerClientOptions

{

/// <summary>

/// The address of the semantic layer gRPC service.

/// </summary>

public string ServiceAddress { get; set; } = "http://localhost:50051";

/// <summary>

/// The timeout for gRPC calls in seconds.

/// </summary>

public int TimeoutSeconds { get; set; } = 30;

/// <summary>

/// The maximum retry attempts for failed gRPC calls.

/// </summary>

public int MaxRetryAttempts { get; set; } = 3;

/// <summary>

/// The initial backoff delay for retries in milliseconds.

/// </summary>

public int InitialBackoffMs { get; set; } = 100;

/// <summary>

/// Indicates whether to use secure connection.

/// </summary>

public bool UseSecureConnection { get; set; } = false;

/// <summary>

/// The path to client certificate file for mutual TLS.

/// </summary>

public string ClientCertificatePath { get; set; }

/// <summary>

/// The client certificate password.

/// </summary>

public string ClientCertificatePassword { get; set; }

/// <summary>

/// Cache configuration options.

/// </summary>

public CacheOptions CacheOptions { get; set; } = new CacheOptions();

}

/// <summary>

/// Configuration options for caching.

/// </summary>

public class CacheOptions

{

/// <summary>

/// Indicates whether caching is enabled.

/// </summary>

public bool Enabled { get; set; } = true;

/// <summary>

/// The absolute expiration time in minutes.

/// </summary>

public int AbsoluteExpirationMinutes { get; set; } = 30;

/// <summary>

/// The sliding expiration time in minutes.

/// </summary>

public int SlidingExpirationMinutes { get; set; } = 10;

/// <summary>

/// The cache size limit in items.

/// </summary>

public int SizeLimit { get; set; } = 1000;

}

}

## **4. GrPC Client Implementation with Caching**

namespace ProgressPlay.NLP.SemanticLayer.Client

{

using System;

using System.Threading;

using System.Threading.Tasks;

using Grpc.Core;

using Grpc.Net.Client;

using Microsoft.Extensions.Caching.Memory;

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Options;

using ProgressPlay.NLP.SemanticLayer.Configuration;

using ProgressPlay.NLP.SemanticLayer.Interfaces;

using ProgressPlay.NLP.SemanticLayer.Models;

using ProgressPlay.NLP.SemanticLayer.Protos;

using Polly;

using Polly.Retry;

using System.Net.Http;

using System.Security.Cryptography.X509Certificates;

using System.Text.Json;

/// <summary>

/// Implementation of the semantic layer client using gRPC.

/// </summary>

public class GrpcSemanticLayerClient : ISemanticLayerClient, IDisposable

{

private readonly SemanticLayerService.SemanticLayerServiceClient \_client;

private readonly IMemoryCache \_cache;

private readonly ILogger<GrpcSemanticLayerClient> \_logger;

private readonly SemanticLayerClientOptions \_options;

private readonly GrpcChannel \_channel;

private readonly AsyncRetryPolicy<SqlTranslationResult> \_translateRetryPolicy;

private readonly AsyncRetryPolicy<QueryValidationResult> \_validateRetryPolicy;

private readonly AsyncRetryPolicy<bool> \_healthRetryPolicy;

/// <summary>

/// Initializes a new instance of the <see cref="GrpcSemanticLayerClient"/> class.

/// </summary>

public GrpcSemanticLayerClient(

IOptions<SemanticLayerClientOptions> options,

IMemoryCache cache,

ILogger<GrpcSemanticLayerClient> logger)

{

\_options = options.Value;

\_cache = cache;

\_logger = logger;

// Create gRPC channel with proper configuration

\_channel = CreateGrpcChannel();

\_client = new SemanticLayerService.SemanticLayerServiceClient(\_channel);

// Configure retry policies

\_translateRetryPolicy = CreateRetryPolicy<SqlTranslationResult>(

(ex) => \_logger.LogWarning(ex, "Error translating to SQL. Retrying..."));

\_validateRetryPolicy = CreateRetryPolicy<QueryValidationResult>(

(ex) => \_logger.LogWarning(ex, "Error validating query. Retrying..."));

\_healthRetryPolicy = CreateRetryPolicy<bool>(

(ex) => \_logger.LogWarning(ex, "Error checking service health. Retrying..."));

}

/// <summary>

/// Translates NLP entities to SQL query.

