# Planificador Agent

**PlanificadorAgent**

El PlanificadorAgent es un componente **fundamental y actualmente funcional** dentro del ecosistema del MCP (Modelo de Ciclo de Prompts). Su función central es tomar un input en forma de Shared.Prompt (una solicitud textual en lenguaje natural) y convertirlo en un **backlog validado y ordenado de tareas técnicas de alto nivel**, listas para ser procesadas por el DesarrolladorAgent.

**6.1.1 Propósito y Misión**

La misión actual del PlanificadorAgent es:

1. **Interpretar Requerimientos:** Analizar la Descripcion del Shared.Prompt para entender el objetivo general del usuario.
2. **Generar Backlog Técnico:** Utilizar GeminiClient con un **prompt cuidadosamente diseñado** para generar una lista de tareas técnicas específicas. **Crucialmente, cada tarea debe representar la creación o modificación de un único artefacto** (archivo .cs, .razor, .csproj o una configuración específica como el registro en Program.cs). **Evita explícitamente generar líneas de código como tareas.**
3. **Validar Tareas:** Filtrar la respuesta de Gemini para descartar líneas que no parezcan tareas técnicas válidas (ej. código suelto, comentarios, líneas muy cortas).
4. **Ordenar Tareas:** Aplicar un **ordenamiento básico por categorías de dependencia** (ej: Modelos -> Datos -> Configuración -> Servicios -> Componentes/Páginas -> Layouts) para mejorar la probabilidad de éxito en la fase de desarrollo.
5. **Manejar Fallos:** Detectar si el proceso de planificación falla (respuesta vacía de Gemini, ninguna tarea válida generada) y devolver un resultado vacío para detener el flujo del Worker.

Su objetivo principal es producir un **plan de trabajo accionable y lógicamente ordenado** para el DesarrolladorAgent, basado en el requerimiento original.

**6.1.2 Interfaces Implementadas**

* Shared.IPlanificadorAgent
  + Task<string[]> ConvertirPromptABacklog(Shared.Prompt prompt): Implementación principal que realiza todo el proceso de planificación, validación y ordenamiento.
  + *Nota:* El método OrganizarTareasAsync mencionado en documentación anterior **no forma parte de la interfaz ni implementación actual.** El ordenamiento se realiza dentro de ConvertirPromptABacklog.

Estas interfaces aseguran una estandarización de sus métodos y promueven la inyección de dependencias limpia.

**6.1.3 Flujo de actividad interno**

graph TD

A[Inicio: Recibe Shared.Prompt] --> B{Construir Prompt Detallado para Gemini};

B --> C{Llamar a GeminiClient.GenerarAsync};

C --> D{Recibir Respuesta (string)};

D --> E{Manejar Errores API / Respuesta Vacía?};

E -- Sí --> Z[Log Error/Warning y Retornar Vacío];

E -- No --> F{Dividir Respuesta en Líneas};

F --> G{Limpiar/Trimmear cada Línea};

G --> H{Validar cada Línea (IsValidTask)};

H --> I{Filtrar Tareas Válidas};

I --> J{Categorizar Tareas Válidas (GetTaskCategory)};

J --> K{Ordenar Tareas por Categoría};

K --> L{Verificar si Backlog Final es Suficiente?};

L -- No --> Z;

L -- Sí --> M{Log Éxito y Devolver Backlog Ordenado (string[])};

M --> X[Fin del Ciclo del Planificador];

Z --> X;

**6.1. 6.1.4 Entradas / Salidas / Errores (Actualizado)**

* **Entradas:**
  + Shared.Prompt prompt: Objeto que contiene Titulo y Descripcion del requerimiento.
* **Salidas:**
  + Task<string[]>: Un array de strings representando el **backlog de tareas técnicas validadas y ordenadas**. Cada string es una tarea de alto nivel (ej: "Crear modelo Models/Producto.cs...", "Añadir enlace Productos en Shared/NavMenu.razor"). Devuelve un array vacío (Array.Empty<string>()) en caso de fallo irrecuperable en la planificación.
* **Errores Comunes Manejados:**
  + Fallo en GeminiClient.GenerarAsync (Errores de API, timeout, cuota excedida - Error 429/503).
  + Respuesta vacía o inválida de Gemini.
  + Fallo de planificación: No se generan tareas válidas después del filtrado.
  + Excepciones durante el procesamiento de la respuesta.
* **Errores Potenciales No Manejados Explícitamente (Dependen de Calidad de Gemini):**
  + Tareas generadas que *pasan* la validación pero son lógicamente incorrectas o incompletas para el requerimiento.
  + Orden de tareas generado por Gemini que, aunque reordenado por categoría, aún podría tener dependencias finas incorrectas.

