

Aprende Semantic Kernel en 25 pasos

De cero a producción con .NET, Azure OpenAI y buenas prácticas

David Cantón Nadales

Índice

Aprende Semantic Kernel en 25 pasos	15
Dedicatoria	15
Página legal	15
Licencia del contenido (libre y gratuito)	15
Licencia del código de ejemplo	15
Atribución sugerida	15
Nota del autor	15
Tabla de contenidos	16
Cómo usar este libro (sin perderte)	16
Dos formas de seguir el libro	16
Opción A (recomendada): un mini-proyecto por paso	16
Opción B: un proyecto acumulativo	17
Convenciones del libro	17
Qué vas a saber hacer al terminar	17
Antes de empezar: entorno, cuentas y verificación	17
1) Instala .NET y herramientas	17
2) Decide proveedor de IA: Azure OpenAI o OpenAI	17
Opción A (rápida): variables de entorno	17
Opción B (recomendada en desarrollo): User Secrets	18
3) (Azure) Crear el recurso Azure OpenAI paso a paso	18
Checklist de Azure (para no atascarte)	18
4) Prueba rápida (smoke test)	18
5) (Opcional) Preparar un directorio de trabajo para los 25 pasos	18
Paso 01 — Introducción a Semantic Kernel con C#: Construyendo tu Primera Aplicación de IA	19
Mapa del paso	19
Ruta guiada (paso a paso)	19
Introducción	19
Requisitos Previos	19
¿Qué es Semantic Kernel?	19
Instalación y Configuración	20
Paso 1: Crear un Nuevo Proyecto	20
Paso 2: Instalar Paquetes NuGet	20
Paso 3: Configurar Azure OpenAI	20
Creando tu Primera Aplicación	20
Construyendo el Kernel	20
Tu Primera Llamada al LLM	20
Usando Prompts Inline	21

Configuración de Parámetros de Ejecución	21
Manejo de Errores	21
Ejemplo Completo: Asistente de Resumen	22
Mejores Prácticas	22
1. Usar Variables de Entorno	22
2. Configurar Timeouts	23
3. Logging Estructurado	23
4. Validar Entradas	23
Siguientes Pasos	23
Recursos Adicionales	23
Conclusión	23
Cierre del paso	23
Verificación rápida	23
Errores frecuentes y cómo desbloquearte	24
Ejercicios (para afianzar)	24
Siguiente paso	24
Paso 02 — Creando Plugins Personalizados en Semantic Kernel	24
Mapa del paso	24
Ruta guiada (paso a paso)	24
Introducción	24
¿Qué son los Plugins?	24
Creando tu Primer Plugin	25
Plugin Simple: Calculadora	25
Registrar el Plugin	25
Plugin Asíncrono: Búsqueda de Datos	25
Plugin con Dependencias: Acceso a Datos	26
Plugin Complejo: Procesamiento de Texto	28
Registro de Plugins con Inyección de Dependencias	29
Invocación Manual de Funciones	29
Plugin con Estado: Sesión de Usuario	30
Mejores Prácticas	31
1. Descripciones Claras	31
2. Manejo de Errores Robusto	31
3. Logging para Debugging	31
4. Parámetros Opcionales	32
Conclusión	32
Cierre del paso	32
Verificación rápida	32
Errores frecuentes y cómo desbloquearte	32
Ejercicios (para afianzar)	32
Siguiente paso	32
Paso 03 — Clasificación de Intenciones con LLMs en .NET	32
Mapa del paso	32
Ruta guiada (paso a paso)	33
Introducción	33
¿Qué es la Clasificación de Intenciones?	33
Arquitectura del Sistema	33
Implementación del Servicio	33
Modelo de Resultado	33
Servicio de Clasificación	34
Uso del Clasificador	38
Ejemplo Básico	38
Integración con Router	38
Clasificación Multi-Intención	39
Clasificación con Contexto	40
Testing del Clasificador	41

Mejores Prácticas	42
1. Temperatura Baja para Consistencia	42
2. Validación de Resultados	42
3. Logging Detallado	42
4. Caché de Clasificaciones	42
Conclusión	43
Cierre del paso	43
Verificación rápida	43
Errores frecuentes y cómo desbloquearte	43
Ejercicios (para afianzar)	43
Siguiente paso	43
Paso 04 — Servicios de Chat Completion con Azure OpenAI y Semantic Kernel	43
Mapa del paso	43
Ruta guiada (paso a paso)	44
Introducción	44
Conceptos Fundamentales	44
Chat History	44
Servicio de Chat Completion	44
Implementación de un Servicio de Chat Completo	45
Modelo de Conversación	45
Servicio de Chat con Estado	45
Streaming de Respuestas	48
Chat con Funciones (Function Calling)	49
Chat Multimodal (Texto e Imágenes)	50
Chat con Memoria Persistente	51
Manejo de Rate Limits	52
Retry con Exponential Backoff	53
Testing de Servicios de Chat	53
Mejores Prácticas	54
1. Limitar Historial de Mensajes	54
2. Validar Longitud de Mensajes	54
3. Sanitizar Entrada del Usuario	54
4. Monitorear Uso de Tokens	55
Conclusión	55
Cierre del paso	55
Verificación rápida	55
Errores frecuentes y cómo desbloquearte	55
Ejercicios (para afianzar)	55
Siguiente paso	55
Paso 05 — Vector Embeddings y Búsqueda Semántica con .NET	55
Mapa del paso	55
Ruta guiada (paso a paso)	56
Introducción	56
¿Qué son los Embeddings?	56
Configuración de Embeddings	56
Instalar Paquetes	56
Configurar Servicio de Embeddings	56
Generar Embeddings	56
Embedding de un Texto	56
Embeddings en Batch	57
Cálculo de Similitud	57
Similitud Coseno	57
Sistema de Búsqueda Semántica en Memoria	58
Búsqueda con Filtros	60
Integración con Base de Datos Vectorial	61
Modelo para Persistencia	61

Servicio con Persistencia	61
RAG (Retrieval Augmented Generation)	63
Ejemplo Completo: Sistema de Preguntas y Respuestas	64
Mejores Prácticas	65
1. Normalización de Texto	65
2. Chunking de Documentos Largos	66
3. Caché de Embeddings	66
4. Manejo de Errores	66
Conclusión	67
Cierre del paso	67
Verificación rápida	67
Errores frecuentes y cómo desbloquearte	67
Ejercicios (para afianzar)	67
Siguiente paso	67
Paso 06 — Prompt Engineering: Mejores Prácticas para LLMs	67
Mapa del paso	67
Ruta guiada (paso a paso)	68
Introducción	68
Principios Fundamentales	68
1. Claridad y Especificidad	68
2. Estructura Clara	68
3. Ejemplos (Few-Shot Learning)	68
Técnicas de Prompt Engineering	69
Chain-of-Thought (Cadena de Pensamiento)	69
Zero-Shot vs Few-Shot vs Many-Shot	69
Role Prompting	69
Structured Output	70
Patrones Avanzados	70
Template Method Pattern	70
Prompt Chaining	71
Self-Consistency	72
Anti-Patrones (Qué Evitar)	73
Prompts Ambiguos	73
Instrucciones Contradicторias	73
Formato Inconsistente	73
Prompts para Casos de Uso Comunes	74
Clasificación	74
Extracción de Información	74
Generación Creativa	75
Análisis y Resumen	75
Testing de Prompts	75
Optimización de Prompts	76
A/B Testing	76
Iteración Basada en Feedback	77
Mejores Prácticas Resumidas	78
1. Estructura	78
2. Claridad	78
3. Contexto	78
4. Testing	78
5. Temperatura	78
Conclusión	78
Cierre del paso	79
Verificación rápida	79
Errores frecuentes y cómo desbloquearte	79
Ejercicios (para afianzar)	79
Siguiente paso	79

Paso 07 — Integración de Azure OpenAI en Aplicaciones .NET	79
Mapa del paso	79
Ruta guiada (paso a paso)	79
Introducción	80
Configuración Inicial	80
Requisitos Previos	80
Obtener Credenciales	80
Instalación de Paquetes	80
Gestión Segura de Credenciales	80
User Secrets (Desarrollo)	80
Configuration en appsettings.json	80
Modelo de Configuración	80
Configuración con Dependency Injection	81
Startup/Program.cs	81
Cliente Azure OpenAI Nativo	82
Usando Azure.AI.OpenAI	82
Streaming de Respuestas	83
Manejo de Rate Limits	84
Implementación con Polly	84
Retry con Exponential Backoff	84
Monitoreo y Telemetría	85
Application Insights	85
Caché de Respuestas	86
Testing	87
Unit Tests con Mocks	87
Integration Tests	88
Mejores Prácticas	89
1. Seguridad	89
2. Timeouts Apropiados	89
3. Logging Estructurado	89
4. Manejo de Costos	89
Conclusión	90
Cierre del paso	90
Verificación rápida	90
Errores frecuentes y cómo desbloquearte	90
Ejercicios (para afianzar)	90
Siguiente paso	90
Paso 08 — Configuración de Temperatura y Tokens en Modelos LLM	90
Mapa del paso	90
Ruta guiada (paso a paso)	90
Introducción	91
Parámetros Principales	91
Temperature (0.0 - 2.0)	91
Max Tokens	91
Top P (Nucleus Sampling)	91
Frequency Penalty y Presence Penalty	92
Configuraciones por Caso de Uso	92
1. Clasificación/Categorización	92
2. Chat Conversacional	92
3. Generación Creativa	92
4. Extracción de Información	93
5. Resumen de Texto	93
6. Código/Programación	93
Servicio de Configuración Dinámica	93
Estimación de Tokens	95
Optimización de Costos	95
Testing de Configuraciones	96

Monitoreo de Configuraciones	97
Mejores Prácticas	98
1. Comenzar Conservador	98
2. Documentar Configuraciones	98
3. A/B Testing	98
Conclusión	98
Cierre del paso	99
Verificación rápida	99
Errores frecuentes y cómo desbloquearte	99
Ejercicios (para afianzar)	99
Siguiente paso	99
Paso 09 — Estrategias de Caché para Servicios de IA	99
Mapa del paso	99
Ruta guiada (paso a paso)	99
Introducción	99
¿Por Qué Cachear?	100
Tipos de Caché	100
1. Caché en Memoria (IMemoryCache)	100
2. Caché Distribuido (Redis)	100
3. Caché de Embeddings	102
Caché Inteligente	103
Caché con Versioning	103
Caché con TTL Dinámico	103
Caché por Similitud Semántica	104
Estrategias de Invalidación	105
Invalidación por Tiempo	105
Invalidación por Evento	105
Invalidación por Tamaño	106
Caché en Múltiples Niveles	106
Monitoreo de Caché	107
Mejores Prácticas	108
1. Definir TTL Apropiado	108
2. Considerar Tamaño de Caché	108
3. Caché Selectivo	108
4. Warming del Caché	109
Conclusión	109
Cierre del paso	109
Verificación rápida	109
Errores frecuentes y cómo desbloquearte	109
Ejercicios (para afianzar)	109
Siguiente paso	109
Paso 10 — Manejo de Errores en Aplicaciones de IA con .NET	109
Mapa del paso	109
Ruta guiada (paso a paso)	110
Introducción	110
Tipos de Errores Comunes	110
1. Errores de Red	110
2. Rate Limiting (429)	110
3. Respuestas Inesperadas del Modelo	111
Patrones de Retry	111
Retry Simple	111
Exponential Backoff con Polly	111
Mejores Prácticas	112
Cierre del paso	112
Verificación rápida	112
Errores frecuentes y cómo desbloquearte	112

Ejercicios (para afianzar)	112
Siguiente paso	112
Paso 11 — Salida JSON Estructurada con LLMs	112
Mapa del paso	112
Ruta guiada (paso a paso)	112
Introducción	113
Conceptos Clave	113
Fundamentos	113
Implementación Práctica	113
Paso 1: Configuración	113
Paso 2: Uso	113
Mejores Prácticas	114
Patrones Avanzados	114
Patrón 1: Factory	114
Patrón 2: Strategy	114
Testing	115
Unit Tests	115
Integration Tests	115
Optimización	115
Performance	115
Escalabilidad	116
Conclusión	116
Recursos Adicionales	116
Cierre del paso	116
Verificación rápida	116
Errores frecuentes y cómo desbloquearte	116
Ejercicios (para afianzar)	116
Siguiente paso	116
Paso 12 — Testing de Servicios de IA en .NET	116
Mapa del paso	116
Ruta guiada (paso a paso)	117
Introducción	117
Conceptos Clave	117
Fundamentos	117
Implementación Práctica	118
Paso 1: Configuración	118
Paso 2: Uso	118
Mejores Prácticas	118
Patrones Avanzados	118
Patrón 1: Factory	118
Patrón 2: Strategy	118
Testing	119
Unit Tests	119
Integration Tests	119
Optimización	120
Performance	120
Escalabilidad	120
Conclusión	120
Recursos Adicionales	120
Cierre del paso	120
Verificación rápida	120
Errores frecuentes y cómo desbloquearte	120
Ejercicios (para afianzar)	121
Siguiente paso	121
Paso 13 — Inyección de Dependencias con Semantic Kernel	121
Mapa del paso	121

Ruta guiada (paso a paso)	121
Introducción	121
Conceptos Clave	121
Fundamentos	121
Implementación Práctica	122
Paso 1: Configuración	122
Paso 2: Uso	122
Mejores Prácticas	122
Patrones Avanzados	122
Patrón 1: Factory	122
Patrón 2: Strategy	123
Testing	123
Unit Tests	123
Integration Tests	124
Optimización	124
Performance	124
Escalabilidad	124
Conclusión	124
Recursos Adicionales	124
Cierre del paso	125
Verificación rápida	125
Errores frecuentes y cómo desbloquearte	125
Ejercicios (para afianzar)	125
Siguiente paso	125
Paso 14 — Comprensión de Consultas con Lenguaje Natural	125
Mapa del paso	125
Ruta guiada (paso a paso)	125
Introducción	125
Conceptos Clave	126
Fundamentos	126
Implementación Práctica	126
Paso 1: Configuración	126
Paso 2: Uso	126
Mejores Prácticas	126
Patrones Avanzados	127
Patrón 1: Factory	127
Patrón 2: Strategy	127
Testing	128
Unit Tests	128
Integration Tests	128
Optimización	128
Performance	128
Escalabilidad	129
Conclusión	129
Recursos Adicionales	129
Cierre del paso	129
Verificación rápida	129
Errores frecuentes y cómo desbloquearte	129
Ejercicios (para afianzar)	129
Siguiente paso	129
Paso 15 — Sistemas de Búsqueda Semántica en Producción	129
Mapa del paso	129
Ruta guiada (paso a paso)	130
Introducción	130
Conceptos Clave	130
Fundamentos	130

Implementación Práctica	131
Paso 1: Configuración	131
Paso 2: Uso	131
Mejores Prácticas	131
Patrones Avanzados	131
Patrón 1: Factory	131
Patrón 2: Strategy	131
Testing	132
Unit Tests	132
Integration Tests	132
Optimización	133
Performance	133
Escalabilidad	133
Conclusión	133
Recursos Adicionales	133
Cierre del paso	133
Verificación rápida	133
Errores frecuentes y cómo desbloquearte	133
Ejercicios (para afianzar)	133
Siguiente paso	133
Paso 16 — Configuración de HttpClient para Servicios de IA	134
Mapa del paso	134
Ruta guiada (paso a paso)	134
Introducción	134
Conceptos Clave	134
Fundamentos	134
Implementación Práctica	135
Paso 1: Configuración	135
Paso 2: Uso	135
Mejores Prácticas	135
Patrones Avanzados	135
Patrón 1: Factory	135
Patrón 2: Strategy	136
Testing	136
Unit Tests	136
Integration Tests	136
Optimización	137
Performance	137
Escalabilidad	137
Conclusión	137
Recursos Adicionales	137
Cierre del paso	138
Verificación rápida	138
Errores frecuentes y cómo desbloquearte	138
Ejercicios (para afianzar)	138
Siguiente paso	138
Paso 17 — Guardrails y Validación en Sistemas de IA	138
Mapa del paso	138
Ruta guiada (paso a paso)	138
Introducción	138
Conceptos Clave	138
Fundamentos	138
Implementación Práctica	139
Paso 1: Configuración	139
Paso 2: Uso	139
Mejores Prácticas	139

Patrones Avanzados	140
Patrón 1: Factory	140
Patrón 2: Strategy	140
Testing	140
Unit Tests	140
Integration Tests	141
Optimización	141
Performance	141
Escalabilidad	141
Conclusión	142
Recursos Adicionales	142
Cierre del paso	142
Verificación rápida	142
Errores frecuentes y cómo desbloquearte	142
Ejercicios (para afianzar)	142
Siguiente paso	142
Paso 18 — Workflows Multi-Paso con IA	142
Mapa del paso	142
Ruta guiada (paso a paso)	143
Introducción	143
Conceptos Clave	143
Fundamentos	143
Implementación Práctica	144
Paso 1: Configuración	144
Paso 2: Uso	144
Mejores Prácticas	144
Patrones Avanzados	144
Patrón 1: Factory	144
Patrón 2: Strategy	144
Testing	145
Unit Tests	145
Integration Tests	145
Optimización	146
Performance	146
Escalabilidad	146
Conclusión	146
Recursos Adicionales	146
Cierre del paso	146
Verificación rápida	146
Errores frecuentes y cómo desbloquearte	146
Ejercicios (para afianzar)	146
Siguiente paso	146
Paso 19 — Optimización de Costos en Aplicaciones de IA	147
Mapa del paso	147
Ruta guiada (paso a paso)	147
Introducción	147
Conceptos Clave	147
Fundamentos	147
Implementación Práctica	148
Paso 1: Configuración	148
Paso 2: Uso	148
Mejores Prácticas	148
Patrones Avanzados	148
Patrón 1: Factory	148
Patrón 2: Strategy	149
Testing	149

Unit Tests	149
Integration Tests	149
Optimización	150
Performance	150
Escalabilidad	150
Conclusión	150
Recursos Adicionales	150
Cierre del paso	151
Verificación rápida	151
Errores frecuentes y cómo desbloquearte	151
Ejercicios (para afianzar)	151
Siguiente paso	151
Paso 20 — Monitoreo y Observabilidad de Servicios de IA	151
Mapa del paso	151
Ruta guiada (paso a paso)	151
Introducción	151
Conceptos Clave	151
Fundamentos	151
Implementación Práctica	152
Paso 1: Configuración	152
Paso 2: Uso	152
Mejores Prácticas	152
Patrones Avanzados	153
Patrón 1: Factory	153
Patrón 2: Strategy	153
Testing	153
Unit Tests	153
Integration Tests	154
Optimización	154
Performance	154
Escalabilidad	154
Conclusión	155
Recursos Adicionales	155
Cierre del paso	155
Verificación rápida	155
Errores frecuentes y cómo desbloquearte	155
Ejercicios (para afianzar)	155
Siguiente paso	155
Paso 21 — Arquitectura de Microservicios con IA	155
Mapa del paso	155
Ruta guiada (paso a paso)	156
Introducción	156
Conceptos Clave	156
Fundamentos	156
Implementación Práctica	157
Paso 1: Configuración	157
Paso 2: Uso	157
Mejores Prácticas	157
Patrones Avanzados	157
Patrón 1: Factory	157
Patrón 2: Strategy	157
Testing	158
Unit Tests	158
Integration Tests	158
Optimización	159
Performance	159

Escalabilidad	159
Conclusión	159
Recursos Adicionales	159
Cierre del paso	159
Verificación rápida	159
Errores frecuentes y cómo desbloquearte	159
Ejercicios (para afianzar)	159
Siguiente paso	159
Paso 22 — Seguridad en Aplicaciones de IA	160
Mapa del paso	160
Ruta guiada (paso a paso)	160
Introducción	160
Conceptos Clave	160
Fundamentos	160
Implementación Práctica	161
Paso 1: Configuración	161
Paso 2: Uso	161
Mejores Prácticas	161
Patrones Avanzados	161
Patrón 1: Factory	161
Patrón 2: Strategy	162
Testing	162
Unit Tests	162
Integration Tests	162
Optimización	163
Performance	163
Escalabilidad	163
Conclusión	163
Recursos Adicionales	163
Cierre del paso	164
Verificación rápida	164
Errores frecuentes y cómo desbloquearte	164
Ejercicios (para afianzar)	164
Siguiente paso	164
Paso 23 — Implementación de Routers Conversacionales Inteligentes	164
Mapa del paso	164
Ruta guiada (paso a paso)	164
Introducción	164
Arquitectura de un Router	165
Modelo de Estado de Conversación	165
Implementación del Router	165
Gestión de Estado con Redis	170
Router con Middleware Pattern	172
Testing del Router	173
Mejores Prácticas	174
1. Prioridad de Comandos	174
2. Timeouts para Flujos	174
3. Context Cleanup	174
Conclusión	175
Cierre del paso	175
Verificación rápida	175
Errores frecuentes y cómo desbloquearte	175
Ejercicios (para afianzar)	175
Siguiente paso	175
Paso 24 — Filtrado por Relevancia Semántica en Búsquedas	175
Mapa del paso	175

Ruta guiada (paso a paso)	175
Introducción	176
Conceptos de Relevancia	176
Tipos de Relevancia	176
Implementación de Filtros	176
Mejores Prácticas	177
Cierre del paso	177
Verificación rápida	177
Errores frecuentes y cómo desbloquearte	177
Ejercicios (para afianzar)	178
Siguiente paso	178
Paso 25 — Normalización y Preprocesamiento de Datos para IA	178
Mapa del paso	178
Ruta guiada (paso a paso)	178
Introducción	178
Normalización de Texto	178
Tokenización	179
Limpieza de Datos	179
Mejores Prácticas	180
Cierre del paso	180
Verificación rápida	180
Errores frecuentes y cómo desbloquearte	180
Ejercicios (para afianzar)	180
Fin del recorrido	180
Contacto del autor	180
Agradecimientos	180

APRENDE

SEMANTIC KERNEL

EN 25 PASOS

De cero a producción con .NET
Azure OpenAI y buenas prácticas

David Cantón Nadales

Microsoft MVP · 2026

Aprende Semantic Kernel en 25 pasos

Subtítulo: De cero a producción con .NET, Azure OpenAI y buenas prácticas

Autor: David Cantón Nadales **Edición:** 1^a / 2026

Dedicatoria

A mi padre, que está viviendo su peor momento.

Página legal

© 2026 David Cantón Nadales. **Algunos derechos reservados.**

Este ebook se ofrece como **recurso educativo**. El software y los ejemplos incluidos se proporcionan “tal cual”, sin garantías.

- **Marcas registradas:** Azure, .NET y Semantic Kernel son marcas de sus respectivos propietarios.
- **Costes:** ejecutar ejemplos con servicios de IA puede generar costes. Revisa precios y cuotas antes de usar entornos de producción.

Licencia del contenido (libre y gratuito)

Salvo que se indique lo contrario, el texto y los recursos de este libro se distribuyen bajo **Creative Commons Atribución 4.0 Internacional (CC BY 4.0)**:

- Puedes **copiar y redistribuir** el material en cualquier medio o formato.
- Puedes **remezclar, transformar y crear a partir** del material, incluso con fines comerciales.
- Condición: debes **atribuir** la autoría, **indicar cambios** si los hay y **mantener la referencia a la licencia**.

Licencia: <https://creativecommons.org/licenses/by/4.0/>

Licencia del código de ejemplo

Salvo que se indique lo contrario, el código de ejemplo incluido en este libro se distribuye bajo la **licencia MIT**.

Atribución sugerida

“Aprende Semantic Kernel en 25 pasos”, David Cantón Nadales (2026). CC BY 4.0.

Nota del autor

Este ebook es una guía práctica para aprender **Semantic Kernel** en .NET de forma progresiva, con foco en:

- fundamentos (kernel, plugins, prompts),
- integración con servicios de IA (Azure OpenAI / OpenAI),
- diseño “production-ready” (errores, testing, guardrails, coste, observabilidad, seguridad),
- y técnicas avanzadas (routers conversacionales, filtrado semántico, normalización de datos).

Soy **David Cantón Nadales**, ingeniero de software de Sevilla (España). Llevo más de 20 años desarrollando productos y compartiendo conocimiento a través de tutoriales, charlas y libros. He sido reconocido como **Microsoft MVP** en la categoría de **.NET**, y soy autor de *Curso práctico con Unity 3D* (Anaya Multimedia) y *Build Your Own Metaverse with Unity* (Packt).

He escrito este libro porque, cuando empezamos con IA y LLMs, lo más difícil no es “hacer la primera llamada”, sino **construir sistemas mantenibles y seguros**: prompts versionados, salida estructurada en JSON, validación, testing, control de costes, observabilidad y guardrails. Mi objetivo es que al terminar estos 25 pasos puedas diseñar y construir aplicaciones con Semantic Kernel con un enfoque realista, listo para llevar a producción.

Este ebook se publica de forma **libre y gratuita**, para que cualquiera pueda aprender y reutilizar el contenido según las licencias indicadas en la página legal.

Gracias a la comunidad por las preguntas y el feedback, al equipo de Semantic Kernel y a todos los proyectos open-source que sostienen este ecosistema. Si este libro te resulta útil, compártelo para que llegue a más personas.

Tabla de contenidos

- Cómo usar este libro
- Antes de empezar: entorno, cuentas y verificación
- Paso 01 — Introducción a Semantic Kernel con C#: Construyendo tu Primera Aplicación de IA
- Paso 02 — Creando Plugins Personalizados en Semantic Kernel
- Paso 03 — Clasificación de Intenciones con LLMs en .NET
- Paso 04 — Servicios de Chat Completion con Azure OpenAI y Semantic Kernel
- Paso 05 — Vector Embeddings y Búsqueda Semántica con .NET
- Paso 06 — Prompt Engineering: Mejores Prácticas para LLMs
- Paso 07 — Integración de Azure OpenAI en Aplicaciones .NET
- Paso 08 — Configuración de Temperatura y Tokens en Modelos LLM
- Paso 09 — Estrategias de Caché para Servicios de IA
- Paso 10 — Manejo de Errores en Aplicaciones de IA con .NET
- Paso 11 — Salida JSON Estructurada con LLMs
- Paso 12 — Testing de Servicios de IA en .NET
- Paso 13 — Inyección de Dependencias con Semantic Kernel
- Paso 14 — Comprensión de Consultas con Lenguaje Natural
- Paso 15 — Sistemas de Búsqueda Semántica en Producción
- Paso 16 — Configuración de HttpClient para Servicios de IA
- Paso 17 — Guardrails y Validación en Sistemas de IA
- Paso 18 — Workflows Multi-Paso con IA
- Paso 19 — Optimización de Costos en Aplicaciones de IA
- Paso 20 — Monitoreo y Observabilidad de Servicios de IA
- Paso 21 — Arquitectura de Microservicios con IA
- Paso 22 — Seguridad en Aplicaciones de IA
- Paso 23 — Implementación de Routers Conversacionales Inteligentes
- Paso 24 — Filtrado por Relevancia Semántica en Búsquedas
- Paso 25 — Normalización y Preprocesamiento de Datos para IA
- Contacto del autor

Cómo usar este libro (sin perderte)

Este ebook está organizado en **25 pasos**. Cada paso está pensado para ser:

- **autocontenido** (puedes leerlo de forma independiente),
- **práctico** (con comandos y C# listo para adaptar),
- y **orientado a producción** (con validaciones y errores frecuentes).

Dos formas de seguir el libro

Opción A (recomendada): un mini-proyecto por paso

Ideal si estás aprendiendo y no quieres pelearte con dependencias entre capítulos.

- En cada paso crearás una app pequeña (console o API) con el mínimo necesario.
- Ventaja: menos fricción.

- Inconveniente: menos continuidad de “producto final”.

Opción B: un proyecto acumulativo

Ideal si quieres acabar con una base de código más realista.

- Mantendrás una solución `.sln` y añadirás carpetas/módulos.
- Ventaja: visión de arquitectura.
- Inconveniente: más decisiones (y más mantenimiento).

Consejo: empieza con la Opción A; cuando domines los pasos 1–8, repite con la Opción B.

Convenciones del libro

- Los bloques de terminal usan `bash`, pero los comandos son equivalentes en Windows (PowerShell) con pequeñas variaciones.
- Cuando veas valores como `https://tu-recurso.openai.azure.com/` o `tu-api-key`, sustitúyelos por tus datos.
- Si un capítulo usa Azure, verás una sección explícita con la lista de recursos y cómo crearlos.

Qué vas a saber hacer al terminar

Al finalizar el paso 25 podrás:

- diseñar aplicaciones con LLMs con control de calidad (guardrails, validación, JSON estructurado),
- integrar chat/embeddings con Semantic Kernel,
- añadir caché, resiliencia, testing, observabilidad y seguridad,
- aplicar patrones de producción (routers, filtrado, normalización, arquitectura de microservicios),
- y tomar decisiones informadas de coste/rendimiento.

Antes de empezar: entorno, cuentas y verificación

El objetivo de esta sección es que llegues al **Paso 01** con todo listo para ejecutar ejemplos sin bloqueos.

1) Instala .NET y herramientas

Recomendación: usa una versión **LTS** de .NET (mínimo .NET 6).

- SDK de .NET
- Visual Studio 2022 o VS Code
- Git

Verifica:

```
dotnet --version
dotnet --info
```

2) Decide proveedor de IA: Azure OpenAI o OpenAI

En el libro verás ejemplos con **Azure OpenAI** (por integración empresarial y despliegues). Si no tienes acceso, puedes adaptar a OpenAI API (en muchos casos, el cambio es de configuración/conector).

