The Reflexive Agent: Pre-Execution Prompt Evaluation in Multi-Agent Software Development

Author: David Pineda **Affiliation:** Sumud Project — Soviet Dev Framework **Date:** April 2026 **Keywords:** multi-agent systems, prompt engineering, software engineering, LLM, code quality, human-computer collaboration

---

Abstract

We introduce the **Reflexive Agent**, a pre-execution gatekeeper in multi-agent software development pipelines that evaluates user prompts before any code generation or task delegation occurs. The agent classifies each prompt into a chess-piece taxonomy (corresponding to specialized downstream agents), scores it across five orthogonal dimensions, detects ambiguity patterns, and either passes the prompt transparently or returns concrete clarifying questions. Implemented as a system-prompt extension and a shell-hook within the Soviet Dev Framework, the Reflexive Agent reduces re-prompting cycles, improves prompt quality longitudinally through implicit feedback, and shifts cognitive load from post-execution debugging to pre-execution clarification. We argue that the cheapest agent in any software multi-agent system should be the one that prevents the most expensive failures: misunderstanding the request itself.

---

1. Introduction

Multi-agent LLM systems for software engineering have demonstrated significant gains in code quality and reliability over single-agent approaches (Qian et al., 2023; Hong et al., 2023). These systems typically organize specialized agents — architects, coders, testers, reviewers — into pipelines or blackboard architectures (Engelmore & Morgan, 1988) where each role contributes to a shared work product.

A common but rarely studied failure mode of such systems is the **upstream ambiguity problem**: the entire pipeline executes correctly on a flawed input. The architect designs a solution to a misunderstood requirement; the implementer codes that solution; the tester validates it against the misunderstood spec; the reviewer approves. Thirty minutes later, the user discovers that the agents built the wrong thing.

The cost of this failure mode is asymmetric. Detecting ambiguity at input requires seconds of analysis. Detecting it at output requires reading and rejecting the entire produced artifact, then re-prompting and re-executing. The ratio of these costs in practice is approximately 100:1.

We propose addressing this asymmetry with a **Reflexive Agent**: a pre-execution evaluator whose sole responsibility is to assess prompt quality and request clarification before downstream agents engage. The agent does not execute tasks. It does not generate code. It does not even propose solutions. Its only output is either transparency (when the prompt is good) or concrete questions (when it is not).

This paper describes the design, scoring framework, classification taxonomy, and integration of the Reflexive Agent within the Soviet Dev Framework, a chess-themed multi-agent system used in production for the Sumud Project codebase.

2. Theoretical Foundations

2.1 The Cost Asymmetry of Specification Errors

Boehm (1981) established that software defects discovered late in the development lifecycle cost 10-100× more to fix than defects caught at requirements time. Modern multi-agent LLM systems compress the entire development lifecycle into minutes, but the cost ratio between early and late detection remains unchanged: a 30-second clarification at input avoids 30 minutes of misdirected execution at output.

2.2 Cognitive Architecture and System 2 Thinking

Kahneman (2011) describes two modes of cognition: System 1 (fast, intuitive, error-prone) and System 2 (slow, deliberate, accurate). LLM agents typically operate in a System-1-like mode by default — they generate responses to whatever input arrives without questioning its validity. The Reflexive Agent is deliberately constructed as a System 2 component: it forces a slow, structured analysis of the prompt before any downstream agent acts.

2.3 Conversational Repair in Human-Computer Interaction

In human conversation, ambiguity is resolved through **other-initiated repair** (Schegloff et al., 1977): when speaker B does not understand speaker A, B asks. LLM systems rarely do this — they prefer to guess and proceed. The Reflexive Agent re-introduces the repair mechanism as a first-class operation, treating "I don't understand yet" as a valid and useful response.

2.4 Prompt Engineering as Specification Engineering

We treat user prompts not as natural-language requests but as **informal specifications**. Like formal specifications, they have measurable quality attributes: clarity, completeness, verifiability, scope. Unlike formal specifications, they are written in seconds and reviewed (if at all) by no one. The Reflexive Agent provides this missing review step.