/// </summary>

public async Task<SqlTranslationResult> TranslateToSqlAsync(

string rawQuery,

QueryEntities entities,

CancellationToken cancellationToken = default)

{

\_logger.LogDebug("Translating query to SQL: {RawQuery}", rawQuery);

// Generate cache key

var cacheKey = $"translate:{GenerateCacheKey(rawQuery, entities)}";

// Try to get from cache first

if (\_options.CacheOptions.Enabled &&

\_cache.TryGetValue(cacheKey, out SqlTranslationResult cachedResult))

{

\_logger.LogDebug("Retrieved translation from cache for query: {RawQuery}", rawQuery);

return cachedResult;

}

// Execute with retry policy

return await \_translateRetryPolicy.ExecuteAsync(async (ct) =>

{

try

{

var entitiesJson = JsonSerializer.Serialize(entities);

var request = new NlpEntitiesRequest

{

RawQuery = rawQuery,

EntitiesJson = entitiesJson

};

var callOptions = CreateCallOptions(ct);

var response = await \_client.TranslateToSqlAsync(request, callOptions);

if (!response.Success)

{

\_logger.LogError("SQL translation failed: {ErrorMessage}", response.ErrorMessage);

throw new SemanticLayerException(response.ErrorMessage);

}

var result = new SqlTranslationResult

{

Sql = response.Sql,

Dataset = response.Dataset,

Metrics = response.Metrics.ToList(),

Dimensions = response.Dimensions.ToList()

};

// Store in cache if enabled

if (\_options.CacheOptions.Enabled)

{

var cacheEntryOptions = new MemoryCacheEntryOptions()

.SetAbsoluteExpiration(TimeSpan.FromMinutes(\_options.CacheOptions.AbsoluteExpirationMinutes))

.SetSlidingExpiration(TimeSpan.FromMinutes(\_options.CacheOptions.SlidingExpirationMinutes))

.SetSize(1);

\_cache.Set(cacheKey, result, cacheEntryOptions);

}

return result;

}

catch (RpcException ex)

{

\_logger.LogError(ex, "RPC error in semantic layer: {StatusCode} - {Message}", ex.StatusCode, ex.Message);

HandleRpcException(ex);

throw;

}

catch (JsonException ex)

{

\_logger.LogError(ex, "JSON serialization error: {Message}", ex.Message);

throw new SemanticLayerException("Failed to serialize or deserialize JSON data", ex);

}

catch (Exception ex) when (ex is not SemanticLayerException)

{

\_logger.LogError(ex, "Unexpected error in semantic layer client: {Message}", ex.Message);

throw new SemanticLayerException("Unexpected error occurred while communicating with semantic layer", ex);

}

}, cancellationToken);

}

/// <summary>

/// Validates a SQL query against security policies.

/// </summary>

public async Task<QueryValidationResult> ValidateQueryAsync(

string sql,

string dataset,

CancellationToken cancellationToken = default)

{

\_logger.LogDebug("Validating SQL query for dataset {Dataset}", dataset);

// Generate cache key

var cacheKey = $"validate:{sql}:{dataset}";

// Try to get from cache first

if (\_options.CacheOptions.Enabled &&

\_cache.TryGetValue(cacheKey, out QueryValidationResult cachedResult))

{

\_logger.LogDebug("Retrieved validation result from cache for SQL: {SqlPrefix}", sql.Substring(0, Math.Min(50, sql.Length)));

return cachedResult;

}

// Execute with retry policy

return await \_validateRetryPolicy.ExecuteAsync(async (ct) =>

{

try

{

var request = new QueryValidationRequest

{

Sql = sql,

Dataset = dataset

};

var callOptions = CreateCallOptions(ct);

var response = await \_client.ValidateQueryAsync(request, callOptions);

var result = new QueryValidationResult

{

IsValid = response.IsValid,

ValidationMessage = response.ValidationMessage,

OptimizedSql = response.OptimizedSql

};

// Store in cache if enabled

if (\_options.CacheOptions.Enabled && result.IsValid)

{

var cacheEntryOptions = new MemoryCacheEntryOptions()