Opción A (rápida): variables de entorno

Define estas variables (nombres orientativos; cada capítulo te lo recordará):

```
export AZURE_OPENAI_ENDPOINT="https://TU-RECURSO.openai.azure.com/"
export AZURE_OPENAI_KEY="TU_API_KEY"
export AZURE_OPENAI_CHAT_DEPLOYMENT="gpt-4"
export AZURE_OPENAI_EMBEDDING_DEPLOYMENT="text-embedding-ada-002"
```

En Windows (PowerShell), usa `setx` o define variables en tu sesión. En macOS/Linux puedes ponerlas en tu `~/.zshrc`/`~/.bashrc`.

Opción B (recomendada en desarrollo): User Secrets

En .NET, **User Secrets** te permite guardar credenciales fuera del repo, por proyecto.

```
dotnet user-secrets init
dotnet user-secrets set "AzureOpenAI:Endpoint" "https://TU-RECURSO.openai.azure.com/"
dotnet user-secrets set "AzureOpenAI:ApiKey" "TU_API_KEY"
dotnet user-secrets set "AzureOpenAI:ChatDeployment" "gpt-4"
dotnet user-secrets set "AzureOpenAI:EmbeddingDeployment" "text-embedding-ada-002"
```

Consejo: usa User Secrets para desarrollo local y variables de entorno/Key Vault para producción.

3) (Azure) Crear el recurso Azure OpenAI paso a paso

Nota: los nombres exactos de pantallas pueden variar; la lógica general se mantiene.

1. Entra al **Portal de Azure**.
2. Si es tu primera vez con Azure OpenAI, puede requerir **solicitud de acceso** o estar limitado por región/tenant. Si no lo ves disponible, revisa la documentación oficial de Azure y el estado de tu suscripción.
3. Crea un recurso **Azure OpenAI** (busca “Azure OpenAI”).
4. Una vez creado, abre el recurso y ve a:
 - **Keys and Endpoint** (o similar) para obtener Endpoint y Key.
5. Abre **Azure AI Studio** (o el “Studio” asociado) para **crear deployments**:
 - un deployment de **chat** (por ejemplo GPT-4 / GPT-35 Turbo, según disponibilidad),
 - y un deployment de **embeddings**.
6. Guarda los nombres de deployment exactamente (son *case-sensitive* en tu configuración).

Checklist de Azure (para no atascarte)

- Tienes Endpoint y Key del recurso.
- Tienes un deployment de **chat** (para `AddAzureOpenAIChatCompletion`).
- Tienes un deployment de **embeddings** (para `AddAzureOpenAITextEmbeddingGeneration`).
- Has confirmado que los nombres de deployment son exactamente iguales a los que configuras en tu app.

4) Prueba rápida (smoke test)

Crea un proyecto y comprueba que puedes llamar al chat completion con Semantic Kernel.

```
mkdir -p sk-smoke-test && cd sk-smoke-test
dotnet new console -n SmokeTest
cd SmokeTest
dotnet add package Microsoft.SemanticKernel
```

Si no quieres escribir código todavía, salta al **Paso 01**, donde está el ejemplo completo.

5) (Opcional) Preparar un directorio de trabajo para los 25 pasos

Si vas a seguir la “Opción A” (un proyecto por paso), una estructura típica es:

```
sk-25-pasos/
  paso-01/
  paso-02/
  ...
  paso-25/
```

Si prefieres la “Opción B” (proyecto acumulativo), empieza con una solución:

```
mkdir -p sk-25-pasos && cd sk-25-pasos
dotnet new sln -n SemanticKernel25Pasos
```

Paso 01 — Introducción a Semantic Kernel con C#: Construyendo tu Primera Aplicación de IA

Mapa del paso

Tiempo estimado: 45–75 min

Resultado que vas a conseguir:

Una consola .NET que crea un Kernel, llama a un chat model y ejecuta un prompt inline.

Objetivos de aprendizaje:

- Entender qué es el Kernel y por qué es el centro de Semantic Kernel.
- Configurar un conector (Azure OpenAI) de forma segura (sin hardcodear claves).
- Ejecutar tu primera llamada a un LLM y controlar parámetros básicos (temperature/tokens).

Prerrequisitos:

- .NET 6+ (ideal LTS) y un editor (VS/VS Code).
- Acceso a Azure OpenAI u OpenAI y una API key.

Ruta guiada (paso a paso)

1. Crea un proyecto de consola con `dotnet new console`.
2. Instala `Microsoft.SemanticKernel` y logging a consola.
3. Configura `AZURE_OPENAI_ENDPOINT` y `AZURE_OPENAI_KEY` en tu entorno (o usa User Secrets).
4. Crea el Kernel y registra el servicio de chat completion.
5. Ejecuta una conversación mínima y verifica que obtienes respuesta.
6. Ajusta `PromptExecutionSettings` y observa el cambio.

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

Introducción

Semantic Kernel es un SDK open-source de Microsoft que permite integrar modelos de lenguaje de gran tamaño (LLMs) como GPT-4 en aplicaciones .NET. En este tutorial aprenderás a configurar y usar Semantic Kernel para crear aplicaciones inteligentes que pueden comprender y generar lenguaje natural.

Requisitos Previos

- .NET 6.0 o superior
- Visual Studio 2022 o VS Code
- Cuenta de Azure OpenAI o acceso a OpenAI API
- Conocimientos básicos de C# y programación asíncrona

¿Qué es Semantic Kernel?

Semantic Kernel es un framework ligero que actúa como orquestador entre tu código y servicios de IA. Te permite:

- Integrar LLMs en aplicaciones .NET
- Crear funciones reutilizables (Skills/Plugins)
- Orquestar llamadas múltiples a modelos de IA
- Gestionar contexto y memoria en conversaciones
- Combinar IA con código tradicional

Instalación y Configuración

Paso 1: Crear un Nuevo Proyecto

```
dotnet new console -n MiAppSemanticKernel  
cd MiAppSemanticKernel
```

Paso 2: Instalar Paquetes NuGet

```
dotnet add package Microsoft.SemanticKernel  
dotnet add package Microsoft.Extensions.Logging.Console
```

Paso 3: Configurar Azure OpenAI

Necesitarás los siguientes valores de tu recurso Azure OpenAI:

- **Endpoint:** URL de tu servicio Azure OpenAI
- **API Key:** Clave de acceso
- **Deployment Name:** Nombre de tu modelo desplegado (ej: gpt-4)

Creando tu Primera Aplicación

Construyendo el Kernel

El Kernel es el componente central de Semantic Kernel. Aquí está cómo crearlo:

```
using Microsoft.SemanticKernel;  
using Microsoft.Extensions.Logging;  
  
// Crear el builder del kernel  
var builder = Kernel.CreateBuilder();  
  
// Configurar logging (opcional pero recomendado)  
builder.Services.AddLogging(c => c.AddConsole().SetMinimumLevel(LogLevel.Information));  
  
// Agregar servicio de chat completion  
builder.AddAzureOpenAIChatCompletion(  
    deploymentName: "gpt-4",  
    endpoint: "https://tu-recurso.openai.azure.com/",  
    apiKey: "tu-api-key");  
  
// Construir el kernel  
var kernel = builder.Build();
```

Tu Primera Llamada al LLM

Una vez configurado el kernel, puedes hacer tu primera llamada:

```
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.ChatCompletion;  
  
var chatService = kernel.GetRequiredService<IChatCompletionService>();  
  
var chatHistory = new ChatHistory();  
chatHistory.AddSystemMessage("Eres un asistente útil que responde de forma concisa.");  
chatHistory.AddUserMessage("¿Cuál es la capital de España?");  
  
var response = await chatService.GetChatMessageContentAsync(  
    chatHistory,  
    kernel: kernel);  
  
Console.WriteLine($"Respuesta: {response.Content}");
```

Usando Prompts Inline

Semantic Kernel permite crear funciones directamente desde prompts:

```
string promptTemplate = """
Eres un experto en {{$tema}}.
Explica el siguiente concepto de forma simple: {{$concepto}}
""";
```

```
var function = kernel.CreateFunctionFromPrompt(promptTemplate);
```

```
var result = await kernel.InvokeAsync(function, new KernelArguments
{
    ["tema"] = "programación",
    ["concepto"] = "recursividad"
});
```

```
Console.WriteLine(result.GetValue<string>());
```

Configuración de Parámetros de Ejecución

Puedes controlar el comportamiento del modelo con `PromptExecutionSettings`:

```
using Microsoft.SemanticKernel.Connectors.OpenAI;
```

```
var settings = new OpenAIPromptExecutionSettings
{
    Temperature = 0.7,           // Creatividad (0.0 - 1.0)
    MaxTokens = 500,             // Longitud máxima de respuesta
    TopP = 0.9,                  // Muestreo nucleus
    FrequencyPenalty = 0.0,      // Penalización por repetición
    PresencePenalty = 0.0        // Penalización por temas ya mencionados
};
```

```
var response = await chatService.GetChatMessageContentAsync(
    chatHistory,
    settings,
    kernel);
```

Manejo de Errores

Es importante manejar errores en aplicaciones de IA:

```
try
{
    var response = await chatService.GetChatMessageContentAsync(
        chatHistory,
        kernel: kernel);

    Console.WriteLine(response.Content);
}
catch (HttpRequestException ex)
{
    Console.WriteLine($"Error de conexión: {ex.Message}");
}
catch (Exception ex)
{
    Console.WriteLine($"Error inesperado: {ex.Message}");
}
```

Ejemplo Completo: Asistente de Resumen

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.Extensions.Logging;

class Program
{
    static async Task Main(string[] args)
    {
        // Configurar kernel
        var builder = Kernel.CreateBuilder();
        builder.Services.AddLogging(c => c.AddConsole());

        builder.AddAzureOpenAIChatCompletion(
            deploymentName: "gpt-4",
            endpoint: Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")!,
            apiKey: Environment.GetEnvironmentVariable("AZURE_OPENAI_KEY")!);

        var kernel = builder.Build();

        // Crear función de resumen
        string promptResumen = """
        Resume el siguiente texto en 3 puntos clave:

        {{$texto}}
        """;

        var funcionResumen = kernel.CreateFunctionFromPrompt(promptResumen);

        // Texto a resumir
        string texto = """
        Semantic Kernel es un SDK ligero que permite integrar modelos de IA
        en aplicaciones .NET. Proporciona abstracciones para trabajar con diferentes
        proveedores de IA, gestionar prompts, y orquestar llamadas complejas.
        Es especialmente útil para desarrolladores que quieren construir aplicaciones
        inteligentes sin reinventar la rueda.
        """;

        // Invocar función
        var resultado = await kernel.InvokeAsync(funcionResumen, new KernelArguments
        {
            ["texto"] = texto
        });

        Console.WriteLine("Resumen:");
        Console.WriteLine(resultado.GetValue<string>());
    }
}
```

Mejores Prácticas

1. Usar Variables de Entorno

Nunca hardcodees API keys en el código:

```
var apiKey = Environment.GetEnvironmentVariable("AZURE_OPENAI_KEY")
?? throw new InvalidOperationException("API key no configurada");
```

2. Configurar Timeouts

```
var httpClient = new HttpClient
{
    Timeout = TimeSpan.FromSeconds(120)
};

builder.AddAzureOpenAIChatCompletion(
    deploymentName: "gpt-4",
    endpoint: endpoint,
    apiKey: apiKey,
    httpClient: httpClient);
```

3. Logging Estructurado

```
builder.Services.AddLogging(c =>
{
    c.AddConsole();
    c.SetMinimumLevel(LogLevel.Information);
});
```

4. Validar Entradas

```
if (string.IsNullOrWhiteSpace(userInput))
{
    throw new ArgumentException("La entrada no puede estar vacía");
}
```

Siguientes Pasos

Ahora que dominas lo básico de Semantic Kernel, puedes:

1. Crear funciones personalizadas (Skills/Plugins)
2. Implementar memoria y contexto para conversaciones
3. Usar embeddings para búsqueda semántica
4. Orquestar llamadas múltiples al modelo
5. Integrar con bases de datos vectoriales

Recursos Adicionales

- Documentación oficial de Semantic Kernel
- GitHub: Semantic Kernel
- Azure OpenAI Service

Conclusión

Semantic Kernel simplifica la integración de IA en aplicaciones .NET. Con los conceptos básicos que has aprendido, estás listo para construir aplicaciones inteligentes que aprovechen el poder de los modelos de lenguaje. En los siguientes tutoriales profundizaremos en técnicas más avanzadas.

Palabras clave: Semantic Kernel, C#, .NET, Azure OpenAI, GPT-4, LLM, inteligencia artificial, tutorial, programación

Cierre del paso

Verificación rápida

- La app imprime una respuesta del modelo sin excepciones.

- El logging muestra que el request llega y vuelve (sin timeouts).

Errores frecuentes y cómo desbloquearte

- 401/403: endpoint/clave incorrectos o sin permisos.
- 404: deployment name incorrecto (no coincide con el desplegado).
- Timeouts: red, DNS corporativo o límites de firewall.

Ejercicios (para afianzar)

- Cambia el mensaje del sistema para forzar un estilo de respuesta.
- Añade manejo de errores y reintentos básicos (con backoff).

Siguiente paso

Cuando estés listo, continúa con el **Paso 02 — Creando Plugins Personalizados en Semantic Kernel**.

Paso 02 — Creando Plugins Personalizados en Semantic Kernel

Mapa del paso

Tiempo estimado: 60–90 min

Resultado que vas a conseguir:

Un plugin reutilizable (funciones nativas y/o semánticas) y su invocación desde el **Kernel**.

Objetivos de aprendizaje:

- Crear plugins (skills) con funciones C# y prompts.
- Entender cómo se importan y versionan plugins.
- Practicar argumentos, validación y logging por función.

Prerrequisitos:

- Haber completado el Paso 01 o tener un **Kernel** funcionando.

Ruta guiada (paso a paso)

1. Crea una clase con métodos que representen capacidades (ej.: calendario, texto, utilidades).
2. Anota/expón funciones para que Semantic Kernel pueda descubrirlas.
3. Importa el plugin en el **Kernel** y lista funciones disponibles.
4. Invoca funciones con **KernelArguments** y valida entradas.
5. Añade una función desde prompt para comparar enfoques.

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

Introducción

Los plugins (anteriormente llamados Skills) son componentes reutilizables en Semantic Kernel que encapsulan funcionalidad específica. Aprenderás a crear plugins que pueden ser invocados por el kernel o por el propio LLM para realizar tareas específicas.

¿Qué son los Plugins?

Un plugin es una colección de funciones que:

- Realizan tareas específicas (buscar datos, realizar cálculos, llamar APIs)
- Pueden ser invocadas por el LLM automáticamente
- Son reutilizables en diferentes contextos
- Combinan código tradicional con capacidades de IA

Creando tu Primer Plugin

Plugin Simple: Calculadora

```
using System.ComponentModel;
using Microsoft.SemanticKernel;

public class CalculadoraPlugin
{
    [KernelFunction("sumar")]
    [Description("Suma dos números")]
    public double Sumar(
        [Description("Primer número")] double a,
        [Description("Segundo número")] double b)
    {
        return a + b;
    }

    [KernelFunction("multiplicar")]
    [Description("Multiplica dos números")]
    public double Multiplicar(
        [Description("Primer número")] double a,
        [Description("Segundo número")] double b)
    {
        return a * b;
    }

    [KernelFunction("calcular_porcentaje")]
    [Description("Calcula el porcentaje de un número")]
    public double CalcularPorcentaje(
        [Description("Número base")] double numero,
        [Description("Porcentaje a calcular")] double porcentaje)
    {
        return numero * (porcentaje / 100);
    }
}
```

Registrar el Plugin

```
var kernel = Kernel.CreateBuilder()
    .AddAzureOpenAIChatCompletion(deploymentName, endpoint, apiKey)
    .Build();

// Registrar plugin como objeto
kernel.Plugins.AddFromObject(new CalculadoraPlugin());

// Ahora el LLM puede usar estas funciones automáticamente
var response = await kernel.InvokePromptAsync(
    "Calcula el 15% de 200 y luego multiplicalo por 3");

Console.WriteLine(response);
```

Plugin Asíncrono: Búsqueda de Datos

```
using System.ComponentModel;
using Microsoft.SemanticKernel;

public class BuscadorPlugin
{
    private readonly HttpClient _httpClient;
```

```

public BuscadorPlugin(HttpClient httpClient)
{
    _httpClient = httpClient;
}

[KernelFunction("buscar_informacion")]
[Description("Busca información sobre un tema específico")]
public async Task<string> BuscarInformacionAsync(
    [Description("Tema a buscar")] string tema,
    CancellationToken cancellationToken = default)
{
    try
    {
        // Simular búsqueda en una API
        var url = $"https://api.ejemplo.com/search?q={Uri.EscapeDataString(tema)}";
        var response = await _httpClient.GetStringAsync(url, cancellationToken);

        return response;
    }
    catch (Exception ex)
    {
        return $"Error al buscar: {ex.Message}";
    }
}

[KernelFunction("verificar_disponibilidad")]
[Description("Verifica si un servicio está disponible")]
public async Task<bool> VerificarDisponibilidadAsync(
    [Description("URL del servicio")] string url,
    CancellationToken cancellationToken = default)
{
    try
    {
        var response = await _httpClient.GetAsync(url, cancellationToken);
        return response.IsSuccessStatusCode;
    }
    catch
    {
        return false;
    }
}
}

```

Plugin con Dependencias: Acceso a Datos

```

using System.ComponentModel;
using Microsoft.SemanticKernel;
using Microsoft.Extensions.Logging;

public class RepositorioPlugin
{
    private readonly ILogger<RepositorioPlugin> _logger;
    private readonly IDatabaseService _database;

    public RepositorioPlugin(
        ILogger<RepositorioPlugin> logger,
        IDatabaseService database)

```

```

{
    _logger = logger;
    _database = database;
}

[KernelFunction("obtener_usuario")]
[Description("Obtiene información de un usuario por ID")]
public async Task<string> ObtenerUsuarioAsync(
    [Description("ID del usuario")] string userId,
    CancellationToken cancellationToken = default)
{
    _logger.LogInformation("Buscando usuario {UserId}", userId);

    try
    {
        var usuario = await _database.GetUserAsync(userId, cancellationToken);

        if (usuario == null)
        {
            return "Usuario no encontrado";
        }

        return $"Usuario: {usuario.Nombre}, Email: {usuario.Email}";
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error obteniendo usuario {UserId}", userId);
        return "Error al obtener usuario";
    }
}

[KernelFunction("guardar_nota")]
[Description("Guarda una nota en la base de datos")]
public async Task<string> GuardarNotaAsync(
    [Description("Contenido de la nota")] string contenido,
    [Description("Categoría de la nota")] string categoria,
    CancellationToken cancellationToken = default)
{
    _logger.LogInformation("Guardando nota en categoría {Categoria}", categoria);

    try
    {
        var nota = new Nota
        {
            Contenido = contenido,
            Categoria = categoria,
            FechaCreacion = DateTime.UtcNow
        };

        await _database.SaveNoteAsync(nota, cancellationToken);

        return $"Nota guardada exitosamente con ID: {nota.Id}";
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error guardando nota");
        return "Error al guardar la nota";
    }
}

```

```
    }
}
```

Plugin Complejo: Procesamiento de Texto

```
using System.ComponentModel;
using System.Text;
using System.Text.RegularExpressions;
using Microsoft.SemanticKernel;

public class TextoPlugin
{
    [KernelFunction("contar_palabras")]
    [Description("Cuenta las palabras en un texto")]
    public int ContarPalabras([Description("Texto a analizar")] string texto)
    {
        if (string.IsNullOrWhiteSpace(texto))
            return 0;

        return texto.Split(new[] { ' ', '\t', '\n', '\r' },
                           StringSplitOptions.RemoveEmptyEntries).Length;
    }

    [KernelFunction("extraer_emails")]
    [Description("Extrae todos los emails de un texto")]
    public List<string> ExtraerEmails([Description("Texto a procesar")] string texto)
    {
        if (string.IsNullOrWhiteSpace(texto))
            return new List<string>();

        var pattern = @"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b";
        var matches = Regex.Matches(texto, pattern);

        return matches.Select(m => m.Value).Distinct().ToList();
    }

    [KernelFunction("formatear_texto")]
    [Description("Formatea un texto según reglas específicas")]
    public string FormatearTexto(
        [Description("Texto a formatear")] string texto,
        [Description("Tipo de formato: uppercase, lowercase, titlecase")] string formato)
    {
        if (string.IsNullOrWhiteSpace(texto))
            return string.Empty;

        return formato.ToLower() switch
        {
            "uppercase" => texto.ToUpper(),
            "lowercase" => texto.ToLower(),
            "titlecase" => System.Globalization.CultureInfo.CurrentCulture.TextInfo.ToTitleCase(texto)
            _ => texto
        };
    }

    [KernelFunction("resumen_estadistico")]
    [Description("Proporciona estadísticas sobre un texto")]
    public string ResumenEstadistico([Description("Texto a analizar")] string texto)
    {
```

```

        if (string.IsNullOrWhiteSpace(texto))
            return "Texto vacío";

        var palabras = ContarPalabras(texto);
        var caracteres = texto.Length;
        var lineas = texto.Split('\n').Length;
        var emails = ExtraerEmails(texto).Count;

        var sb = new StringBuilder();
        sb.AppendLine($"Palabras: {palabras}");
        sb.AppendLine($"Caracteres: {caracteres}");
        sb.AppendLine($"Lineas: {lineas}");
        sb.AppendLine($"Emails encontrados: {emails}");

        return sb.ToString();
    }
}

```

Registro de Plugins con Inyección de Dependencias

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;

// Configurar servicios
var services = new ServiceCollection();
services.AddHttpClient();
services.AddLogging();
services.AddSingleton<IDatabaseService, DatabaseService>();

// Crear kernel con DI
var builder = Kernel.CreateBuilder();
builder.Services.AddSingleton(services.BuildServiceProvider());

builder.AddAzureOpenAIChatCompletion(deploymentName, endpoint, apiKey);

var kernel = builder.Build();

// Registrar plugins con dependencias resueltas por DI
var serviceProvider = kernel.Services.GetRequiredService<IServiceProvider>();
var httpClient = serviceProvider.GetRequiredService<HttpClient>();
var database = serviceProvider.GetRequiredService<IDatabaseService>();
var logger = serviceProvider.GetRequiredService<ILogger<RepositorioPlugin>>();

kernel.Plugins.AddFromObject(new BuscadorPlugin(httpClient));
kernel.Plugins.AddFromObject(new RepositorioPlugin(logger, database));
kernel.Plugins.AddFromObject(new TextoPlugin());
kernel.Plugins.AddFromObject(new CalculadoraPlugin());

```

Invocación Manual de Funciones

```

// Invocar función específica directamente
var resultado = await kernel.InvokeAsync(
    "CalculadoraPlugin",
    "sumar",
    new KernelArguments
    {
        ["a"] = 10,
        ["b"] = 20
    }
)

```

```

    });

Console.WriteLine($"Resultado: {resultado}");

// Invocar con tipo fuerte
var textoPlugin = new TextoPlugin();
kernel.Plugins.AddFromObject(textoPlugin, "TextoPlugin");

var estadisticas = await kernel.InvokeAsync(
    "TextoPlugin",
    "resumen_estadistico",
    new KernelArguments
    {
        ["texto"] = "Este es un texto de ejemplo para analizar."
    });

Console.WriteLine(estadisticas);

```

Plugin con Estado: Sesión de Usuario

```

using System.ComponentModel;
using Microsoft.SemanticKernel;

public class SesionPlugin
{
    private readonly Dictionary<string, object> _sessionData = new();

    [KernelFunction("guardar_en_sesion")]
    [Description("Guarda un valor en la sesión del usuario")]
    public string GuardarEnSesion(
        [Description("Clave para el valor")] string clave,
        [Description("Valor a guardar")] string valor)
    {
        _sessionData[clave] = valor;
        return $"Valor guardado con clave '{clave}'";
    }

    [KernelFunction("obtener_de_sesion")]
    [Description("Obtiene un valor de la sesión del usuario")]
    public string ObtenerDeSesion([Description("Clave del valor")] string clave)
    {
        if (_sessionData.TryGetValue(clave, out var valor))
        {
            return valor?.ToString() ?? "Valor nulo";
        }

        return $"No se encontró valor para la clave '{clave}'";
    }

    [KernelFunction("limpiar_sesion")]
    [Description("Limpia todos los datos de la sesión")]
    public string LimpiarSesion()
    {
        var count = _sessionData.Count;
        _sessionData.Clear();
        return $"Sesión limpia. Se eliminaron {count} valores.";
    }
}

```

Mejores Prácticas

1. Descripciones Claras

Las descripciones ayudan al LLM a decidir cuándo usar cada función:

```
[KernelFunction("buscar_productos")]
[Description("Busca productos en el catálogo que coincidan con los criterios especificados")]
public async Task<string> BuscarProductosAsync(
    [Description("Término de búsqueda, puede ser nombre, categoría o descripción")] string busqueda,
    [Description("Precio máximo en euros, opcional")] double? precioMax = null)
{
    // Implementación
}
```

2. Manejo de Errores Robusto

```
[KernelFunction("procesar_pedido")]
[Description("Procesa un pedido de cliente")]
public async Task<string> ProcesarPedidoAsync(
    string pedidoId,
    CancellationToken cancellationToken)
{
    try
    {
        // Validación
        if (string.IsNullOrWhiteSpace(pedidoId))
            return "Error: ID de pedido inválido";

        // Procesamiento
        var resultado = await _service.ProcessOrderAsync(pedidoId, cancellationToken);

        return $"Pedido {pedidoId} procesado exitosamente";
    }
    catch (OperationCanceledException)
    {
        return "Operación cancelada por el usuario";
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error procesando pedido {PedidoId}", pedidoId);
        return $"Error al procesar el pedido: {ex.Message}";
    }
}
```

3. Logging para Debugging

```
[KernelFunction("analizar_sentimiento")]
[Description("Analiza el sentimiento de un texto")]
public async Task<string> AnalizarSentimientoAsync(
    string texto,
    CancellationToken cancellationToken)
{
    _logger.LogInformation("Analizando sentimiento de texto con {Length} caracteres", texto?.Length

    var resultado = await _sentimentService.AnalyzeAsync(texto, cancellationToken);

    _logger.LogInformation("Sentimiento detectado: {Sentiment} (confianza: {Confidence})",
        resultado.Sentiment, resultado.Confidence);
```

```

    return resultado.ToString();
}

4. Parámetros Opcionales

[KernelFunction("buscar_articulos")]
[Description("Busca artículos en el blog")]
public async Task<string> BuscarArticulosAsync(
    [Description("Término de búsqueda")] string termino,
    [Description("Categoría opcional para filtrar")] string? categoria = null,
    [Description("Número máximo de resultados, default 10")] int limite = 10,
    CancellationToken cancellationToken = default)
{
    // Implementación con parámetros opcionales
}

```

Conclusión

Los plugins son fundamentales para extender Semantic Kernel con funcionalidad personalizada. Te permiten combinar la inteligencia de los LLMs con lógica de negocio específica, acceso a datos y servicios externos. Con los conceptos aprendidos puedes crear plugins robustos y reutilizables para tus aplicaciones de IA.

Palabras clave: Semantic Kernel plugins, C# skills, KernelFunction, custom functions, AI integration, .NET

Cierre del paso

Verificación rápida

- Puedes invocar al menos 2 funciones del plugin y ver resultados coherentes.
- Al pasar argumentos inválidos, recibes errores claros y controlados.

Errores frecuentes y cómo desbloquearte

- Nombres de función duplicados o colisiones de plugin.
- Falta de validación de entradas (provoca respuestas basura).

Ejercicios (para afianzar)

- Crea un plugin de “normalización de texto” y úsalo antes de llamar al LLM.
- Versiona tu plugin con una interfaz y una implementación alternativa.

Siguiente paso

Cuando estés listo, continúa con el **Paso 03 — Clasificación de Intenciones con LLMs en .NET**.

Paso 03 — Clasificación de Intenciones con LLMs en .NET

Mapa del paso

Tiempo estimado: 75–120 min

Resultado que vas a conseguir:

Un clasificador de intenciones que devuelve JSON estructurado con confianza y entidades.

Objetivos de aprendizaje:

- Diseñar un prompt robusto para clasificación.
- Forzar salida estructurada (JSON) y validarla.
- Aplicar guardrails mínimos (umbral de confianza, intents permitidas).

Prerrequisitos:

- Kernel con chat completion configurado.
- Conocimientos básicos de serialización JSON en .NET.

Ruta guiada (paso a paso)

1. Define el contrato de salida (`IntentClassificationResult`) y una lista cerrada de intents.
 2. Construye un system prompt que describa reglas y ejemplos.
 3. Solicita respuesta JSON y parsea el resultado con `System.Text.Json`.
 4. Aplica guardrails: umbral de confianza y fallback a `unknown`.
 5. Registra métricas/logs para entender fallos y ambigüedad.
-

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

Introducción

La clasificación de intenciones es fundamental en aplicaciones conversacionales. Permite determinar qué quiere hacer el usuario basándose en su mensaje. En este tutorial aprenderás a implementar un sistema robusto de clasificación de intenciones usando Semantic Kernel y Azure OpenAI.

¿Qué es la Clasificación de Intenciones?