**6.1.5 Algoritmos Clave y Pseudocódigo (Actualizado)**

public async Task<string[]> ConvertirPromptABacklog(Shared.Prompt prompt)

{

// 1. Construir Prompt Específico para Gemini (Ver código real para detalles)

// - Incluye rol, restricciones, ejemplos buenos/malos, descripción del prompt.

string promptParaGemini = ConstruirPromptPlanificador(prompt);

string respuestaGemini;

try

{

// 2. Llamar a Gemini

respuestaGemini = await \_gemini.GenerarAsync(promptParaGemini);

if (string.IsNullOrWhiteSpace(respuestaGemini))

{

\_logger.LogWarning("Respuesta vacía de Gemini.");

return Array.Empty<string>();

}

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error llamando a Gemini para planificar.");

return Array.Empty<string>();

}

// 3. Procesar Respuesta

var lineasIniciales = respuestaGemini

.Split(new[] { '\r', '\n' }, StringSplitOptions.RemoveEmptyEntries)

.Select(l => l.Trim(' ', '-', '\*', '.'))

.Select(l => l.Trim())

.Where(l => !string.IsNullOrWhiteSpace(l))

.ToList();

// 4. Validar Tareas (Mejora #1)

var tareasValidas = new List<string>();

foreach (var linea in lineasIniciales)

{

if (IsValidTask(linea)) // Usa lógica de validación (ver código)

{

tareasValidas.Add(linea);

}

else

{

\_logger.LogWarning("Línea descartada por inválida: {Linea}", linea);

}

}

// 5. Ordenar Tareas (Mejora #2)

var tareasOrdenadas = tareasValidas

.Select(t => new { Task = t, Category = GetTaskCategory(t) }) // Categoriza (ver código)

.OrderBy(item => item.Category) // Ordena por Enum de categoría

.ThenBy(item => item.Task) // Orden secundario alfabético

.Select(item => item.Task)

.ToList();

// 6. Verificar Resultado Final (Mejora #3)

const int minRequiredTasks = 1;

if (tareasOrdenadas.Count < minRequiredTasks)

{

\_logger.LogError("FALLO PLANIFICACIÓN: No hay tareas válidas suficientes después de filtrar/ordenar.");

return Array.Empty<string>();

}

\_logger.LogInformation("Backlog generado y ordenado con {Count} tareas.", tareasOrdenadas.Count);

return tareasOrdenadas.ToArray();

}

private bool IsValidTask(string line)

{

// Implementación: Chequea longitud mínima, ausencia de prefijos de código,

// presencia de verbos de acción y artefactos conocidos. (Ver código real)

// ...

return true; // Placeholder

}

private TaskCategory GetTaskCategory(string task)

{

// Implementación: Asigna una categoría (enum) basada en palabras clave

// y rutas inferidas (Models, Data, Services, Pages, etc.). (Ver código real)

// ...

return TaskCategory.Other; // Placeholder

}

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).C#

**6.1.6 Métricas Reportadas**

(Sin cambios respecto a la documentación original - son métricas *potenciales* a implementar)

* Prompts procesados por hora
* Tiempo promedio de análisis por prompt
* % de prompts que requieren intervención manual (o resultan en backlog vacío)
* Tareas promedio por prompt (después de validación)
* % de tareas aceptadas directamente por DesarrolladorAgent sin modificación (métrica ideal a futuro)

**6.1.7 Dependencias Externas (Actualizado)**

* Infraestructura.GeminiClient: Para interactuar con la API de Gemini.
* Microsoft.Extensions.Logging.ILogger<PlanificadorAgent>: Para logging.
* Shared.Prompt: Tipo de dato de entrada.
* Shared.IPlanificadorAgent: Interfaz que implementa.

**6.1.8 Mejoras Futuras**

(La lista original sigue siendo relevante, pero algunas prioridades cambian con el estado actual):

* **Prioridad Alta:**
  + Inferencia de dependencias más complejas (#1).
  + Soporte para modificar proyectos existentes (contextualización) (#3, #11).
  + Mejor manejo de errores/reintentos de API Gemini (#17, #18).
  + Feedback de ejecución real (#16) para refinar prompts/lógica.
* **Prioridad Media:**
  + Clasificación más granular (#2).
  + Estimación de dificultad (#4).
  + Priorización (#5).
  + Integración con herramientas externas (Jira/Trello) (#14).
  + Validación cruzada (#19).
* **Prioridad Baja/Opcional:**
  + Gestión de versiones (#6, #7, #15).
  + Curación humana (#8).
  + Análisis histórico (#9).
  + Soporte multilingüe (#10).
  + Predicción de errores (#12).
  + Integración con docs (#13).
  + Notificaciones (#17).
  + Modo educativo (#20).

**6.1.9 Objetivo estratégico**

A futuro, se espera que el PlanificadorAgent evolucione hacia una entidad completamente autónoma, capaz de:

* Comprender contexto histórico y técnico del usuario.
* Priorizar según impacto.
* Asegurar la coherencia técnica de los planes propuestos.
* Adaptarse a distintos dominios (frontend, backend, mobile, IA, IoT...)