.SetAbsoluteExpiration(TimeSpan.FromMinutes(\_options.CacheOptions.AbsoluteExpirationMinutes))

.SetSlidingExpiration(TimeSpan.FromMinutes(\_options.CacheOptions.SlidingExpirationMinutes))

.SetSize(1);

\_cache.Set(cacheKey, result, cacheEntryOptions);

}

return result;

}

catch (RpcException ex)

{

\_logger.LogError(ex, "RPC error in semantic layer: {StatusCode} - {Message}", ex.StatusCode, ex.Message);

HandleRpcException(ex);

throw;

}

catch (Exception ex) when (ex is not SemanticLayerException)

{

\_logger.LogError(ex, "Unexpected error in semantic layer client: {Message}", ex.Message);

throw new SemanticLayerException("Unexpected error occurred while communicating with semantic layer", ex);

}

}, cancellationToken);

}

/// <summary>

/// Gets the health status of the semantic layer service.

/// </summary>

public async Task<bool> IsHealthyAsync(CancellationToken cancellationToken = default)

{

\_logger.LogDebug("Checking semantic layer service health");

// Execute with retry policy

return await \_healthRetryPolicy.ExecuteAsync(async (ct) =>

{

try

{

var request = new HealthCheckRequest();

var callOptions = CreateCallOptions(ct);

var response = await \_client.CheckHealthAsync(request, callOptions);

return response.Status == HealthStatus.Healthy;

}

catch (RpcException ex)

{

\_logger.LogError(ex, "RPC error checking health: {StatusCode} - {Message}", ex.StatusCode, ex.Message);

return false;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Unexpected error checking health: {Message}", ex.Message);

return false;

}

}, cancellationToken);

}

/// <summary>

/// Disposes the client and channel.

/// </summary>

public void Dispose()

{

\_channel?.Dispose();

}

#region Private helper methods

private GrpcChannel CreateGrpcChannel()

{

var channelOptions = new GrpcChannelOptions

{

MaxReceiveMessageSize = null, // No limit

MaxSendMessageSize = null // No limit

};

if (\_options.UseSecureConnection && !string.IsNullOrEmpty(\_options.ClientCertificatePath))

{

// Configure TLS with client certificate

var handler = new HttpClientHandler();

var clientCertificate = new X509Certificate2(

\_options.ClientCertificatePath,

\_options.ClientCertificatePassword);

handler.ClientCertificates.Add(clientCertificate);

// Allow self-signed certificates in development

if (Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT") == "Development")

{

handler.ServerCertificateCustomValidationCallback =

HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

}

channelOptions.HttpHandler = handler;

}

return GrpcChannel.ForAddress(\_options.ServiceAddress, channelOptions);

}

private CallOptions CreateCallOptions(CancellationToken cancellationToken)

{

var deadline = DateTime.UtcNow.AddSeconds(\_options.TimeoutSeconds);

return new CallOptions(deadline: deadline, cancellationToken: cancellationToken);

}

private AsyncRetryPolicy<T> CreateRetryPolicy<T>(Action<Exception> onRetry)

{

return Policy<T>

.Handle<RpcException>(ex =>

ex.StatusCode == StatusCode.Unavailable ||

ex.StatusCode == StatusCode.DeadlineExceeded ||

ex.StatusCode == StatusCode.ResourceExhausted)

.WaitAndRetryAsync(

\_options.MaxRetryAttempts,

attempt => TimeSpan.FromMilliseconds(\_options.InitialBackoffMs \* Math.Pow(2, attempt - 1)),

(ex, timeSpan, retryCount, context) =>

{

onRetry(ex);

});

}

private void HandleRpcException(RpcException ex)

{

switch (ex.StatusCode)

{

case StatusCode.InvalidArgument:

throw new SemanticLayerException("Invalid arguments provided to semantic layer service", ex);

case StatusCode.PermissionDenied:

throw new SemanticLayerException("Permission denied by semantic layer service", ex);

case StatusCode.Unauthenticated:

throw new SemanticLayerException("Authentication failed with semantic layer service", ex);

case StatusCode.Unimplemented:

throw new SemanticLayerException("Requested operation is not implemented by semantic layer service", ex);

case StatusCode.Internal:

throw new SemanticLayerException("Internal error in semantic layer service", ex);

case StatusCode.Unavailable:

throw new SemanticLayerException("Semantic layer service is unavailable", ex);

case StatusCode.DeadlineExceeded:

throw new SemanticLayerException("Operation timed out", ex);

default:

throw new SemanticLayerException($"gRPC error: {ex.StatusCode} - {ex.Message}", ex);