La clasificación de intenciones determina la **acción** que el usuario desea realizar. Por ejemplo:

- “Quiero comprar un libro” → Intención: `comprar`
- “¿Cuál es el estado de mi pedido?” → Intención: `consultar_estado`
- “Necesito ayuda” → Intención: `solicitar_ayuda`

Arquitectura del Sistema

Un clasificador de intenciones efectivo tiene estos componentes:

1. **Servicio de Clasificación:** Procesa el mensaje y determina la intención
2. **Sistema de Prompts:** Define las instrucciones para el LLM
3. **Modelo de Resultado:** Estructura que contiene la intención y metadatos
4. **Guardrails:** Validaciones para garantizar resultados confiables

Implementación del Servicio

Modelo de Resultado

```
public record IntentClassificationResult
{
    public required string Intent { get; init; }
    public double Confidence { get; init; }
    public Dictionary<string, object>? Entities { get; init; }
    public string? Category { get; init; }
}

// Constantes para intenciones
public static class Intents
{
    public const string BuyProduct = "buy_product";
    public const string CheckStatus = "check_status";
```

```

    public const string RequestHelp = "request_help";
    public const string CancelOrder = "cancel_order";
    public const string Unknown = "unknown";
}

```

Servicio de Clasificación

```

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.Extensions.Logging;
using System.Text.Json;

public class IntentClassificationService
{
    private readonly Kernel _kernel;
    private readonly ILogger<IntentClassificationService> _logger;

    private const double MinimumConfidenceThreshold = 0.65;

    public IntentClassificationService(
        Kernel kernel,
        ILogger<IntentClassificationService> logger)
    {
        _kernel = kernel;
        _logger = logger;
    }

    public async Task<IntentClassificationResult> ClassifyAsync(
        string messageText,
        CancellationToken cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(messageText))
        {
            return CreateUnknownResult("Mensaje vacío");
        }

        try
        {
            var chatService = _kernel.GetRequiredService<IChatCompletionService>();

            var systemPrompt = BuildSystemPrompt();
            var userPrompt = BuildUserPrompt(messageText);

            var chatHistory = new ChatHistory();
            chatHistory.AddSystemMessage(systemPrompt);
            chatHistory.AddUserMessage(userPrompt);

            // Configuración para respuesta JSON estructurada
            var settings = new OpenAIPromptExecutionSettings
            {
                Temperature = 0.1, // Baja temperatura para clasificación consistente
                ResponseFormat = "json_object"
            };

            var response = await chatService.GetChatMessageContentAsync(
                chatHistory,
                settings,

```

```

        _kernel,
        cancellationToken);

    var jsonContent = response.Content ?? string.Empty;
    _logger.LogDebug("Respuesta de clasificación: {Response}", jsonContent);

    var result = ParseClassificationResponse(jsonContent, messageText);
    result = ApplyGuardrails(result, messageText);

    _logger.LogInformation(
        "Intención clasificada: {Intent} (confianza: {Confidence:F2})",
        result.Intent, result.Confidence);

    return result;
}
catch (Exception ex)
{
    _logger.LogError(ex, "Error clasificando intención");
    return CreateUnknownResult("Error en clasificación");
}
}

private string BuildSystemPrompt()
{
    return """
Eres un clasificador experto de intenciones para un sistema de e-commerce.

Tu tarea es determinar QUÉ quiere hacer el usuario.

Tipos de intención (en orden de prioridad):
1. **buy_product**: Usuario quiere comprar algo (ej: "quiero comprar un libro", "añadir al carrito")
2. **check_status**: Usuario consulta el estado de algo (ej: "¿dónde está mi pedido?", "está pendiente")
3. **cancel_order**: Usuario quiere cancelar (ej: "cancelar mi pedido", "no quiero el producto")
4. **request_help**: Usuario pide ayuda o información (ej: "ayuda", "¿cómo funciona?", "sopor")
5. **unknown**: No está clara la intención

REGLAS CRÍTICAS:
- Analiza el CONTEXTO completo del mensaje
- Si detectas producto mencionado en compra, extrae el nombre en "product_name"
- Si hay número de pedido, extráelo en "order_id"
- Confidence debe ser 0.0-1.0 basado en tu certeza
- Si la intención es ambigua, usa "unknown"

Responde SOLO con JSON válido:
{
    "intent": "buy_product|check_status|cancel_order|request_help|unknown",
    "confidence": 0.0-1.0,
    "entities": {
        "product_name": "string|null",
        "order_id": "string|null",
        "quantity": number|null
    },
    "category": "string|null"
}

Ejemplos:
- "Quiero comprar tres camisetas" → {"intent": "buy_product", "confidence": 0.95, "entities": {"product_name": "camisetas", "quantity": 3}}
- "¿Dónde está mi pedido 12345?" → {"intent": "check_status", "confidence": 0.98, "entities": {"order_id": "12345"}}

```

```

        - "Ayuda con mi cuenta" → {"intent":"request_help","confidence":0.90,"entities":{}}
    """
}

private string BuildUserPrompt(string messageText)
{
    return $"""
    Clasifica la intención de este mensaje:

    Mensaje: "{messageText}"

    Responde con JSON válido siguiendo el esquema.
"""
}
}

private IntentClassificationResult ParseClassificationResponse(
    string jsonContent,
    string originalText)
{
    try
    {
        // Limpiar respuesta de markdown si existe
        jsonContent = jsonContent.Trim();
        if (jsonContent.StartsWith("```json"))
            jsonContent = jsonContent[7..];
        if (jsonContent.StartsWith("```"))
            jsonContent = jsonContent[3..];
        if (jsonContent.EndsWith("```"))
            jsonContent = jsonContent[..^3];
        jsonContent = jsonContent.Trim();

        var options = new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        };

        var parsed = JsonSerializer.Deserialize<JsonClassificationResult>(
            jsonContent, options);

        if (parsed == null)
            throw new JsonException("Resultado nulo");

        return new IntentClassificationResult
        {
            Intent = NormalizeIntent(parsed.Intent),
            Confidence = parsed.Confidence,
            Entities = parsed.Entities,
            Category = parsed.Category
        };
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, "Error parseando JSON: {Json}", jsonContent);
        return CreateUnknownResult("Error parsing JSON");
    }
}

private IntentClassificationResult ApplyGuardrails(

```

```

IntentClassificationResult result,
string originalText)
{
    // Guardrail 1: Confianza baja → marcar como unknown
    if (result.Confidence < MinimumConfidenceThreshold)
    {
        _logger.LogInformation(
            "Confianza baja ({Confidence:F2}) para '{Text}', marcando como unknown",
            result.Confidence, originalText);

        return result with { Intent = Intents.Unknown };
    }

    // Guardrail 2: Validar entidades requeridas
    if (result.Intent == Intents.BuyProduct)
    {
        if (result.Entities == null ||
            !result.Entities.ContainsKey("product_name"))
        {
            _logger.LogWarning(
                "buy_product sin product_name para '{Text}'", originalText);
            return result with { Intent = Intents.Unknown };
        }
    }

    return result;
}

private string NormalizeIntent(string? intent)
{
    if (string.IsNullOrWhiteSpace(intent))
        return Intents.Unknown;

    return intent.ToLowerInvariant().Replace("_", "").Replace("-", "") switch
    {
        "buyproduct" or "buy" or "purchase" => Intents.BuyProduct,
        "checkstatus" or "status" => Intents.CheckStatus,
        "cancelorder" or "cancel" => Intents.CancelOrder,
        "requesthelp" or "help" or "ayuda" => Intents.RequestHelp,
        _ => Intents.Unknown
    };
}

private IntentClassificationResult CreateUnknownResult(string reason)
{
    _logger.LogInformation("Creando resultado unknown: {Reason}", reason);

    return new IntentClassificationResult
    {
        Intent = Intents.Unknown,
        Confidence = 0.0,
        Entities = null,
        Category = null
    };
}

// Modelo interno para deserialización
private class JsonClassificationResult

```

```

    {
        public string? Intent { get; set; }
        public double Confidence { get; set; }
        public Dictionary<string, object>? Entities { get; set; }
        public string? Category { get; set; }
    }
}

```

Uso del Clasificador

Ejemplo Básico

```

var classifier = new IntentClassificationService(kernel, logger);

var result = await classifier.ClassifyAsync("Quiero comprar un libro de programación");

Console.WriteLine($"Intención: {result.Intent}");
Console.WriteLine($"Confianza: {result.Confidence:F2}");

if (result.Entities != null && result.Entities.ContainsKey("product_name"))
{
    Console.WriteLine($"Producto: {result.Entities["product_name"]}");
}

```

Integración con Router

```

public class MessageRouter
{
    private readonly IntentClassificationService _classifier;
    private readonly Dictionary<string, Func<IntentClassificationResult, Task<string>>> _handlers;

    public MessageRouter(IntentClassificationService classifier)
    {
        _classifier = classifier;
        _handlers = new()
        {
            [Intents.BuyProduct] = HandleBuyProduct,
            [Intents.CheckStatus] = HandleCheckStatus,
            [Intents.CancelOrder] = HandleCancelOrder,
            [Intents.RequestHelp] = HandleRequestHelp,
            [Intents.Unknown] = HandleUnknown
        };
    }

    public async Task<string> RouteMessageAsync(string message)
    {
        var classification = await _classifier.ClassifyAsync(message);

        if (_handlers.TryGetValue(classification.Intent, out var handler))
        {
            return await handler(classification);
        }

        return "No pude procesar tu mensaje. ¿Puedes reformularlo?";
    }

    private async Task<string> HandleBuyProduct(IntentClassificationResult result)
    {
        if (result.Entities?.ContainsKey("product_name") == true)

```

```

    {
        var productName = result.Entities["product_name"].ToString();
        return $"Perfecto, te ayudo a comprar {productName}. ¿Cuántas unidades necesitas?";
    }

    return "¿Qué producto te gustaría comprar?";
}

private async Task<string> HandleCheckStatus(IntentClassificationResult result)
{
    if (result.Entities?.ContainsKey("order_id") == true)
    {
        var orderId = result.Entities["order_id"].ToString();
        return $"Buscando información del pedido {orderId}...";
    }

    return "Por favor, proporciona tu número de pedido para consultar el estado.";
}

private async Task<string> HandleCancelOrder(IntentClassificationResult result)
{
    return "Entiendo que quieres cancelar un pedido. ¿Puedes darme el número de pedido?";
}

private async Task<string> HandleRequestHelp(IntentClassificationResult result)
{
    return "¡Estoy aquí para ayudarte! ¿Qué necesitas saber?";
}

private async Task<string> HandleUnknown(IntentClassificationResult result)
{
    return "No estoy seguro de entender. ¿Puedes explicar qué necesitas?";
}
}

```

Clasificación Multi-Intención

Para casos más complejos donde un mensaje puede tener múltiples intenciones:

```

public class MultiIntentClassificationService
{
    private readonly Kernel _kernel;

    public async Task<List<IntentClassificationResult>> ClassifyMultipleAsync(
        string messageText,
        CancellationToken cancellationToken = default)
    {
        var systemPrompt = """
        Analiza el mensaje y extrae TODAS las intenciones presentes.
        Un mensaje puede tener múltiples intenciones.

        Por ejemplo: "Quiero comprar un libro y cancelar mi pedido anterior"
        Tiene dos intenciones: buy_product y cancel_order

        Responde con JSON array:
        [
            {"intent": "buy_product", "confidence": 0.95, "entities": {"product_name": "libro"}},
            {"intent": "cancel_order", "confidence": 0.90, "entities": {}}
        ]
    }
}

```

```

""";  
  

var chatService = _kernel.GetRequiredService<IChatCompletionService>();  

var chatHistory = new ChatHistory();  

chatHistory.AddSystemMessage(systemPrompt);  

chatHistory.AddUserMessage($"Mensaje: {messageText}");  
  

var settings = new OpenAIPromptExecutionSettings  

{  

    Temperature = 0.1,  

    ResponseFormat = "json_object"  

};  
  

var response = await chatService.GetChatMessageContentAsync(  

    chatHistory, settings, _kernel, cancellationToken);  
  

// Parsear array de intenciones  

var results = JsonSerializer.Deserialize<List<JsonClassificationResult>>(  

    response.Content ?? "[]");  
  

return results?.Select(r => new IntentClassificationResult  

{  

    Intent = r.Intent ?? Intents.Unknown,  

    Confidence = r.Confidence,  

    Entities = r.Entities,  

    Category = r.Category  

}).ToList() ?? new List<IntentClassificationResult>();  

}  

}
}

```

Clasificación con Contexto

```

public class ContextAwareIntentClassifier  

{  

    private readonly Kernel _kernel;  
  

    public async Task<IntentClassificationResult> ClassifyWithContextAsync(  

        string messageText,  

        List<string> conversationHistory,  

        CancellationToken cancellationToken = default)  

    {  

        var chatHistory = new ChatHistory();  

        chatHistory.AddSystemMessage(BuildSystemPrompt());  
  

        // Agregar historial de conversación  

        foreach (var previousMessage in conversationHistory.TakeLast(5))  

        {  

            chatHistory.AddUserMessage(previousMessage);  

        }  
  

        // Agregar mensaje actual  

        chatHistory.AddUserMessage($"Clasifica esta intención: {messageText}");  
  

        var chatService = _kernel.GetRequiredService<IChatCompletionService>();  

        var response = await chatService.GetChatMessageContentAsync(  

            chatHistory,  

            kernel: _kernel,  

            cancellationToken: cancellationToken);
    }
}

```

```

        return ParseResponse(response.Content ?? "");
    }

    private string BuildSystemPrompt()
    {
        return """
            Clasifica la intención considerando el CONTEXTO de la conversación.

            Si el usuario dice "sí" o "no", determina a qué se refiere basándose
            en los mensajes anteriores.

            Responde con JSON válido.
            """;
    }

    private IntentClassificationResult ParseResponse(string content)
    {
        // Implementación de parsing
        return new IntentClassificationResult
        {
            Intent = Intents.Unknown,
            Confidence = 0.0
        };
    }
}

```

Testing del Clasificador

```

using Xunit;
using Microsoft.Extensions.Logging.Abstractions;

public class IntentClassificationServiceTests
{
    [Theory]
    [InlineData("Quiero comprar un libro", Intents.BuyProduct)]
    [InlineData("¿Dónde está mi pedido?", Intents.CheckStatus)]
    [InlineData("Cancelar mi orden", Intents.CancelOrder)]
    [InlineData("Necesito ayuda", Intents.RequestHelp)]
    public async Task ClassifyAsync_IdentifiesCorrectIntent(
        string message,
        string expectedIntent)
    {
        // Arrange
        var kernel = CreateTestKernel();
        var logger = NullLogger<IntentClassificationService>.Instance;
        var classifier = new IntentClassificationService(kernel, logger);

        // Act
        var result = await classifier.ClassifyAsync(message);

        // Assert
        Assert.Equal(expectedIntent, result.Intent);
        Assert.True(result.Confidence >= 0.65);
    }

    [Fact]
    public async Task ClassifyAsync_ExtractsEntities()

```

```

{
    // Arrange
    var kernel = CreateTestKernel();
    var logger = NullLogger<IntentClassificationService>.Instance;
    var classifier = new IntentClassificationService(kernel, logger);

    // Act
    var result = await classifier.ClassifyAsync("Quiero comprar 3 libros");

    // Assert
    Assert.Equal(Intents.BuyProduct, result.Intent);
    Assert.NotNull(result.Entities);
    Assert.True(result.Entities.ContainsKey("product_name"));
    Assert.Equal("libros", result.Entities["product_name"]);
}

private Kernel CreateTestKernel()
{
    // Configuración de kernel para tests
    var builder = Kernel.CreateBuilder();
    // Configurar con mock o test deployment
    return builder.Build();
}
}

```

Mejores Prácticas

1. Temperatura Baja para Consistencia

```

var settings = new OpenAIPromptExecutionSettings
{
    Temperature = 0.1 // Muy baja para clasificación consistente
};

```

2. Validación de Resultados

```

private bool IsValidResult(IntentClassificationResult result)
{
    if (result.Confidence < 0.0 || result.Confidence > 1.0)
        return false;

    if (string.IsNullOrWhiteSpace(result.Intent))
        return false;

    return true;
}

```

3. Logging Detallado

```

_logger.LogInformation(
    "Clasificación: Intent={Intent}, Confidence={Confidence:F2}, Entities={Entities}",
    result.Intent,
    result.Confidence,
    JsonSerializer.Serialize(result.Entities));

```

4. Caché de Clasificaciones

```

public class CachedIntentClassifier
{
    private readonly IMemoryCache _cache;

```

```

private readonly IntentClassificationService _classifier;

public async Task<IntentClassificationResult> ClassifyAsync(string message)
{
    var cacheKey = $"intent_{message.GetHashCode()}";

    if (_cache.TryGetValue<IntentClassificationResult>(cacheKey, out var cached))
    {
        return cached;
    }

    var result = await _classifier.ClassifyAsync(message);

    _cache.Set(cacheKey, result, TimeSpan.FromMinutes(30));

    return result;
}
}

```

Conclusión

La clasificación de intenciones es crucial para aplicaciones conversacionales efectivas. Con Semantic Kernel y Azure OpenAI puedes crear clasificadores robustos que entienden el contexto y manejan casos complejos. Los guardrails y validaciones aseguran resultados confiables en producción.

Palabras clave: intent classification, NLU, Semantic Kernel, Azure OpenAI, chatbot, conversational AI, C#

Cierre del paso

Verificación rápida

- Mensajes de prueba se clasifican con intent correcta y confianza razonable.
- Respuestas no válidas (JSON roto) terminan en `unknown` sin romper la app.

Errores frecuentes y cómo desbloquearte

- JSON no válido por texto adicional: limpiar fences y validar esquema.
- Temperatura alta: reduce `Temperature` para tareas deterministas.

Ejercicios (para afianzar)

- Añade soporte de idiomas y sinónimos sin cambiar la lista de intents.
- Introduce tests con mocks del servicio de chat (contratos).

Siguiente paso

Cuando estés listo, continúa con el **Paso 04 — Servicios de Chat Completion con Azure OpenAI y Semantic Kernel**.

Paso 04 — Servicios de Chat Completion con Azure OpenAI y Semantic Kernel

Mapa del paso

Tiempo estimado: 60–90 min

Resultado que vas a conseguir:

Un servicio de chat completion con historial, system prompts y settings por petición.

Objetivos de aprendizaje:

- Trabajar con ChatHistory y roles (system/user/assistant).
- Separar plantilla de prompt, settings y lógica de negocio.
- Añadir límites: tokens, truncado, y manejo de contexto.

Prerrequisitos:

- Kernel con chat completion configurado.

Ruta guiada (paso a paso)

1. Crea un ChatService que encapsule ChatHistory.
2. Define un system message estable (políticas del asistente).
3. Añade mensajes del usuario y ejecuta GetChatMessageContentAsync.
4. Controla MaxTokens y Temperature por escenario.
5. Implementa truncado del historial (por tamaño o tokens).

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

Introducción

Los servicios de Chat Completion permiten crear conversaciones naturales con modelos de IA. En este tutorial aprenderás a implementar servicios de chat robustos y escalables usando Azure OpenAI y Semantic Kernel.

Conceptos Fundamentales

Chat History

El historial de chat mantiene el contexto de la conversación:

```
using Microsoft.SemanticKernel.ChatCompletion;

var chatHistory = new ChatHistory();

// Mensaje del sistema: define el comportamiento del asistente
chatHistory.AddSystemMessage("Eres un experto en programación .NET.");

// Mensajes del usuario
chatHistory.AddUserMessage("¿Qué es async/await?");

// Respuestas del asistente
chatHistory.AddAssistantMessage("async/await es un patrón para programación asíncrona...");
```

Servicio de Chat Completion

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;

var kernel = Kernel.CreateBuilder()
    .AddAzureOpenAIChatCompletion(
        deploymentName: "gpt-4",
        endpoint: "https://tu-recurso.openai.azure.com/",
        apiKey: "tu-api-key")
    .Build();
```

```

var chatService = kernel.GetRequiredService<IChatCompletionService>();

var response = await chatService.GetChatMessageContentAsync(
    chatHistory,
    kernel: kernel);

Console.WriteLine(response.Content);

```

Implementación de un Servicio de Chat Completo

Modelo de Conversación

```

public class ConversationMessage
{
    public required string Role { get; init; } // "user", "assistant", "system"
    public required string Content { get; init; }
    public DateTime Timestamp { get; init; } = DateTime.UtcNow;
    public Dictionary<string, object>? Metadata { get; init; }
}

public class Conversation
{
    public string Id { get; init; } = Guid.NewGuid().ToString();
    public List<ConversationMessage> Messages { get; init; } = new();
    public DateTime CreatedAt { get; init; } = DateTime.UtcNow;
    public DateTime LastUpdated { get; set; } = DateTime.UtcNow;
    public Dictionary<string, object>? Context { get; set; }
}

```

Servicio de Chat con Estado

```

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.Extensions.Logging;

public class ChatService
{
    private readonly Kernel _kernel;
    private readonly ILogger<ChatService> _logger;
    private readonly Dictionary<string, Conversation> _conversations;
    private readonly string _systemPrompt;

    public ChatService(
        Kernel kernel,
        ILogger<ChatService> logger,
        string systemPrompt = "Eres un asistente útil y amigable.")
    {
        _kernel = kernel;
        _logger = logger;
        _conversations = new Dictionary<string, Conversation>();
        _systemPrompt = systemPrompt;
    }

    public string CreateConversation(Dictionary<string, object>? context = null)
    {
        var conversation = new Conversation
        {

```

```

        Context = context
    };

    _conversations[conversation.Id] = conversation;

    _logger.LogInformation("Conversación creada: {ConversationId}", conversation.Id);

    return conversation.Id;
}

public async Task<string> SendMessageAsync(
    string conversationId,
    string message,
    CancellationToken cancellationToken = default)
{
    if (!_conversations.TryGetValue(conversationId, out var conversation))
    {
        throw new InvalidOperationException($"Conversación {conversationId} no encontrada");
    }

    try
    {
        // Agregar mensaje del usuario
        conversation.Messages.Add(new ConversationMessage
        {
            Role = "user",
            Content = message
        });

        // Construir historial de chat
        var chatHistory = BuildChatHistory(conversation);

        // Obtener respuesta del modelo
        var chatService = _kernel.GetRequiredService<IChatCompletionService>();

        var settings = new OpenAIPromptExecutionSettings
        {
            Temperature = 0.7,
            MaxTokens = 800,
            TopP = 0.9
        };

        var response = await chatService.GetChatMessageContentAsync(
            chatHistory,
            settings,
            _kernel,
            cancellationToken);

        var assistantMessage = response.Content ?? string.Empty;

        // Guardar respuesta del asistente
        conversation.Messages.Add(new ConversationMessage
        {
            Role = "assistant",
            Content = assistantMessage,
            Metadata = new Dictionary<string, object>
            {
                ["model"] = response.ModelId ?? "unknown",
            }
        });
    }
}

```

```

        ["tokens"] = response.Metadata?.ContainsKey("Usage") == true
            ? response.Metadata["Usage"]
            : null
    }
});

conversation.LastUpdated = DateTime.UtcNow;

_logger.LogInformation(
    "Mensaje procesado en conversación {ConversationId}. Tokens: {Tokens}",
    conversationId,
    response.Metadata?.ContainsKey("Usage") == true
        ? response.Metadata["Usage"]
        : "N/A");

return assistantMessage;
}
catch (Exception ex)
{
    _logger.LogError(ex, "Error procesando mensaje en conversación {ConversationId}",
        conversationId);
    throw;
}
}

public Conversation? GetConversation(string conversationId)
{
    return _conversations.TryGetValue(conversationId, out var conversation)
        ? conversation
        : null;
}

public void ClearConversation(string conversationId)
{
    if (_conversations.TryGetValue(conversationId, out var conversation))
    {
        conversation.Messages.Clear();
        conversation.LastUpdated = DateTime.UtcNow;

        _logger.LogInformation("Conversación {ConversationId} limpiada", conversationId);
    }
}

public void DeleteConversation(string conversationId)
{
    _conversations.Remove(conversationId);
    _logger.LogInformation("Conversación {ConversationId} eliminada", conversationId);
}

private ChatHistory BuildChatHistory(Conversation conversation)
{
    var chatHistory = new ChatHistory();

    // Agregar mensaje del sistema
    chatHistory.AddSystemMessage(_systemPrompt);

    // Agregar contexto si existe
    if (conversation.Context != null && conversation.Context.Count > 0)

```

```

    {
        var contextInfo = string.Join(", ",
            conversation.Context.Select(kvp => $"{kvp.Key}: {kvp.Value}"));
        chatHistory.AddSystemMessage($"Contexto: {contextInfo}");
    }

// Agregar mensajes de la conversación (últimos N mensajes para no exceder límite)
var recentMessages = conversation.Messages.TakeLast(10);

foreach (var msg in recentMessages)
{
    switch (msg.Role.ToLower())
    {
        case "user":
            chatHistory.AddUserMessage(msg.Content);
            break;
        case "assistant":
            chatHistory.AddAssistantMessage(msg.Content);
            break;
        case "system":
            chatHistory.AddSystemMessage(msg.Content);
            break;
    }
}

return chatHistory;
}
}

```

Streaming de Respuestas

Para respuestas en tiempo real:

```

public class StreamingChatService
{
    private readonly Kernel _kernel;

    public async IAsyncEnumerable<string> StreamMessageAsync(
        string conversationId,
        string message,
        [EnumeratorCancellation] CancellationToken cancellationToken = default)
    {
        var chatService = _kernel.GetRequiredService<IChatCompletionService>();
        var chatHistory = BuildChatHistory(conversationId);

        chatHistory.AddUserMessage(message);

        var settings = new OpenAIPromptExecutionSettings
        {
            Temperature = 0.7,
            MaxTokens = 800
        };

        var fullResponse = new StringBuilder();

        await foreach (var chunk in chatService.GetStreamingChatMessageContentsAsync(
            chatHistory,
            settings,
            _kernel,

```

```

        cancellationToken))
    {
        var content = chunk.Content ?? string.Empty;
        fullResponse.Append(content);
        yield return content;
    }

    // Guardar mensaje completo después del streaming
    SaveAssistantMessage(conversationId, fullResponse.ToString());
}

private void SaveAssistantMessage(string conversationId, string message)
{
    // Implementación para guardar el mensaje
}
}

```

Chat con Funciones (Function Calling)

```

public class FunctionCallingChatService
{
    private readonly Kernel _kernel;

    public FunctionCallingChatService(Kernel kernel)
    {
        _kernel = kernel;

        // Registrar funciones disponibles
        _kernel.Plugins.AddFromObject(new WeatherPlugin());
        _kernel.Plugins.AddFromObject(new CalculatorPlugin());
    }

    public async Task<string> ChatWithFunctionsAsync(
        string message,
        CancellationToken cancellationToken = default)
    {
        var chatService = _kernel.GetRequiredService<IChatCompletionService>();
        var chatHistory = new ChatHistory();

        chatHistory.AddSystemMessage("""
            Eres un asistente que puede usar funciones para responder preguntas.
            Tienes acceso a:
            - WeatherPlugin: para obtener información del clima
            - CalculatorPlugin: para realizar cálculos

            Usa las funciones cuando sea necesario.
            """);

        chatHistory.AddUserMessage(message);

        var settings = new OpenAIPromptExecutionSettings
        {
            Temperature = 0.7,
            ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
        };

        var response = await chatService.GetChatMessageContentAsync(
            chatHistory,

```

```

        settings,
        _kernel,
        cancellationToken);

    return response.Content ?? "No pude generar una respuesta";
}
}

// Plugins de ejemplo
public class WeatherPlugin
{
    [KernelFunction("get_weather")]
    [Description("Obtiene el clima actual de una ciudad")]
    public string GetWeather([Description("Nombre de la ciudad")] string city)
    {
        // Simular llamada a API de clima
        return $"En {city}: 22°C, soleado";
    }
}

public class CalculatorPlugin
{
    [KernelFunction("calculate")]
    [Description("Realiza un cálculo matemático")]
    public double Calculate(
        [Description("Primera operando")] double a,
        [Description("Operador: +, -, *, /")] string op,
        [Description("Segundo operando")] double b)
    {
        return op switch
        {
            "+" => a + b,
            "-" => a - b,
            "*" => a * b,
            "/" => a / b,
            _ => 0
        };
    }
}

```

Chat Multimodal (Texto e Imágenes)

```

using Microsoft.SemanticKernel.ChatCompletion;

public class MultimodalChatService
{
    private readonly Kernel _kernel;

    public async Task<string> AnalyzeImageAsync(
        string imageUrl,
        string question,
        CancellationToken cancellationToken = default)
    {
        var chatService = _kernel.GetRequiredService<IChatCompletionService>();
        var chatHistory = new ChatHistory();

        chatHistory.AddSystemMessage("Eres un experto en análisis de imágenes.");
    }
}

```

```

    // Agregar imagen y pregunta
    var message = new ChatMessageContent(
        AuthorRole.User,
        new ChatMessageContentItemCollection
    {
        new TextContent(question),
        new ImageContent(new Uri(imageUrl))
    });

    chatHistory.Add(message);

    var response = await chatService.GetChatMessageContentAsync(
        chatHistory,
        kernel: _kernel,
        cancellationToken: cancellationToken);

    return response.Content ?? "No pude analizar la imagen";
}
}
}

```

Chat con Memoria Persistente

```

using System.Text.Json;

public class PersistentChatService
{
    private readonly ChatService _chatService;
    private readonly string _storageDirectory;

    public PersistentChatService(ChatService chatService, string storageDirectory)
    {
        _chatService = chatService;
        _storageDirectory = storageDirectory;

        Directory.CreateDirectory(storageDirectory);
    }

    public async Task SaveConversationAsync(string conversationId)
    {
        var conversation = _chatService.GetConversation(conversationId);
        if (conversation == null)
            throw new InvalidOperationException("Conversación no encontrada");

        var filePath = Path.Combine(_storageDirectory, $"{conversationId}.json");
        var json = JsonSerializer.Serialize(conversation, new JsonSerializerOptions
        {
            WriteIndented = true
        });

        await File.WriteAllTextAsync(filePath, json);
    }

    public async Task<string> LoadConversationAsync(string conversationId)
    {
        var filePath = Path.Combine(_storageDirectory, $"{conversationId}.json");

        if (!File.Exists(filePath))
            throw new FileNotFoundException("Conversación no encontrada");
    }
}

```