Su objetivo no es sólo planificar, sino **anticiparse a los obstáculos, optimizar el flujo de valor y garantizar el éxito del ciclo de desarrollo de principio a fin**.

**IngenieroPromptAgent**

El **IngenieroPromptAgent** es una incorporación crítica en la arquitectura MCP, cuyo propósito es refinar, optimizar y reescribir iterativamente los prompts recibidos por parte del usuario o generados automáticamente por el sistema. Su función principal es maximizar la calidad semántica y contextual de las instrucciones antes de que lleguen al PlanificadorAgent, asegurando que cada prompt sea claro, específico, libre de ambigüedades y alineado con la intención deseada.

**6.2.1 Propósito y misión**

Este agente tiene la misión de:

1. **Iterar sobre un prompt original para transformarlo en una versión mejorada**.
2. **Aplicar distintas estrategias de reformulación usando LLMs para que las instrucciones sean más efectivas.**
3. **Hacerlo configurable: permitir definir cuántas iteraciones se aplican para refinar el prompt.**
4. **Evaluar automáticamente la calidad del prompt mejorado con base en heurísticas de claridad, cobertura y completitud.**

**6.2.2 Interfaces implementadas**

* IIngenieroPromptAgent
  + Task<string> IterarYMejorarPromptAsync(string promptOriginal, int iteraciones = 3)

**6.2.3 Flujo de actividad interno**

+---------------------------+

| Prompt Original |

+------------+-------------+

↓

Iteración 1 → LLM mejora claridad y enfoque

↓

Iteración 2 → LLM agrega detalles técnicos faltantes

↓

Iteración 3 → LLM evalúa redundancias y optimiza estilo

↓

Prompt Final Mejorado

↓

→ Se envía al PlanificadorAgent

Cada iteración utiliza un modelo LLM como Gemini para reescribir el prompt anterior, evaluando progresivamente su efectividad.

**6.2.4 Entradas / salidas / errores**

* **Entradas**:
  + Prompt original en texto plano.
  + Número de iteraciones a aplicar (int).
* **Salidas**:
  + Prompt final mejorado.
  + Historial de iteraciones (opcional).
* **Errores comunes**:
  + Tiempo de espera excedido con LLMs.
  + Reescrituras que introducen errores semánticos (a mitigar).

**6.2.5 Algoritmos clave**

public async Task<string> IterarYMejorarPromptAsync(string promptOriginal, int iteraciones)

{

string actual = promptOriginal;

for (int i = 0; i < iteraciones; i++)

{

actual = await \_geminiClient.MejorarPromptAsync(actual);

\_logger.LogInformation($"Iteración {i + 1} aplicada al prompt.");

}

return actual;

}

**6.2.6 Métricas reportadas**

* Tiempo medio de optimización
* Longitud del prompt final vs original
* Número de iteraciones configuradas vs efectivas
* % de prompts aceptados sin edición adicional por el PlanificadorAgent

**6.2.7 Dependencias externas**

* GeminiClient
* ILogger<IngenieroPromptAgent>

**6.2.8 Mejoras futuras (20)**

1. **Comparación entre versiones del prompt para resaltar cambios**.
2. **Evaluador automático de calidad semántica mediante embeddings**.
3. **Uso de ejemplos previos del usuario como base estilística**.
4. **Detección de prompts redundantes o ya solucionados anteriormente**.
5. **Exportación a formatos de dataset (CSV, JSONL) para fine-tuning**.
6. **Vista previa visual con diferencias destacadas entre versiones**.
7. **Integración con motores sintácticos para asegurar corrección gramatical**.
8. **Soporte multilingüe bidireccional**.
9. **Asistente visual para iterar prompts paso a paso en GUI**.
10. **Estadísticas históricas por usuario sobre evolución de calidad**.
11. **Simulación de ejecución futura para validar viabilidad del prompt**.
12. **Modo adversarial para generar prompts edge-case**.
13. **Conversión de objetivos de negocio a prompts técnicos automáticamente**.
14. **Versionado de prompts con rollback histórico**.
15. **Explicaciones educativas sobre cómo mejorar cada prompt**.
16. **Entrenamiento incremental personalizado por usuario**.
17. **Uso de modelos internos si no hay conexión a Gemini**.
18. **Control granular de estilo (formal, directo, amigable, técnico)**.
19. **Restricción automática de longitud máxima compatible**.
20. **Prevalidación de la compatibilidad del prompt con cada agente**.

**6.2.9 Objetivo estratégico**

El IngenieroPromptAgent se posiciona como un filtro de calidad que garantiza que el sistema trabaje con instrucciones claras, efectivas y bien definidas. Su existencia aumenta la productividad de todos los agentes posteriores y mejora la satisfacción del usuario final al ver sus requerimientos interpretados correctamente desde el inicio.

A largo plazo, este agente podrá evolucionar hacia un **optimizador continuo de instrucciones**, ayudando a construir una memoria de buenas prácticas de prompt engineering y sirviendo como mentor para usuarios humanos que deseen aprender cómo redactar mejores instrucciones para IA.