}

}

private string GenerateCacheKey(string rawQuery, QueryEntities entities)

{

return $"{rawQuery}:{JsonSerializer.Serialize(entities)}".GetHashCode().ToString();

}

#endregion

}

/// <summary>

/// Factory for creating semantic layer clients.

/// </summary>

public class SemanticLayerClientFactory : ISemanticLayerClientFactory

{

private readonly IServiceProvider \_serviceProvider;

public SemanticLayerClientFactory(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public ISemanticLayerClient CreateClient(string instanceName = "Default")

{

return \_serviceProvider.GetRequiredService<ISemanticLayerClient>();

}

}

}

## **5. Extended Protocol Buffer Definitions**

syntax = "proto3";

package progressplay.nlp;

service SemanticLayerService {

// Translates NLP entities to SQL

rpc TranslateToSql (NlpEntitiesRequest) returns (SqlTranslationResponse);

// Validates a SQL query against security policies

rpc ValidateQuery (QueryValidationRequest) returns (QueryValidationResponse);

// Gets detailed metadata about available metrics and dimensions

rpc GetMetadata (MetadataRequest) returns (MetadataResponse);

// Checks service health

rpc CheckHealth (HealthCheckRequest) returns (HealthCheckResponse);

}

message NlpEntitiesRequest {

string raw\_query = 1;

string entities\_json = 2;

}

message SqlTranslationResponse {

string sql = 1;

string dataset = 2;

repeated string metrics = 3;

repeated string dimensions = 4;

repeated string filters = 5;

string error\_message = 6;

bool success = 7;

int32 estimated\_execution\_cost = 8;

}

message QueryValidationRequest {

string sql = 1;

string dataset = 2;

}

message QueryValidationResponse {

bool is\_valid = 1;

string validation\_message = 2;

string optimized\_sql = 3;

}

message MetadataRequest {

string dataset = 1;

}

message MetadataResponse {

repeated MetricInfo metrics = 1;

repeated DimensionInfo dimensions = 2;

}

message MetricInfo {

string name = 1;

string display\_name = 2;

string description = 3;

string data\_type = 4;

string expression = 5;

}

message DimensionInfo {

string name = 1;

string display\_name = 2;

string description = 3;

string data\_type = 4;

bool is\_hierarchical = 5;

string parent\_dimension = 6;

}

message HealthCheckRequest {

}

enum HealthStatus {

UNKNOWN = 0;

HEALTHY = 1;

DEGRADED = 2;

UNHEALTHY = 3;

}

message HealthCheckResponse {

HealthStatus status = 1;

string details = 2;

}

## **6. Service Registration Extensions**

namespace ProgressPlay.NLP.SemanticLayer.Extensions

{

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using ProgressPlay.NLP.SemanticLayer.Client;

using ProgressPlay.NLP.SemanticLayer.Configuration;

using ProgressPlay.NLP.SemanticLayer.Interfaces;

using ProgressPlay.NLP.SemanticLayer.Monitoring;

using Polly;

using Polly.Extensions.Http;

using System;

using System.Net.Http;

using OpenTelemetry.Trace;

using OpenTelemetry.Metrics;

/// <summary>

/// Extension methods for registering semantic layer services.

/// </summary>

public static class ServiceCollectionExtensions

{

/// <summary>

/// Adds semantic layer services to the service collection.