```

var json = await File.ReadAllTextAsync(filePath);
var conversation = JsonSerializer.Deserialize<Conversation>(json);

if (conversation == null)
    throw new InvalidOperationException("Error deserializando conversación");

// Cargar en el servicio de chat
// Implementación específica según tu arquitectura

return conversationId;
}

public async Task<List<string>> ListConversationsAsync()
{
    var files = Directory.GetFiles(_storageDirectory, "*.json");
    return files.Select(Path.GetFileNameWithoutExtension).ToList()!;
}
}

```

Manejo de Rate Limits

```

public class RateLimitedChatService
{
    private readonly ChatService _chatService;
    private readonly SemaphoreSlim _semaphore;
    private readonly int _maxConcurrent;

    public RateLimitedChatService(ChatService chatService, int maxConcurrent = 5)
    {
        _chatService = chatService;
        _maxConcurrent = maxConcurrent;
        _semaphore = new SemaphoreSlim(maxConcurrent, maxConcurrent);
    }

    public async Task<string> SendMessageAsync(
        string conversationId,
        string message,
        CancellationToken cancellationToken = default)
    {
        await _semaphore.WaitAsync(cancellationToken);

        try
        {
            return await _chatService.SendMessageAsync(
                conversationId,
                message,
                cancellationToken);
        }
        finally
        {
            _semaphore.Release();
        }
    }
}

```

Retry con Exponential Backoff

```
using Polly;
using Polly.Retry;

public class ResilientChatService
{
    private readonly ChatService _chatService;
    private readonly AsyncRetryPolicy _retryPolicy;

    public ResilientChatService(ChatService chatService)
    {
        _chatService = chatService;

        _retryPolicy = Policy
            .Handle<HttpRequestException>()
            .Or<TimeoutException>()
            .WaitAndRetryAsync(
                retryCount: 3,
                sleepDurationProvider: attempt => TimeSpan.FromSeconds(Math.Pow(2, attempt)),
                onRetry: (exception, timeSpan, retryCount, context) =>
                {
                    Console.WriteLine($"Intento {retryCount} después de {timeSpan.TotalSeconds}s");
                });
    }

    public async Task<string> SendMessageWithRetryAsync(
        string conversationId,
        string message,
        CancellationToken cancellationToken = default)
    {
        return await _retryPolicy.ExecuteAsync(async () =>
            await _chatService.SendMessageAsync(conversationId, message, cancellationToken));
    }
}
```

Testing de Servicios de Chat

```
using Xunit;
using Moq;

public class ChatServiceTests
{
    [Fact]
    public async Task SendMessageAsync_ReturnsResponse()
    {
        // Arrange
        var kernel = CreateTestKernel();
        var logger = Mock.Of<ILogger<ChatService>>();
        var chatService = new ChatService(kernel, logger);

        var conversationId = chatService.CreateConversation();

        // Act
        var response = await chatService.SendMessageAsync(
            conversationId,
            "Hola, ¿cómo estás?");

        // Assert
    }
}
```

```

        Assert.NotNull(response);
        Assert.NotEmpty(response);
    }

    [Fact]
    public void CreateConversation_GeneratesUniqueId()
    {
        // Arrange
        var kernel = CreateTestKernel();
        var logger = Mock.Of<ILogger<ChatService>>();
        var chatService = new ChatService(kernel, logger);

        // Act
        var id1 = chatService.CreateConversation();
        var id2 = chatService.CreateConversation();

        // Assert
        Assert.NotEqual(id1, id2);
    }

    private Kernel CreateTestKernel()
    {
        var builder = Kernel.CreateBuilder();
        // Configurar kernel de prueba
        return builder.Build();
    }
}

```

Mejores Prácticas

1. Limitar Historial de Mensajes

```

var recentMessages = conversation.Messages
    .TakeLast(10) // Solo últimos 10 mensajes
    .ToList();

```

2. Validar Longitud de Mensajes

```

public async Task<string> SendMessageAsync(string conversationId, string message)
{
    const int maxLength = 4000;

    if (message.Length > maxLength)
    {
        throw new ArgumentException($"Mensaje excede {maxLength} caracteres");
    }

    // Procesar mensaje
}

```

3. Sanitizar Entrada del Usuario

```

private string SanitizeInput(string input)
{
    // Eliminar caracteres peligrosos o no deseados
    return input
        .Replace("<script>", "")
        .Replace("</script>", "")
        .Trim();
}

```

}

4. Monitorear Uso de Tokens

```
private void LogTokenUsage(ChatMessageContent response)
{
    if (response.Metadata?.ContainsKey("Usage") == true)
    {
        var usage = response.Metadata["Usage"];
        _logger.LogInformation("Tokens usados: {Usage}", usage);
    }
}
```

Conclusión

Los servicios de chat completion son la base de aplicaciones conversacionales modernas. Con Semantic Kernel puedes crear servicios robustos que manejan estado, funciones, streaming y más. Las prácticas de manejo de errores, rate limiting y persistencia aseguran un sistema confiable en producción.

Palabras clave: chat completion, Azure OpenAI, Semantic Kernel, conversational AI, chatbot, streaming, function calling, C#

Cierre del paso

Verificación rápida

- La conversación mantiene contexto entre turnos (sin crecer indefinidamente).
- Puedes variar settings y ver cambios en estilo/longitud.

Errores frecuentes y cómo desbloquearte

- Historial infinito: truncar o resumir para evitar costes/latencia.
- System prompts inconsistentes: centralízalos.

Ejercicios (para afianzar)

- Añade un comando /reset que reinicie historial.
- Implementa un resumen automático cada N mensajes.

Siguiente paso

Cuando estés listo, continúa con el **Paso 05 — Vector Embeddings y Búsqueda Semántica con .NET**.

Paso 05 — Vector Embeddings y Búsqueda Semántica con .NET

Mapa del paso

Tiempo estimado: 75–120 min

Resultado que vas a conseguir:

Una búsqueda semántica básica con embeddings y ranking por similitud.

Objetivos de aprendizaje:

- Generar embeddings para textos y consultas.
- Calcular similitud coseno y rankear resultados.
- Diseñar un mini-índice en memoria como primer prototipo.

Prerrequisitos:

- Deployment de embeddings en Azure/OpenAI.
- Comprender que embeddings != chat: son vectores numéricos.

Ruta guiada (paso a paso)

1. Configura el servicio de embeddings en el Kernel.
 2. Crea un método que genere embeddings para un texto.
 3. Indexa una lista de documentos (texto + embedding).
 4. Calcula similitud coseno entre query y documentos y ordena.
 5. Devuelve top-k con score y muestra resultados.
-

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

Introducción

Los embeddings vectoriales transforman texto en representaciones numéricas que capturan significado semántico. Esto permite buscar contenido por similitud de significado, no solo por palabras clave. En este tutorial aprenderás a implementar búsqueda semántica con Azure OpenAI y Semantic Kernel.

¿Qué son los Embeddings?

Un embedding es una representación vectorial (array de números) de texto. Textos con significados similares tienen vectores similares:

- “El gato está durmiendo” → [0.23, -0.45, 0.67, ...]
- “El felino descansa” → [0.25, -0.43, 0.69, ...] (similar)
- “Python es un lenguaje” → [0.89, 0.12, -0.34, ...] (diferente)

Configuración de Embeddings

Instalar Paquetes

```
dotnet add package Microsoft.SemanticKernel
dotnet add package System.Numerics.Tensors
```

Configurar Servicio de Embeddings

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Embeddings;

var builder = Kernel.CreateBuilder();

// Configurar servicio de text embeddings
builder.AddAzureOpenAITextEmbeddingGeneration(
    deploymentName: "text-embedding-ada-002",
    endpoint: "https://tu-recurso.openai.azure.com/",
    apiKey: "tu-api-key");

var kernel = builder.Build();

// Obtener servicio de embeddings
var embeddingService = kernel.GetRequiredService<ITextEmbeddingGenerationService>();
```

Generar Embeddings

Embedding de un Texto

```
using Microsoft.SemanticKernel.Embeddings;
```

```

public class EmbeddingService
{
    private readonly ITextEmbeddingGenerationService _embeddingService;

    public EmbeddingService(ITextEmbeddingGenerationService embeddingService)
    {
        _embeddingService = embeddingService;
    }

    public async Task<ReadOnlyMemory<float>> GenerateEmbeddingAsync(
        string text,
        CancellationToken cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(text))
        {
            throw new ArgumentException("El texto no puede estar vacío");
        }

        var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
            new[] { text },
            kernel: null,
            cancellationToken);

        return embeddings.First();
    }
}

```

Embeddings en Batch

```

public async Task<IList<ReadOnlyMemory<float>>> GenerateBatchEmbeddingsAsync(
    IEnumerable<string> texts,
    CancellationToken cancellationToken = default)
{
    var textList = texts.ToList();

    if (textList.Count == 0)
    {
        return new List<ReadOnlyMemory<float>>();
    }

    return await _embeddingService.GenerateEmbeddingsAsync(
        textList,
        kernel: null,
        cancellationToken);
}

```

Cálculo de Similitud

Similitud Coseno

```

using System.Numerics.Tensors;

public class SimilarityCalculator
{
    public static double CalculateCosineSimilarity(
        ReadOnlyMemory<float> vector1,
        ReadOnlyMemory<float> vector2)
    {
        var span1 = vector1.Span;

```

```

    var span2 = vector2.Span;

    if (span1.Length != span2.Length)
    {
        throw new ArgumentException("Los vectores deben tener la misma dimensión");
    }

    // Producto punto
    float dotProduct = TensorPrimitives.Dot(span1, span2);

    // Magnitudes
    float magnitude1 = TensorPrimitives.Norm(span1);
    float magnitude2 = TensorPrimitives.Norm(span2);

    if (magnitude1 == 0 || magnitude2 == 0)
    {
        return 0;
    }

    return dotProduct / (magnitude1 * magnitude2);
}

public static double CalculateEuclideanDistance(
    ReadOnlyMemory<float> vector1,
    ReadOnlyMemory<float> vector2)
{
    var span1 = vector1.Span;
    var span2 = vector2.Span;

    if (span1.Length != span2.Length)
    {
        throw new ArgumentException("Los vectores deben tener la misma dimensión");
    }

    float sumSquaredDiff = 0;
    for (int i = 0; i < span1.Length; i++)
    {
        float diff = span1[i] - span2[i];
        sumSquaredDiff += diff * diff;
    }

    return Math.Sqrt(sumSquaredDiff);
}
}

```

Sistema de Búsqueda Semántica en Memoria

```

using System.Collections.Concurrent;

public class Document
{
    public required string Id { get; init; }
    public required string Content { get; init; }
    public ReadOnlyMemory<float> Embedding { get; set; }
    public Dictionary<string, string>? Metadata { get; init; }
}

public class SearchResult

```

```

{
    public required Document Document { get; init; }
    public double Score { get; init; }
}

public class InMemorySemanticSearch
{
    private readonly ITextEmbeddingGenerationService _embeddingService;
    private readonly ConcurrentDictionary<string, Document> _documents;

    public InMemorySemanticSearch(ITextEmbeddingGenerationService embeddingService)
    {
        _embeddingService = embeddingService;
        _documents = new ConcurrentDictionary<string, Document>();
    }

    public async Task IndexDocumentAsync(
        Document document,
        CancellationToken cancellationToken = default)
    {
        // Generar embedding para el documento
        var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
            new[] { document.Content },
            kernel: null,
            cancellationToken);

        document.Embedding = embeddings.First();

        _documents[document.Id] = document;
    }

    public async Task IndexDocumentsAsync(
        IEnumerable<Document> documents,
        CancellationToken cancellationToken = default)
    {
        var docList = documents.ToList();
        var texts = docList.Select(d => d.Content).ToList();

        // Generar embeddings en batch
        var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
            texts,
            kernel: null,
            cancellationToken);

        for (int i = 0; i < docList.Count; i++)
        {
            docList[i].Embedding = embeddings[i];
            _documents[docList[i].Id] = docList[i];
        }
    }

    public async Task<List<SearchResult>> SearchAsync(
        string query,
        int topK = 5,
        CancellationToken cancellationToken = default)
    {
        // Generar embedding de la consulta
        var queryEmbeddings = await _embeddingService.GenerateEmbeddingsAsync(

```

```

        new[] { query },
        kernel: null,
        cancellationToken);

var queryEmbedding = queryEmbeddings.First();

// Calcular similitud con todos los documentos
var results = new List<SearchResult>();

foreach (var doc in _documents.Values)
{
    var similarity = SimilarityCalculator.CalculateCosineSimilarity(
        queryEmbedding,
        doc.Embedding);

    results.Add(new SearchResult
    {
        Document = doc,
        Score = similarity
    });
}

// Ordenar por score descendente y tomar top K
return results
    .OrderByDescending(r => r.Score)
    .Take(topK)
    .ToList();
}

public bool RemoveDocument(string documentId)
{
    return _documents.TryRemove(documentId, out_);
}

public void Clear()
{
    _documents.Clear();
}

public int Count => _documents.Count;
}

```

Búsqueda con Filtros

```

public class FilteredSemanticSearch : InMemorySemanticSearch
{
    public FilteredSemanticSearch(ITextEmbeddingGenerationService embeddingService)
        : base(embeddingService)
    {
    }

    public async Task<List<SearchResult>> SearchWithFiltersAsync(
        string query,
        Dictionary<string, string>? filters = null,
        int topK = 5,
        double minScore = 0.0,
        CancellationToken cancellationToken = default)
    {

```

```

    var results = await SearchAsync(query, topK * 2, cancellationToken);

    // Aplicar filtros de metadata
    if (filters != null && filters.Count > 0)
    {
        results = results
            .Where(r => MatchesFilters(r.Document, filters))
            .ToList();
    }

    // Aplicar score mínimo
    results = results
        .Where(r => r.Score >= minScore)
        .Take(topK)
        .ToList();

    return results;
}

private bool MatchesFilters(Document document, Dictionary<string, string> filters)
{
    if (document.Metadata == null)
        return false;

    foreach (var filter in filters)
    {
        if (!document.Metadata.TryGetValue(filter.Key, out var value) ||
            value != filter.Value)
        {
            return false;
        }
    }

    return true;
}
}

```

Integración con Base de Datos Vectorial

Modelo para Persistencia

```

public class VectorDocument
{
    public string Id { get; set; } = Guid.NewGuid().ToString();
    public required string Content { get; set; }
    public required float[] Embedding { get; set; }
    public Dictionary<string, string>? Metadata { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
}

```

Servicio con Persistencia

```

using System.Text.Json;

public class PersistentSemanticSearch
{
    private readonly ITextEmbeddingGenerationService _embeddingService;
    private readonly string _storageDirectory;

```

```

public PersistentSemanticSearch(
    ITextEmbeddingGenerationService embeddingService,
    string storageDirectory)
{
    _embeddingService = embeddingService;
    _storageDirectory = storageDirectory;

    Directory.CreateDirectory(storageDirectory);
}

public async Task IndexAndSaveAsync(
    Document document,
    CancellationToken cancellationToken = default)
{
    // Generar embedding
    var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
        new[] { document.Content },
        kernel: null,
        cancellationToken);

    document.Embedding = embeddings.First();

    // Guardar en disco
    var vectorDoc = new VectorDocument
    {
        Id = document.Id,
        Content = document.Content,
        Embedding = document.Embedding.ToArray(),
        Metadata = document.Metadata
    };

    var json = JsonSerializer.Serialize(vectorDoc);
    var filePath = Path.Combine(_storageDirectory, $"{document.Id}.json");
    await File.WriteAllTextAsync(filePath, json, cancellationToken);
}

public async Task<List<SearchResult>> SearchAsync(
    string query,
    int topK = 5,
    CancellationToken cancellationToken = default)
{
    // Generar embedding de la consulta
    var queryEmbeddings = await _embeddingService.GenerateEmbeddingsAsync(
        new[] { query },
        kernel: null,
        cancellationToken);

    var queryEmbedding = queryEmbeddings.First();

    // Cargar y buscar en todos los documentos
    var files = Directory.GetFiles(_storageDirectory, "*.json");
    var results = new List<SearchResult>();

    foreach (var file in files)
    {
        var json = await File.ReadAllTextAsync(file, cancellationToken);
        var vectorDoc = JsonSerializer.Deserialize<VectorDocument>(json);

```

```

    if (vectorDoc == null) continue;

    var docEmbedding = new ReadOnlyMemory<float>(vectorDoc.Embedding);
    var similarity = SimilarityCalculator.CalculateCosineSimilarity(
        queryEmbedding,
        docEmbedding);

    results.Add(new SearchResult
    {
        Document = new Document
        {
            Id = vectorDoc.Id,
            Content = vectorDoc.Content,
            Embedding = docEmbedding,
            Metadata = vectorDoc.Metadata
        },
        Score = similarity
    });
}

return results
    .OrderByDescending(r => r.Score)
    .Take(topK)
    .ToList();
}
}

```

RAG (Retrieval Augmented Generation)

Combinar búsqueda semántica con generación:

```

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;

public class RAGService
{
    private readonly InMemorySemanticSearch _search;
    private readonly Kernel _kernel;

    public RAGService(
        InMemorySemanticSearch search,
        Kernel kernel)
    {
        _search = search;
        _kernel = kernel;
    }

    public async Task<string> AskAsync(
        string question,
        CancellationToken cancellationToken = default)
    {
        // 1. Buscar documentos relevantes
        var searchResults = await _search.SearchAsync(
            question,
            topK: 3,
            cancellationToken);

        // 2. Construir contexto con documentos encontrados
        var context = string.Join("\n\n",

```

```

    searchResults.Select(r => $"- {r.Document.Content}"));

// 3. Generar respuesta basada en el contexto
var chatService = _kernel.GetRequiredService<IChatCompletionService>();

var chatHistory = new ChatHistory();
chatHistory.AddSystemMessage("""
    Responde la pregunta del usuario basándote ÚNICAMENTE en el contexto proporcionado.
    Si la información no está en el contexto, di que no tienes suficiente información.
""");

chatHistory.AddUserMessage($"""
    Contexto:
    {context}

    Pregunta: {question}
""");

var response = await chatService.GetChatMessageContentAsync(
    chatHistory,
    kernel: _kernel,
    cancellationToken: cancellationToken);

return response.Content ?? "No pude generar una respuesta";
}
}
}

```

Ejemplo Completo: Sistema de Preguntas y Respuestas

```

using Microsoft.SemanticKernel;
using Microsoft.Extensions.Logging;

class Program
{
    static async Task Main(string[] args)
    {
        // Configurar kernel
        var builder = Kernel.CreateBuilder();

        builder.AddAzureOpenAITextEmbeddingGeneration(
            deploymentName: "text-embedding-ada-002",
            endpoint: Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")!,
            apiKey: Environment.GetEnvironmentVariable("AZURE_OPENAI_KEY")!);

        builder.AddAzureOpenAIChatCompletion(
            deploymentName: "gpt-4",
            endpoint: Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")!,
            apiKey: Environment.GetEnvironmentVariable("AZURE_OPENAI_KEY")!);

        var kernel = builder.Build();

        // Configurar búsqueda semántica
        var embeddingService = kernel.GetRequiredService<ITextEmbeddingGenerationService>();
        var search = new InMemorySemanticSearch(embeddingService);

        // Indexar documentos de conocimiento
        var documents = new[]
        {

```

```

new Document
{
    Id = "1",
    Content = "C# es un lenguaje de programación orientado a objetos desarrollado por Microsoft. Se caracteriza por su tipado fuerte y su sintaxis similar al C++.",
    Metadata = new Dictionary<string, string> { ["category"] = "programming" }
},
new Document
{
    Id = "2",
    Content = "Async/await en C# permite escribir código asíncrono de manera más legible y mantenible que el código tradicional de bucle while o thread pooling.",
    Metadata = new Dictionary<string, string> { ["category"] = "programming" }
},
new Document
{
    Id = "3",
    Content = "Semantic Kernel es un SDK para integrar LLMs en aplicaciones .NET.", que permite interactuar con modelos de lenguaje como Qwen, Claude y GPT-4.",
    Metadata = new Dictionary<string, string> { ["category"] = "ai" }
}
};

await search.IndexDocumentsAsync(documents);

// Crear servicio RAG
var ragService = new RAGService(search, kernel);

// Hacer preguntas
Console.WriteLine("Sistema de preguntas y respuestas listo.");
Console.WriteLine("Escribe 'salir' para terminar.\n");

while (true)
{
    Console.Write("Pregunta: ");
    var question = Console.ReadLine();

    if (string.IsNullOrWhiteSpace(question) || question.ToLower() == "salir")
        break;

    var answer = await ragService.AskAsync(question);
    Console.WriteLine($"\\nRespuesta: {answer}\\n");
}
}
}

```

Mejores Prácticas

1. Normalización de Texto

```

private string NormalizeText(string text)
{
    return text
        .ToLowerInvariant()
        .Trim()
        .Replace("\\n", " ")
        .Replace("\\r", " ");
}

```

2. Chunking de Documentos Largos

```
public List<string> ChunkText(string text, int chunkSize = 500, int overlap = 50)
{
    var words = text.Split(' ', StringSplitOptions.RemoveEmptyEntries);
    var chunks = new List<string>();

    for (int i = 0; i < words.Length; i += chunkSize - overlap)
    {
        var chunk = string.Join(" ", words.Skip(i).Take(chunkSize));
        chunks.Add(chunk);
    }

    return chunks;
}
```

3. Caché de Embeddings

```
using Microsoft.Extensions.Caching.Memory;

public class CachedEmbeddingService
{
    private readonly ITextEmbeddingGenerationService _embeddingService;
    private readonly IMemoryCache _cache;

    public async Task<ReadOnlyMemory<float>> GetEmbeddingAsync(string text)
    {
        var cacheKey = $"emb_{text.GetHashCode()}";

        if (_cache.TryGetValue<ReadOnlyMemory<float>>(cacheKey, out var cached))
        {
            return cached;
        }

        var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
            new[] { text },
            kernel: null);

        var embedding = embeddings.First();

        _cache.Set(cacheKey, embedding, TimeSpan.FromHours(24));
    }
}
```

4. Manejo de Errores

```
public async Task<ReadOnlyMemory<float>> GenerateEmbeddingWithRetryAsync(
    string text,
    int maxRetries = 3)
{
    for (int attempt = 0; attempt < maxRetries; attempt++)
    {
        try
        {
            return await GenerateEmbeddingAsync(text);
        }
        catch (HttpRequestException ex) when (attempt < maxRetries - 1)
```

```

    {
        await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, attempt)));
    }
}

throw new Exception("No se pudo generar embedding después de varios intentos");
}

```

Conclusión

Los embeddings vectoriales y la búsqueda semántica son fundamentales para aplicaciones de IA modernas. Permiten buscar por significado, no solo por palabras clave, y son la base de sistemas RAG que combinan recuperación de información con generación de texto. Con Semantic Kernel y Azure OpenAI puedes implementar búsqueda semántica robusta en tus aplicaciones .NET.

Palabras clave: vector embeddings, semantic search, RAG, Azure OpenAI, Semantic Kernel, similarity search, C#

Cierre del paso

Verificación rápida

- Una query recupera documentos ‘relacionados’ aunque no comparten palabras exactas.

Errores frecuentes y cómo desbloquearte

- Deployment de embeddings incorrecto: revisa nombre y modelo.
- Vectores de distinta dimensión: no mezclar modelos/embeddings.

Ejercicios (para afianzar)

- Añade normalización previa (lowercase, trim, eliminación de ruido).
- Persiste el índice en disco (JSON) y recárgalo al iniciar.

Siguiente paso

Cuando estés listo, continúa con el **Paso 06 — Prompt Engineering: Mejores Prácticas para LLMs**.

Paso 06 — Prompt Engineering: Mejores Prácticas para LLMs

Mapa del paso

Tiempo estimado: 60–90 min

Resultado que vas a conseguir:

Prompts más fiables: plantillas, ejemplos, delimitadores y checklist de calidad.

Objetivos de aprendizaje:

- Reducir ambigüedad con instrucciones específicas y formato de salida.
- Aplicar técnicas de prompt engineering con pruebas A/B.
- Diseñar prompts mantenibles (variables, versiones, comentarios).

Prerrequisitos:

- Experiencia mínima invocando chat completion (Pasos 01–04).

Ruta guiada (paso a paso)

1. Define un prompt base y un conjunto de casos de prueba.
 2. Introduce delimitadores (triple backticks) y formato de salida.
 3. Añade ejemplos (few-shot) solo si aportan señal.
 4. Documenta límites: qué NO debe hacer el modelo.
 5. Versiona el prompt y compara resultados.
-

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

Introducción

El prompt engineering es el arte de diseñar instrucciones efectivas para modelos de lenguaje. Un buen prompt puede ser la diferencia entre resultados mediocres y excepcionales. En este tutorial aprenderás técnicas probadas para crear prompts efectivos.

Principios Fundamentales

1. Claridad y Especificidad

Prompt vago:

Escribe sobre perros

Prompt específico:

Escribe un artículo de 500 palabras sobre los beneficios de tener un perro como mascota, enfocándose en aspectos de salud mental, ejercicio físico y compañía.

2. Estructura Clara

```
var promptTemplate = """
```

CONTEXTO: Eres un experto en {{\$dominio}}.

TAREA: {{\$tarea}}

RESTRICCIONES:

- Máximo {{\$maxPalabras}} palabras
- Usa ejemplos concretos
- Formato {{\$formato}}

EJEMPLO:

```
{{$ejemplo}}
```

```
""";
```

3. Ejemplos (Few-Shot Learning)

```
var promptConEjemplos = """
```

Clasifica el sentimiento de estos mensajes como positivo, negativo o neutral.

Ejemplos:

- "Me encanta este producto" → positivo
- "No funciona bien" → negativo
- "Es de color azul" → neutral

```
Ahora clasifica:="{{$mensaje}}"
""";
```

Técnicas de Prompt Engineering

Chain-of-Thought (Cadena de Pensamiento)

Pedir al modelo que “piense paso a paso”:

```
var chainOfThoughtPrompt = """
```

Resuelve este problema paso a paso:

Problema: {{\$problema}}

Piensa en voz alta y muestra tu razonamiento:

1. ¿Qué información tengo?
 2. ¿Qué necesito calcular?
 3. ¿Cómo lo resuelvo?
 4. ¿Cuál es la respuesta final?
- ```
""";
```

### Zero-Shot vs Few-Shot vs Many-Shot

```
// Zero-Shot: Sin ejemplos
```

```
var zeroShot = "Traduce al inglés: {{$texto}}";
```

```
// Few-Shot: Pocos ejemplos (2-5)
```

```
var fewShot = """
```

Traduce al inglés:

- "Hola" → "Hello"
  - "Adiós" → "Goodbye"
  - "{{\$texto}}" →
- ```
""";
```

```
// Many-Shot: Muchos ejemplos (10+)
```

```
var manyShot = """
```

Traduce al inglés:

- "Buenos días" → "Good morning"
 - "Buenas tardes" → "Good afternoon"
 - "Buenas noches" → "Good night"
 - "Por favor" → "Please"
 - "Gracias" → "Thank you"
 - "De nada" → "You're welcome"
 - [... más ejemplos ...]
 - "{{\$texto}}" →
- ```
""";
```

### Role Prompting

Asignar un rol específico al modelo:

```
var rolePrompt = """
```

```
Eres un {{$rol}} con {{$experiencia}} años de experiencia.
```

Tu estilo de comunicación es:

- {{\$estilo1}}
- {{\$estilo2}}
- {{\$estilo3}}

Responde a: {{\$pregunta}}

```
""";
```

```
// Ejemplo concreto
```

```
var expertPrompt = """
```

Eres un arquitecto de software senior con 15 años de experiencia en sistemas distribuidos.

Tu estilo de comunicación es:

- Técnico pero accesible
- Usa ejemplos concretos
- Menciona trade-offs y consideraciones importantes

Responde a: ¿Cómo diseñarías un sistema de caché distribuido?