# **DesarrolladorAgent**

El DesarrolladorAgent es uno de los componentes más cruciales en el ecosistema de MCP (Model Context Protocol), encargado específicamente de transformar los planes estratégicos generados por el PlanificadorAgent en código funcional y operativo. Este agente desempeña un papel fundamental en la automatización del desarrollo de software, asegurando que las aplicaciones cumplan rigurosamente con los requisitos técnicos y funcionales definidos previamente.

**Descripción Detallada del Comportamiento**

Este agente recibe del PlanificadorAgent un backlog estructurado en tareas específicas, cada una detallada con requisitos claros y parámetros técnicos específicos. El DesarrolladorAgent utiliza modelos LLM avanzados, como GPT-4o y Gemini-2.0-flash, para interpretar estos requisitos y traducirlos en código fuente claro, eficiente y alineado con las mejores prácticas de desarrollo.

Su ciclo operativo incluye los siguientes pasos:

1. **Recepción de Tareas:** Toma el backlog generado y priorizado por el PlanificadorAgent.
2. **Interpretación y Análisis:** Utiliza modelos LLM para interpretar detalladamente los requisitos técnicos y funcionales de cada tarea.
3. **Generación Inicial de Código:** Basado en la interpretación, genera fragmentos iniciales de código estructurados, claros y siguiendo estrictamente los estándares establecidos en la organización.
4. **Auto-revisión de Código:** Antes de desplegar cualquier código, realiza una auto-revisión exhaustiva mediante análisis estático y dinámico, identificando posibles errores de sintaxis, semántica o problemas lógicos.
5. **Iteración de Mejora:** Si detecta errores o mejoras potenciales, realiza iteraciones adicionales para perfeccionar el código generado.
6. **Pruebas Preliminares:** Ejecuta pruebas unitarias básicas y simulaciones de entornos limitados para asegurar que el código generado es operacional.
7. **Entrega del Código:** Finalmente, una vez validado, el código se entrega al siguiente agente o se despliega directamente en entornos controlados para pruebas más avanzadas.

**Objetivos Principales**

* Garantizar la generación de código fuente limpio, eficiente y de alta calidad.
* Automatizar el proceso completo de desarrollo, minimizando la intervención humana en tareas repetitivas o de bajo valor agregado.
* Incrementar considerablemente la velocidad de desarrollo, reduciendo así los ciclos de entrega.
* Mantener la consistencia técnica y arquitectónica a lo largo de los diferentes módulos y funcionalidades desarrolladas.

**Mejoras Futuras Propuestas**

1. **Optimización Automática:** Implementar técnicas avanzadas de inteligencia artificial para optimizar automáticamente el rendimiento del código.
2. **Integración con Herramientas CI/CD:** Profundizar la integración con plataformas de integración continua y entrega continua para automatizar completamente el pipeline.
3. **Soporte Multilenguaje:** Expandir capacidades para soportar generación y mantenimiento de código en múltiples lenguajes de programación simultáneamente.
4. **Auto-documentación:** Generar documentación técnica automáticamente y mantenerla actualizada con cada cambio en el código.
5. **Gestión Proactiva de Errores:** Desarrollar mecanismos inteligentes para predecir y gestionar errores potenciales antes de que ocurran.
6. **Retroalimentación Inteligente:** Integrar feedback continuo y dinámico para mejorar la calidad del código mediante aprendizaje automático.
7. **Estrategias Avanzadas de Debugging:** Incorporar capacidades avanzadas de depuración automatizada para identificar y corregir errores de manera más eficiente.
8. **Mejoras en Interpretación de Requisitos:** Aumentar la precisión en la interpretación de requisitos para reducir la necesidad de iteraciones posteriores.
9. **Seguridad Integrada:** Incorporar de forma predeterminada prácticas avanzadas de seguridad en la generación del código.
10. **Revisión Colaborativa de Código:** Facilitar revisiones automatizadas y colaborativas con agentes especializados en seguridad y optimización.
11. **Aprendizaje Continuo:** Permitir al agente aprender de las tareas previamente ejecutadas para mejorar continuamente la calidad y precisión del código.
12. **Soporte de Arquitecturas Distribuidas:** Optimizar el código generado específicamente para entornos distribuidos y de alta escalabilidad.
13. **Automatización de Migraciones:** Facilitar la generación automática de scripts para migraciones y adaptaciones a nuevos frameworks y tecnologías.
14. **Extensibilidad Modular:** Diseñar una arquitectura interna más modular que permita agregar nuevas funcionalidades y características de forma sencilla.
15. **Análisis Predictivo:** Usar análisis predictivos para anticipar futuras necesidades de mantenimiento o mejoras del software.
16. **Adaptación Contextual:** Desarrollar mecanismos para adaptar automáticamente el código a diferentes contextos operativos y entornos específicos.
17. **Incrementar Robustez en Pruebas:** Mejorar considerablemente la cobertura y profundidad de las pruebas preliminares automáticas.
18. **Autosupervisión en Tiempo Real:** Establecer monitoreo continuo en tiempo real del desempeño del código generado en producción.
19. **Generación de Interfaces de Usuario Automatizadas:** Avanzar en la capacidad del agente para crear interfaces de usuario intuitivas y altamente personalizables.
20. **Optimización en el Consumo de Recursos:** Implementar técnicas avanzadas para generar código altamente eficiente en términos de consumo de recursos computacionales.