/// </summary>

public static IServiceCollection AddSemanticLayer(

this IServiceCollection services,

IConfiguration configuration,

Action<SemanticLayerClientOptions> configureOptions = null)

{

// Register options from configuration

services.AddOptions<SemanticLayerClientOptions>()

.Bind(configuration.GetSection("SemanticLayer"))

.ValidateDataAnnotations();

// Allow additional configuration

if (configureOptions != null)

{

services.Configure(configureOptions);

}

// Add memory cache for client

services.AddMemoryCache(options =>

{

var cacheOptions = configuration

.GetSection("SemanticLayer:CacheOptions")

.Get<CacheOptions>() ?? new CacheOptions();

options.SizeLimit = cacheOptions.SizeLimit;

});

// Register client interfaces

services.AddTransient<ISemanticLayerClient, GrpcSemanticLayerClient>();

services.AddSingleton<ISemanticLayerClientFactory, SemanticLayerClientFactory>();

// Register health check services

services.AddSingleton<ISemanticLayerHealthCheck, SemanticLayerHealthCheck>();

// Add client telemetry

services.AddSingleton<ISemanticLayerTelemetry, SemanticLayerTelemetry>();

// Configure OpenTelemetry

services.AddOpenTelemetry()

.WithTracing(builder => builder

.AddSource("ProgressPlay.NLP.SemanticLayer")

.AddGrpcClientInstrumentation())

.WithMetrics(builder => builder

.AddMeter("ProgressPlay.NLP.SemanticLayer"));

return services;

}

/// <summary>

/// Adds semantic layer health checks to the service collection.

/// </summary>

public static IHealthChecksBuilder AddSemanticLayerHealthCheck(

this IHealthChecksBuilder builder,

string name = "semantic-layer")

{

return builder.AddCheck<SemanticLayerHealthCheckService>(name);

}

}

}

## **7. Health Checks and Monitoring**

namespace ProgressPlay.NLP.SemanticLayer.Monitoring

{

using System;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

using Microsoft.Extensions.Diagnostics.HealthChecks;

using Microsoft.Extensions.Logging;

using ProgressPlay.NLP.SemanticLayer.Interfaces;

using OpenTelemetry.Metrics;

using OpenTelemetry.Trace;

using System.Diagnostics.Metrics;

/// <summary>

/// Interface for semantic layer health check.

/// </summary>

public interface ISemanticLayerHealthCheck

{

Task<bool> CheckHealthAsync(CancellationToken cancellationToken = default);

}

/// <summary>

/// Implementation of semantic layer health check.

/// </summary>

public class SemanticLayerHealthCheck : ISemanticLayerHealthCheck

{

private readonly ISemanticLayerClient \_client;

private readonly ILogger<SemanticLayerHealthCheck> \_logger;

public SemanticLayerHealthCheck(ISemanticLayerClient client, ILogger<SemanticLayerHealthCheck> logger)

{

\_client = client;

\_logger = logger;

}

public async Task<bool> CheckHealthAsync(CancellationToken cancellationToken = default)

{

try

{

return await \_client.IsHealthyAsync(cancellationToken);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error checking semantic layer health");

return false;

}

}

}

/// <summary>

/// Health check service for semantic layer.

/// </summary>

public class SemanticLayerHealthCheckService : IHealthCheck

{

private readonly ISemanticLayerHealthCheck \_healthCheck;

public SemanticLayerHealthCheckService(ISemanticLayerHealthCheck healthCheck)

{

\_healthCheck = healthCheck;

}

public async Task<HealthCheckResult> CheckHealthAsync(

HealthCheckContext context,

CancellationToken cancellationToken = default)

{

bool isHealthy = await \_healthCheck.CheckHealthAsync(cancellationToken);

if (isHealthy)

{

return HealthCheckResult.Healthy("Semantic layer service is healthy");

}

else

{

return HealthCheckResult.Unhealthy("Semantic layer service is unhealthy");

}

}

}

/// <summary>

/// Interface for semantic layer telemetry.

/// </summary>

public interface ISemanticLayerTelemetry

{

void RecordTranslationDuration(double milliseconds);

void RecordValidationDuration(double milliseconds);

void IncrementErrorCount(string errorType);

void IncrementCacheHit();

void IncrementCacheMiss();

}

/// <summary>

/// Implementation of semantic layer telemetry.