""";

## Structured Output

Solicitar formato específico:

`var structuredPrompt = """`

Analiza el siguiente texto y responde en formato JSON:

```
{
 "resumen": "string (máximo 100 palabras)",
 "temas_principales": ["string"],
 "sentimiento": "positivo|negativo|neutral",
 "entidades": [{"nombre": "string", "tipo": "string"}],
 "palabras_clave": ["string"]
}
```

Texto: {{\$texto}}

""";

## Patrones Avanzados

### Template Method Pattern

```
public abstract class PromptTemplate
{
 public string Build(Dictionary<string, string> variables)
 {
 var prompt = GetSystemMessage();
 prompt += "\n\n" + GetTaskDescription(variables);
 prompt += "\n\n" + GetConstraints();
 prompt += "\n\n" + GetExamples();
 prompt += "\n\n" + GetUserInput(variables);

 return prompt;
 }

 protected abstract string GetSystemMessage();
 protected abstract string GetTaskDescription(Dictionary<string, string> variables);
 protected abstract string GetConstraints();
 protected abstract string GetExamples();
 protected abstract string GetUserInput(Dictionary<string, string> variables);
}

public class ClassificationPromptTemplate : PromptTemplate
{
 protected override string GetSystemMessage()
 {
 return "Eres un clasificador experto que analiza texto y lo categoriza con precisión.";
 }

 protected override string GetTaskDescription(Dictionary<string, string> variables)
```

```

{
 return $"Clasifica el siguiente texto en una de estas categorías: {variables["categories"]}";
}

protected override string GetConstraints()
{
 return """
 REGLAS:
 - Selecciona solo UNA categoría
 - Justifica tu decisión brevemente
 - Si no estás seguro, indica tu nivel de confianza
 """;
}

protected override string GetExamples()
{
 return """
 EJEMPLOS:
 Texto: "Necesito un fontanero urgente"
 Categoría: servicios_hogar
 Confianza: 95 %

 Texto: "¿Dónde puedo comprar pan?"
 Categoría: alimentación
 Confianza: 90 %
 """;
}

protected override string GetUserInput(Dictionary<string, string> variables)
{
 return $"Texto a clasificar: {variables["text"]}";
}
}

```

## Prompt Chaining

Encadenar múltiples prompts para tareas complejas:

```

public class PromptChainService
{
 private readonly Kernel _kernel;

 public async Task<string> AnalyzeArticleAsync(string article)
 {
 // Paso 1: Extraer temas principales
 var topicsPrompt = $$"
 Extrae los 3 temas principales de este artículo.
 Responde solo con la lista de temas, uno por línea.

 Artículo: {article}
 """;

 var topics = await InvokePromptAsync(topicsPrompt);

 // Paso 2: Resumir cada tema
 var summaries = new List<string>();
 foreach (var topic in topics.Split('\n'))
 {
 var summaryPrompt = $$"

```

```

Resume cómo se trata el tema "{topic}" en el siguiente artículo.
Usa máximo 50 palabras.

Artículo: {article}
""";
var summary = await InvokePromptAsync(summaryPrompt);
summaries.Add($"{topic}: {summary}");
}

// Paso 3: Crear resumen ejecutivo
var executiveSummaryPrompt = $"""
Basándose en estos resúmenes por tema, crea un resumen ejecutivo del artículo.

{string.Join("\n\n", summaries)}
""";

return await InvokePromptAsync(executiveSummaryPrompt);
}

private async Task<string> InvokePromptAsync(string prompt)
{
 var function = _kernel.CreateFunctionFromPrompt(prompt);
 var result = await _kernel.InvokeAsync(function);
 return result.GetValue<string>() ?? "";
}
}

```

## Self-Consistency

Generar múltiples respuestas y tomar la más común:

```

public class SelfConsistentService
{
 private readonly Kernel _kernel;

 public async Task<string> GetConsistentAnswerAsync(
 string question,
 int iterations = 5)
 {
 var answers = new List<string>();

 var prompt = $$"
Responde la siguiente pregunta:

{question}

Piensa paso a paso y proporciona tu respuesta final.
""";

 // Generar múltiples respuestas con temperatura alta
for (int i = 0; i < iterations; i++)
{
 var settings = new OpenAIPromptExecutionSettings
 {
 Temperature = 0.8
 };

 var function = _kernel.CreateFunctionFromPrompt(prompt, settings);

```

```

 var result = await _kernel.InvokeAsync(function);
 answers.Add(result.GetValue<string>() ?? "");
 }

 // Encontrar respuesta más común o consenso
 return FindConsensus(answers);
}

private string FindConsensus(List<string> answers)
{
 // Implementación simplificada: retornar la más frecuente
 return answers
 .GroupBy(a => a)
 .OrderByDescending(g => g.Count())
 .First()
 .Key;
}
}

```

## Anti-Patrones (Qué Evitar)

### Prompts Ambiguos

```

// MAL
var malPrompt = "Háblame de eso";

// BIEN
var buenPrompt = """
Proporciona una explicación detallada sobre {{$tema}}, incluyendo:
1. Definición
2. Casos de uso principales
3. Ventajas y desventajas
4. Ejemplo práctico
""";

```

### Instrucciones Contradicторias

```

// MAL: Instrucciones que se contradicen
var contradictorio = """
Sé muy breve pero incluye todos los detalles posibles.
""";

// BIEN: Instrucciones claras y coherentes
var claro = """
Proporciona un resumen ejecutivo de 200 palabras que destaque
los 3 puntos más importantes.
""";

```

### Formato Inconsistente

```

// MAL
var inconsistente = """
dame info sobre:
Tema: {{$tema}}
quiero saber todo
formato: lista
""";

```

```
// BIEN
```

```
var consistente = """
TEMA: {{$tema}}
```

#### FORMATO REQUERIDO:

- Lista con viñetas
- Máximo 5 puntos
- Cada punto: 1-2 oraciones

#### PROFUNDIDAD:

Proporciona información intermedia, adecuada para alguien con conocimientos básicos del tema.

```
""";
```

## Prompts para Casos de Uso Comunes

### Clasificación

```
var classificationPrompt = """
```

Clasifica el siguiente mensaje según su intención.

#### CATEGORÍAS POSIBLES:

- pregunta: El usuario hace una pregunta
- queja: El usuario expresa insatisfacción
- elogio: El usuario expresa satisfacción
- solicitud: El usuario pide algo específico
- otro: No encaja en las categorías anteriores

#### REGLAS:

1. Responde con UNA sola palabra (el nombre de la categoría)
2. Si hay duda entre dos, elige la más específica

```
MENSAJE: {{$mensaje}}
```

#### CATEGORÍA:

```
""";
```

### Extracción de Información

```
var extractionPrompt = """
```

Extrae información estructurada del siguiente texto.

#### INFORMACIÓN A EXTRAER:

- Nombre de persona(s)
- Fechas
- Ubicaciones
- Organizaciones
- Números de contacto

#### FORMATO DE SALIDA:

```
{
 "personas": ["string"],
 "fechas": ["string"],
 "ubicaciones": ["string"],
 "organizaciones": ["string"],
 "contactos": ["string"]
}
```

Si no encuentras información de algún tipo, usa un array vacío [] .

TEXTO: {{ \$texto }}

JSON:

```
""";
```

### Generación Creativa

```
var creativePrompt = """
```

Genera {{\$tipo}} con las siguientes características:

ESTILO: {{\$estilo}}

TONO: {{\$tono}}

LONGITUD: {{\$longitud}} palabras

AUDIENCIA: {{\$audiencia}}

REQUISITOS:

- Original y creativo
- Apropriado para la audiencia
- Mantén el tono consistente
- {{#if incluir\_ejemplos}}
- Incluye ejemplos concretos
- {{/if}}

TEMA: {{\$tema}}

  {{\$tipo}}:

```
""";
```

### Análisis y Resumen

```
var analysisPrompt = """
```

Analiza el siguiente contenido y proporciona:

1. RESUMEN (100 palabras máximo)
  - Idea principal
  - Puntos clave
2. ANÁLISIS
  - Fortalezas del contenido
  - Debilidades o áreas de mejora
  - Público objetivo
3. RECOMENDACIONES
  - 3 sugerencias específicas de mejora

CONTENIDO:

```
{{$contenido}}
```

----

ANÁLISIS:

```
""";
```

### Testing de Prompts

```
public class PromptTester
{
 private readonly Kernel _kernel;
```

```

public async Task<PromptTestResult> TestPromptAsync(
 string promptTemplate,
 Dictionary<string, string> testCases)
{
 var results = new List<TestCase>();

 foreach (var testCase in testCases)
 {
 var prompt = promptTemplate.Replace("{{\$input}}", testCase.Value);

 var function = _kernel.CreateFunctionFromPrompt(prompt);
 var result = await _kernel.InvokeAsync(function);
 var output = result.GetValue<string>() ?? "";

 results.Add(new TestCase
 {
 Input = testCase.Value,
 Output = output,
 Expected = testCase.Key
 });
 }

 return new PromptTestResult
 {
 TestCases = results,
 SuccessRate = CalculateSuccessRate(results)
 };
}

private double CalculateSuccessRate(List<TestCase> results)
{
 // Implementar lógica de evaluación
 return 0.0;
}
}

public class TestCase
{
 public required string Input { get; init; }
 public required string Output { get; init; }
 public required string Expected { get; init; }
}

public class PromptTestResult
{
 public required List<TestCase> TestCases { get; init; }
 public double SuccessRate { get; init; }
}

```

## Optimización de Prompts

### A/B Testing

```

public class PromptOptimizer
{
 private readonly Kernel _kernel;

 public async Task<string> FindBestPromptAsync(
 List<string> promptVariants,

```

```

 List<string> testInputs)
{
 var scores = new Dictionary<string, double>();

 foreach (var prompt in promptVariants)
 {
 var totalScore = 0.0;

 foreach (var input in testInputs)
 {
 var result = await TestPromptWithInputAsync(prompt, input);
 totalScore += result.Quality;
 }

 scores[prompt] = totalScore / testInputs.Count;
 }

 return scores.OrderByDescending(kvp => kvp.Value).First().Key;
}

private async Task<(string Output, double Quality)> TestPromptWithInputAsync(
 string prompt,
 string input)
{
 // Implementar testing y scoring
 return ("", 0.0);
}
}

```

### Iteración Basada en Feedback

```

public class PromptIterator
{
 public string ImprovePrompt(
 string originalPrompt,
 List<PromptFeedback> feedbacks)
 {
 var improvements = new List<string>();

 // Analizar feedback común
 var commonIssues = feedbacks
 .SelectMany(f => f.Issues)
 .GroupBy(i => i)
 .OrderByDescending(g => g.Count())
 .Take(3)
 .Select(g => g.Key);

 foreach (var issue in commonIssues)
 {
 improvements.Add(GetImprovementForIssue(issue));
 }

 return ApplyImprovements(originalPrompt, improvements);
 }

 private string GetImprovementForIssue(string issue)
 {
 return issue switch

```

```

 {
 "too_vague" => "Aregar más especificidad y ejemplos",
 "inconsistent" => "Estandarizar formato y estructura",
 "too_long" => "Simplificar y reducir instrucciones",
 _ => "Revisar claridad general"
 };
}

private string ApplyImprovements(string prompt, List<string> improvements)
{
 // Implementar lógica de mejora
 return prompt;
}
}

public class PromptFeedback
{
 public required string PromptVersion { get; init; }
 public required List<string> Issues { get; init; }
 public int Rating { get; init; }
}

```

## Mejores Prácticas Resumidas

### 1. Estructura

- Usa secciones claramente delimitadas
- Instrucciones antes de datos
- Formato consistente

### 2. Claridad

- Sé específico sobre lo que quieres
- Proporciona ejemplos cuando sea útil
- Define el formato de salida esperado

### 3. Contexto

- Da contexto relevante al modelo
- Define el rol o perspectiva
- Establece restricciones claras

### 4. Testing

- Prueba con múltiples entradas
- Itera basándote en resultados
- Mantén registro de versiones

### 5. Temperatura

- Baja (0.0-0.3) para tareas determinísticas
- Media (0.4-0.7) para balance
- Alta (0.8-1.0) para creatividad

## Conclusión

El prompt engineering es una habilidad esencial para trabajar efectivamente con LLMs. Buenos prompts son claros, estructurados y específicos. La práctica y la iteración basada en resultados te ayudarán a crear prompts cada vez más efectivos. Recuerda que no existe un “prompt perfecto” universal - cada caso de uso requiere adaptación y optimización.

---

**Palabras clave:** prompt engineering, LLM, GPT, Azure OpenAI, few-shot learning, chain-of-thought, structured output

---

## Cierre del paso

### Verificación rápida

- Las respuestas siguen el formato con menos variabilidad y menos alucinación.

### Errores frecuentes y cómo desbloquearte

- Prompts largos sin estructura: separar instrucciones, datos y formato.
- Pedir ‘hazlo perfecto’ sin criterios: define checklist de aceptación.

### Ejercicios (para afianzar)

- Crea 3 versiones del mismo prompt y mide consistencia.
- Añade validación automática del formato de salida.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 07 — Integración de Azure OpenAI en Aplicaciones .NET**.

## Paso 07 — Integración de Azure OpenAI en Aplicaciones .NET

### Mapa del paso

**Tiempo estimado:** 75–120 min

### Resultado que vas a conseguir:

Integración segura de Azure OpenAI: opciones, User Secrets, DI y HttpClient.

### Objetivos de aprendizaje:

- Crear una configuración tipada (`IOptions`) para Azure OpenAI.
- Gestionar credenciales de forma segura en dev y prod.
- Registrar Semantic Kernel y conectores con DI.

### Prerrequisitos:

- Suscripción Azure activa y recurso Azure OpenAI con deployments.
- Conocer lo básico de DI en .NET (se cubre más adelante).

### Ruta guiada (paso a paso)

- Crea el recurso Azure OpenAI y copia endpoint/keys.
- Crea deployments para chat y embeddings y guarda sus nombres.
- Configura User Secrets o variables de entorno.
- Registra opciones (`IOptions<T>`) y un `HttpClient` con timeout.
- Registra `Kernel` en DI con chat y embeddings.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Azure OpenAI Service proporciona acceso a modelos avanzados de IA como GPT-4 a través de APIs REST. En este tutorial aprenderás a integrar Azure OpenAI en aplicaciones .NET de forma segura y escalable.

## Configuración Inicial

### Requisitos Previos

1. Suscripción de Azure activa
2. Recurso Azure OpenAI creado
3. Modelos desplegados (GPT-4, GPT-3.5-Turbo, embeddings)
4. .NET 6.0 o superior

### Obtener Credenciales

En el portal de Azure, necesitas:

Endpoint: <https://tu-recurso.openai.azure.com/>

API Key: tu-clave-api

Deployment Names:

- gpt-4
- gpt-35-turbo
- text-embedding-ada-002

### Instalación de Paquetes

```
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.SemanticKernel
dotnet add package Microsoft.Extensions.Configuration
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

## Gestión Segura de Credenciales

### User Secrets (Desarrollo)

```
dotnet user-secrets init
dotnet user-secrets set "AzureOpenAI:Endpoint" "https://tu-recurso.openai.azure.com/"
dotnet user-secrets set "AzureOpenAI:ApiKey" "tu-clave-api"
dotnet user-secrets set "AzureOpenAI:ChatDeployment" "gpt-4"
dotnet user-secrets set "AzureOpenAI:EmbeddingDeployment" "text-embedding-ada-002"
```

### Configuration en appsettings.json

```
{
 "AzureOpenAI": {
 "Endpoint": "",
 "ApiKey": "",
 "ChatDeployment": "gpt-4",
 "EmbeddingDeployment": "text-embedding-ada-002",
 "MaxRetries": 3,
 "TimeoutSeconds": 120
 }
}
```

### Modelo de Configuración

```
public class AzureOpenAIOptions
{
 public const string SectionName = "AzureOpenAI";
```

```

 public required string Endpoint { get; set; }
 public required string ApiKey { get; set; }
 public required string ChatDeployment { get; set; }
 public required string EmbeddingDeployment { get; set; }
 public int MaxRetries { get; set; } = 3;
 public int TimeoutSeconds { get; set; } = 120;
}

```

## Configuración con Dependency Injection

### Startup/Program.cs

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.SemanticKernel;

var builder = WebApplication.CreateBuilder(args);

// Configurar opciones
builder.Services.Configure<AzureOpenAIOptions>(
 builder.Configuration.GetSection(AzureOpenAIOptions.SectionName));

// Registrar HttpClient configurado
builder.Services.AddHttpClient("AzureOpenAI", (serviceProvider, client) =>
{
 var options = serviceProvider
 .GetRequiredService<IOptions<AzureOpenAIOptions>>()
 .Value;

 client.Timeout = TimeSpan.FromSeconds(options.TimeoutSeconds);
});

// Registrar Semantic Kernel
builder.Services.AddScoped<Kernel>(serviceProvider =>
{
 var options = serviceProvider
 .GetRequiredService<IOptions<AzureOpenAIOptions>>()
 .Value;

 var httpClientFactory = serviceProvider
 .GetRequiredService<IHttpClientFactory>();

 var httpClient = httpClientFactory.CreateClient("AzureOpenAI");

 var kernelBuilder = Kernel.CreateBuilder();

 kernelBuilder.AddAzureOpenAIChatCompletion(
 deploymentName: options.ChatDeployment,
 endpoint: options.Endpoint,
 apiKey: options.ApiKey,
 httpClient: httpClient);

 kernelBuilder.AddAzureOpenAITextEmbeddingGeneration(
 deploymentName: options.EmbeddingDeployment,
 endpoint: options.Endpoint,
 apiKey: options.ApiKey,
 httpClient: httpClient);
}

```

```

 return kernelBuilder.Build();
 });

var app = builder.Build();

Cliente Azure OpenAI Nativo

Usando Azure.AI.OpenAI


```

using Azure;
using Azure.AI.OpenAI;

public class AzureOpenAIClient
{
    private readonly OpenAIClient _client;
    private readonly string _chatDeployment;
    private readonly string _embeddingDeployment;
    private readonly ILogger<AzureOpenAIClient> _logger;

    public AzureOpenAIClient(
        IOptions<AzureOpenAIOptions> options,
        ILogger<AzureOpenAIClient> logger)
    {
        var config = options.Value;

        _client = new OpenAIClient(
            new Uri(config.Endpoint),
            new AzureKeyCredential(config.ApiKey));

        _chatDeployment = config.ChatDeployment;
        _embeddingDeployment = config.EmbeddingDeployment;
        _logger = logger;
    }

    public async Task<string> GetChatCompletionAsync(
        string prompt,
        CancellationToken cancellationToken = default)
    {
        var chatOptions = new ChatCompletionsOptions
        {
            DeploymentName = _chatDeployment,
            Messages =
            {
                new ChatRequestSystemMessage("Eres un asistente útil."),
                new ChatRequestUserMessage(prompt)
            },
            Temperature = 0.7f,
            MaxTokens = 800,
            NucleusSamplingFactor = 0.9f
        };

        try
        {
            var response = await _client.GetChatCompletionsAsync(
                chatOptions,
                cancellationToken);

            var choice = response.Value.Choices.FirstOrDefault();
            if (choice == null)

```


```

```

 {
 _logger.LogWarning("No se recibieron respuestas del modelo");
 return string.Empty;
 }

 _logger.LogInformation(
 "Chat completion generado. Tokens: {PromptTokens} prompt, {CompletionTokens} complet
 response.Value.Usage.PromptTokens,
 response.Value.Usage.CompletionTokens);

 return choice.Message.Content;
}
catch (RequestFailedException ex)
{
 _logger.LogError(ex, "Error llamando a Azure OpenAI");
 throw;
}
}

public async Task<float[]> GetEmbeddingAsync(
 string text,
 CancellationToken cancellationToken = default)
{
 var embeddingsOptions = new EmbeddingsOptions(
 _embeddingDeployment,
 new List<string> { text });

 try
 {
 var response = await _client.GetEmbeddingsAsync(
 embeddingsOptions,
 cancellationToken);

 return response.Value.Data[0].Embedding.ToArray();
 }
 catch (RequestFailedException ex)
 {
 _logger.LogError(ex, "Error generando embedding");
 throw;
 }
}
}

```

## Streaming de Respuestas

```

public async IAsyncEnumerable<string> GetChatCompletionStreamAsync(
 string prompt,
 [EnumeratorCancellation] CancellationToken cancellationToken = default)
{
 var chatOptions = new ChatCompletionsOptions
 {
 DeploymentName = _chatDeployment,
 Messages =
 {
 new ChatRequestSystemMessage("Eres un asistente útil."),
 new ChatRequestUserMessage(prompt)
 }
 };
}

```

```

var response = await _client.GetChatCompletionsStreamingAsync(
 chatOptions,
 cancellationToken);

await foreach (var update in response.WithCancellation(cancellationToken))
{
 if (update.ContentUpdate != null)
 {
 yield return update.ContentUpdate;
 }
}
}

```

## Manejo de Rate Limits

### Implementación con Polly

```

using Polly;
using Polly.RateLimit;

public class RateLimitedAzureOpenAIClient
{
 private readonly AzureOpenAIClient _client;
 private readonly AsyncRateLimitPolicy _rateLimitPolicy;

 public RateLimitedAzureOpenAIClient(AzureOpenAIClient client)
 {
 _client = client;

 // Limitar a 60 llamadas por minuto
 _rateLimitPolicy = Policy.RateLimitAsync(
 numberOfExecutions: 60,
 perTimeSpan: TimeSpan.FromMinutes(1),
 maxBurst: 10);
 }

 public async Task<string> GetChatCompletionAsync(
 string prompt,
 CancellationToken cancellationToken = default)
 {
 return await _rateLimitPolicy.ExecuteAsync(
 async ct => await _client.GetChatCompletionAsync(prompt, ct),
 cancellationToken);
 }
}

```

### Retry con Exponential Backoff

```

using Polly;
using Polly.Retry;

public class ResilientAzureOpenAIClient
{
 private readonly AzureOpenAIClient _client;
 private readonly AsyncRetryPolicy _retryPolicy;
 private readonly ILogger<ResilientAzureOpenAIClient> _logger;

 public ResilientAzureOpenAIClient(

```

```

 AzureOpenAIClient client,
 ILogger<ResilientAzureOpenAIClient> logger)
 {
 _client = client;
 _logger = logger;

 _retryPolicy = Policy
 .Handle<RequestFailedException>(ex =>
 ex.Status == 429 || // Too Many Requests
 ex.Status == 503 || // Service Unavailable
 ex.Status == 504) // Gateway Timeout
 .WaitAndRetryAsync(
 retryCount: 3,
 sleepDurationProvider: attempt => TimeSpan.FromSeconds(Math.Pow(2, attempt)),
 onRetry: (exception, timeSpan, retryCount, context) =>
 {
 _logger.LogWarning(
 "Intento {RetryCount} después de {Delay}s debido a: {Error}",
 retryCount,
 timeSpan.TotalSeconds,
 exception.Message);
 });
 }

 public async Task<string> GetChatCompletionAsync(
 string prompt,
 CancellationToken cancellationToken = default)
 {
 return await _retryPolicy.ExecuteAsync(
 async ct => await _client.GetChatCompletionAsync(prompt, ct),
 cancellationToken);
 }
}

```

## Monitoreo y Telemetría

### Application Insights

```

using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;

public class MonitoredAzureOpenAIClient
{
 private readonly AzureOpenAIClient _client;
 private readonly TelemetryClient _telemetryClient;

 public MonitoredAzureOpenAIClient(
 AzureOpenAIClient client,
 TelemetryClient telemetryClient)
 {
 _client = client;
 _telemetryClient = telemetryClient;
 }

 public async Task<string> GetChatCompletionAsync(
 string prompt,
 CancellationToken cancellationToken = default)
 {
 var operation = _telemetryClient.StartOperation<DependencyTelemetry>("AzureOpenAI.ChatComple")

```

```
operation.Telemetry.Type = "HTTP";
operation.Telemetry.Target = "Azure OpenAI";

var stopwatch = Stopwatch.StartNew();

try
{
 var result = await _client.GetChatCompletionAsync(prompt, cancellationToken);

 stopwatch.Stop();

 operation.Telemetry.Success = true;
 operation.Telemetry.Duration = stopwatch.Elapsed;

 _telemetryClient.TrackMetric("AzureOpenAI.PromptLength", prompt.Length);
 _telemetryClient.TrackMetric("AzureOpenAI.ResponseLength", result.Length);
 _telemetryClient.TrackMetric("AzureOpenAI.LatencyMs", stopwatch.ElapsedMilliseconds);

 return result;
}
catch (Exception ex)
{
 stopwatch.Stop();

 operation.Telemetry.Success = false;
 operation.Telemetry.Duration = stopwatch.Elapsed;

 _telemetryClient.TrackException(ex, new Dictionary<string, string>
 {
 ["PromptLength"] = prompt.Length.ToString(),
 ["Operation"] = "ChatCompletion"
 });

 throw;
}
finally
{
 _telemetryClient.StopOperation(operation);
}
}
```

## Caché de Respuestas

```
using Microsoft.Extensions.Caching.Memory;
using System.Security.Cryptography;
using System.Text;

public class CachedAzureOpenAIclient
{
 private readonly AzureOpenAIclient _client;
 private readonly IMemoryCache _cache;
 private readonly ILogger<CachedAzureOpenAIclient> _logger;
 private readonly TimeSpan _cacheDuration;

 public CachedAzureOpenAIclient(
 AzureOpenAIclient client,
 IMemoryCache cache,
```

```

 ILogger<CachedAzureOpenAIClient> logger,
 TimeSpan? cacheDuration = null)
{
 _client = client;
 _cache = cache;
 _logger = logger;
 _cacheDuration = cacheDuration ?? TimeSpan.FromHours(1);
}

public async Task<string> GetChatCompletionAsync(
 string prompt,
 bool useCache = true,
 CancellationToken cancellationToken = default)
{
 if (!useCache)
 {
 return await _client.GetChatCompletionAsync(prompt, cancellationToken);
 }

 var cacheKey = GenerateCacheKey(prompt);

 if (_cache.TryGetValue<string>(cacheKey, out var cachedResult))
 {
 _logger.LogInformation("Cache HIT para prompt: {PromptHash}", cacheKey);
 return cachedResult;
 }

 _logger.LogInformation("Cache MISS para prompt: {PromptHash}", cacheKey);

 var result = await _client.GetChatCompletionAsync(prompt, cancellationToken);

 _cache.Set(cacheKey, result, _cacheDuration);

 return result;
}

private string GenerateCacheKey(string prompt)
{
 using var sha256 = SHA256.Create();
 var hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(prompt));
 return $"aoai_{BitConverter.ToString(hashBytes).Replace("-", "")}.Substring(0, 16).ToLowerInvariant";
}
}

```

## Testing

### Unit Tests con Mocks

```

using Moq;
using Xunit;

public class AzureOpenAIclientTests
{
 [Fact]
 public async Task GetChatCompletionAsync_ReturnsResponse()
 {
 // Arrange
 var mockOptions = new Mock<IOptions<AzureOpenAIOptions>>();
 mockOptions.Setup(o => o.Value).Returns(new AzureOpenAIOptions

```

```

 {
 Endpoint = "https://test.openai.azure.com/",
 ApiKey = "test-key",
 ChatDeployment = "gpt-4",
 EmbeddingDeployment = "embedding"
 });

 var mockLogger = new Mock<ILogger<AzureOpenAIClient>>();

 var client = new AzureOpenAIClient(mockOptions.Object, mockLogger.Object);

 // Act & Assert se requiere configuración adicional con cliente real o mock
 // En tests de integración usarías el cliente real
}
}

```

## Integration Tests

```

[Collection("AzureOpenAI")]
public class AzureOpenAIIntegrationTests
{
 private readonly AzureOpenAIClient _client;

 public AzureOpenAIIntegrationTests()
 {
 var configuration = new ConfigurationBuilder()
 .AddUserSecrets<AzureOpenAIIntegrationTests>()
 .Build();

 var options = Options.Create(new AzureOpenAIOptions
 {
 Endpoint = configuration["AzureOpenAI:Endpoint"]!,
 ApiKey = configuration["AzureOpenAI:ApiKey"]!,
 ChatDeployment = configuration["AzureOpenAI:ChatDeployment"]!,
 EmbeddingDeployment = configuration["AzureOpenAI:EmbeddingDeployment"]!
 });

 var logger = NullLogger<AzureOpenAIClient>.Instance;
 _client = new AzureOpenAIClient(options, logger);
 }

 [Fact]
 public async Task GetChatCompletionAsync_WithSimplePrompt_ReturnsResponse()
 {
 // Arrange
 var prompt = "¿Cuál es la capital de España?";

 // Act
 var result = await _client.GetChatCompletionAsync(prompt);

 // Assert
 Assert.NotNull(result);
 Assert.NotEmpty(result);
 Assert.Contains("Madrid", result, StringComparison.OrdinalIgnoreCase);
 }

 [Fact]
 public async Task GetEmbeddingAsync_WithText_ReturnsVector()

```

```

{
 // Arrange
 var text = "Este es un texto de prueba";

 // Act
 var embedding = await _client.GetEmbeddingAsync(text);

 // Assert
 Assert.NotNull(embedding);
 Assert.True(embedding.Length > 0);
 Assert.Equal(1536, embedding.Length); // text-embedding-ada-002 dimension
}
}

```

## Mejores Prácticas

### 1. Seguridad

```

// Usar variables de entorno o Azure Key Vault
var apiKey = Environment.GetEnvironmentVariable("AZURE_OPENAI_KEY");

// Nunca hardcodear credenciales
// var apiKey = "sk-...";

```

### 2. Timeouts Apropiados

```

var httpClient = new HttpClient
{
 Timeout = TimeSpan.FromSeconds(120) // 2 minutos para operaciones de IA
};

```

### 3. Logging Estructurado

```

_logger.LogInformation(
 "Azure OpenAI request: Model={Model}, PromptTokens={PromptTokens}, CompletionTokens={CompletionTokens},
 deployment,
 promptTokens,
 completionTokens);

```

### 4. Manejo de Costos

```

public class CostTrackingClient
{
 private long _totalPromptTokens;
 private long _totalCompletionTokens;

 public void TrackUsage(int promptTokens, int completionTokens)
 {
 Interlocked.Add(ref _totalPromptTokens, promptTokens);
 Interlocked.Add(ref _totalCompletionTokens, completionTokens);

 _logger.LogInformation(
 "Total tokens: {TotalPromptTokens} prompt, {TotalCompletionTokens} completion",
 _totalPromptTokens,
 _totalCompletionTokens);
 }
}

```

## Conclusión

Integrar Azure OpenAI en aplicaciones .NET requiere atención a seguridad, resiliencia y monitoreo. Usa dependency injection, maneja errores apropiadamente, implementa rate limiting y caché cuando sea apropiado. El resultado será una integración robusta y lista para producción.

---

**Palabras clave:** Azure OpenAI, .NET integration, GPT-4, secure API, rate limiting, resilience, caching, monitoring

---

## Cierre del paso

### Verificación rápida

- La app resuelve `Kernel` desde DI y puede llamar chat/embeddings.

### Errores frecuentes y cómo desbloquearte

- Mezclar endpoints o regiones: usa el endpoint del recurso correcto.
- Key expuesta en código o repositorio: usa secrets/KeyVault.