**Objetivo a Largo Plazo**

El objetivo final del DesarrolladorAgent después de implementar las 20 mejoras mencionadas es alcanzar un nivel de autonomía y calidad tal que permita reducir al mínimo la intervención humana, asegurando entregas rápidas, consistentes y altamente eficientes, posicionando a MCP como líder en soluciones automatizadas de desarrollo de software.

# **ErrorFixerAgent**

El ErrorFixerAgent desempeña un papel crucial en el sistema MCP, enfocado específicamente en la identificación, diagnóstico y corrección automática de errores en el código generado por los otros agentes, especialmente por el DesarrolladorAgent. Este agente tiene como propósito principal asegurar la integridad, estabilidad y calidad del código generado, reduciendo considerablemente la necesidad de intervención manual y garantizando un proceso de entrega continua fluido y confiable.

**Propósito y charter**

El ErrorFixerAgent se encarga de revisar continuamente los logs de compilación y de pruebas, detectar patrones comunes de errores, aplicar correcciones automáticamente y verificar la eficacia de esas correcciones mediante nuevas compilaciones automáticas. Su misión es minimizar el impacto de errores técnicos en el ciclo de vida del desarrollo y facilitar un flujo continuo desde la generación inicial hasta la implementación final en producción.

**Interfaces implementadas**

* IErrorFixer
  + CorregirErroresAsync()
  + RecompilarProyectoAsync()
* ILogger
  + Uso extensivo para registrar acciones, resultados y diagnósticos en tiempo real.
* ICompilerService
  + Para realizar compilaciones automáticas post-corrección y asegurar la estabilidad.

**Flujo interno (diagrama de actividad)**

Inicio

↓

Lectura del log de errores (build\_errors.log)

↓

Identificación y clasificación de errores por archivo

↓

Generación de solicitud de corrección para Gemini API

↓

Recepción de correcciones sugeridas por Gemini API

↓

Aplicación automática de correcciones en archivos fuente

↓

Recompilación automática del proyecto

↓

Verificación del resultado de la recompilación

↓

Generación de nuevo log (build\_errors\_after\_fix.log)

↓

Evaluación de éxito o fracaso

↓

Informe final del resultado

Fin

**Entradas / salidas / errores**

* **Entradas:**
  + Archivo build\_errors.log
  + Archivos de código fuente identificados
* **Salidas:**
  + Archivos corregidos
  + Archivo build\_errors\_after\_fix.log
* **Errores posibles:**
  + Fallo en la API Gemini (quota, errores de conexión)
  + Imposibilidad de corregir un error específico
  + Errores de compilación recurrentes que necesitan intervención manual

**Algoritmos clave / pseudocódigo**

var errores = ParseBuildLog("build\_errors.log");

foreach (var archivoError in errores.GroupBy(e => e.Archivo))

{

var promptCorreccion = CrearPromptCorreccion(archivoError);

var correcciones = await GeminiClient.SolicitarCorreccionAsync(promptCorreccion);

AplicarCorrecciones(correcciones, archivoError.Key);

var resultadoCompilacion = await CompilarAsync();

if (!resultadoCompilacion.Exitoso)

{

RegistrarErrores(resultadoCompilacion.Errores);

ContinuarCorreccion();

}

}

**Métricas que reporta**

* Cantidad de errores detectados vs. corregidos
* Tiempo medio de corrección por error
* Porcentaje de éxito en correcciones automáticas
* Frecuencia de errores recurrentes (para identificar problemas estructurales)

**Dependencias externas**

* **Gemini API:** Para generar soluciones basadas en IA.
* **Compiler Service:** Para validar correcciones mediante recompilación.
* **Sistema de archivos:** Para manipular y almacenar logs de errores y archivos corregidos.

