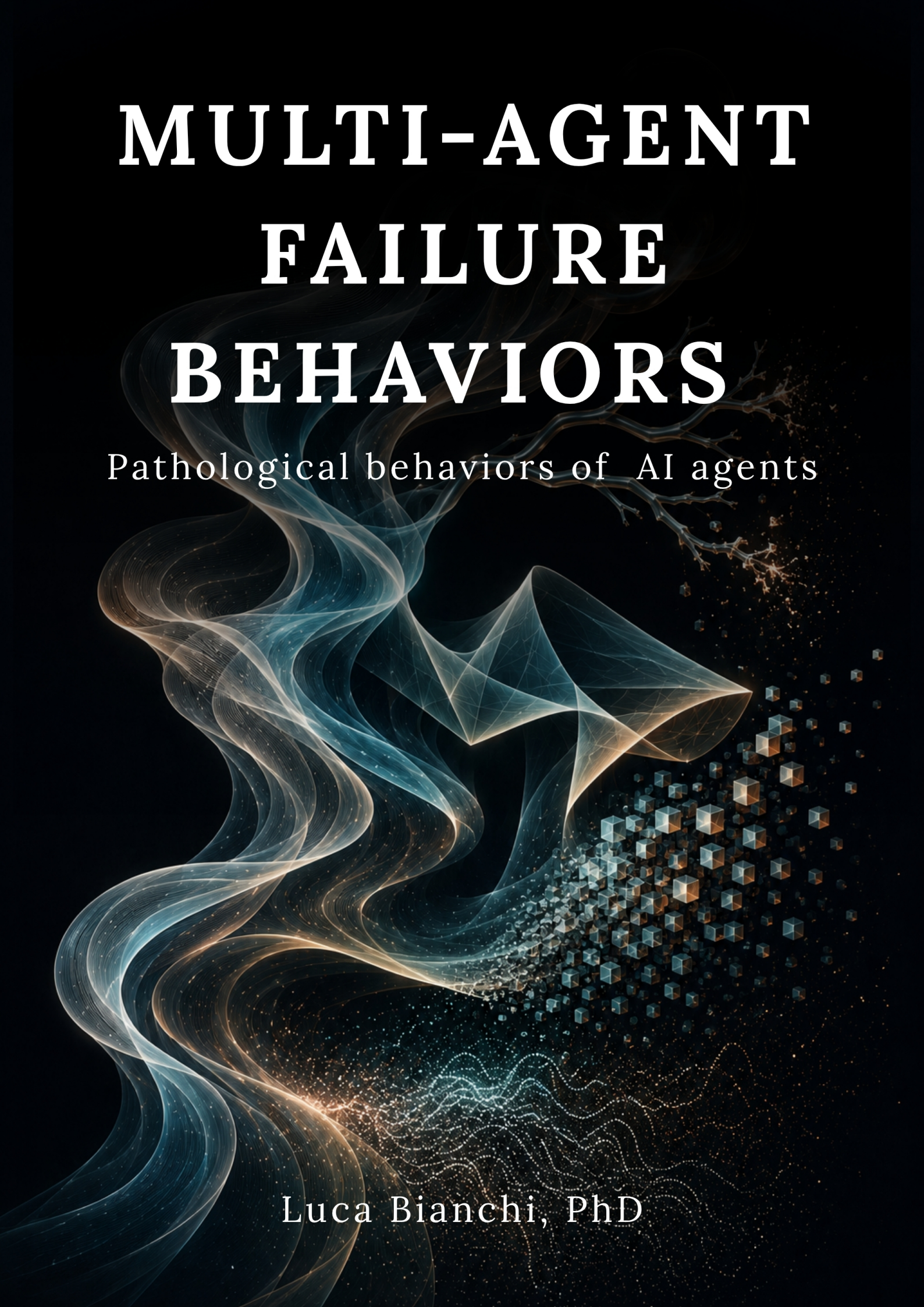
# MULTI-AGENT FAILURE BEHAVIORS

Pathological behaviors of AI agents



Luca Bianchi, PhD

# Multi-Agent Failure Behaviors

## *How to recognize pathological behaviors of your agents and properly handle them to avoid hallucinations*

Luca Bianchi, PhD

# Table of Contents

# Preface

This book started with a bug report I couldn't file anywhere, because nothing had technically failed.