### Ejercicios (para afianzar)

- Añade `MaxRetries` con políticas resilientes (retry/backoff).
- Separa configuración por entorno (`Development`, `Production`).

## Siguiente paso

Cuando estés listo, continúa con el **Paso 08 — Configuración de Temperatura y Tokens en Modelos LLM**.

## Paso 08 — Configuración de Temperatura y Tokens en Modelos LLM

### Mapa del paso

**Tiempo estimado:** 45–75 min

### Resultado que vas a conseguir:

Control fino del comportamiento del LLM (temperature, max tokens, top\_p, penalties).

### Objetivos de aprendizaje:

- Elegir settings según el tipo de tarea (creativa vs determinista).
- Crear presets (perfils) de settings para casos de uso.
- Evitar costes innecesarios con límites y truncado.

### Prerrequisitos:

- Haber ejecutado chat completion al menos una vez.

## Ruta guiada (paso a paso)

1. Crea una clase `PromptProfile` con settings por caso de uso.
  2. Define perfils: clasificación (temp baja), redacción (temp media), brainstorming (alta).
  3. Añade `MaxTokens` y observa la longitud.
  4. Registra el uso de tokens si tu proveedor lo devuelve.
-

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Los parámetros de configuración de LLMs como temperatura, max\_tokens, top\_p controlan el comportamiento y la calidad de las respuestas. En este tutorial aprenderás a configurar estos parámetros para diferentes casos de uso.

## Parámetros Principales

### Temperature (0.0 - 2.0)

Controla la aleatoriedad de las respuestas:

```
// Determinístico (misma respuesta siempre)
var settingsDeterminista = new OpenAIPromptExecutionSettings
{
 Temperature = 0.0
};

// Balanceado
var settingsBalanceado = new OpenAIPromptExecutionSettings
{
 Temperature = 0.7
};

// Muy creativo
var settingsCreativo = new OpenAIPromptExecutionSettings
{
 Temperature = 1.5
};
```

**Casos de uso:** - **0.0-0.3:** Clasificación, extracción de datos, respuestas precisas - **0.4-0.7:** Conversaciones naturales, asistentes - **0.8-1.2:** Contenido creativo, brainstorming - **1.3-2.0:** Muy experimental (raramente usado)

### Max Tokens

Limita la longitud de la respuesta:

```
var settings = new OpenAIPromptExecutionSettings
{
 MaxTokens = 150 // Aproximadamente 100-120 palabras
};

// Reglas generales:
// - 1 token 0.75 palabras en inglés
// - 1 token 0.6 palabras en español
// - GPT-4: max 8,192 tokens (entrada + salida)
// - GPT-4-32k: max 32,768 tokens
```

### Top P (Nucleus Sampling)

Controla la diversidad considerando probabilidad acumulativa:

```
var settings = new OpenAIPromptExecutionSettings
{
 TopP = 0.9 // Considera tokens que suman hasta 90% de probabilidad
};
```

```
// Valores típicos:
// - 0.1: Muy conservador
// - 0.9: Estándar (recomendado)
// - 1.0: Considera todas las opciones
```

**Nota:** Generalmente se usa temperature O top\_p, no ambos.

### Frequency Penalty y Presence Penalty

```
var settings = new OpenAIPromptExecutionSettings
{
 FrequencyPenalty = 0.5f, // Reduce repetición de tokens frecuentes (-2.0 a 2.0)
 PresencePenalty = 0.5f // Fomenta temas nuevos (-2.0 a 2.0)
};

// Frequency Penalty: Penaliza por frecuencia de uso
// Presence Penalty: Penaliza por presencia (sí/no), sin importar frecuencia
```

## Configuraciones por Caso de Uso

### 1. Clasificación/Categorización

```
var classificationSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.1, // Muy baja para consistencia
 MaxTokens = 50, // Respuestas cortas
 TopP = 0.1, // Muy conservador
 FrequencyPenalty = 0,
 PresencePenalty = 0
};

var prompt = """
Clasifica el siguiente texto en una de estas categorías: deportes, tecnología, política.

Texto: {{$text}}

Categoría:
""";
```

### 2. Chat Conversacional

```
var chatSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.7, // Balance entre creatividad y coherencia
 MaxTokens = 500, // Respuestas medianas
 TopP = 0.9,
 FrequencyPenalty = 0.3f, // Evitar repetición moderada
 PresencePenalty = 0.3f // Fomentar variedad
};
```

### 3. Generación Creativa

```
var creativeSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.9, // Alta creatividad
 MaxTokens = 1000, // Respuestas largas
 TopP = 0.95,
 FrequencyPenalty = 0.5f,
 PresencePenalty = 0.5f
};
```

#### 4. Extracción de Información

```
var extractionSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.0, // Determinístico
 MaxTokens = 200,
 ResponseFormat = "json_object", // Formato estructurado
 FrequencyPenalty = 0,
 PresencePenalty = 0
};
```

#### 5. Resumen de Texto

```
var summarySettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.3, // Baja, pero permite algo de variación
 MaxTokens = 300,
 TopP = 0.8,
 FrequencyPenalty = 0.2f,
 PresencePenalty = 0.2f
};
```

#### 6. Código/Programación

```
var codeSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.2, // Preciso pero con algo de flexibilidad
 MaxTokens = 1500,
 TopP = 0.9,
 FrequencyPenalty = 0,
 PresencePenalty = 0
};
```

### Servicio de Configuración Dinámica

```
public class LLMConfigurationService
{
 public enum TaskType
 {
 Classification,
 Conversation,
 Creative,
 Extraction,
 Summary,
 Code,
 Translation
 }

 public OpenAIPromptExecutionSettings GetSettingsForTask(TaskType taskType)
 {
 return taskType switch
 {
 TaskType.Classification => new OpenAIPromptExecutionSettings
 {
 Temperature = 0.1f,
 MaxTokens = 50,
 TopP = 0.1f,
 FrequencyPenalty = 0,
 PresencePenalty = 0
 }
 };
 }
}
```

```

 },

TaskType.Conversation => new OpenAIPromptExecutionSettings
{
 Temperature = 0.7f,
 MaxTokens = 500,
 TopP = 0.9f,
 FrequencyPenalty = 0.3f,
 PresencePenalty = 0.3f
} ,

TaskType.Creative => new OpenAIPromptExecutionSettings
{
 Temperature = 0.9f,
 MaxTokens = 1000,
 TopP = 0.95f,
 FrequencyPenalty = 0.5f,
 PresencePenalty = 0.5f
} ,

TaskType.Extraction => new OpenAIPromptExecutionSettings
{
 Temperature = 0.0f,
 MaxTokens = 200,
 ResponseFormat = "json_object",
 FrequencyPenalty = 0,
 PresencePenalty = 0
} ,

TaskType.Summary => new OpenAIPromptExecutionSettings
{
 Temperature = 0.3f,
 MaxTokens = 300,
 TopP = 0.8f,
 FrequencyPenalty = 0.2f,
 PresencePenalty = 0.2f
} ,

TaskType.Code => new OpenAIPromptExecutionSettings
{
 Temperature = 0.2f,
 MaxTokens = 1500,
 TopP = 0.9f,
 FrequencyPenalty = 0,
 PresencePenalty = 0
} ,

TaskType.Translation => new OpenAIPromptExecutionSettings
{
 Temperature = 0.3f,
 MaxTokens = 1000,
 TopP = 0.9f,
 FrequencyPenalty = 0,
 PresencePenalty = 0
} ,

_ => new OpenAIPromptExecutionSettings
{
}

```

```
 Temperature = 0.7f,
 MaxTokens = 500
 }
};
};
}
```

## Estimación de Tokens

```
public class TokenEstimator
{
 // Estimación aproximada (no exacta)
 public int EstimateTokens(string text)
 {
 // Regla simple: ~1.3 caracteres por token en español
 return (int)Math.Ceiling(text.Length / 1.3);
 }

 public bool WillFitInContext(string prompt, string response, int maxContextTokens = 8192)
 {
 var promptTokens = EstimateTokens(prompt);
 var responseTokens = EstimateTokens(response);

 return (promptTokens + responseTokens) < maxContextTokens;
 }

 public int CalculateMaxResponseTokens(string prompt, int maxContextTokens = 8192, int safetyMargin = 0)
 {
 var promptTokens = EstimateTokens(prompt);
 return Math.Max(0, maxContextTokens - promptTokens - safetyMargin);
 }
}
```

## Optimización de Costos

```
public class CostOptimizedLLMService
{
 private readonly Kernel _kernel;
 private readonly TokenEstimator _tokenEstimator;

 public async Task<string> GenerateWithCostLimitAsync(
 string prompt,
 int maxCostTokens = 1000)
 {
 var estimatedPromptTokens = _tokenEstimator.EstimateTokens(prompt);
 var remainingTokens = maxCostTokens - estimatedPromptTokens;

 if (remainingTokens <= 0)
 {
 throw new InvalidOperationException("Prompt excede límite de costo");
 }

 var settings = new OpenAIPromptExecutionSettings
 {
 Temperature = 0.7f,
 MaxTokens = Math.Min(remainingTokens, 500) // Limitar respuesta
 };
 }
}
```

```

 var function = _kernel.CreateFunctionFromPrompt(prompt, settings);
 var result = await _kernel.InvokeAsync(function);

 return result.GetValue<string>() ?? string.Empty;
 }
}

```

## Testing de Configuraciones

```

public class SettingsTester
{
 private readonly Kernel _kernel;

 public async Task<SettingsTestResult> TestSettingsAsync(
 string prompt,
 List<OpenAIPromptExecutionSettings> settingsVariations,
 int iterations = 3)
 {
 var results = new List<TestIteration>();

 foreach (var settings in settingsVariations)
 {
 var responses = new List<string>();

 for (int i = 0; i < iterations; i++)
 {
 var function = _kernel.CreateFunctionFromPrompt(prompt, settings);
 var result = await _kernel.InvokeAsync(function);
 responses.Add(result.GetValue<string>() ?? "");
 }

 results.Add(new TestIteration
 {
 Settings = settings,
 Responses = responses,
 Consistency = CalculateConsistency(responses),
 AverageLength = responses.Average(r => r.Length)
 });
 }

 return new SettingsTestResult
 {
 Iterations = results,
 BestSettings = results.OrderByDescending(r => r.Consistency).First().Settings
 };
 }

 private double CalculateConsistency(List<string> responses)
 {
 // Implementación simplificada
 if (responses.Count < 2) return 1.0;

 var firstLength = responses[0].Length;
 var variance = responses.Average(r => Math.Abs(r.Length - firstLength));

 return 1.0 - (variance / firstLength);
 }
}

```

```

public class TestIteration
{
 public required OpenAIPromptExecutionSettings Settings { get; init; }
 public required List<string> Responses { get; init; }
 public double Consistency { get; init; }
 public double AverageLength { get; init; }
}

public class SettingsTestResult
{
 public required List<TestIteration> Iterations { get; init; }
 public required OpenAIPromptExecutionSettings BestSettings { get; init; }
}

```

## Monitoreo de Configuraciones

```

public class SettingsMonitor
{
 private readonly Dictionary<string, SettingsMetrics> _metrics = new();

 public void TrackRequest(
 string configName,
 OpenAIPromptExecutionSettings settings,
 int promptTokens,
 int completionTokens,
 TimeSpan latency)
 {
 if (!_metrics.ContainsKey(configName))
 {
 _metrics[configName] = new SettingsMetrics { ConfigName = configName };

 var metrics = _metrics[configName];
 metrics.TotalRequests++;
 metrics.TotalPromptTokens += promptTokens;
 metrics.TotalCompletionTokens += completionTokens;
 metrics.TotalLatencyMs += latency.TotalMilliseconds;
 metrics.LastUsed = DateTime.UtcNow;
 }

 var metrics = _metrics[configName];
 metrics.TotalRequests++;
 metrics.TotalPromptTokens += promptTokens;
 metrics.TotalCompletionTokens += completionTokens;
 metrics.TotalLatencyMs += latency.TotalMilliseconds;
 metrics.LastUsed = DateTime.UtcNow;
 }

 public SettingsMetrics GetMetrics(string configName)
 {
 return _metrics.GetValueOrDefault(configName) ?? new SettingsMetrics { ConfigName = configName };
 }

 public List<SettingsMetrics> GetAllMetrics()
 {
 return _metrics.Values.OrderByDescending(m => m.TotalRequests).ToList();
 }
}

public class SettingsMetrics
{
 public required string ConfigName { get; init; }
 public long TotalRequests { get; set; }
 public long TotalPromptTokens { get; set; }
 public long TotalCompletionTokens { get; set; }
}

```

```

 public double TotalLatencyMs { get; set; }
 public DateTime? LastUsed { get; set; }

 public double AveragePromptTokens => TotalRequests > 0 ? (double)TotalPromptTokens / TotalRequests : 0;
 public double AverageCompletionTokens => TotalRequests > 0 ? (double)TotalCompletionTokens / TotalRequests : 0;
 public double AverageLatencyMs => TotalRequests > 0 ? TotalLatencyMs / TotalRequests : 0;
}

```

## Mejores Prácticas

### 1. Comenzar Conservador

```

// Empezar con settings conservadores
var initialSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.5f, // Medio
 MaxTokens = 300 // Moderado
};

// Ajustar basándose en resultados

```

### 2. Documentar Configuraciones

```

/// <summary>
/// Configuración para clasificación de tickets de soporte.
/// Temperature baja (0.1) para consistencia en categorías.
/// MaxTokens limitado (50) ya que solo necesitamos nombre de categoría.
/// </summary>
var ticketClassificationSettings = new OpenAIPromptExecutionSettings
{
 Temperature = 0.1f,
 MaxTokens = 50
};

```

### 3. A/B Testing

```

public async Task<string> ABTestSettingsAsync(string prompt)
{
 var settingsA = new OpenAIPromptExecutionSettings { Temperature = 0.5f };
 var settingsB = new OpenAIPromptExecutionSettings { Temperature = 0.7f };

 // Alternar entre configuraciones y medir resultados
 var useA = Random.Shared.Next(2) == 0;
 var settings = useA ? settingsA : settingsB;

 // Trackear qué configuración se usó
 TrackConfigUsage(useA ? "A" : "B");

 var function = _kernel.CreateFunctionFromPrompt(prompt, settings);
 return (await _kernel.InvokeAsync(function)).GetValue<string>() ?? "";
}

```

## Conclusión

La configuración correcta de temperatura, tokens y otros parámetros es crucial para obtener resultados óptimos de LLMs. Cada caso de uso requiere un balance diferente entre creatividad, consistencia y costo. Experimenta con diferentes configuraciones, mide resultados, y ajusta basándote en datos reales.

**Palabras clave:** LLM configuration, temperature, tokens, top\_p, GPT-4, prompt settings, OpenAI parameters

---

## Cierre del paso

### Verificación rápida

- Con el mismo prompt, perfiles distintos producen resultados coherentes con el objetivo.

### Errores frecuentes y cómo desbloquearte

- Usar temperatura alta para JSON: causa formatos rotos.
- No limitar tokens: respuestas largas y caras.

### Ejercicios (para afianzar)

- Crea un endpoint que reciba `mode=creative|strict` y aplique settings.
- Implementa un ‘cap’ de tokens basado en presupuesto por petición.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 09 — Estrategias de Caché para Servicios de IA**.

## Paso 09 — Estrategias de Caché para Servicios de IA

### Mapa del paso

**Tiempo estimado:** 60–90 min

### Resultado que vas a conseguir:

Caché para respuestas/embeddings con estrategias y claves bien diseñadas.

### Objetivos de aprendizaje:

- Elegir qué cachear (embeddings, prompts deterministas, resultados parciales).
- Diseñar claves de caché seguras y estables.
- Evitar cachear datos sensibles por error.

### Prerrequisitos:

- Entender qué inputs afectan al resultado (prompt + settings + modelo).

### Ruta guiada (paso a paso)

1. Identifica funciones candidatas a caché (idempotentes o semideterministas).
  2. Diseña una clave: hash(prompt + settings + modelo + versión).
  3. Aplica TTL según caso (minutos/horas/días).
  4. Mide hit-rate y controla invalidación por versión.
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

El caché es esencial para optimizar costos y rendimiento en aplicaciones de IA. Las llamadas a LLMs son costosas en tiempo y dinero. En este tutorial aprenderás estrategias efectivas de caché para servicios de IA.

## ¿Por Qué Cachear?

1. **Reducción de costos:** Menos llamadas al API = menor gasto
2. **Mejor rendimiento:** Respuestas instantáneas desde caché
3. **Resiliencia:** Funciona aunque el servicio esté caído
4. **Reducción de latencia:** Microsegundos vs segundos

## Tipos de Caché

### 1. Caché en Memoria (IMemoryCache)

```
using Microsoft.Extensions.Caching.Memory;

public class MemoryCachedAIService
{
 private readonly IMemoryCache _cache;
 private readonly IAIService _aiService;

 public MemoryCachedAIService(IMemoryCache cache, IAIService aiService)
 {
 _cache = cache;
 _aiService = aiService;
 }

 public async Task<string> GetResponseAsync(string prompt)
 {
 var cacheKey = GenerateCacheKey(prompt);

 if (_cache.TryGetValue<string>(cacheKey, out var cachedResponse))
 {
 return cachedResponse;
 }

 var response = await _aiService.GenerateAsync(prompt);

 var cacheOptions = new MemoryCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = TimeSpan.FromHours(24),
 SlidingExpiration = TimeSpan.FromHours(6),
 Priority = CacheItemPriority.Normal
 };

 _cache.Set(cacheKey, response, cacheOptions);

 return response;
 }

 private string GenerateCacheKey(string prompt)
 {
 using var sha256 = SHA256.Create();
 var hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(prompt));
 return $"ai_{BitConverter.ToString(hashBytes).Replace("-", "")}.Substring(0, 16)}";
 }
}
```

### 2. Caché Distribuido (Redis)

```
using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;
```

```

public class DistributedCachedAIService
{
 private readonly IDistributedCache _cache;
 private readonly IAIService _aiService;
 private readonly ILogger<DistributedCachedAIService> _logger;

 public DistributedCachedAIService(
 IDistributedCache cache,
 IAIService aiService,
 ILogger<DistributedCachedAIService> logger)
 {
 _cache = cache;
 _aiService = aiService;
 _logger = logger;
 }

 public async Task<AIResponse> GetResponseAsync(
 string prompt,
 CancellationToken cancellationToken = default)
 {
 var cacheKey = GenerateCacheKey(prompt);

 // Intentar obtener del caché
 var cachedBytes = await _cache.GetAsync(cacheKey, cancellationToken);

 if (cachedBytes != null)
 {
 _logger.LogInformation("Cache HIT: {CacheKey}", cacheKey);
 var cachedResponse = JsonSerializer.Deserialize<AIResponse>(cachedBytes);
 if (cachedResponse != null)
 {
 cachedResponse.FromCache = true;
 return cachedResponse;
 }
 }

 _logger.LogInformation("Cache MISS: {CacheKey}", cacheKey);

 // Generar respuesta
 var response = await _aiService.GenerateAsync(prompt, cancellationToken);

 // Guardar en caché
 var responseBytes = JsonSerializer.SerializeToUtf8Bytes(response);

 var cacheOptions = new DistributedCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = TimeSpan.FromDays(7),
 SlidingExpiration = TimeSpan.FromDays(1)
 };

 await _cache.SetAsync(cacheKey, responseBytes, cacheOptions, cancellationToken);

 response.FromCache = false;
 return response;
 }

 private string GenerateCacheKey(string prompt)
 {

```

```

 using var sha256 = SHA256.Create();
 var hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(prompt));
 return $"ai_{BitConverter.ToString(hashBytes).Replace("-", "")}";
 }
}

public class AIResponse
{
 public required string Content { get; set; }
 public DateTime GeneratedAt { get; set; } = DateTime.UtcNow;
 public bool FromCache { get; set; }
 public Dictionary<string, string>? Metadata { get; set; }
}

```

### 3. Caché de Embeddings

```

public class EmbeddingCacheService
{
 private readonly IDistributedCache _cache;
 private readonly ITextEmbeddingGenerationService _embeddingService;

 public async Task<ReadOnlyMemory<float>> GetEmbeddingAsync(
 string text,
 CancellationToken cancellationToken = default)
 {
 var cacheKey = $"emb_{ComputeHash(text)}";

 var cachedBytes = await _cache.GetAsync(cacheKey, cancellationToken);

 if (cachedBytes != null)
 {
 // Deserializar embedding
 var floatArray = new float[cachedBytes.Length / sizeof(float)];
 Buffer.BlockCopy(cachedBytes, 0, floatArray, 0, cachedBytes.Length);
 return new ReadOnlyMemory<float>(floatArray);
 }

 // Generar embedding
 var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
 new[] { text },
 kernel: null,
 cancellationToken);

 var embedding = embeddings.First();

 // Serializar y cachear
 var floats = embedding.ToArray();
 var bytes = new byte[floats.Length * sizeof(float)];
 Buffer.BlockCopy(floats, 0, bytes, 0, bytes.Length);

 await _cache.SetAsync(
 cacheKey,
 bytes,
 new DistributedCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = TimeSpan.FromDays(30)
 },
 cancellationToken);
 }
}

```

```

 return embedding;
 }

 private string ComputeHash(string text)
 {
 using var sha256 = SHA256.Create();
 var hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(text));
 return BitConverter.ToString(hashBytes).Replace("-", "");
 }
}

```

## Caché Inteligente

### Caché con Versioning

```

public class VersionedCacheService
{
 private readonly IDistributedCache _cache;
 private readonly string _version;

 public VersionedCacheService(IDistributedCache cache, string version = "v1")
 {
 _cache = cache;
 _version = version;
 }

 private string GenerateCacheKey(string key)
 {
 return $"{_version}_{key}";
 }

 public async Task InvalidateVersionAsync()
 {
 // Cambiar versión invalida todo el caché anterior
 // Implementar incremento de versión
 }
}

```

### Caché con TTL Dinámico

```

public class DynamicTTLCacheService
{
 private readonly IDistributedCache _cache;

 public async Task<string> GetWithDynamicTTLAsync(
 string prompt,
 Func<string, TimeSpan> ttlCalculator)
 {
 var cacheKey = GenerateCacheKey(prompt);
 var cached = await _cache.GetStringAsync(cacheKey);

 if (cached != null)
 {
 return cached;
 }

 var response = await GenerateResponseAsync(prompt);

```

```

 // TTL basado en características de la respuesta
 var ttl = ttlCalculator(response);

 await _cache.SetStringAsync(
 cacheKey,
 response,
 new DistributedCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = ttl
 });
}

return response;
}
}

// Uso
var response = await cacheService.GetWithDynamicTTLAsync(
 prompt,
 response => response.Length > 1000
 ? TimeSpan.FromDays(7) // Respuestas largas: TTL largo
 : TimeSpan.FromHours(1)); // Respuestas cortas: TTL corto

```

### Caché por Similitud Semántica

```

public class SemanticCacheService
{
 private readonly IDistributedCache _cache;
 private readonly ITextEmbeddingGenerationService _embeddingService;
 private readonly double _similarityThreshold = 0.95;

 public async Task<(bool Found, string? Response)> TryGetSimilarAsync(
 string prompt,
 CancellationToken cancellationToken = default)
 {
 // Generar embedding del prompt
 var embeddings = await _embeddingService.GenerateEmbeddingsAsync(
 new[] { prompt },
 kernel: null,
 cancellationToken);

 var promptEmbedding = embeddings.First();

 // Buscar prompts similares en caché (simplificado)
 // En producción, usar una base de datos vectorial
 var cachedPrompts = await GetCachedPromptsAsync();

 foreach (var cachedPrompt in cachedPrompts)
 {
 var similarity = CalculateCosineSimilarity(
 promptEmbedding,
 cachedPrompt.Embedding);

 if (similarity >= _similarityThreshold)
 {
 var response = await _cache.GetStringAsync(cachedPrompt.Key);
 if (response != null)
 {
 return (true, response);
 }
 }
 }
 }
}

```

```

 }
 }

 return (false, null);
}

private double CalculateCosineSimilarity(
 ReadOnlyMemory<float> v1,
 ReadOnlyMemory<float> v2)
{
 // Implementación de similitud coseno
 return 0.0;
}

private Task<List<CachedPrompt>> GetCachedPromptsAsync()
{
 // Implementación
 return Task.FromResult(new List<CachedPrompt>());
}
}

public class CachedPrompt
{
 public required string Key { get; init; }
 public required ReadOnlyMemory<float> Embedding { get; init; }
}

```

## Estrategias de Invalidación

### Invalidación por Tiempo

```

public class TimeBasedInvalidation
{
 private readonly IMemoryCache _cache;

 public void SetWithExpiration<T>(string key, T value, TimeSpan expiration)
 {
 _cache.Set(key, value, new MemoryCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = expiration
 });
 }
}

```

### Invalidación por Evento

```

public class EventBasedInvalidation
{
 private readonly IMemoryCache _cache;

 public void InvalidateOnEvent(string pattern)
 {
 // Invalidar todas las claves que coincidan con el patrón
 // Nota: IMemoryCache no soporta pattern matching nativamente
 // Necesitarías mantener un registro de claves
 }

 public void InvalidateRelated(string entityId)

```

```

{
 // Invalidar caché relacionado con una entidad
 var relatedKeys = new[]
 {
 $"entity_{entityId}",
 $"list_with_{entityId}",
 $"summary_of_{entityId}"
 };

 foreach (var key in relatedKeys)
 {
 _cache.Remove(key);
 }
}
}

```

### Invalidación por Tamaño

```

public class SizeLimitedCache
{
 private readonly IMemoryCache _cache;
 private readonly long _maxSizeInBytes;

 public SizeLimitedCache(IMemoryCache cache, long maxSizeInBytes)
 {
 _cache = cache;
 _maxSizeInBytes = maxSizeInBytes;
 }

 public void Set<T>(string key, T value, long estimatedSize)
 {
 _cache.Set(key, value, new MemoryCacheEntryOptions
 {
 Size = estimatedSize,
 Priority = CacheItemPriority.Normal
 });
 }
}

```

### Caché en Múltiples Niveles

```

public class MultiLevelCacheService
{
 private readonly IMemoryCache _l1Cache; // Nivel 1: Memoria local
 private readonly IDistributedCache _l2Cache; // Nivel 2: Redis
 private readonly IAIService _aiService; // Nivel 3: Servicio IA

 public async Task<string> GetResponseAsync(
 string prompt,
 CancellationToken cancellationToken = default)
 {
 var cacheKey = GenerateCacheKey(prompt);

 // Nivel 1: Memoria local
 if (_l1Cache.TryGetValue<string>(cacheKey, out var l1Response))
 {
 return l1Response;
 }
 }
}

```

```

 // Nivel 2: Caché distribuido
 var l2Response = await _l2Cache.GetStringAsync(cacheKey, cancellationToken);
 if (l2Response != null)
 {
 // Poblar L1
 _l1Cache.Set(cacheKey, l2Response, TimeSpan.FromMinutes(5));
 return l2Response;
 }

 // Nivel 3: Generar desde IA
 var response = await _aiService.GenerateAsync(prompt, cancellationToken);

 // Poblar ambos niveles
 _l1Cache.Set(cacheKey, response, TimeSpan.FromMinutes(5));
 await _l2Cache.SetStringAsync(
 cacheKey,
 response,
 new DistributedCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = TimeSpan.FromHours(24)
 },
 cancellationToken);

 return response;
}

private string GenerateCacheKey(string prompt)
{
 using var sha256 = SHA256.Create();
 var hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(prompt));
 return BitConverter.ToString(hashBytes).Replace("-", "");
}
}

```

## Monitoreo de Caché

```

public class CacheMonitor
{
 private long _hits;
 private long _misses;
 private readonly ILogger<CacheMonitor> _logger;

 public void RecordHit()
 {
 Interlocked.Increment(ref _hits);
 }

 public void RecordMiss()
 {
 Interlocked.Increment(ref _misses);
 }

 public CacheStatistics GetStatistics()
 {
 var totalRequests = _hits + _misses;
 var hitRate = totalRequests > 0 ? (double)_hits / totalRequests : 0;

 return new CacheStatistics
 }
}

```

```

 {
 Hits = _hits,
 Misses = _misses,
 TotalRequests = totalRequests,
 HitRate = hitRate
 };
}

public void LogStatistics()
{
 var stats = GetStatistics();
 _logger.LogInformation(
 "Cache Stats: {Hits} hits, {Misses} misses, {HitRate:P2} hit rate",
 stats.Hits,
 stats.Misses,
 stats.HitRate);
}

public class CacheStatistics
{
 public long Hits { get; init; }
 public long Misses { get; init; }
 public long TotalRequests { get; init; }
 public double HitRate { get; init; }
}

```

## Mejores Prácticas

### 1. Definir TTL Apropriado

```

// TTL basado en volatilidad de datos
var staticDataTTL = TimeSpan.FromDays(7);
var dynamicDataTTL = TimeSpan.FromMinutes(5);
var realtimeDataTTL = TimeSpan.FromSeconds(30);

```

### 2. Considerar Tamaño de Caché

```

// Limitar tamaño para evitar problemas de memoria
services.AddMemoryCache(options =>
{
 options.SizeLimit = 1024; // Límite en unidades arbitrarias
});

```

### 3. Caché Selectivo

```

// Solo cachear operaciones costosas
public async Task<string> GetDataAsync(string id, bool useCache = true)
{
 if (!useCache || IsRealtimeRequired(id))
 {
 return await FetchFromSourceAsync(id);
 }

 return await GetFromCacheAsync(id);
}

```

#### 4. Warming del Caché

```
public class CacheWarmer : IHostedService
{
 public async Task StartAsync(CancellationToken cancellationToken)
 {
 // Pre-cargar datos frecuentes al iniciar
 await WarmFrequentQueriesAsync(cancellationToken);
 }

 private async Task WarmFrequentQueriesAsync(CancellationToken cancellationToken)
 {
 var frequentQueries = await GetFrequentQueriesAsync();

 foreach (var query in frequentQueries)
 {
 await _cachedService.GetResponseAsync(query, cancellationToken);
 }
 }

 public Task StopAsync(CancellationToken cancellationToken) => Task.CompletedTask;
}
```

### Conclusión

El caché es esencial para aplicaciones de IA eficientes. Implementa caché en múltiples niveles, monitorea hit rates, y ajusta TTLs basándose en patrones de uso. Una estrategia de caché bien diseñada puede reducir costos hasta 80 % y mejorar significativamente el rendimiento.