/// </summary>

public class SemanticLayerTelemetry : ISemanticLayerTelemetry

{

private static readonly ActivitySource \_activitySource =

new ActivitySource("ProgressPlay.NLP.SemanticLayer");

private readonly Meter \_meter = new Meter("ProgressPlay.NLP.SemanticLayer");

private readonly Counter<long> \_translationCounter;

private readonly Counter<long> \_validationCounter;

private readonly Counter<long> \_errorCounter;

private readonly Counter<long> \_cacheHitCounter;

private readonly Counter<long> \_cacheMissCounter;

private readonly Histogram<double> \_translationDurationHistogram;

private readonly Histogram<double> \_validationDurationHistogram;

public SemanticLayerTelemetry()

{

\_translationCounter = \_meter.CreateCounter<long>("semantic\_layer.translations.count");

\_validationCounter = \_meter.CreateCounter<long>("semantic\_layer.validations.count");

\_errorCounter = \_meter.CreateCounter<long>("semantic\_layer.errors.count");

\_cacheHitCounter = \_meter.CreateCounter<long>("semantic\_layer.cache.hits");

\_cacheMissCounter = \_meter.CreateCounter<long>("semantic\_layer.cache.misses");

\_translationDurationHistogram = \_meter.CreateHistogram<double>("semantic\_layer.translation.duration.ms");

\_validationDurationHistogram = \_meter.CreateHistogram<double>("semantic\_layer.validation.duration.ms");

}

public void RecordTranslationDuration(double milliseconds)

{

\_translationCounter.Add(1);

\_translationDurationHistogram.Record(milliseconds);

}

public void RecordValidationDuration(double milliseconds)

{

\_validationCounter.Add(1);

\_validationDurationHistogram.Record(milliseconds);

}

public void IncrementErrorCount(string errorType)

{

\_errorCounter.Add(1, new KeyValuePair<string, object>("error\_type", errorType));

}

public void IncrementCacheHit()

{

\_cacheHitCounter.Add(1);

}

public void IncrementCacheMiss()

{

\_cacheMissCounter.Add(1);

}

}

}

## **8. Example Usage**

// Program.cs or Startup.cs

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Configuration;

using ProgressPlay.NLP.SemanticLayer.Extensions;

public class Startup

{

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public void ConfigureServices(IServiceCollection services)

{

// Add semantic layer services

services.AddSemanticLayer(Configuration, options =>

{

options.ServiceAddress = "https://semantic-layer-service:50051";

options.UseSecureConnection = true;

options.TimeoutSeconds = 60;

// Configure caching

options.CacheOptions.AbsoluteExpirationMinutes = 15;

options.CacheOptions.SlidingExpirationMinutes = 5;

});

// Add health checks

services.AddHealthChecks()

.AddSemanticLayerHealthCheck();

// Other service registrations...

}

}

// Example usage in a controller or service

public class NlpQueryService

{

private readonly ISemanticLayerClient \_semanticClient;

private readonly ILogger<NlpQueryService> \_logger;

public NlpQueryService(

ISemanticLayerClientFactory clientFactory,

ILogger<NlpQueryService> logger)

{

\_semanticClient = clientFactory.CreateClient();

\_logger = logger;

}

public async Task<QueryResult> ProcessNaturalLanguageQueryAsync(

string query,

QueryEntities entities,

CancellationToken cancellationToken = default)

{

try

{

// Translate to SQL using semantic layer

var translationResult = await \_semanticClient.TranslateToSqlAsync(

query,

entities,

cancellationToken);

// Validate the generated SQL

var validationResult = await \_semanticClient.ValidateQueryAsync(

translationResult.Sql,

translationResult.Dataset,

cancellationToken);

if (!validationResult.IsValid)

{

\_logger.LogWarning("Invalid query: {Message}", validationResult.ValidationMessage);

return new QueryResult

{

Success = false,

ErrorMessage = validationResult.ValidationMessage

};

}

// Use optimized SQL if available

string sqlToExecute = validationResult.OptimizedSql ?? translationResult.Sql;

// Execute the SQL query and return results

// (implementation details omitted)

return new QueryResult

{

Success = true,

Sql = sqlToExecute,

Dataset = translationResult.Dataset,

// Add query execution results

};

}

catch (SemanticLayerException ex)

{

\_logger.LogError(ex, "Semantic layer error: {Message}", ex.Message);

return new QueryResult

{

Success = false,

ErrorMessage = "Failed to process natural language query"

};

