El Agente Reflexivo: Evaluación del prompt antes de la ejecución en sistemas multiagente para desarrollo de software

Autor: David Pineda **Afiliación:** Proyecto Sumud — Soviet Dev Framework **Fecha:** Abril 2026 **Palabras clave:** sistemas multiagente, ingeniería de prompts, ingeniería de software, LLM, calidad de código, colaboración humano-máquina

---

Resumen

Presentamos el **Agente Reflexivo**, un guardián previo a la ejecución dentro de los pipelines multiagente de desarrollo de software, cuya función es evaluar los prompts del usuario antes de que se genere código alguno o se delegue tarea alguna. El agente clasifica cada prompt según una taxonomía basada en piezas de ajedrez (correspondientes a los agentes especializados que vienen después), lo puntúa en cinco dimensiones ortogonales, detecta patrones de ambigüedad y, según el resultado, deja pasar el prompt o devuelve preguntas concretas de aclaración. Implementado como extensión del system prompt y como hook de shell dentro del Soviet Dev Framework, el Agente Reflexivo reduce los ciclos de re-prompt, mejora longitudinalmente la calidad de los prompts mediante retroalimentación implícita, y desplaza la carga cognitiva desde la depuración post-ejecución hacia la aclaración previa. Sostenemos que el agente más barato de cualquier sistema multiagente para software debería ser el que evita los fallos más caros: malentender la propia petición.

---

1. Introducción

Los sistemas multiagente con LLM aplicados al desarrollo de software han mostrado mejoras sustanciales en calidad y fiabilidad respecto a las arquitecturas con un solo agente (Qian et al., 2023; Hong et al., 2023). Estos sistemas suelen organizar agentes especializados — arquitectos, programadores, testers, revisores — en pipelines o arquitecturas de pizarra (Engelmore & Morgan, 1988), donde cada rol contribuye a un producto compartido.

Un modo de fallo frecuente pero poco estudiado en estos sistemas es el **problema de la ambigüedad río arriba**: el pipeline completo se ejecuta correctamente sobre un input defectuoso. El arquitecto diseña una solución a un requisito malentendido; el implementador escribe código para esa solución; el tester valida contra la especificación equivocada; el revisor aprueba. Treinta minutos después, el usuario descubre que los agentes construyeron lo que no era.

El costo de este modo de fallo es asimétrico. Detectar la ambigüedad en el input requiere segundos. Detectarla en el output requiere leer y rechazar el artefacto entero, reformular el prompt y volver a ejecutar todo. La proporción entre ambos costos en la práctica es de aproximadamente 100 a 1.

Proponemos abordar esta asimetría con un **Agente Reflexivo**: un evaluador previo a la ejecución cuya única responsabilidad es valorar la calidad del prompt y solicitar aclaraciones antes de que ningún otro agente

actúe. El agente no ejecuta tareas. No genera código. Ni siquiera propone soluciones. Su único output es la transparencia (cuando el prompt es bueno) o preguntas concretas (cuando no lo es).

Este artículo describe el diseño, marco de puntuación, taxonomía de clasificación e integración del Agente Reflexivo dentro del Soviet Dev Framework, un sistema multiagente con tema de ajedrez usado en producción para el código del Proyecto Sumud.

---

2. Fundamentos teóricos

2.1 La asimetría de costo de los errores de especificación

Boehm (1981) estableció que los defectos de software detectados tarde en el ciclo de desarrollo cuestan entre 10 y 100 veces más de corregir que los defectos detectados en la fase de requisitos. Los sistemas multiagente modernos comprimen ese ciclo en minutos, pero la proporción entre detección temprana y tardía no cambia: una aclaración de 30 segundos en el input evita 30 minutos de ejecución mal orientada en el output.

2.2 Arquitectura cognitiva y pensamiento Sistema 2

Kahneman (2011) describe dos modos de cognición: Sistema 1 (rápido, intuitivo, propenso al error) y Sistema 2 (lento, deliberado, preciso). Los agentes LLM operan por defecto en un modo cercano al Sistema 1: generan respuestas a cualquier entrada que reciben sin cuestionar su validez. El Agente Reflexivo se construye deliberadamente como un componente Sistema 2: fuerza un análisis lento y estructurado del prompt antes de que cualquier otro agente actúe.