---

**Palabras clave:** caching strategies, distributed cache, Redis, memory cache, AI optimization, performance, cost reduction

---

### Cierre del paso

#### Verificación rápida

- Peticiones repetidas reducen latencia y llamadas al proveedor.

#### Errores frecuentes y cómo desbloquearte

- Clave incompleta: cambios de settings devuelven respuestas incorrectas.
- Cachear PII o secretos: enmascara o evita por política.

#### Ejercicios (para afianzar)

- Añade un ‘cache busting’ por versión de prompt.
- Cachea embeddings de documentos y compara coste antes/después.

#### Siguiente paso

Cuando estés listo, continúa con el **Paso 10 — Manejo de Errores en Aplicaciones de IA con .NET**.

## Paso 10 — Manejo de Errores en Aplicaciones de IA con .NET

#### Mapa del paso

**Tiempo estimado:** 45–75 min

## Resultado que vas a conseguir:

Manejo de errores robusto: timeouts, rate limits, respuestas vacías y fallbacks.

## Objetivos de aprendizaje:

- Clasificar errores (transitorios vs permanentes).
- Definir reintentos con backoff y límites.
- Devolver mensajes útiles sin filtrar detalles sensibles.

## Prerrequisitos:

- Haber llamado a un proveedor LLM desde .NET.

## Ruta guiada (paso a paso)

1. Envuelve llamadas al proveedor y captura excepciones relevantes.
2. Añade reintentos para 429/5xx con backoff.
3. Define un fallback (respuesta segura o modo degradado).
4. Registra correlación (request id) y contexto mínimo.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Las aplicaciones de IA enfrentan errores únicos: rate limits, timeouts, respuestas inesperadas del modelo, y fallos de red. Este tutorial cubre estrategias robustas de manejo de errores para aplicaciones de IA en producción.

## Tipos de Errores Comunes

### 1. Errores de Red

```
try
{
 var response = await aiService.GenerateAsync(prompt);
}
catch (HttpRequestException ex)
{
 _logger.LogError(ex, "Error de conexión al servicio de IA");
 // Retry o fallback
}
catch (TaskCanceledException ex)
{
 _logger.LogError(ex, "Timeout al llamar al servicio de IA");
 // Timeout handling
}
```

### 2. Rate Limiting (429)

```
catch (RequestFailedException ex) when (ex.Status == 429)
{
 var retryAfter = ex.GetRawResponse()?.Headers
 .TryGetValue("Retry-After", out var value) == true
 ? int.Parse(value) : 60;

 _logger.LogWarning("Rate limit alcanzado. Reintentar en {Seconds}s", retryAfter);
 await Task.Delay(TimeSpan.FromSeconds(retryAfter));
 // Retry
}
```

### 3. Respuestas Inesperadas del Modelo

```
var response = await aiService.GenerateAsync(prompt);

if (string.IsNullOrWhiteSpace(response))
{
 throw new AIServiceException("Modelo retornó respuesta vacía");
}

if (!IsValidFormat(response))
{
 _logger.LogWarning("Respuesta en formato inesperado: {Response}", response);
 // Intentar parsear o usar respuesta por defecto
}
```

## Patrones de Retry

### Retry Simple

```
public async Task<string> GenerateWithRetryAsync(
 string prompt,
 int maxRetries = 3)
{
 for (int attempt = 0; attempt < maxRetries; attempt++)
 {
 try
 {
 return await _aiService.GenerateAsync(prompt);
 }
 catch (Exception ex) when (attempt < maxRetries - 1)
 {
 _logger.LogWarning(ex, "Intento {Attempt} falló, reintentando...", attempt + 1);
 await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, attempt)));
 }
 }

 throw new AIServiceException($"Fallo después de {maxRetries} intentos");
}
```

### Exponential Backoff con Polly

```
using Polly;

var retryPolicy = Policy
 .Handle<HttpRequestException>()
 .Or<RequestFailedException>(ex => ex.Status == 429 || ex.Status == 503)
 .WaitAndRetryAsync(
 retryCount: 3,
 sleepDurationProvider: attempt => TimeSpan.FromSeconds(Math.Pow(2, attempt)),
 onRetry: (exception, timeSpan, retryCount, context) =>
 {
 _logger.LogWarning(
 "Retry {RetryCount} después de {Delay}s: {Error}",
 retryCount, timeSpan.TotalSeconds, exception.Message);
 });

var response = await retryPolicy.ExecuteAsync(
 async () => await _aiService.GenerateAsync(prompt));
```

## Mejores Prácticas

1. Siempre loggear errores con contexto
2. Implementar circuit breakers para servicios externos
3. Tener fallbacks para funcionalidad crítica
4. Validar respuestas antes de usarlas

**Palabras clave:** error handling, retry patterns, Polly, resilience, exception handling, AI services

---

## Cierre del paso

### Verificación rápida

- Ante fallos simulados, la app no se cae y entrega un fallback controlado.

### Errores frecuentes y cómo desbloquearte

- Reintentar sin límite: tormenta de reintentos.
- Loggear prompts con datos sensibles: sanitiza.

### Ejercicios (para afianzar)

- Implementa un circuito (circuit breaker) para fallos repetidos.
- Añade un ‘modo offline’ que devuelva respuestas dummy para desarrollo.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 11 — Salida JSON Estructurada con LLMs**.

## Paso 11 — Salida JSON Estructurada con LLMs

### Mapa del paso

**Tiempo estimado:** 60–90 min

### Resultado que vas a conseguir:

Salida JSON estructurada: esquema, validación y reparación de respuestas.

### Objetivos de aprendizaje:

- Forzar formato de salida para integración con código.
- Validar el JSON y mapear a DTOs.
- Diseñar recuperación: re-prompt o corrección cuando falle el parseo.

### Prerrequisitos:

- Conocer serialización JSON en .NET.

### Ruta guiada (paso a paso)

1. Define un esquema (campos, tipos, reglas) y DTOs.
  2. Configura el modelo para responder como `json_object` si está disponible.
  3. Parsea y valida; si falla, aplica ‘repair’ o reintentó con prompt de corrección.
  4. Crea tests de casos malos (texto extra, campos faltantes).
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Salida JSON Estructurada con LLMs en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}
```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
```

```
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}
```

```
 }
}
```

## Testing

### Unit Tests

```
using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}
```

### Integration Tests

```
[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}
```

## Optimización

### Performance

1. Usa caché cuando sea apropiado

2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
  - Guías de Azure OpenAI
  - Patrones de diseño en .NET
  - Mejores prácticas de arquitectura
- 

**Palabras clave:** structured output, JSON, schema validation, type safety, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

### Cierre del paso

#### Verificación rápida

- El consumidor recibe DTOs válidos o un error controlado; nunca JSON ‘a medias’.

#### Errores frecuentes y cómo desbloquearte

- Confiar ciegamente en el LLM: siempre valida.
- No versionar el esquema: rompe compatibilidad.

#### Ejercicios (para afianzar)

- Añade validación semántica (rangos, enumeraciones, dependencias).
- Implementa ‘strict mode’: rechaza campos desconocidos.

#### Siguiente paso

Cuando estés listo, continúa con el **Paso 12 — Testing de Servicios de IA en .NET**.

## Paso 12 — Testing de Servicios de IA en .NET

#### Mapa del paso

**Tiempo estimado:** 60–90 min

#### Resultado que vas a conseguir:

Testing de servicios IA: unit tests con mocks + tests de contrato para prompts.

#### Objetivos de aprendizaje:

- Separar lógica de negocio del proveedor LLM.
- Testear prompts con casos fijos (golden tests).
- Medir regresiones cuando cambias prompt/settings/modelo.

#### Prerrequisitos:

- Conocer el framework de tests que uses (xUnit/NUnit/MSTest).

### Ruta guiada (paso a paso)

1. Extrae interfaces para chat/embeddings.
  2. Mockea respuestas del LLM para unit tests deterministas.
  3. Crea tests de contrato para salida JSON (schema).
  4. Introduce un conjunto de ‘fixtures’ de prompts y entradas.
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Testing de Servicios de IA en .NET en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

### Conceptos Clave

#### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
```

```

 _logger.LogError(ex, "Error procesando input");
 throw;
 }
}
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
```

```

 Task<string> ProcessAsync(string input);
 }

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);
 }
}

```

```

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
}

private Kernel CreateRealKernel()
{
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
- Guías de Azure OpenAI
- Patrones de diseño en .NET
- Mejores prácticas de arquitectura

**Palabras clave:** testing, mocking, integration tests, AI services, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

## Cierre del paso

### Verificación rápida

- Los tests se ejecutan sin llamar a Azure/OpenAI (modo offline).

### Errores frecuentes y cómo desbloquearte

- Tests frágiles por texto exacto: compara estructura/heurísticas.
- No aislar el proveedor: tests lentos y caros.

## Ejercicios (para afianzar)

- Añade un test que falle si el JSON no cumple el esquema.
- Añade un pipeline que ejecute tests al cambiar prompts.

## Siguiente paso

Cuando estés listo, continúa con el **Paso 13** — Inyección de Dependencias con Semantic Kernel.

# Paso 13 — Inyección de Dependencias con Semantic Kernel

## Mapa del paso

**Tiempo estimado:** 60–90 min

**Resultado que vas a conseguir:**

Arquitectura con DI: composición limpia del Kernel, servicios y plugins.

**Objetivos de aprendizaje:**

- Configurar servicios con DI y scopes correctos.
- Evitar singlettons peligrosos con estado mutable de chat.
- Diseñar servicios pequeños y testeables.

**Prerrequisitos:**

- Conocer Microsoft.Extensions.DependencyInjection a nivel básico.

## Ruta guiada (paso a paso)

1. Define interfaces para servicios (chat, embeddings, router).
2. Registra configuración tipada y `HttpClientFactory`.
3. Registra el `Kernel` y plugins según tu estrategia.
4. Separa ‘infra’ (proveedores) de ‘app’ (casos de uso).

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Inyección de Dependencias con Semantic Kernel en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }
}
```

```

public async Task<string> ProcessAsync(string input)
{
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
}
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```

builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();

```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```

var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);

```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```

public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory

```

```

{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}

```

## Patrón 2: Strategy

```

public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");
 }
}

```

```

 // Assert
 Assert.NotNull(result);
 }
}

Integration Tests

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
- Guías de Azure OpenAI
- Patrones de diseño en .NET
- Mejores prácticas de arquitectura

---

**Palabras clave:** dependency injection, DI, ASP.NET Core, Semantic Kernel, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Puedes construir el contenedor y resolver servicios sin exceptions.

### Errores frecuentes y cómo desbloquearte

- Guardar `ChatHistory` en singleton: fuga de estado entre usuarios.
- Dependencias circulares: revisar capas.

### Ejercicios (para afianzar)

- Crea un `ChatSessionFactory` por usuario.
- Introduce decoradores para logging/metrics.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 14 — Comprensión de Consultas con Lenguaje Natural**.

## Paso 14 — Comprensión de Consultas con Lenguaje Natural

### Mapa del paso

**Tiempo estimado:** 60–90 min

#### Resultado que vas a conseguir:

Comprensión de consultas: extracción de intención + parámetros para ejecutar acciones.

#### Objetivos de aprendizaje:

- Convertir lenguaje natural en acciones con parámetros.
- Diseñar contratos de ‘consulta interpretada’ (JSON).
- Validar y normalizar entradas antes de ejecutar.

#### Prerrequisitos:

- Pasos 03 (intenciones) y 11 (JSON) recomendados.

### Ruta guiada (paso a paso)

1. Define el esquema: acción + filtros + orden + límites.
2. Construye prompt para transformar texto a JSON.
3. Valida y aplica defaults (limit, sort).
4. Conecta con un ‘executor’ que traduzca a consultas reales (DB/API).

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Comprensión de Consultas con Lenguaje Natural en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}
```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas

2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}
```

## Testing

### Unit Tests

```
using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}
```

### Integration Tests

```
[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}
```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

## Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
- Guías de Azure OpenAI
- Patrones de diseño en .NET
- Mejores prácticas de arquitectura

---

**Palabras clave:** NLU, query understanding, intent extraction, entity recognition, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Consultas en lenguaje natural se traducen a una estructura ejecutable.

### Errores frecuentes y cómo desbloquearte

- Inputs ambiguos: pide aclaración o devuelve ‘necesito más datos’.
- Ejecutar sin validar: riesgos de seguridad/coste.

### Ejercicios (para afianzar)

- Añade una fase de confirmación (“¿Quieres decir X?”).
- Implementa un catálogo de acciones permitidas (allow-list).

## Siguiente paso

Cuando estés listo, continúa con el **Paso 15** — Sistemas de Búsqueda Semántica en Producción.

## Paso 15 — Sistemas de Búsqueda Semántica en Producción

### Mapa del paso

**Tiempo estimado:** 75–120 min

### Resultado que vas a conseguir:

Búsqueda semántica en producción: chunking, metadata, re-ranking y evaluación.

### Objetivos de aprendizaje:

- Diseñar pipeline: ingesta → embeddings → indexado → consulta → ranking.
- Añadir metadata y filtros (tenant, idioma, fecha).
- Definir métricas de calidad (precision@k, recall@k).

### Prerrequisitos:

- Paso 05 (embeddings) recomendado.

## Ruta guiada (paso a paso)

1. Define estrategia de chunking (tamaño/solape).
  2. Genera embeddings por chunk y almacena metadata.
  3. Implementa filtros antes del ranking.
  4. Añade re-ranking (si aplica) y evalúa con dataset pequeño.
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Sistemas de Búsqueda Semántica en Producción en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}
```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
```

```

 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }
}

```

```

private Kernel CreateRealKernel()
{
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
}
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
  - Guías de Azure OpenAI
  - Patrones de diseño en .NET
  - Mejores prácticas de arquitectura
- 

**Palabras clave:** semantic search, vector database, production ready, scalability, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Puedes explicar por qué un resultado aparece (metadata + score).

### Errores frecuentes y cómo desbloquearte

- Chunks demasiado grandes: pierdes granularidad y sube coste.
- No evaluar: mejoras ‘a ojo’ que empeoran métricas.

### Ejercicios (para afianzar)

- Crea 20 queries y mide precision@5 antes/después de chunking.
- Añade un score mínimo para descartar resultados débiles.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 16** — Configuración de HttpClient para Servicios de IA.

# Paso 16 — Configuración de HttpClient para Servicios de IA

## Mapa del paso

Tiempo estimado: 45–75 min

Resultado que vas a conseguir:

HttpClient correcto para IA: timeouts, pooling, resiliencia y trazabilidad.

Objetivos de aprendizaje:

- Evitar `new HttpClient()` por llamada (sockets agotados).
- Configurar timeouts realistas para LLMs.
- Añadir headers de correlación y logging mínimo.

Prerrequisitos:

- Uso básico de `IHttpClientFactory` recomendado.

## Ruta guiada (paso a paso)

1. Registra `HttpClient` con `IHttpClientFactory`.
2. Define timeouts por tipo de operación (chat vs embeddings).
3. Configura reintentos para fallos transitorios.
4. Añade un `DelegatingHandler` para correlación.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Configuración de `HttpClient` para Servicios de IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");
 }
 }
}
```

```

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
}
catch (Exception ex)
{
 _logger.LogError(ex, "Error procesando input");
 throw;
}
}
}
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```

public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

```

 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}

```

## Patrón 2: Strategy

```

public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests

```

```

{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
- Guías de Azure OpenAI
- Patrones de diseño en .NET
- Mejores prácticas de arquitectura

---

**Palabras clave:** HttpClient, timeout, retry, connection pooling, performance, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- La app mantiene estabilidad bajo carga sin agotar sockets.

### Errores frecuentes y cómo desbloquearte

- Timeouts demasiado bajos: cortes falsos en respuestas largas.
- Reintentos agresivos: duplicas coste y saturas el proveedor.

### Ejercicios (para afianzar)

- Añade logs estructurados con duración y tamaño de input.
- Separa clientes: uno para chat y otro para embeddings.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 17 — Guardrails y Validación en Sistemas de IA**.

## Paso 17 — Guardrails y Validación en Sistemas de IA

### Mapa del paso

**Tiempo estimado:** 60–90 min

**Resultado que vas a conseguir:**

Guardrails: validación, políticas, límites y controles para reducir riesgo.

**Objetivos de aprendizaje:**

- Definir allow-lists y validaciones de entrada/salida.
- Mitigar prompt injection y salidas no deseadas.
- Aplicar ‘defense in depth’: reglas + validación + observabilidad.

**Prerrequisitos:**

- Pasos 10 y 11 recomendados.

### Ruta guiada (paso a paso)

1. Define políticas: límites de longitud, lenguaje permitido, formato.
2. Implementa validación de output (schema, enumeraciones).
3. Añade un ‘policy layer’ antes de llamar al LLM.
4. Registra violaciones y crea un modo de bloqueo/alerta.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Guardrails y Validación en Sistemas de IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```

// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs

6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}
```

## Testing

### Unit Tests

```
using Xunit;
using Moq;

public class ExampleServiceTests
```

```

{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

## Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
  - Guías de Azure OpenAI
  - Patrones de diseño en .NET
  - Mejores prácticas de arquitectura
- 

**Palabras clave:** guardrails, validation, safety, content filtering, moderation, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Entradas maliciosas se detectan y no llegan al modelo (o se neutralizan).

### Errores frecuentes y cómo desbloquearte

- Confiar solo en el prompt: necesitas validación real.
- No separar datos de instrucciones: facilita la injection.

### Ejercicios (para afianzar)

- Crea 10 ejemplos de prompt injection y asegúrate de bloquearlos.
- Añade 'red teaming' básico en tests automatizados.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 18 — Workflows Multi-Paso con IA**.

## Paso 18 — Workflows Multi-Paso con IA

### Mapa del paso

**Tiempo estimado:** 75–120 min

### Resultado que vas a conseguir:

Workflows multi-paso: planificación, ejecución por fases y manejo de estado.

### Objetivos de aprendizaje:

- Descomponer una tarea grande en pasos verificables.
- Almacenar estado intermedio y reintentar fases fallidas.
- Diseñar trazabilidad por etapa.

### Prerrequisitos:

- Pasos 4, 10 y 13 recomendados.

## Ruta guiada (paso a paso)

1. Define un workflow con etapas (plan → ejecutar → validar).
  2. Modela el estado (DTO) y persiste si es necesario.
  3. Implementa timeouts y reintentos por etapa.
  4. Añade validación entre etapas (gates).
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Workflows Multi-Paso con IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}
```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
```

```

 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }
}

```

```

private Kernel CreateRealKernel()
{
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
}
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
  - Guías de Azure OpenAI
  - Patrones de diseño en .NET
  - Mejores prácticas de arquitectura
- 

**Palabras clave:** workflows, orchestration, multi-step, chaining, pipelines, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Puedes observar cada etapa y su output; los fallos no rompen todo el flujo.

### Errores frecuentes y cómo desbloquearte

- Workflows ‘mágicos’ sin checkpoints: difíciles de depurar.
- Mezclar responsabilidades: separar orquestación y ejecución.

### Ejercicios (para afianzar)

- Añade un modo ‘dry-run’ que no llame al LLM.
- Implementa compensación: si falla etapa 3, revierte etapa 2.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 19 — Optimización de Costos en Aplicaciones de IA**.

# Paso 19 — Optimización de Costos en Aplicaciones de IA

## Mapa del paso

**Tiempo estimado:** 60–90 min

**Resultado que vas a conseguir:**

Optimización de costes: presupuestos, batch, caché y límites por usuario/tenant.

**Objetivos de aprendizaje:**

- Entender drivers de coste (tokens, llamadas, embeddings).
- Aplicar caché y batching donde aporta.
- Diseñar presupuestos (budget) y cuotas.

**Prerrequisitos:**

- Paso 9 (caché) recomendado.

## Ruta guiada (paso a paso)

1. Instrumenta tokens/requests por endpoint o caso de uso.
2. Define presupuestos diarios/semanales y límites por usuario.
3. Aplica batch embeddings y reduce prompts redundantes.
4. Crea alertas por anomalías de coste.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Optimización de Costos en Aplicaciones de IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");
 }
 }
}
```

```

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
}
catch (Exception ex)
{
 _logger.LogError(ex, "Error procesando input");
 throw;
}
}
}
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```

public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

```

 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}

```

## Patrón 2: Strategy

```

public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests

```

```

{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
- Guías de Azure OpenAI
- Patrones de diseño en .NET
- Mejores prácticas de arquitectura

**Palabras clave:** cost optimization, token management, batching, efficiency, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

## Cierre del paso

### Verificación rápida

- Puedes estimar coste por petición y ver tendencias por día.

### Errores frecuentes y cómo desbloquearte

- Optimizar sin medir: primero instrumenta.
- Reducir tokens rompiendo calidad: valida con métricas.

### Ejercicios (para afianzar)

- Implementa ‘modo económico’ con resumen y prompts más cortos.
- Añade un límite duro por usuario y un mensaje de ‘cuota alcanzada’.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 20** — Monitoreo y Observabilidad de Servicios de IA.

## Paso 20 — Monitoreo y Observabilidad de Servicios de IA

### Mapa del paso

**Tiempo estimado:** 60–90 min

**Resultado que vas a conseguir:**

Observabilidad: logs, métricas, trazas y dashboards para IA.

**Objetivos de aprendizaje:**

- Registrar latencia, tokens y errores por operación.
- Añadir trazas distribuidas y correlación.
- Detectar degradación (calidad/latencia) antes del usuario.

**Prerrequisitos:**

- Conocer logging en .NET y conceptos básicos de observabilidad.

### Ruta guiada (paso a paso)

1. Define un modelo de logs: request id, user id anonimizado, operation, duración.
2. Mide latencia y rate limits (429) como métricas.
3. Introduce trazas (activity) por fase del workflow.
4. Añade alertas: p95 latencia, tasa de error, coste.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Monitoreo y Observabilidad de Servicios de IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```

// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs

6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}
```

## Testing

### Unit Tests

```
using Xunit;
using Moq;

public class ExampleServiceTests
```

```

{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

## Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
  - Guías de Azure OpenAI
  - Patrones de diseño en .NET
  - Mejores prácticas de arquitectura
- 

**Palabras clave:** monitoring, observability, Application Insights, logging, metrics, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Puedes seguir una petición de extremo a extremo con correlación.

### Errores frecuentes y cómo desbloquearte

- Loggear prompts completos: filtra/sanitiza para privacidad.
- Sin cardinalidad controlada: métricas inutilizables.

### Ejercicios (para afianzar)

- Añade un dashboard con p50/p95/p99 y error rate.
- Introduce muestreo (sampling) para trazas en producción.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 21 — Arquitectura de Microservicios con IA**.

## Paso 21 — Arquitectura de Microservicios con IA

### Mapa del paso

**Tiempo estimado:** 75–120 min

### Resultado que vas a conseguir:

Arquitectura de microservicios con IA: límites, contratos y despliegue por capacidades.

### Objetivos de aprendizaje:

- Separar responsabilidades (chat, embeddings, retrieval, policy).
- Definir contratos y versionado entre servicios.
- Evitar acoplamiento al proveedor IA en toda la plataforma.

### Prerrequisitos:

- Experiencia básica en arquitectura de servicios y DI.

## Ruta guiada (paso a paso)

1. Identifica capacidades como servicios (gateway, orchestrator, retrieval).
  2. Define contratos (DTO) y versionado.
  3. Introduce colas para tareas largas (si aplica).
  4. Diseña resiliencia: timeouts, circuit breakers, bulkheads.
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Arquitectura de Microservicios con IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
 }
 catch (Exception ex)
 {
 _logger.LogError(ex, "Error procesando input");
 throw;
 }
 }
}
```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```
public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

### Patrón 2: Strategy

```
public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
```

```

 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests
{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }
}

```

```

private Kernel CreateRealKernel()
{
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
}
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
  - Guías de Azure OpenAI
  - Patrones de diseño en .NET
  - Mejores prácticas de arquitectura
- 

**Palabras clave:** microservices, architecture, distributed systems, AI integration, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Puedes explicar qué servicio falla y cómo se degrada sin caída total.

### Errores frecuentes y cómo desbloquearte

- Microservicios por moda: hazlo por límites claros y escalabilidad.
- Sin gobernanza: versionado y observabilidad son obligatorios.

### Ejercicios (para afianzar)

- Diseña un diagrama de servicios y define SLAs por operación.
- Crea un contrato 'IAResponse' común con metadata de coste/latencia.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 22** — Seguridad en Aplicaciones de IA.

## Paso 22 — Seguridad en Aplicaciones de IA

### Mapa del paso

Tiempo estimado: 60–90 min

Resultado que vas a conseguir:

Seguridad: secretos, PII, permisos, prompt injection y cumplimiento.

Objetivos de aprendizaje:

- Gestionar secretos correctamente (Key Vault/secret stores).
- Reducir exposición de datos sensibles al modelo.
- Aplicar controles de acceso, auditoría y cumplimiento.

Prerrequisitos:

- Pasos 10, 17 y 20 recomendados.

### Ruta guiada (paso a paso)

1. Clasifica datos: qué puede salir al modelo y qué no.
2. Enmascara PII antes de prompts; rehidrata si procede.
3. Aplica roles/permisos por operación (no todo usuario puede todo).
4. Audita accesos y rotación de claves.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Este tutorial cubre aspectos esenciales de Seguridad en Aplicaciones de IA en aplicaciones .NET con Semantic Kernel y Azure OpenAI.

## Conceptos Clave

### Fundamentos

La implementación correcta de estos conceptos es crucial para aplicaciones de IA robustas y escalables.

```
// Ejemplo de implementación
public class ExampleService
{
 private readonly Kernel _kernel;
 private readonly ILogger<ExampleService> _logger;

 public ExampleService(Kernel kernel, ILogger<ExampleService> logger)
 {
 _kernel = kernel;
 _logger = logger;
 }

 public async Task<string> ProcessAsync(string input)
 {
 try
 {
 _logger.LogInformation("Procesando input: {Input}", input);

 var function = _kernel.CreateFunctionFromPrompt(
 "Procesa: {{\}}");
 }
 }
}
```

```

 var result = await _kernel.InvokeAsync(function, new KernelArguments
 {
 ["input"] = input
 });

 return result.GetValue<string>() ?? string.Empty;
}
catch (Exception ex)
{
 _logger.LogError(ex, "Error procesando input");
 throw;
}
}
}
}

```

## Implementación Práctica

### Paso 1: Configuración

Configura los componentes necesarios en tu aplicación:

```
builder.Services.AddScoped<ExampleService>();
builder.Services.AddLogging();
```

### Paso 2: Uso

Implementa la lógica específica para tu caso de uso:

```
var service = serviceProvider.GetRequiredService<ExampleService>();
var result = await service.ProcessAsync("ejemplo");
Console.WriteLine(result);
```

## Mejores Prácticas

1. **Validación:** Siempre valida las entradas y salidas
2. **Logging:** Implementa logging estructurado
3. **Manejo de Errores:** Usa try-catch apropiadamente
4. **Testing:** Escribe tests unitarios e de integración
5. **Documentación:** Documenta tu código y APIs
6. **Monitoreo:** Implementa métricas y alertas
7. **Seguridad:** Protege credenciales y datos sensibles

## Patrones Avanzados

### Patrón 1: Factory

```

public interface IServiceFactory
{
 ExampleService Create();
}

public class ServiceFactory : IServiceFactory
{
 private readonly IServiceProvider _serviceProvider;

 public ServiceFactory(IServiceProvider serviceProvider)
 {
 _serviceProvider = serviceProvider;
 }

 public ExampleService Create()
 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}
```

```

 {
 return _serviceProvider.GetRequiredService<ExampleService>();
 }
}

```

## Patrón 2: Strategy

```

public interface IProcessingStrategy
{
 Task<string> ProcessAsync(string input);
}

public class StandardStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación estándar
 return await Task.FromResult(input);
 }
}

public class AdvancedStrategy : IProcessingStrategy
{
 public async Task<string> ProcessAsync(string input)
 {
 // Implementación avanzada
 return await Task.FromResult(input.ToUpper());
 }
}

```

## Testing

### Unit Tests

```

using Xunit;
using Moq;

public class ExampleServiceTests
{
 [Fact]
 public async Task ProcessAsync_WithValidInput_ReturnsResult()
 {
 // Arrange
 var mockKernel = new Mock<Kernel>();
 var mockLogger = new Mock<ILogger<ExampleService>>();
 var service = new ExampleService(mockKernel.Object, mockLogger.Object);

 // Act
 var result = await service.ProcessAsync("test");

 // Assert
 Assert.NotNull(result);
 }
}

```

### Integration Tests

```

[Collection("Integration")]
public class ExampleServiceIntegrationTests

```

```

{
 [Fact]
 public async Task ProcessAsync_EndToEnd_Success()
 {
 // Configurar servicio real
 var kernel = CreateRealKernel();
 var logger = NullLogger<ExampleService>.Instance;
 var service = new ExampleService(kernel, logger);

 // Ejecutar
 var result = await service.ProcessAsync("integration test");

 // Verificar
 Assert.NotEmpty(result);
 }

 private Kernel CreateRealKernel()
 {
 // Configuración real para tests de integración
 return Kernel.CreateBuilder().Build();
 }
}

```

## Optimización

### Performance

1. Usa caché cuando sea apropiado
2. Implementa batching para operaciones múltiples
3. Considera async/await correctamente
4. Monitorea métricas de rendimiento

### Escalabilidad

1. Diseña para procesamiento distribuido
2. Usa message queues para operaciones asíncronas
3. Implementa load balancing
4. Considera sharding de datos

## Conclusión

La implementación correcta de estos conceptos es fundamental para aplicaciones de IA robustas y escalables. Sigue las mejores prácticas, implementa testing exhaustivo, y monitorea constantemente el rendimiento en producción.