}

}

}

## **9. Configuration for Microservice Architecture**

{

"SemanticLayer": {

"ServiceAddress": "http://semantic-layer-service:50051",

"TimeoutSeconds": 30,

"MaxRetryAttempts": 3,

"InitialBackoffMs": 100,

"UseSecureConnection": true,

"ClientCertificatePath": "/app/certs/client.pfx",

"ClientCertificatePassword": "",

"CacheOptions": {

"Enabled": true,

"AbsoluteExpirationMinutes": 30,

"SlidingExpirationMinutes": 10,

"SizeLimit": 1000

}

},

"DistributedCache": {

"RedisConnection": "redis:6379",

"InstanceName": "SemanticLayer:"

}

}

## **10. Advanced Distributed Caching Implementation**

namespace ProgressPlay.NLP.SemanticLayer.Caching

{

using System;

using System.Text.Json;

using System.Threading.Tasks;

using Microsoft.Extensions.Caching.Distributed;

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Options;

using ProgressPlay.NLP.SemanticLayer.Configuration;

using ProgressPlay.NLP.SemanticLayer.Monitoring;

/// <summary>

/// Distributed cache service for semantic layer operations.

/// </summary>

public class SemanticLayerDistributedCache : ISemanticLayerCache

{

private readonly IDistributedCache \_distributedCache;

private readonly CacheOptions \_cacheOptions;

private readonly ISemanticLayerTelemetry \_telemetry;

private readonly ILogger<SemanticLayerDistributedCache> \_logger;

public SemanticLayerDistributedCache(

IDistributedCache distributedCache,

IOptions<SemanticLayerClientOptions> options,

ISemanticLayerTelemetry telemetry,

ILogger<SemanticLayerDistributedCache> logger)

{

\_distributedCache = distributedCache;

\_cacheOptions = options.Value.CacheOptions;

\_telemetry = telemetry;

\_logger = logger;

}

public async Task<T> GetOrSetAsync<T>(

string key,

Func<Task<T>> factory,

Func<T, bool> shouldCache = null)

{

if (!\_cacheOptions.Enabled)

{

return await factory();

}

// Try to get from cache

string cachedValue = await \_distributedCache.GetStringAsync(key);

if (!string.IsNullOrEmpty(cachedValue))

{

try

{

\_telemetry.IncrementCacheHit();

T value = JsonSerializer.Deserialize<T>(cachedValue);

\_logger.LogDebug("Cache hit for key: {Key}", key);

return value;

}

catch (JsonException ex)

{

\_logger.LogWarning(ex, "Failed to deserialize cached value for key: {Key}", key);

// Continue with factory call

}

}

// Cache miss, get from factory

\_telemetry.IncrementCacheMiss();

\_logger.LogDebug("Cache miss for key: {Key}", key);

T result = await factory();

// Cache the result if enabled and meets criteria

if (shouldCache == null || shouldCache(result))

{

try

{

string serialized = JsonSerializer.Serialize(result);

var options = new DistributedCacheEntryOptions

{

AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(\_cacheOptions.AbsoluteExpirationMinutes),

SlidingExpiration = TimeSpan.FromMinutes(\_cacheOptions.SlidingExpirationMinutes)

};

await \_distributedCache.SetStringAsync(key, serialized, options);

}

catch (Exception ex)

{

\_logger.LogWarning(ex, "Failed to cache value for key: {Key}", key);

// Continue with result

}

}

return result;

}

public async Task InvalidateAsync(string keyPattern)

{

// Note: Redis does not support wildcard key deletion via IDistributedCache

// This is a simplified version that only works with exact keys

try

{

await \_distributedCache.RemoveAsync(keyPattern);

}

catch (Exception ex)

{

\_logger.LogWarning(ex, "Failed to invalidate cache for key pattern: {KeyPattern}", keyPattern);

}

}

}

/// <summary>

/// Interface for semantic layer cache operations.

/// </summary>

public interface ISemanticLayerCache

{

Task<T> GetOrSetAsync<T>(

string key,

Func<Task<T>> factory,

Func<T, bool> shouldCache = null);

Task InvalidateAsync(string keyPattern);

}

}

## **11. Advanced Circuit Breaking and Load Balancing**

namespace ProgressPlay.NLP.SemanticLayer.Resilience

{

using System;

using System.Collections.Generic;

using System.Threading;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

using Polly;

using Polly.CircuitBreaker;

using Polly.Bulkhead;

/// <summary>

/// Provides circuit breaking for semantic layer operations.