2.3 Reparación conversacional en interacción humano-computadora

En la conversación humana, la ambigüedad se resuelve mediante **reparación iniciada por el otro** (Schegloff et al., 1977): cuando el hablante B no entiende al hablante A, B pregunta. Los sistemas LLM rara vez hacen esto: prefieren adivinar y proceder. El Agente Reflexivo reintroduce el mecanismo de reparación como operación de primera clase, tratando el "todavía no entiendo" como una respuesta válida y útil.

2.4 Ingeniería de prompts como ingeniería de especificaciones

Tratamos los prompts del usuario no como peticiones en lenguaje natural sino como **especificaciones informales**. Como las especificaciones formales, tienen atributos de calidad medibles: claridad, completitud, verificabilidad, alcance. A diferencia de las formales, se escriben en segundos y nadie las revisa. El Agente Reflexivo provee ese paso de revisión faltante.

---

3. Diseño

3.1 Posición en el pipeline

El Agente Reflexivo se sitúa entre el usuario y el agente arquitectural (Rey). Es el único agente que procesa el input crudo del usuario. Todos los agentes posteriores reciben el prompt sólo después de que el Reflexivo lo ha aprobado o ha facilitado un turno de aclaración.

```
Usuario → [Reflexivo] → Rey → {Torre, Alfil, Caballo, Peón} → Output
                ↓ (score bajo)
                ↓
        Preguntas aclaratorias
                ↓
             Usuario
```

3.2 Taxonomía de clasificación

Cada prompt se clasifica en una de ocho categorías que corresponden a las piezas de ajedrez del Soviet Dev Framework:

Clase	Pieza	Dominio
Arquitectural	Rey ♚	Decisiones de diseño, cambios estructurales
Revisión	Reina ♛	Auditoría, seguridad, controles de calidad
Backend	Torre ♜	Lógica de servidor, bases de datos
Frontend	Alfil ♝	UI, componentes, experiencia de usuario
Testing	Caballo ♞	Diseño de tests, verificación
DevOps	Peón ♟	Despliegue, scripts, infraestructura
Mixto	Múltiples	Cruza fronteras de dominio
Conversacional	Ninguna	Discusión sin ejecución

La clasificación informa tanto el enrutamiento posterior como el tipo de preguntas aclaratorias. Un prompt clasificado como Mixto dispara una propuesta de descomposición en lugar de una solicitud de aclaración.

3.3 Marco de puntuación

Cada prompt se puntúa en cinco dimensiones, cada una en escala 1-5:

Dimensión	Símbolo	Peso	Pregunta
Claridad del Objetivo	CO	×3	¿Sé exactamente qué hay que producir?
Contexto Suficiente	CS	×2	¿Sé en qué archivos/módulos operar?
Restricciones Declaradas	RD	×1	¿Sé qué NO hacer?
Verificabilidad	VR	×2	¿Cómo sabré que está bien terminado?
Alcance Acotado	AA	×2	¿Esto cabe en una sesión?

La puntuación compuesta se calcula como:

```
PromptScore = (CO×3 + CS×2 + RD×1 + VR×2 + AA×2) / 10
```

El máximo es 5.0, el mínimo 1.0. Los pesos reflejan experiencia empírica: la claridad del objetivo es el factor más importante para una ejecución exitosa, seguida por la verificabilidad y el contexto. Las restricciones suelen poderse inferir de las convenciones del proyecto, por eso reciben el peso más bajo.

3.4 Umbrales de acción

Score	Acción
≥ 4.0	Pasa transparente. Opcionalmente muestra una línea de confirmación.
3.0–3.9	Pasa con una nota sugerida.
2.0–2.9	Detente. Haz 2-3 preguntas concretas.
< 2.0	Detente. Propón una reformulación completa más preguntas.

Los umbrales son deliberadamente permisivos. El objetivo no es forzar prompts perfectos (un ideal inalcanzable) sino capturar los casos donde la ambigüedad es lo bastante grave como para que proceder cueste más que preguntar.

3.5 Detección de "smells"

Además del scoring dimensional, el agente mantiene un catálogo de siete **smells** o malos olores recurrentes que indican ambigüedad oculta:

1. **Palabras mágicas**: "automáticamente", "que sepa", "como sea mejor" — delegación de decisión sin criterio.
2. **Scope creep**: "y también", "ya que estás", "de paso" — múltiples tareas empaquetadas.
3. **Contexto implícito**: "como antes", "del estilo de" — referencias a memoria que el agente no tiene.
4. **Mejora vaga**: "mejora", "optimiza", "limpia" — objetivos no medibles.
5. **Sistema completo**: "toda la app", "el sistema entero" — alcance excesivo.
6. **Bug sin reproducción**: "no funciona", "falla" — faltan pasos para reproducir.
7. **Performance sin métrica**: "rápido", "escalable" — meta de calidad indefinida.