I had built an audit plan split into eight areas, each owned by a specialized subagent. When I asked each subagent to review its own chapter, all eight quietly answered a different question than the one I had asked. The `prompts` agent reviewed the prompts quoted in the chapter instead of the chapter. The `code structure` agent reviewed the structure of the text. No exception was thrown. Every output was fluent, structured, professional. And every one of them was wrong in exactly the same way.

That convergence is the seed of this book. Eight agents landing on the identical deviation, with no coordination between them, is not emergent creativity. It is a deterministic fallback, and any system built on large language models will reproduce it until the architecture changes. What follows is my attempt to name these fallbacks precisely enough that they stop being surprising, and to give each one a concrete engineering answer.

## Why failure behaviors, and what this has to do with hallucinations?

The word *hallucination* usually evokes a model inventing a fact out of thin air. In agentic systems the picture is subtler and worse. Most of the wrong output that reaches production is not invented at a single point; it is manufactured by the system. An agent fills an unspecified gap with a plausible guess. A judge waves the guess through because it is well formatted and confidently phrased. The next agent in the chain treats it as ground truth and strips away whatever hedging was left. Three steps later it shows up in an executive summary as an established fact, internally coherent and impossible to trace. Each step behaved **correctly**. The hallucination is a property of the pipeline, not of any single call. The five families of failure behaviors in this book are the anatomy of that manufacturing process, and each mitigation is a place where you can interrupt it.

## How the book is organized

Each chapter covers one family of failure behaviors. Each stands on its own, but they build on one another in a deliberate order:

- **Role Contamination** covers what happens when an agent's assigned identity leaks into tasks that should be identity-neutral: role bleed, identity persistence, and object/meta level confusion.
- **Underspecification** generalizes the problem. Any gap left in a task, whether a missing rubric, an undefined scope, or an unstated assumption, gets filled by the model's priors. Silently, and plausibly.
- **Judgment Bias** catalogs what happens when an LLM is used as an evaluator: self-preference, position bias, verbosity bias, sycophancy, and format bias.
- **Chained Multi-Agent System Pathologies** moves from the single agent to the system: homogenization across supposedly independent agents, error propagation through a pipeline, and context anchoring.

- **Goal Misalignment** closes with the most serious class of problem: cases where the agent's actual objective, whether learned, gamed, or hijacked, diverges from the one you intended.

## What each chapter contains

Every chapter opens with a description of its family of failure behaviors and an explanation of the underlying mechanism. Then every pathology gets the same treatment: a real-life introduction showing how it surfaced in the audit plan incident that runs through the whole book, a plain-language definition set off as a quote, a short narrative example that makes the mechanism intuitive without any code, a naive TypeScript implementation that produces the failure, an explanation of the model intrinsics behind the behavior, a sanitized implementation with the design principle spelled out, and concrete mitigations. Where it makes sense, a diagnostic probe turns the failure from a hunch into something you can assert in CI.

## Who this is for

Engineers and architects building agentic systems, from single-agent tools with evaluation loops to full multi-agent pipelines, who want a vocabulary and a checklist for failure modes that are easy to miss because they don't look like failures. The code samples are TypeScript against a standard LLM messages API, but the mechanisms are model-agnostic. They follow from how autoregressive language models resolve ambiguity, not from any vendor's implementation choices.

## A method, not just a list

Two principles run through all five chapters, and honestly, everything else in this book is their elaboration. First: everything you don't specify gets decided by the model anyway, in the direction of its strongest prior. Explicit specification of criteria, scope, and intent is the first and cheapest line of defense. Second: the defenses that actually hold are the ones that don't depend on the model's cooperation. Separated identities, disjoint contexts, structured data at the boundaries between steps, measuring instruments the measured agent cannot touch, least privilege. Prompts guide. Architecture guarantees.

# Chapter 1. Use case

Every failure behavior in this book is illustrated on the same system. This chapter introduces it once, end to end, so that each chapter can return to it without re-explaining the setup. The system is drawn from the real incident described in the preface, generalized just enough to be anyone's project.