/// </summary>

public class SemanticLayerCircuitBreaker

{

private readonly AsyncCircuitBreakerPolicy \_circuitBreakerPolicy;

private readonly AsyncBulkheadPolicy \_bulkheadPolicy;

private readonly ILogger<SemanticLayerCircuitBreaker> \_logger;

public SemanticLayerCircuitBreaker(

ILogger<SemanticLayerCircuitBreaker> logger)

{

\_logger = logger;

// Configure circuit breaker

\_circuitBreakerPolicy = Policy

.Handle<Exception>(ex => !(ex is OperationCanceledException))

.CircuitBreakerAsync(

exceptionsAllowedBeforeBreaking: 5,

durationOfBreak: TimeSpan.FromSeconds(30),

onBreak: (ex, timespan) =>

{

\_logger.LogWarning(

ex,

"Circuit breaker tripped for semantic layer for {Timespan} seconds",

timespan.TotalSeconds);

},

onReset: () =>

{

\_logger.LogInformation("Circuit breaker reset for semantic layer");

},

onHalfOpen: () =>

{

\_logger.LogInformation("Circuit breaker half-open for semantic layer");

});

// Configure bulkhead to limit concurrent calls

\_bulkheadPolicy = Policy.BulkheadAsync(

maxParallelization: 20,

maxQueuingActions: 40,

onBulkheadRejectedAsync: context =>

{

\_logger.LogWarning("Bulkhead rejected execution for semantic layer");

return Task.CompletedTask;

});

}

/// <summary>

/// Executes the specified action with circuit breaking.

/// </summary>

public async Task<T> ExecuteAsync<T>(

Func<CancellationToken, Task<T>> action,

CancellationToken cancellationToken = default)

{

return await \_bulkheadPolicy.ExecuteAsync(async ct =>

await \_circuitBreakerPolicy.ExecuteAsync(async innerCt =>

await action(innerCt),

ct),

cancellationToken);

}

/// <summary>

/// Gets the current state of the circuit breaker.

/// </summary>

public CircuitState State => \_circuitBreakerPolicy.CircuitState;

}

}

## **12. Deployment and Scaling Considerations**

To deploy the semantic layer service in a scalable microservice architecture:

1. **Containerization**
   * Package both the Python semantic layer and the C# application as Docker containers
   * Use Docker Compose or Kubernetes for orchestration
2. **Scaling Strategy**
   * Horizontally scale the semantic layer service with multiple replicas
   * Deploy Redis for distributed caching across instances
   * Implement proper load balancing with gRPC
3. **Sample Docker Compose Configuration**

version: '3.8'

services:

semantic-layer:

build: ./semantic-layer

image: progressplay/semantic-layer:latest

deploy:

replicas: 3

resources:

limits:

cpus: '1'

memory: 2G

environment:

- PYTHONUNBUFFERED=1

- LOG\_LEVEL=INFO

healthcheck:

test: ["CMD", "python", "healthcheck.py"]

interval: 30s

timeout: 10s

retries: 3

start\_period: 40s

networks:

- nlp-network

dotnet-api:

build: ./dotnet-api

image: progressplay/nlp-api:latest

deploy:

replicas: 3

resources:

limits:

cpus: '1'

memory: 2G

ports:

- "8080:80"

environment:

- ASPNETCORE\_ENVIRONMENT=Production

- SemanticLayer\_\_ServiceAddress=http://semantic-layer:50051

depends\_on:

- semantic-layer

- redis

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost/health"]

interval: 30s

timeout: 10s

retries: 3

networks:

- nlp-network

redis:

image: redis:alpine

command: redis-server --appendonly yes

deploy:

resources:

limits:

cpus: '0.5'

memory: 1G

volumes:

- redis-data:/data

networks:

- nlp-network

networks:

nlp-network:

driver: overlay

volumes:

redis-data:

This comprehensive C# class library provides a robust, scalable, and efficient integration between Python semantic layer components and .NET applications using gRPC. It includes features for distributed caching, circuit breaking, telemetry, health checks, and service registration to support a microservice architecture.