---

3. Design

3.1 Position in the Pipeline

The Reflexive Agent sits between the user and the architectural agent (Rey). It is the only agent that processes raw user input. All downstream agents receive the prompt only after the Reflexive has either approved it or facilitated a clarification turn.

```
User → [Reflexive] → Rey → {Torre, Alfil, Caballo, Peon} → Output
               ↓ (low score)
               ↓
        Clarifying questions
               ↓
             User
```

3.2 Classification Taxonomy

Each prompt is classified into one of eight categories that correspond to the chess pieces of the Soviet Dev Framework:

Class	Chess Piece	Domain
Architectural	Rey ♚	Design decisions, structural changes
Review	Reina ♛	Audit, security, quality gates
Backend	Torre ♜	Server-side logic, databases
Frontend	Alfil ♝	UI, components, user experience
Testing	Caballo ♞	Test design, verification
DevOps	Peón ♟	Deploy, scripts, infrastructure
Mixed	Multiple	Crosses domain boundaries
Conversational	None	Discussion without execution

The classification informs both the downstream routing and the type of clarifying questions asked. A prompt classified as Mixed triggers a decomposition proposal rather than a clarification request.

3.3 Scoring Framework

Each prompt is scored along five dimensions, each on a 1-5 scale:

Dimension	Symbol	Weight	Question
Clarity of Objective	CO	×3	Do I know exactly what to produce?
Context Sufficiency	CS	×2	Do I know which files/modules to operate on?
Restrictions Declared	RD	×1	Do I know what NOT to do?
Verifiability	VR	×2	How will I know it is done correctly?
Acotated Scope	AA	×2	Does this fit in one session?

The composite score is computed as:

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

The maximum score is 5.0; the minimum is 1.0. The weighting reflects empirical experience: clarity of objective is the single most important factor in successful execution, followed by verifiability and context. Restrictions can usually be inferred from project conventions, so they receive the lowest weight.

3.4 Action Thresholds

Score	Action
≥ 4.0	Pass through. Optionally display a one-line confirmation.
3.0–3.9	Pass through with one suggested note.
2.0–2.9	Halt. Ask 2-3 concrete questions.
< 2.0	Halt. Propose a full reformulation plus questions.

The thresholds are deliberately permissive. The goal is not to force perfect prompts (an unreachable ideal) but to catch the cases where ambiguity is severe enough that proceeding would waste more time than asking would.

3.5 Smell Detection

In addition to dimensional scoring, the agent maintains a catalog of seven prompt **smells** — recurring patterns that indicate hidden ambiguity:

1. **Magic words**: "automatically", "should know", "as appropriate" — delegating decisions without criteria.
2. **Scope creep**: "and also", "while you're at it", "in passing" — multiple tasks bundled.
3. **Implicit context**: "like before", "the usual way" — references to memory the agent does not have.
4. **Vague improvement**: "improve", "optimize", "clean up" — non-measurable objectives.
5. **Total system**: "the whole app", "all the code" — excessive scope.
6. **Bug without reproduction**: "it's broken", "doesn't work" — missing steps to reproduce.
7. **Performance without metric**: "make it faster", "scalable" — undefined quality target.

Each smell triggers a specific question template designed to convert the implicit constraint into an explicit one.

3.6 Clarifying Question Principles

When the score is below the pass threshold, the agent generates 1-3 questions following four principles:

1. **Maximum three per turn.** More becomes interrogation.
2. **Concrete, not open-ended.** "Which table?" beats "What do you want to modify?"
3. **Multiple choice when possible.** "A, B, or C?" is faster than free response.
4. **No condescension.** "To execute this well, I need..." not "I don't understand."

The questions are accompanied by a proposed reformulation — a parameterized version of the original prompt with placeholders for the missing information. The user can either answer the questions individually or fill in the reformulation directly.

4. Integration

4.1 Implementation Strategies

Three implementation strategies are described, each with different trade-offs:

Strategy A: System Prompt Extension. The Reflexive protocol is embedded into the system prompt of the LLM agent (e.g., a `CLAUDE.md` global instructions file). Every new user message triggers the analysis as the first internal step. Pros: works in any tool; no infrastructure required. Cons: consumes context window on every turn.

Strategy B: Shell Hook. A lightweight shell script (`UserPromptSubmit` hook in Claude Code) intercepts each prompt and runs heuristic regex checks for the most common smells. If two or more smells are detected, the hook injects a note into the model's context. Pros: invisible until needed; no LLM cost. Cons: tool-specific; only catches lexical ambiguity, not semantic.