**Mejoras pendientes (20 ítems)**

1. **Integración con herramientas adicionales de análisis estático (SonarQube).**
2. **Capacidad para sugerir refactorizaciones más profundas del código.**
3. **Histórico de correcciones para aprendizaje y mejora continua.**
4. **Detección proactiva de patrones recurrentes de errores.**
5. **Alertas inteligentes ante errores críticos recurrentes.**
6. **Manejo automático y robusto de fallos de API Gemini (reintentos automáticos).**
7. **Corrección de errores en paralelo para múltiples proyectos simultáneos.**
8. **Integración de test de regresión automáticos post-corrección.**
9. **Panel de control para visualización de errores corregidos en tiempo real.**
10. **Capacidad de integración con pipelines externos de CI/CD (Jenkins, GitHub Actions).**
11. **Módulo de feedback hacia Gemini para optimizar prompts de corrección.**
12. **Implementación de auditoría automática sobre la eficacia de las correcciones realizadas.**
13. **Integración con Slack/Teams para notificaciones inmediatas ante fallos críticos.**
14. **Personalización dinámica de reglas de corrección según tipo de proyecto o tecnología usada.**
15. **Mejor manejo de errores específicos relacionados con dependencias y conflictos de paquetes.**
16. **Incorporación de recomendaciones preventivas basadas en errores históricos.**
17. **Generación automática de documentación técnica sobre correcciones aplicadas.**
18. **Sistema avanzado de caching para optimizar rendimiento en correcciones frecuentes similares.**
19. **Despliegue de ambiente aislado (sandbox) para verificar correcciones antes de aplicación definitiva.**
20. **Integración profunda con herramientas IDE como Visual Studio o JetBrains Rider para visualización y edición en tiempo real.**

# **PromptEngineerAgent**

**6.X.1 Propósito y Alcance**

**PromptEngineerAgent** es un agente especializado dentro del ecosistema MCP cuyo objetivo es **mejorar iterativamente los prompts** generados por el usuario o por otros agentes, aplicando estrategias de prompt engineering asistidas por IA. Su finalidad es optimizar el rendimiento de los modelos LLM en tareas complejas, maximizando la calidad de las respuestas generadas.

**6.X.2 Interfaces Implementadas**

* IPromptOptimizer → expone el método MejorarPromptAsync(string promptOriginal, int iteraciones)

**6.X.3 Flujo Interno (Diagrama de Actividad)**

1. Recibe un Prompt crudo.
2. Ejecuta una primera validación sintáctica y semántica.
3. Inicia un ciclo for parametrizable basado en la cantidad de iteraciones.
4. En cada ciclo:
   * Llama al modelo LLM con instrucciones especiales para refactorizar el prompt.
   * Analiza los cambios con heurísticas (claridad, completitud, enfoque).
   * Registra cada versión iterada en un historial.
5. Devuelve el mejor prompt final + historial.

**6.X.4 Entradas / Salidas / Errores**

**Entradas**

* string promptOriginal
* int iteraciones

**Salidas**

* PromptOptimizado: estructura con:
  + string MejorPrompt
  + List<string> Iteraciones

**Errores**

* Prompt vacío o sin sentido semántico.
* Límites de tokens superados en una iteración.

**6.X.5 Algoritmos Clave / Pseudocódigo**

for i in 1..n:

promptActual = LLM.Reescribir(promptAnterior, estrategia="mejorar claridad y enfoque")

if Evaluar(promptActual) > Evaluar(promptAnterior):

promptAnterior = promptActual

GuardarHistorial()

return promptAnterior

**6.X.6 Métricas Reportadas**

* Número de iteraciones realizadas
* Tiempo total de optimización
* Evaluación heurística inicial y final
* Nivel de cambio entre prompt inicial y final (Levenshtein distance)

**6.X.7 Dependencias Externas**

* GeminiClient o OpenAIClient (según LLM elegido para mejorar el prompt)
* PromptHeuristicsEvaluator (módulo auxiliar interno)
* Logger (ILogger)

**6.X.8 Mejoras Futuras (20 propuestas descriptivas)**

1. **Selector de modelo automático según tipo de prompt.** Elegir entre GPT-4, Gemini o Claude según si es técnico, narrativo o matemático.
2. **Heurística de legibilidad basada en Flesch-Kincaid.** Mejora prompts demasiado crípticos.
3. **Clasificación semántica del prompt.** Detectar si es instrucción, pregunta, resumen, etc.
4. **Estrategias de "Chain of Thought".** Insertar cadenas de razonamiento si se detecta ambigüedad.
5. **Uso de embeddings para comparar versiones.** Medir distancia semántica entre iteraciones.
6. **Parámetros adicionales: nivel de formalidad, longitud deseada.**
7. **Modo "escolar" para generar prompts adaptados a secundaria/universidad.**
8. **Integración con agente validador.** Confirmar que el prompt cumple con requerimientos antes de aprobarlo.
9. **Refuerzo positivo.** Aplicar técnicas de reinforcement learning para aprender de cada iteración.
10. **Historial visualizable en la interfaz.** Ver cada versión del prompt como "commits".
11. **Análisis de ambigüedad automático.**
12. **Comprobación de contexto perdido.** Detectar si una iteración omite información clave del original.
13. **Mejorar prompts con ejemplos (few-shot).** Agregar casos cuando se detecta que puede ayudar.
14. **Conversión a formato JSON estructurado para prompts complejos.**
15. **Explicación natural del prompt para usuarios no técnicos.**
16. **Exportación del historial en markdown.**
17. **Plugin de VSCode para visualizar y mejorar prompts.**
18. **Compatibilidad con prompts multimodales (texto + imagen).**
19. **Modo competitivo: varias versiones optimizadas y una votación.**
20. **Evaluación posterior al resultado: ¡el prompt realmente generó mejor output?**

✅ Este agente cierra el ciclo de calidad del sistema, asegurando que la entrada (el prompt) sea tan buena como la salida que se espera obtener.