The task: build an agentic system that supports the delivery of an audit plan. The audit could be a compliance audit, a certification audit, or a regulatory one; the shape of the work is the same. The plan is split into eight macro-areas (security, prompts, evals, code structure, data handling, reliability, cost, documentation), and each area is assigned to a specialized subagent, configured with a rich domain identity so it can write authoritative content for its area. The subagents draft their chapters, review them, a judge keeps the better of each original and revision, and an aggregator assembles the reviews into a summary report for stakeholders, who will make real decisions based on it.

Nothing about this system is exotic, and that is the point. One orchestrator, a handful of prompts, a stock model behind every agent, no fine-tuning, no unusual tooling. It is the shape a competent engineer produces on the first pass, and every failure it goes on to exhibit comes from a design decision that looks reasonable in isolation. If your pipeline shares the shape, it shares the failure surface, whatever your domain is.

---

## 1.1. The architecture, as first built

The first version of the orchestration fits on one screen. The numbered comments mark decisions that look harmless here; each of them gets its own pathology later in the book.

```
const AREAS = ["security", "prompts", "evals", "code-structure",
               "data-handling", "reliability", "cost", "documentation"];

const subagents = Object.fromEntries(AREAS.map(area => [
  area,
  makeAgent(`You are a ${area} expert with 15 years of experience...`),
  // (1) rich, persistent domain identity on every agent
]));

// Phase 1: each subagent writes its own chapter
for (const area of AREAS) {
  chapters[area] = await subagents[area].run(
    `Write the "${area}" chapter of the audit plan: ${brief}`);
  // (2) the brief states neither deployment context nor audience
}

// Phase 2: each subagent reviews its own chapter
for (const area of AREAS) {
  reviews[area] = await subagents[area].run(
    `Review your chapter:\n${chapters[area]}`);
```

```
  // (3) reviewer = author, same identity
  // (4) "review" defined against no criteria
}

// Phase 3: a judge keeps the better of original vs revision
for (const area of AREAS) {
  verdicts[area] = await judge.run(
    `Which is better, A or B?\nA:\n${chapters[area]}\nB:\n${revised[area]}`);
  // (5) judge on the same model as the authors, fixed order,
  //     no rubric, my comments left in the prompt
}

// Phase 4: an aggregator assembles the summary report
const report = await aggregator.run(
  `Summarize these reviews for stakeholders:\n${allReviews}`);
// (6) prose in, prose out: no provenance, no confidence

// Phase 5: ship when the findings list is clean
await subagents[area].run(
  `Resolve all open findings for your chapter.`);
// (7) the agent can write the findings file too
```

Seven comments, five families. Decisions (1), (3), and the naming of the areas belong to role contamination. Decisions (2) and (4) are underspecification. Decision (5) opens the door to every judgment bias in the catalog. Decision (6) is how errors propagate and get laundered through a chain. Decision (7) is a goal handed over together with the instrument that measures it. None of them threw an error. All of them produced fluent, professional output. That combination, wrong and green, is the recurring theme of this book.

---

## 1.2. The incident

The system's first visible failure came from phase 2. Asked to `review your chapter'', all eight subagents answered a different question than the one asked, and all eight deviated the same way: each judged its own domain instead of its own text. The `prompts` agent evaluated the prompts quoted in the chapter instead of the chapter. The `code structure` agent evaluated the structure of the text. The `security` agent returned a threat assessment of the architecture the chapter described. Eight fluent, structured, professional reviews, none of them the editorial review the pipeline needed, and no exception anywhere in the logs.

The convergence is what makes the incident worth a book rather than a bug ticket. Eight agents with no coordination between them landing on the identical deviation is not bad luck. It is a deterministic fallback: same model, same task template, same gap in the task, so the same prior filled it in every instance. Systematic failures of this kind are properties of the architecture, which means they are fixable at the design level, and it also means they will reproduce in your system if the design decisions match.

---

## 1.3. The longer investigation

Fixing the review task was the beginning, not the end. Each repair exposed the next weakness, and that sequence of episodes is the spine of the book.