Strategy C: Dedicated MCP Server. A Model Context Protocol server exposes a `reflexivo.evaluate(prompt)` tool that the main agent calls explicitly. Pros: opt-in, doesn't pollute context. Cons: requires installation and trust that the agent will invoke it.

We recommend **Strategy A as the baseline** (always active, lightweight) combined with **Strategy C for advanced workflows** where opt-in deeper analysis is desirable.

4.2 State Maintained

The agent maintains a small longitudinal state per user:

- Last 10 prompt scores (rolling window)
- Smell frequency histogram
- Improvement trend (rising/stable/declining)
- Active clarification queue

This state supports two features: detecting longitudinal improvement (and gently acknowledging it) and avoiding redundant questions when the same ambiguity has already been clarified earlier in the session.

4.3 Coexistence with Other Agents

The Reflexive does not replace any existing agent. It precedes them. Once a prompt passes the threshold, the architectural agent (Rey) receives:

1. The original user prompt
2. The score and classification
3. Any clarifications added by the user
4. The Reflexive's confidence note

This enriched context allows downstream agents to inherit the Reflexive's analysis without re-doing it.

5. Anti-Patterns

The following patterns violate the spirit of the Reflexive and should be avoided:

Anti-pattern	Symptom	Mitigation
Paranoid Reflexive	Asks for clarification on every prompt, even good ones	Strict score threshold (≥ 4.0 = pass)
Robotic Reflexive	Mechanical, formulaic responses	Vary phrasing; warm tone
Censoring Reflexive	Rejects, judges, condescends	Never reject; always dialogue
Lazy Reflexive	Counts words instead of understanding semantics	Mandatory System 2 analysis
Invisible Reflexive	User never knows it exists	Display score on good prompts too
Gamifying Reflexive	Turns scoring into competitive ranking	Score is feedback, not rank

The Censoring Anti-pattern is the most damaging. A Reflexive that makes the user feel judged for asking questions will be circumvented within days. The agent must be experienced as a colleague, not a gatekeeper.

---

6. Discussion

6.1 Why a Separate Agent?

One could argue that prompt evaluation should be the architect's responsibility — the Rey simply asks for clarification when needed. We considered this and rejected it for two reasons.

First, **role separation enables minimal authority**. The Reflexive cannot write code, design architecture, or run builds. This radical limitation forces it to focus exclusively on prompt quality. A Rey that also evaluates prompts will, under load, prefer to act on whatever input arrives.

Second, **the cost asymmetry warrants a dedicated cheap agent**. Running a small System-2 analysis on every prompt is acceptable when that analysis is the agent's only job. Running it as a side-task of the architect inflates every architectural turn.

6.2 Limitations

The Reflexive is not omniscient. It cannot detect semantic ambiguity that requires deep domain knowledge ("when you say 'user', do you mean authenticated or registered?"). It cannot detect contradictions with the existing codebase ("you're asking for X but X is already implemented as Y"). These remain the architect's responsibility.

It is also vulnerable to **adversarial prompts that look clear but encode subtle assumptions**. A user who has internalized the scoring framework can write prompts that pass formally while hiding semantic problems. We do not consider this a serious risk in cooperative human-agent collaboration but note it for completeness.

6.3 Generalization Beyond Code

The dimensional scoring framework — Clarity, Context, Restrictions, Verifiability, Scope — generalizes to many domains beyond software engineering: technical writing, project planning, support tickets, even personal communication. The chess-piece taxonomy is domain-specific, but the underlying evaluation logic is not. We believe the Reflexive pattern can serve as a building block for any LLM system that processes goal-directed user input.

6.4 Refinements after the pilot deployment (release 2026-04-09)

After three months of real use in an active project (Sumud), we observed a structural bias of the rubric: experienced users working with high shared context between turns systematically received low scores on prompts that **were perfectly valid in their conversational context**. An 8-commit successful session could log `avg_score = 2.4` purely because prompts were short ("do the plan", "1 and 2", "do") even though the context lived in the assistant's prior turn.

We diagnosed two causes and added two modes:

Mode 1 — Continuation detection. The original rubric scored as "reformulate" prompts that simply accepted or selected from the menu the agent had just presented. We extended the continuation token list with regex patterns that cover numeric selections (`1`, `1 and 2`, `1, 2 and 3`), letter selections (`a and c`), and short imperatives (`do the plan`, `run the recommended thing`, `haz el plan`). These prompts now `bypass` scoring entirely — they return `skipped: true, action: pass`. The asymmetry with long prompts is deliberate: a >6-word prompt that contains `do it` inside is NOT a continuation, it is a new instruction that deserves normal evaluation.