# **BasicFunctionValidatorAgent**

**6.X.1 Propósito y Charter**

El **BasicFunctionValidatorAgent** tiene como propósito principal asegurar la completitud funcional de todos los módulos desarrollados dentro del ecosistema de SUPERDESARROLLADOR AUTÓNOMO, validando la presencia y correcta implementación de las funciones básicas propias de cualquier sistema CRUD: Create, Read, Update, Delete (CRUD). Su función es detectar omisiones, errores de implementación, falta de pruebas o incoherencias en estas operaciones esenciales, y garantizar que cada entidad desarrollada cumpla con los estándares funcionales mínimos.

**6.X.2 Interfaces implementadas**

* IValidadorAgent
* ILoggerAware
* ICicloOrquestable

**6.X.3 Flujo interno (diagrama de actividad)**

1. Recibe entidad objetivo y contexto de validación.
2. Localiza carpeta del proyecto y analiza clases, controladores y servicios.
3. Verifica existencia de:
   * Método de creación (Create/POST)
   * Lectura general (Get/List/GET)
   * Lectura individual (GetById/GET?id=)
   * Actualización (Update/PUT/PATCH)
   * Eliminación (Delete/DELETE)
4. Evalúa la estructura de cada método:
   * Validaciones presentes
   * Conexión a capa de datos
   * Códigos HTTP retornados
5. Busca pruebas automatizadas para cada operación CRUD.
6. Genera un CRUDValidationReport.
7. Si alguna operación está ausente o incompleta:
   * Reenvía prompt al DesarrolladorAgent.
   * Adjunta recomendación de mejora.

**6.X.4 Entradas / Salidas / Errores**

**Entradas:**

* Nombre de la entidad (ej. Producto)
* Ruta del proyecto generado
* Configuración del entorno

**Salidas:**

* Informe de validación CRUD (CRUDValidationReport)
* Nueva tarea si falta alguna función (FixTaskPrompt)

**Errores comunes tratados:**

* Métodos detectados pero vacíos
* Pruebas faltantes
* Rutas de API incorrectas
* Uso de verbos HTTP no estándar

**6.X.5 Algoritmos clave / pseudocódigo**

List<string> funcionesRequeridas = ["Create", "Read", "ReadById", "Update", "Delete"];

Dictionary<string, bool> funcionesEncontradas = new();

foreach (var archivo in DirectorioProyecto)

{

foreach (var funcion in funcionesRequeridas)

{

if (ContieneMetodo(archivo, funcion))

funcionesEncontradas[funcion] = true;

else

funcionesEncontradas[funcion] = false;

}

}

if (funcionesEncontradas.ContainsValue(false))

{

CrearTareaCorreccion(funcionesEncontradas);

ReenviarPromptADesarrollador();

}

**6.X.6 Métricas que reporta**

* % de funciones CRUD presentes por entidad
* Tiempo promedio entre detección y corrección
* Total de prompts reenviados por carencia CRUD
* Cobertura de test por operación CRUD

**6.X.7 Dependencias externas**

* FileSystemScanner
* CodeAnalyzerService
* TestCoverageParser
* ILogger
* DesarrolladorAgent
* PromptMemory

**6.X.8 Mejoras pendientes (20 ítems)**

1. **Soporte a métodos no convencionales** (ej. GuardarProducto, EliminarUsuario)
2. **Reconocimiento semántico por IA** de funciones CRUD
3. **Validación de contratos de entrada** (DTOs correctos, validaciones presentes)
4. **Generación automática de tests unitarios CRUD si faltan**
5. **Soporte a GraphQL**: queries y mutations equivalentes a CRUD
6. **Detección de rutas protegidas (autenticación requerida)**
7. **Reporte de cobertura por tipo de entidad** (complejidad)
8. **Gráficos de completitud CRUD por proyecto** (heatmap)
9. **Exportación del informe CRUD a PDF/Excel para QA**
10. **Validación de status HTTP retornados en controladores**
11. **Modo estricto: bloquea entrega si falta una función**
12. **Validación de comentarios y documentación XML en métodos**
13. **Generación automática de Swagger actualizado**
14. **Integración con Prometheus para métricas de completitud**
15. **Verificación de relaciones entre entidades (ej. FK, List<>)**
16. **Reconocimiento de operaciones CRUD asincrónicas**
17. **Validación de que los campos obligatorios estén cubiertos**
18. **Simulación automatizada de flujo completo CRUD** con mocks
19. **Reportes comparativos entre versiones del módulo**
20. **Modo "auditor": revisa proyectos legacy para medir completitud CRUD**

# **CitricAgent - Revisor Crítico de Calidad de Código**

**1. Propósito General**

El **CitricAgent** (Agente Crítico) es una entidad autónoma especializada en la inspección crítica del código fuente generado por otros agentes dentro del ecosistema SUPERDESARROLLADOR AUTÓNOMO. Su objetivo es **identificar errores, code smells, violaciones a buenas prácticas, problemas de rendimiento, seguridad, y patrones incorrectos**, con una mirada áspera pero constructiva.