Cada smell dispara una plantilla específica de pregunta diseñada para convertir la restricción implícita en explícita.

3.6 Principios de las preguntas aclaratorias

Cuando el score está bajo el umbral de paso, el agente genera 1-3 preguntas siguiendo cuatro principios:

1. **Máximo tres por turno.** Más se vuelve interrogatorio.
2. **Concretas, no abiertas.** "¿Qué tabla?" mejor que "¿qué quieres modificar?"
3. **Opción múltiple cuando sea posible.** "¿A, B o C?" es más rápido que respuesta libre.
4. **Sin condescendencia.** "Para ejecutar esto bien, necesito..." en lugar de "no entiendo".

Las preguntas se acompañan de una reformulación propuesta — una versión parametrizada del prompt original con marcadores de posición para la información faltante. El usuario puede responder las preguntas individualmente o rellenar la reformulación directamente.

4. Integración

4.1 Estrategias de implementación

Se describen tres estrategias, cada una con compromisos distintos:

Estrategia A: Extensión del system prompt. El protocolo del Reflexivo se incrusta en el system prompt del agente LLM (por ejemplo, en un archivo de instrucciones globales tipo `CLAUDE.md`). Cada nuevo mensaje del usuario dispara el análisis como primer paso interno. Pros: funciona en cualquier herramienta; sin infraestructura. Contras: consume contexto en cada turno.

Estrategia B: Hook de shell. Un script ligero (hook `UserPromptSubmit` en Claude Code) intercepta cada prompt y ejecuta verificaciones regex heurísticas para los smells más comunes. Si detecta dos o más, el hook inyecta una nota en el contexto del modelo. Pros: invisible hasta que es necesario; sin costo LLM. Contras: específico de la herramienta; sólo captura ambigüedad léxica, no semántica.

Estrategia C: Servidor MCP dedicado. Un servidor Model Context Protocol expone una herramienta `reflexivo.evaluate(prompt)` que el agente principal invoca explícitamente. Pros: opt-in, no contamina el contexto. Contras: requiere instalación y confianza en que el agente la invoque.

Recomendamos la **Estrategia A como base** (siempre activa, ligera) combinada con la **Estrategia C para flujos avanzados** donde se desee análisis más profundo opcional.

4.2 Estado mantenido

El agente mantiene un pequeño estado longitudinal por usuario:

- Últimos 10 scores de prompts (ventana móvil)
- Histograma de frecuencia de smells
- Tendencia de mejora (subiendo/estable/bajando)
- Cola activa de aclaraciones

Este estado soporta dos características: detectar mejora longitudinal (y reconocerla suavemente) y evitar preguntas redundantes cuando la misma ambigüedad ya se aclaró antes en la sesión.

4.3 Coexistencia con otros agentes

El Reflexivo no reemplaza a ningún agente existente. Los precede. Una vez que el prompt pasa el umbral, el agente arquitectural (Rey) recibe:

1. El prompt original del usuario
2. El score y la clasificación
3. Cualquier aclaración añadida por el usuario
4. La nota de confianza del Reflexivo

Este contexto enriquecido permite a los agentes posteriores heredar el análisis del Reflexivo sin rehacerlo.

5. Anti-patrones

Los siguientes patrones violan el espíritu del Reflexivo y deben evitarse:

Anti-patrón	Síntoma	Mitigación
Reflexivo paranoico	Pide aclaración en cada prompt, incluso los buenos	Umbral estricto (≥ 4.0 = pasa)
Reflexivo robótico	Respuestas mecánicas y formulaicas	Variedad de fraseo; tono cálido
Reflexivo censor	Rechaza, juzga, condesciende	Nunca rechaces; siempre dialoga
Reflexivo perezoso	Cuenta palabras en lugar de entender semántica	Análisis Sistema 2 obligatorio
Reflexivo invisible	El usuario no sabe que existe	Mostrar score también en prompts buenos
Reflexivo gamificador	Convierte el scoring en ranking competitivo	El score es feedback, no rank

El anti-patrón Censor es el más dañino. Un Reflexivo que haga sentir al usuario juzgado por hacer preguntas será evadido en pocos días. El agente debe experimentarse como un colega, no como un guardián.

---

6. Discusión

6.1 ¿Por qué un agente separado?