## Recursos Adicionales

- Documentación oficial de Semantic Kernel
- Guías de Azure OpenAI
- Patrones de diseño en .NET
- Mejores prácticas de arquitectura

---

**Palabras clave:** security, API keys, authentication, data protection, privacy, .NET, C#, Semantic Kernel, Azure OpenAI, best practices

---

## Cierre del paso

### Verificación rápida

- Puedes demostrar que PII no se loggea ni se envía cuando no toca.

### Errores frecuentes y cómo desbloquearte

- Filtrar secrets en logs/telemetría: sanitiza siempre.
- No limitar herramientas/acciones: riesgo de abuso.

### Ejercicios (para afianzar)

- Implementa un ‘PII detector’ simple y bloquea envíos.
- Añade rotación de claves y prueba caída controlada.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 23** — Implementación de Routers Conversacionales Inteligentes.

## Paso 23 — Implementación de Routers Conversacionales Inteligentes

### Mapa del paso

**Tiempo estimado:** 90–150 min

**Resultado que vas a conseguir:**

Un router conversacional que decide qué ‘habilidad’ ejecutar según intención y contexto.

**Objetivos de aprendizaje:**

- Diseñar un router con clasificación + reglas + fallback.
- Separar routing (decisión) de ejecución (acciones).
- Medir tasa de acierto y manejar ambigüedad.

**Prerrequisitos:**

- Pasos 03, 13 y 17 altamente recomendados.

### Ruta guiada (paso a paso)

1. Define intents/capacidades que el router puede ejecutar (allow-list).
2. Crea un clasificador (LLM) con salida JSON y confianza.
3. Añade reglas deterministas encima (prioridades, overrides).
4. Implementa fallback: pedir aclaración o respuesta genérica segura.
5. Instrumenta: qué ruta se eligió y por qué.

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

Los routers conversacionales son el cerebro de un chatbot: deciden cómo manejar cada mensaje del usuario. En este tutorial aprenderás a implementar un router conversacional robusto que gestiona estados, intenciones, y flujos complejos.

## Arquitectura de un Router

Un router conversacional efectivo necesita:

1. **Gestión de Estado**: Recordar el contexto de la conversación
2. **Resolución de Intenciones**: Determinar qué quiere hacer el usuario
3. **Enrutamiento**: Dirigir a los handlers apropiados
4. **Comandos Globales**: Manejar comandos prioritarios (menú, ayuda)
5. **Fallbacks**: Respuestas cuando no se entiende el mensaje

## Modelo de Estado de Conversación

```
public class ConversationState
{
 public required string UserId { get; init; }
 public string? CurrentFlow { get; set; }
 public Dictionary<string, object> Context { get; set; } = new();
 public DateTime LastInteraction { get; set; } = DateTime.UtcNow;
 public string? PendingAction { get; set; }
 public int MessageCount { get; set; }
}

public interface IConversationStateStore
{
 Task<ConversationState> GetAsync(string userId, CancellationToken cancellationToken = default);
 Task SaveAsync(ConversationState state, CancellationToken cancellationToken = default);
 Task ClearAsync(string userId, CancellationToken cancellationToken = default);
}
```

## Implementación del Router

```
using Microsoft.Extensions.Logging;

public class ConversationRouter
{
 private readonly IConversationStateStore _stateStore;
 private readonly IIIntentClassificationService _intentClassifier;
 private readonly ILogger<ConversationRouter> _logger;
 private readonly Dictionary<string, Func<RouteContext, Task<string>>> _handlers;

 public ConversationRouter(
 IConversationStateStore stateStore,
 IIIntentClassificationService intentClassifier,
 ILogger<ConversationRouter> logger)
 {
 _stateStore = stateStore;
 _intentClassifier = intentClassifier;
 _logger = logger;

 // Registrar handlers
 _handlers = new()
 {
 ["menu"] = HandleMenuAsync,
 ["search"] = HandleSearchAsync,
 ["help"] = HandleHelpAsync,
 ["buy"] = HandleBuyAsync,
 ["status"] = HandleStatusAsync
 };
 }
}
```

```

public async Task<string> RouteAsync(
 string userId,
 string message,
 CancellationToken cancellationToken = default)
{
 _logger.LogInformation("Routing message for user {UserId}: {Message}", userId, message);

 // 1. Obtener estado
 var state = await _stateStore.GetAsync(userId, cancellationToken);

 // 2. Manejar comandos globales (prioridad máxima)
 var globalResponse = await TryHandleGlobalCommandAsync(message, state);
 if (globalResponse != null)
 {
 state.MessageCount++;
 await _stateStore.SaveAsync(state, cancellationToken);
 return globalResponse;
 }

 // 3. Manejar flujo activo
 if (!string.IsNullOrEmpty(state.CurrentFlow))
 {
 var flowResponse = await TryHandleActiveFlowAsync(message, state, cancellationToken);
 if (flowResponse != null)
 {
 state.MessageCount++;
 await _stateStore.SaveAsync(state, cancellationToken);
 return flowResponse;
 }
 }
}

// 4. Clasificar intención
var classification = await _intentClassifier.ClassifyAsync(message, cancellationToken);

_logger.LogInformation(
 "Intent classified: {Intent} (confidence: {Confidence})",
 classification.Intent,
 classification.Confidence);

// 5. Enrutar al handler apropiado
if (_handlers.TryGetValue(classification.Intent, out var handler))
{
 var context = new RouteContext
 {
 UserId = userId,
 Message = message,
 State = state,
 Classification = classification
 };

 var response = await handler(context);

 state.MessageCount++;
 await _stateStore.SaveAsync(state, cancellationToken);

 return response;
}

```

```

// 6. Fallback
_logger.LogWarning("No handler found for intent: {Intent}", classification.Intent);
return "No estoy seguro de entender. ¿Puedes reformular tu pregunta?";
}

private async Task<string?> TryHandleGlobalCommandAsync(
 string message,
 ConversationState state)
{
 var normalized = message.ToLowerInvariant().Trim();

 // Comando de menú
 if (normalized is "menu" or "menú" or "ayuda" or "help" or "inicio")
 {
 state.CurrentFlow = null;
 state.Context.Clear();
 return BuildMenuMessage();
 }

 // Comando de cancelar
 if (normalized is "cancelar" or "cancel" or "salir" or "exit")
 {
 state.CurrentFlow = null;
 state.Context.Clear();
 return "Operación cancelada. ¿En qué puedo ayudarte?";
 }

 return null;
}

private async Task<string?> TryHandleActiveFlowAsync(
 string message,
 ConversationState state,
 CancellationToken cancellationToken)
{
 // Manejar flujo activo basado en state.CurrentFlow
 return state.CurrentFlow switch
 {
 "purchase" => await HandlePurchaseFlowAsync(message, state, cancellationToken),
 "registration" => await HandleRegistrationFlowAsync(message, state, cancellationToken),
 "feedback" => await HandleFeedbackFlowAsync(message, state, cancellationToken),
 _ => null
 };
}

private async Task<string> HandleMenuAsync(RouteContext context)
{
 context.State.CurrentFlow = null;
 context.State.Context.Clear();
 return BuildMenuMessage();
}

private async Task<string> HandleSearchAsync(RouteContext context)
{
 var query = context.Classification.Entities?.ContainsKey("query") == true
 ? context.Classification.Entities["query"].ToString()
 : context.Message;
}

```

```

_logger.LogInformation("Handling search for: {Query}", query);

// Implementar búsqueda
return $"Buscando: {query}...";
}

private async Task<string> HandleHelpAsync(RouteContext context)
{
 return """";
 Puedo ayudarte con:

 Buscar productos o servicios
 Realizar compras
 Consultar el estado de pedidos
 Ver el menú de opciones

 ¿Qué necesitas?
 """";
}

private async Task<string> HandleBuyAsync(RouteContext context)
{
 // Iniciar flujo de compra
context.State.CurrentFlow = "purchase";
context.State.Context["step"] = "product_selection";

 return "¿Qué producto te gustaría comprar?";
}

private async Task<string> HandleStatusAsync(RouteContext context)
{
 var orderId = context.Classification.Entities?.ContainsKey("order_id") == true
 ? context.Classification.Entities["order_id"].ToString()
 : null;

 if (orderId == null)
 {
 return "Por favor, proporciona tu número de pedido.";
 }

 return $"Consultando estado del pedido {orderId}...";
}

private async Task<string> HandlePurchaseFlowAsync(
 string message,
 ConversationState state,
 CancellationToken cancellationToken)
{
 var step = state.Context.TryGetValue("step", out var stepObj)
 ? stepObj?.ToString()
 : "product_selection";

 return step switch
 {
 "product_selection" => await HandleProductSelectionAsync(message, state),
 "quantity_selection" => await HandleQuantitySelectionAsync(message, state),
 "confirmation" => await HandlePurchaseConfirmationAsync(message, state),
 };
}

```

```

 _ => "Flujo de compra no válido."
 };
}

private async Task<string> HandleProductSelectionAsync(
 string message,
 ConversationState state)
{
 state.Context["selected_product"] = message;
 state.Context["step"] = "quantity_selection";

 return $"Has seleccionado: {message}\n\n¿Cuántas unidades deseas?";
}

private async Task<string> HandleQuantitySelectionAsync(
 string message,
 ConversationState state)
{
 if (!int.TryParse(message, out var quantity) || quantity <= 0)
 {
 return "Por favor, indica una cantidad válida.";
 }

 state.Context["quantity"] = quantity;
 state.Context["step"] = "confirmation";

 var product = state.Context["selected_product"];

 return $"""
 Resumen de tu pedido:
 • Producto: {product}
 • Cantidad: {quantity}

 ¿Confirmas la compra? (sí/no)
 """;
}

private async Task<string> HandlePurchaseConfirmationAsync(
 string message,
 ConversationState state)
{
 var normalized = message.ToLowerInvariant().Trim();

 if (normalized is "sí" or "si" or "s" or "yes" or "y")
 {
 var product = state.Context["selected_product"];
 var quantity = state.Context["quantity"];

 // Procesar pedido
 _logger.LogInformation(
 "Processing purchase: Product={Product}, Quantity={Quantity}",
 product,
 quantity);

 // Lestionar flujo
 state.CurrentFlow = null;
 state.Context.Clear();
 }
}

```

```

 return $" Pedido confirmado: {quantity}x {product}\n\nRecibirás un correo con los detalles";
 }
 else
 {
 // Cancelar
 state.CurrentFlow = null;
 state.Context.Clear();

 return "Pedido cancelado. ¿Puedo ayudarte con algo más?";
 }
}

private async Task<string> HandleRegistrationFlowAsync(
 string message,
 ConversationState state,
 CancellationToken cancellationToken)
{
 // Implementar flujo de registro
 return "Flujo de registro...";
}

private async Task<string> HandleFeedbackFlowAsync(
 string message,
 ConversationState state,
 CancellationToken cancellationToken)
{
 // Implementar flujo de feedback
 return "Flujo de feedback...";
}

private string BuildMenuMessage()
{
 return """
 Menú Principal

 Opciones disponibles:
 1 Buscar productos
 2 Ver mis pedidos
 3 Hacer una compra
 4 Ayuda

 Escribe el número o describe lo que necesitas.
 """;
}
}

public class RouteContext
{
 public required string UserId { get; init; }
 public required string Message { get; init; }
 public required ConversationState State { get; init; }
 public required IntentClassificationResult Classification { get; init; }
}

```

## Gestión de Estado con Redis

```

using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;

```

```

public class RedisConversationStateStore : IConversationStateStore
{
 private readonly IDistributedCache _cache;
 private readonly TimeSpan _stateExpiration = TimeSpan.FromHours(24);

 public RedisConversationStateStore(IDistributedCache cache)
 {
 _cache = cache;
 }

 public async Task<ConversationState> GetAsync(
 string userId,
 CancellationToken cancellationToken = default)
 {
 var key = GetCacheKey(userId);
 var json = await _cache.GetStringAsync(key, cancellationToken);

 if (json == null)
 {
 return new ConversationState { UserId = userId };
 }

 return JsonSerializer.Deserialize<ConversationState>(json)
 ?? new ConversationState { UserId = userId };
 }

 public async Task SaveAsync(
 ConversationState state,
 CancellationToken cancellationToken = default)
 {
 var key = GetCacheKey(state.UserId);
 var json = JsonSerializer.Serialize(state);

 await _cache.SetStringAsync(
 key,
 json,
 new DistributedCacheEntryOptions
 {
 AbsoluteExpirationRelativeToNow = _stateExpiration
 },
 cancellationToken);
 }

 public async Task ClearAsync(
 string userId,
 CancellationToken cancellationToken = default)
 {
 var key = GetCacheKey(userId);
 await _cache.RemoveAsync(key, cancellationToken);
 }

 private string GetCacheKey(string userId)
 {
 return $"conversation:state:{userId}";
 }
}

```

## Router con Middleware Pattern

```
public interface IRouterMiddleware
{
 Task<string?> ProcessAsync(
 RouteContext context,
 Func<RouteContext, Task<string>> next);
}

public class LoggingMiddleware : IRouterMiddleware
{
 private readonly ILogger<LoggingMiddleware> _logger;

 public LoggingMiddleware(ILogger<LoggingMiddleware> logger)
 {
 _logger = logger;
 }

 public async Task<string?> ProcessAsync(
 RouteContext context,
 Func<RouteContext, Task<string>> next)
 {
 _logger.LogInformation(
 "Processing message from {UserId}: {Message}",
 context.UserId,
 context.Message);

 var stopwatch = Stopwatch.StartNew();
 var response = await next(context);
 stopwatch.Stop();

 _logger.LogInformation(
 "Response generated in {Elapsed}ms",
 stopwatch.ElapsedMilliseconds);

 return response;
 }
}

public class RateLimitMiddleware : IRouterMiddleware
{
 private readonly Dictionary<string, Queue<DateTime>> _userRequests = new();
 private readonly int _maxRequestsPerMinute = 10;

 public async Task<string?> ProcessAsync(
 RouteContext context,
 Func<RouteContext, Task<string>> next)
 {
 if (!_userRequests.ContainsKey(context.UserId))
 {
 _userRequests[context.UserId] = new Queue<DateTime>();
 }

 var userQueue = _userRequests[context.UserId];
 var now = DateTime.UtcNow;
 var oneMinuteAgo = now.AddMinutes(-1);

 // Limpia requests antiguos
 while (userQueue.Count > 0 && userQueue.Peek() < oneMinuteAgo)
```

```

 {
 userQueue.Dequeue();
 }

 if (userQueue.Count >= _maxRequestsPerMinute)
 {
 return "Has enviado demasiados mensajes. Por favor, espera un momento.";
 }

 userQueue.Enqueue(now);

 return await next(context);
}
}

```

## Testing del Router

```

using Xunit;
using Moq;

public class ConversationRouterTests
{
 [Fact]
 public async Task RouteAsync_MenuCommand_ReturnsMenuMessage()
 {
 // Arrange
 var mockStateStore = new Mock<IConversationStateStore>();
 mockStateStore
 .Setup(x => x.GetAsync(It.IsAny<string>(), It.IsAny<CancellationToken>()))
 .ReturnsAsync(new ConversationState { UserId = "test-user" });

 var mockClassifier = new Mock<IIIntentClassificationService>();
 var logger = Mock.Of<	ILogger<ConversationRouter>>();

 var router = new ConversationRouter(
 mockStateStore.Object,
 mockClassifier.Object,
 logger);

 // Act
 var response = await router.RouteAsync("test-user", "menu");

 // Assert
 Assert.Contains("Menú Principal", response);
 }

 [Fact]
 public async Task RouteAsync_PurchaseFlow_CompletesSuccessfully()
 {
 // Arrange
 var state = new ConversationState { UserId = "test-user" };
 var mockStateStore = new Mock<IConversationStateStore>();
 mockStateStore
 .Setup(x => x.GetAsync(It.IsAny<string>(), It.IsAny<CancellationToken>()))
 .ReturnsAsync(state);

 var mockClassifier = new Mock<IIIntentClassificationService>();
 mockClassifier

```

```

 .Setup(x => x.ClassifyAsync(It.IsAny<string>(), It.IsAny<CancellationToken>()))
 .ReturnsAsync(new IntentClassificationResult
 {
 Intent = "buy",
 Confidence = 0.95
 });
}

var logger = Mock.Of<ILogger<ConversationRouter>>();
var router = new ConversationRouter(
 mockStateStore.Object,
 mockClassifier.Object,
 logger);

// Act - Start purchase
var response1 = await router.RouteAsync("test-user", "quiero comprar");

// Act - Select product
var response2 = await router.RouteAsync("test-user", "libro");

// Act - Select quantity
var response3 = await router.RouteAsync("test-user", "2");

// Act - Confirm
var response4 = await router.RouteAsync("test-user", "sí");

// Assert
Assert.Contains("producto", response1, StringComparison.OrdinalIgnoreCase);
Assert.Contains("unidades", response2, StringComparison.OrdinalIgnoreCase);
Assert.Contains("Resumen", response3, StringComparison.OrdinalIgnoreCase);
Assert.Contains("confirmado", response4, StringComparison.OrdinalIgnoreCase);
}
}

```

## Mejores Prácticas

## 1. Prioridad de Comandos

```
// Global > Flow > Intent > Fallback
private async Task<string> RouteWithPriorityAsync(string message, ConversationState state)
{
 return await TryHandleGlobalCommandAsync(message, state)
 ?? await TryHandleActiveFlowAsync(message, state)
 ?? await TryHandleIntentAsync(message, state)
 ?? GetFallbackMessage();
}
```

## 2. Timeouts para Flujos

```
public bool IsFlowExpired(ConversationState state)
{
 const int FlowTimeoutMinutes = 15;
 return state.LastInteraction.AddMinutes(FlowTimeoutMinutes) < DateTime.UtcNow;
}
```

### 3. Context Cleanup

```
public void CleanupExpiredState(ConversationState state)
{
 if (IsFlowExpired(state))
```

```

 {
 state.CurrentFlow = null;
 state.Context.Clear();
 }
}

```

## Conclusión

Un router conversacional bien diseñado es esencial para chatbots robustos. Gestiona estados, enruta correctamente, maneja flujos complejos, y proporciona fallbacks apropiados. El uso de middleware permite agregar funcionalidades como logging y rate limiting de forma modular.

---

**Palabras clave:** conversational router, chatbot architecture, state management, intent routing, flow handling, middleware pattern

---

## Cierre del paso

### Verificación rápida

- Casos de prueba se enrutan al handler correcto y los ambiguos piden aclaración.

### Errores frecuentes y cómo desbloquearte

- Router opaco: sin trazabilidad no podrás mejorarlo.
- Routing directo a herramientas peligrosas: usa allow-list y confirmación.

### Ejercicios (para afianzar)

- Añade un modo ‘simulación’ que devuelva la decisión sin ejecutar.
- Introduce evaluación offline: dataset de mensajes + intent esperada.

### Siguiente paso

Cuando estés listo, continúa con el **Paso 24 — Filtrado por Relevancia Semántica en Búsquedas**.

## Paso 24 — Filtrado por Relevancia Semántica en Búsquedas

### Mapa del paso

**Tiempo estimado:** 60–90 min

**Resultado que vas a conseguir:**

Filtrado por relevancia: umbrales, re-ranking y control de ruido en resultados.

**Objetivos de aprendizaje:**

- Reducir falsos positivos en búsqueda semántica.
- Aplicar umbrales, top-k dinámico y heurísticas.
- Mejorar UX: explicar por qué algo no aparece.

**Prerrequisitos:**

- Paso 05 y/o 15 recomendados.

### Ruta guiada (paso a paso)

1. Define un score mínimo y top-k inicial.
2. Aplica filtros por metadata antes del ranking semántico.
3. Añade re-ranking o segunda pasada si es necesario.

4. Mide impacto con un set de queries.
- 

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

El filtrado por relevancia semántica mejora la calidad de resultados de búsqueda eliminando matches irrelevantes. En este tutorial aprenderás a implementar filtros inteligentes que combinan similitud vectorial con lógica de negocio.

## Conceptos de Relevancia

### Tipos de Relevancia

1. **Relevancia Vectorial:** Similitud coseno entre embeddings
2. **Relevancia Léxica:** Coincidencia de palabras clave
3. **Relevancia de Dominio:** Match entre categoría de búsqueda y negocio
4. **Relevancia Contextual:** Considerando ubicación, horarios, etc.

## Implementación de Filtros

```
public class SemanticRelevanceFilter
{
 private const double MinVectorScore = 0.35;
 private const double MinLexicalOverlap = 0.2;

 public async Task<List<SearchResult>> FilterRelevantResultsAsync(
 string query,
 List<SearchResult> candidates,
 SearchContext context)
 {
 var queryTokens = TokenizeQuery(query);
 var filtered = new List<SearchResult>();

 foreach (var candidate in candidates)
 {
 var relevanceScore = CalculateRelevance(
 query,
 queryTokens,
 candidate,
 context);

 if (relevanceScore.IsRelevant)
 {
 candidate.RelevanceScore = relevanceScore.Score;
 candidate.RelevanceReason = relevanceScore.Reason;
 filtered.Add(candidate);
 }
 }

 return filtered.OrderByDescending(r => r.RelevanceScore).ToList();
 }

 private RelevanceScore CalculateRelevance(
 string query,
 HashSet<string> queryTokens,
 SearchResult candidate,
```

```

 SearchContext context)
{
 // 1. Vector score check
 if (candidate.VectorScore < MinVectorScore)
 {
 return RelevanceScore.NotRelevant("Vector score too low");
 }

 // 2. Lexical overlap check
 var candidateTokens = TokenizeText(candidate.Description);
 var overlap = CalculateLexicalOverlap(queryTokens, candidateTokens);

 if (overlap < MinLexicalOverlap)
 {
 return RelevanceScore.NotRelevant("Insufficient lexical overlap");
 }

 // 3. Domain match check
 if (context.Domain != null)
 {
 if (!MatchesDomain(candidate, context.Domain))
 {
 return RelevanceScore.NotRelevant("Domain mismatch");
 }
 }

 // Calculate final score
 var finalScore = (candidate.VectorScore * 0.6) +
 (overlap * 0.3) +
 (DomainBonus(candidate, context) * 0.1);

 return RelevanceScore.Relevant(finalScore, "Passed all checks");
}
}

```

## Mejores Prácticas

1. Combinar múltiples señales de relevancia
2. Ajustar thresholds basándose en métricas de calidad
3. Logging detallado para debugging
4. A/B testing de configuraciones

---

**Palabras clave:** semantic relevance, filtering, search quality, vector similarity, lexical overlap

---

## Cierre del paso

### Verificación rápida

- Resultados irrelevantes disminuyen sin perder demasiado recall.

### Errores frecuentes y cómo desbloquearte

- Umbral demasiado alto: ‘no hay resultados’ frecuente.
- Sin evaluación: mejoras subjetivas que empeoran métricas.

## Ejercicios (para afianzar)

- Implementa un modo ‘strict’ y ‘relaxed’ para comparar.
- Añade explicación: score + criterios de filtrado aplicados.

## Siguiente paso

Cuando estés listo, continúa con el **Paso 25** — Normalización y Preprocesamiento de Datos para IA.

# Paso 25 — Normalización y Preprocesamiento de Datos para IA

## Mapa del paso

**Tiempo estimado:** 60–90 min

**Resultado que vas a conseguir:**

Pipeline de normalización de datos para IA: limpieza, tokenización básica y consistencia.

**Objetivos de aprendizaje:**

- Normalizar texto antes de embeddings y prompts.
- Reducir ruido: duplicados, caracteres raros, espacios, casing.
- Preparar datasets para evaluación y producción.

**Prerrequisitos:**

- Paso 05 recomendado si trabajas con embeddings.

## Ruta guiada (paso a paso)

1. Define reglas de normalización (lowercase, trim, unicode, stopwords si aplica).
2. Crea funciones reutilizables y tests con casos edge.
3. Aplica normalización antes de indexar y antes de consultar.
4. Mide impacto en calidad de búsqueda (precision/recall).

---

A continuación tienes el contenido principal del paso. Léelo con calma, ejecuta los bloques de código en orden y vuelve a la sección **Cierre del paso** para validar y practicar.

## Introducción

La calidad de los datos es crucial para sistemas de IA. En este tutorial aprenderás técnicas de normalización y preprocesamiento que mejoran significativamente los resultados de tus modelos.

## Normalización de Texto

```
public class TextNormalizer
{
 public string Normalize(string text)
 {
 if (string.IsNullOrWhiteSpace(text))
 return string.Empty;

 // 1. Lowercase
 text = text.ToLowerInvariant();

 // 2. Remover acentos
 text = RemoveAccents(text);

 // 3. Remover caracteres especiales
 text = RemoveSpecialCharacters(text);
 }
}
```

```

// 4. Normalizar espacios
text = NormalizeWhitespace(text);

 return text.Trim();
}

private string RemoveAccents(string text)
{
 var normalizedString = text.Normalize(NormalizationForm.FormD);
 var stringBuilder = new StringBuilder();

 foreach (var c in normalizedString)
 {
 var unicodeCategory = CharUnicodeInfo.GetUnicodeCategory(c);
 if (unicodeCategory != UnicodeCategory.NonSpacingMark)
 {
 stringBuilder.Append(c);
 }
 }

 return stringBuilder.ToString().Normalize(NormalizationForm.FormC);
}
}

```

## Tokenización

```

public class Tokenizer
{
 private readonly HashSet<string> _stopWords = new()
 {
 "el", "la", "de", "que", "y", "a", "en", "un", "ser", "se", "no"
 };

 public List<string> Tokenize(string text, bool removeStopWords = true)
 {
 var tokens = text
 .Split(new[] { ' ', '\t', '\n', '\r', ',', '.', ';', ':', '!', '?' }, StringSplitOptions.RemoveEmptyEntries)
 .Select(t => t.ToLowerInvariant())
 .Where(t => t.Length > 2);

 if (removeStopWords)
 {
 tokens = tokens.Where(t => !_stopWords.Contains(t));
 }

 return tokens.ToList();
 }
}

```

## Limpieza de Datos

```

public class DataCleaner
{
 public string CleanBusinessHours(string hours)
 {
 // Normalizar horarios de negocio

```

```

 // "Lunes a Viernes: 9-14h" → "L-V: 09:00-14:00"
 return hours;
}

public string CleanPhoneNumber(string phone)
{
 // Normalizar números de teléfono
 // "+34 666 123 456" → "666123456"
 return Regex.Replace(phone, @"\D", "");
}

```

## Mejores Prácticas

1. Normalizar antes de generar embeddings
2. Mantener consistencia en todo el pipeline
3. Documentar transformaciones aplicadas
4. Validar datos antes y después de normalizar

---

**Palabras clave:** data normalization, text preprocessing, tokenization, data cleaning, NLP

---

## Cierre del paso

### Verificación rápida

- La misma frase con variaciones ('tildes', espacios) produce resultados estables.

### Errores frecuentes y cómo desbloquearte

- Normalizar en exceso: perder información útil (ej.: IDs, mayúsculas relevantes).
- No testear edge cases: emojis, unicode, HTML, URLs.

### Ejercicios (para afianzar)

- Añade soporte para limpieza de HTML/Markdown.
- Introduce un reporte de calidad del dataset (duplicados, longitudes, idiomas).

### Fin del recorrido

Has completado los 25 pasos. A continuación revisa el material final del libro (contacto del autor, recursos y siguientes pasos).

## Contacto del autor

Puedes contactarme si tienes alguna duda o pregunta

- Nombre: David Cantón Nadales
- Web: <https://www.davidcanton.net>
- Email: [davidcantonnadales@gmail.com](mailto:davidcantonnadales@gmail.com)
- LinkedIn: <https://www.linkedin.com/in/davidcantonnadales/>
- GitHub: <https://github.com/davidcantonnadales>
- Telegram: @davidcanton

## Agradecimientos

Gracias a la comunidad por las preguntas y el feedback, al equipo de Semantic Kernel y a todos los proyectos open-source que sostienen este ecosistema. Si este libro te resulta útil, compártelo para que llegue a más personas.

## Sobre este libro

Aprende Semantic Kernel en 25 pasos es una guía práctica para construir aplicaciones con LLMs en .NET sin perderte en la teoría.

Comenzamos desde los fundamentos —kernel, plugins y chat— y avanzamos hasta escenarios reales de producción: JSON estructurado, guardrails, cache, testing, observabilidad, métricas y seguridad.

Este libro está diseñado para desarrolladores que quieren resultados reales, no solo conceptos.

---

### ☑ Qué aprenderás

- ✓ Configurar Azure OpenAI / OpenAI con Semantic Kernel
- ✓ Diseñar prompts robustos y forzar salida JSON validable
- ✓ Construir routers conversacionales y búsqueda semántica
- ✓ Aplicar prácticas de producción: coste, errores, caché y seguridad

---

De cero a producción real con LLMs en .NET.

---

### Autor

**David Cantón Nadales**

Microsoft MVP

Web: [www.davidcanton.net](http://www.davidcanton.net)

GitHub: [github.com/davidcantonnadales](https://github.com/davidcantonnadales)