Este agente funciona como una *segunda capa de revisión* luego de que el DesarrolladorAgent y el Fixer hayan generado y eventualmente corregido una solución. Emula la función de un **senior reviewer implacable** que asegura la máxima calidad del producto antes de ser validado por el tester.

**2. Flujo de Ejecución**

**Diagrama General:**

Prompt recibido

└── PlanificadorAgent genera backlog

└── DesarrolladorAgent genera código por tarea

└── FixerAgent corrige errores de build

└── CitricAgent evalúa críticamente el código

└── TestingAgent (si pasa calidad)

**Pasos Detallados:**

1. **Entrada**: Recibe el código fuente generado por el DesarrolladorAgent (después del Fixer si existió build\_error).
2. **Análisis**:
   * Ejecuta chequeos estáticos en base a reglas configurables (parsing con Roslyn o analizador propio).
   * Detecta:
     + Código duplicado
     + Funciones con demasiadas responsabilidades
     + Abuso de estados globales
     + Nombres poco descriptivos
     + Uso incorrecto de estructuras de datos
     + Falta de comentarios clave
     + Problemas de complejidad ciclomática
     + Errores comunes de seguridad (e.g., string concatenation con SQL)
3. **Reporte**:
   * Emite un informe CriticReport con:
     + Lista de observaciones críticas
     + Sugerencias de refactor
     + Grado de severidad (bajo, medio, alto, bloqueante)
     + Puntaje de calidad (0-100)
   * Devuelve IsApproved = false si hay alguna observación bloqueante.
4. **Acción según resultado**:
   * Si aprueba, pasa a TestingAgent.
   * Si rechaza, se regenera o corrige según backlog.

**3. Interfaces**

* **Input:**
* {
* "archivo": "path/to/Class.cs",
* "contenido": "public class...",
* "promptOriginal": "Generar un CRUD..."
* }
* **Output:**
* {
* "criticidad": "Alta",
* "puntuacion": 68,
* "comentarios": [
* "El método 'Actualizar' hace 4 cosas distintas.",
* "Evitar usar nombres como 'data1' o 'temp'.",
* "No se usa using para el context."
* ],
* "isApproved": false
* }

**4. Comportamiento Interno (Pseudocódigo)**

if (DetectaCodeSmells(archivo))

{

calcularCriticidad();

generarCriticReport();

return isApproved = false;

}

else

{

return isApproved = true;

}

**5. Dependencias**

* Parser C# (Roslyn o similar)
* Reglas configurables desde un critic.config.json
* Acceso a Prompt original
* Logger interno para traza de decisiones

**6. Posibles errores**

* Archivos vacíos
* Código malformado
* Lenguaje no soportado

**7. Métricas que expone**

* Tiempo medio de análisis por archivo
* Puntaje promedio de calidad por proyecto
* % de aprobaciones directas vs. rechazadas

**8. Mejoras Futuras (20)**

1. **Integrar GPT-4 para refinamiento semántico.**
2. **Autogenerar sugerencias de refactor automático.**
3. **Soportar otros lenguajes: JS, Python, etc.**
4. **Comparación con versiones anteriores para detectar regresiones de calidad.**
5. **Adaptación de reglas según tipo de proyecto (API, UI, Worker, etc.).**
6. **Plugin para IDEs que simule a Citric localmente.**
7. **Visualización de resultados en dashboard Prometheus/Grafana.**
8. **Modo "humano implacable": estilo crítico extremo.**
9. **Modo "mentoring": explica las sugerencias como lo haría un senior coach.**
10. **Benchmark de calidad por agente generador.**
11. **Historial de mejoras por archivo con git diff.**
12. **Autoaprendizaje de patrones de baja calidad frecuentes.**
13. **Exportación de reportes a PDF/CSV.**
14. **Integración con SonarQube o herramientas similares.**
15. **Evaluación de documentación técnica asociada.**
16. **Soporte para reglas personalizadas por usuario/organismo.**
17. **Parámetros de sensibilidad para entornos de desarrollo vs. producción.**
18. **Sugerencia de nombres más semánticos automáticamente.**
19. **Modo silencioso que no bloquea, solo advierte.**
20. **Autojuste de score según perfil del consumidor (cliente interno/externo).**

**9. Objetivo General del Agente**

El CitricAgent debe actuar como el último filtro de calidad antes del testing formal. Su función no es solo bloquear código malo, sino mejorar progresivamente la calidad de todo el ciclo de generación de software mediante críticas fundamentadas, trazables y automatizadas. En su versión futura, puede operar en modo continuo en entornos CI/CD validando calidad antes del despliegue.