Mode 2 — Live-context bonus. When the prior assistant turn ends with a numbered or bulleted list AND the next user prompt is short (≤15 words), we add `+1.0` to the composite score (capped at 5.0). The detector looks for at least two list markers (`^- `, `^\d+\. `, `^* `, markdown table rows) in the last ~1KB of the prior turn, which the calling agent passes explicitly as a `prev_assistant_preview` parameter. The justification: the rubric assumes the prompt is self-contained; when the menu lives in the prior turn, the user's prompt only needs to **select**, not **re-specify**. The bonus is conservative — it never fires without unambiguous evidence that there is a menu to respond to.

Empirical validation: we re-evaluated the 7 most representative prompts of the pilot session. The average score went from **2.4 to 3.8**, without touching the math of the original rubric, without adding false negatives (vague prompts still detect as such), and without allowing adversarial bypass (a vague prompt without a prior list still scores low). The general pattern: the original rubric was correct in what it measured, but it measured too restrictive a thing — *atomic prompts in isolation*. Modes 1 and 2 extend it to *prompts in conversation*.

Implication for prompt-evaluator designers: a system that only sees the user's current text string is condemned to flag as "ambiguous" all the continuation traffic of any real collaboration. The signal "this belongs to a live conversation" must be a first-class input of the evaluator, not an optional parameter or an inference.

6.5 Post-session layer: the Curador

The Reflexivo prevents bad **input**. But there was a symmetric gap left: nobody captured **what was learned** during the session so the next one would start with an advantage. Each new session re-learned user preferences from scratch, draining the first 5-10 minutes in "let me see what we were doing".

We added a complementary agent, the **Curador** 📚 (Memory Curator), which operates in the post-session layer (after closing). It reads `git log` from the session start, the Reflexivo evaluations in SQLite, and the transcript when available, and proposes diffs over the user's memory file. The human confirms `y/n` per proposal before any write.

The Curador follows the same principles as the Reflexivo: **minimal authority** (only reads analytics and proposes diffs, executes nothing), **single responsibility** (curate memory, not produce code), and **asymmetric cost** (capturing lessons while fresh costs seconds, recovering them in future sessions costs minutes to hours). It is the temporal counterpart of the Reflexivo: if the Reflexivo is the agent that prevents you from acting on bad input, the Curador is the agent that prevents you from forgetting what last session did right.

6.6 Scope-Governor: atomicity as an independent verification

We identified a third failure mode not covered by the Reflexivo: prompts that pass the rubric (clear, contextualized, verifiable) but contain **multiple non-atomic tasks** that in practice result in batched commits hard to revert and review. The Reflexivo flagged these as a `scope_creep` smell but the action was advisory — the executing agent could (and usually did) proceed anyway.

We inserted a second gate, the **Scope-Governor** ✂️, which operates between Reflexivo and Rey. It is deterministic (pure regex, no LLM, no tokens) and its signals are structural: numeric ranges over lists (`1 to 5`), multiple conjunctions (`and...and...`), ≥3 distinct action verbs, embedded lists in the prompt. When it triggers, it does not propose changes — it proposes a **numbered table** and asks the user for the order. It has an explicit override (`all together`) which the Curador can later collect as feedback ("this user prefers batches when tests cover everything").

The role separation between Reflexivo and Scope-Governor reflects the principle of single responsibility: Reflexivo asks *"is what you want clear?"*, Scope-Governor asks *"is this one thing?"*. The two questions are orthogonal and a single agent would conflate them.

---

7. Conclusion

We introduced the Reflexive Agent, a pre-execution gatekeeper for multi-agent software development pipelines. By scoring user prompts on five dimensions, classifying them by domain, detecting recurring ambiguity patterns, and generating concrete clarifying questions when needed, the agent prevents the most expensive failure mode of multi-agent systems: building the wrong thing perfectly. The agent is cheap, transparent when not needed, and educational over time. Its core insight is that the most valuable agent in a multi-agent system is often the one that prevents the others from acting prematurely.

References

- 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). *Thinking, Fast and Slow*. Farrar, Straus and Giroux.
- 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).

This paper is part of the Soviet Dev Framework documentation for the Sumud Project.