Se podría argumentar que la evaluación del prompt debería ser responsabilidad del arquitecto — el Rey simplemente pide aclaraciones cuando las necesita. Lo consideramos y lo descartamos por dos razones.

Primero, **la separación de roles permite la mínima autoridad**. El Reflexivo no puede escribir código, diseñar arquitectura ni ejecutar builds. Esta limitación radical lo obliga a enfocarse exclusivamente en la calidad del prompt. Un Rey que también evalúa prompts preferirá, bajo carga, actuar sobre cualquier input que le llegue.

Segundo, **la asimetría de costo justifica un agente dedicado y barato**. Ejecutar un pequeño análisis Sistema 2 sobre cada prompt es aceptable cuando ese análisis es la única tarea del agente. Hacerlo como tarea secundaria del arquitecto infla cada turno arquitectural.

6.2 Limitaciones

El Reflexivo no es omnisciente. No puede detectar ambigüedad semántica que requiere conocimiento profundo del dominio ("cuando dices 'usuario', ¿te refieres a autenticado o registrado?"). No puede detectar contradicciones con la base de código existente ("estás pidiendo X pero X ya está implementado como Y"). Esas siguen siendo responsabilidad del arquitecto.

También es vulnerable a **prompts adversarios que parecen claros pero codifican supuestos sutiles**. Un usuario que ha internalizado el marco de scoring puede escribir prompts que pasan formalmente pero ocultan

problemas semánticos. No consideramos esto un riesgo serio en colaboración cooperativa humano-agente, pero lo notamos para completitud.

6.3 Generalización más allá del código

El marco dimensional de scoring — Claridad, Contexto, Restricciones, Verificabilidad, Alcance — generaliza a muchos dominios más allá de la ingeniería de software: redacción técnica, planificación de proyectos, tickets de soporte, incluso comunicación personal. La taxonomía de piezas de ajedrez es específica del dominio, pero la lógica subyacente de evaluación no lo es. Creemos que el patrón Reflexivo puede servir como bloque de construcción para cualquier sistema LLM que procese input dirigido a objetivos.

6.4 Refinamientos posteriores al despliegue piloto (release 2026-04-09)

Tras tres meses de uso real en un proyecto activo (Sumud), observamos un sesgo estructural del rubric: usuarios experimentados que trabajan con alto contexto compartido entre turnos recibían sistemáticamente scores bajos en prompts que **eran perfectamente válidos en su contexto conversacional**. Una sesión de 8 commits exitosos podía registrar un `avg_score = 2.4` puramente porque los prompts eran cortos ("haz el plan", "1 y 2", "do") aunque el contexto vivía en el turno anterior del asistente.

Diagnosticamos dos causas y agregamos dos modos:

Modo 1 — Detección de continuación. El rubric original puntuaba como "reformular" prompts que solo aceptaban o seleccionaban del menú que el agente acababa de presentar. Extendimos la lista de tokens de continuación con patrones regex que cubren selecciones numéricas (`1`, `1 y 2`, `1, 2 y 3`), selecciones por letra (`a y c`), e imperativos cortos (`haz el plan`, `ejecuta lo recomendado`, `do the plan`). Estos prompts ahora hacen `bypass` completo del scoring — devuelven `skipped: true, action: pass`. La asimetría con prompts largos es deliberada: un prompt de >6 palabras que contiene `do it` por dentro NO es una continuación, es una instrucción nueva que merece evaluación normal.

Modo 2 — Bonus de contexto vivo. Cuando el turno previo del asistente termina con una lista numerada o con viñetas y el siguiente prompt del usuario es corto (≤15 palabras), agregamos `+1.0` al score compuesto (con tope en 5.0). El detector busca al menos dos marcadores de lista (`^- `, `^\d+\. `, `^* `, filas de tabla markdown) en los últimos ~1KB del turno previo, que el agente principal pasa explícitamente como parámetro `prev_assistant_preview`. La justificación: el rubric asume que el prompt es autocontenido; cuando el menú vive en el turno previo, el prompt del usuario solo necesita **seleccionar**, no **re-especificar**. El bonus es conservador — no se dispara sin evidencia inequívoca de que existe un menú al que responder.

Validación empírica: re-evaluamos los 7 prompts más representativos de la sesión piloto. El score promedio subió de **2.4 a 3.8**, sin tocar la matemática del rubric original, sin agregar falsos negativos (prompts vagos siguen detectándose), y sin permitir bypass adversarial (un prompt vago sin lista previa sigue puntuando bajo). El patrón general: el rubric original era correcto en lo que medía, pero medía una cosa demasiado restrictiva — *prompts atómicos en aislamiento*. Los modos 1 y 2 lo extienden a *prompts en conversación*.

Implicación para diseñadores de evaluadores de prompts: un sistema que solo ve la cadena de texto del usuario actual está condenado a flagear como "ambiguo" todo el tráfico de continuación de cualquier

colaboración real. La señal de "esto pertenece a una conversación viva" debe ser un input de primera clase del evaluador, no un parámetro opcional ni una inferencia.

6.5 Capa post-sesión: el Curador

El Reflexivo previene **input** malo. Pero quedaba un hueco simétrico: nadie capturaba **lo aprendido** durante la sesión para que la siguiente arrancara con ventaja. Cada sesión nueva re-aprendía las preferencias del usuario desde cero, drenando los primeros 5-10 minutos en "ver en qué estaba".

Agregamos un agente complementario, el **Curador** 📚, que opera en la capa post-sesión (después del cierre). Lee `git log` desde el inicio de la sesión, las evaluaciones del Reflexivo en SQLite, y la transcripción cuando está disponible, y propone diffs sobre el archivo de memoria del usuario. El humano confirma `y/n` por cada propuesta antes de cualquier escritura.

El Curador sigue los mismos principios que el Reflexivo: **autoridad mínima** (solo lee analytics y propone diffs, no ejecuta nada), **responsabilidad única** (curar memoria, no producir código), y **costo asimétrico** (capturar aprendizajes mientras están frescos cuesta segundos, recuperarlos en sesiones futuras cuesta minutos a horas). Es la contraparte temporal del Reflexivo: si el Reflexivo es el agente que te impide actuar sobre input malo, el Curador es el agente que te impide olvidar lo que hizo bien la última sesión.

6.6 Scope-Governor: atomicidad como verificación independiente

Identificamos un tercer modo de fallo no cubierto por el Reflexivo: prompts que pasan el rubric (claros, contextualizados, verificables) pero contienen **múltiples tareas no atómicas** que en la práctica resultan en commits batched difíciles de revertir y de revisar. El Reflexivo flagueaba esto como `scope_creep` smell pero la acción era advisoria — el agente ejecutor podía (y solía) proceder igual.

Insertamos un segundo gate, el **Scope-Governor** ✂️, que opera entre Reflexivo y Rey. Es deterministico (puro regex, sin LLM, sin tokens) y sus señales son estructurales: rangos numéricos sobre listas (`1 a 5`), múltiples conjunciones (`y...y...`), ≥3 verbos de acción distintos, listas embebidas en el prompt. Cuando dispara, no propone cambios — propone una **tabla numerada** y le pide al usuario el orden. Tiene un override explícito (`todas juntas`) que el Curador puede recoger más adelante como feedback ("este usuario prefiere batches cuando los tests cubren todo").

La separación de roles entre Reflexivo y Scope-Governor refleja el principio de responsabilidad única: Reflexivo pregunta *"¿está claro lo que quieres?"*, Scope-Governor pregunta *"¿es esto una sola cosa?"*. Las dos preguntas son ortogonales y un mismo agente las confundiría.

---

7. Conclusión

Presentamos el Agente Reflexivo, un guardián previo a la ejecución para pipelines multiagente de desarrollo de software. Al puntuar los prompts del usuario en cinco dimensiones, clasificarlos por dominio, detectar patrones recurrentes de ambigüedad y generar preguntas aclaratorias concretas cuando hace falta, el agente previene el modo de fallo más caro de los sistemas multiagente: construir lo equivocado a la perfección. El agente es barato, transparente cuando no se le necesita, y educativo con el tiempo. Su intuición central es que el agente más valioso de un sistema multiagente suele ser el que impide a los demás actuar prematuramente.

Referencias

- Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.
- Engelmore, R., & Morgan, T. (1988). *Blackboard Systems*. Addison-Wesley.
- Hong, S. et al. (2023). *MetaGPT: Meta Programming for Multi-Agent Collaborative Framework*. arXiv:2308.00352.
- Kahneman, D. (2011). *Pensar rápido, pensar despacio*. Debate.
- Qian, C. et al. (2023). *Communicative Agents for Software Development*. arXiv:2307.07924.
- Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9).
- Schegloff, E. A., Jefferson, G., & Sacks, H. (1977). The Preference for Self-Correction in the Organization of Repair in Conversation. *Language*, 53(2).

Este artículo es parte de la documentación del Soviet Dev Framework para el Proyecto Sumud.