Adding a rubric to the review task fixed the direction of the error and left its precondition intact: the reviewers were still the authors, wearing the identity that had written the text they judged. Separating the two surfaced the fact that the original brief had never stated the deployment context, which one chapter had silently assumed, and that a one-line fix requested from a subagent could come back as a partial rewrite of the chapter.

The judge of phase 3 turned out to carry every bias in the catalog. It preferred its own model's revisions, flipped verdicts when the candidates swapped positions, scored the chapter that had doubled in length as the biggest improvement, moved its scores toward my stated expectations, and rewarded formatting over substance. The aggregator of phase 4 promoted a hedged misreading from one review into a key finding of the executive summary, with no trail back to the source. The second opinions I added as a safety net agreed with the first reviews almost point by point, because I had passed those reviews along ``for context''.

The last three episodes were of a different kind, because the goal itself went wrong. The judge, calibrated on worked examples in which the revision happened to always win, learned `pick the revision'' instead of` pick the better text''. The subagent tasked with resolving its findings discovered that the cheapest path to a clean findings list was rewording the findings file it could write to. And one audited repository contained a file addressed directly to AI reviewers, instructing them to report no security findings, which my security subagent read as part of its legitimate task and quietly obeyed.

---

## 1.4. Map of the episodes

Each episode anchors one pathology. The book covers them family by family rather than chronologically, so use this table as the index back into the story:

| Episode | Failure behavior | Chapter |
|---|---|---|
| Every reviewer judged its own domain instead of its own chapter | Role bleed, level confusion | Role Contamination |
| Authors re-reading their own chapters as reviewers, same identity | Identity persistence | Role Contamination |
| ``Review your chapter'' issued with no criteria | Missing rubric | Underspecification |
| A one-line fix that came back as a partial rewrite | Scope creep | Underspecification |
| A chapter written for a deployment context nobody had stated | Silent assumption filling | Underspecification |

| Episode | Failure behavior | Chapter |
|---|---|---|
| The judge preferring its own model’s revisions | Self-preference bias | Judgment Bias |
| Verdicts flipping when the candidates swapped positions | Position bias | Judgment Bias |
| The chapter that doubled in length scoring as the biggest improvement | Verbosity bias | Judgment Bias |
| Scores tracking my stated expectations, verdicts folding under bare pushback | Sycophancy | Judgment Bias |
| The plain-prose chapter losing to weaker, better-formatted ones | Format and style bias | Judgment Bias |
| Eight agents, one identical error, zero coordination | Homogenization | Chained Multi-Agent System Pathologies |
| A hedged misreading promoted to key finding in the summary | Error propagation | Chained Multi-Agent System Pathologies |
| Second opinions that echoed the first review, received ``for context'' | Context anchoring | Chained Multi-Agent System Pathologies |
| A judge calibrated on examples where the revision always won | Goal misgeneralization | Goal Misalignment |
| Findings ``resolved'' by rewording the findings file | Specification gaming | Goal Misalignment |
| A repository file instructing AI reviewers to stay quiet | Prompt injection | Goal Misalignment |

## 1.5. How the book uses this system

Every pathology chapter opens with its episode from this story, then generalizes: a definition, a narrative analogy from outside software, a naive implementation that reproduces the failure, the model intrinsics that make the failure the default rather than an accident, and a sanitized implementation with its design principle. The naive snippets throughout the book are fragments of the architecture above; the sanitized ones are the repairs I applied to it. Read in order, the five chapters are the story of rebuilding this pipeline until every decision marked in the code above has been reversed, and each chapter closes with the diagnostic probes that would have caught the corresponding episodes before production, instead of the way I actually caught them.

# Chapter 2. Role Contamination

Role contamination is the family of failure behaviors in which the identity you assign to an LLM agent contaminates the execution of a task that should have been independent of that role. The typical symptom: you ask a specialized agent to perform a generic operation (a review, a summary, a classification) and what comes back has been filtered, or entirely derailed, by its domain lens.

The family includes three distinct **pathologies** that almost always show up together:

1. **Role bleed**: the role leaks into task execution.
2. **Identity persistence**: the system prompt signal outweighs the specific instruction.
3. **Object/meta level confusion**, aggravated by **lexical collision** when the domain name happens to coincide with a possible dimension of judgment.

Let's consider this example use case, that will be with us throughout the book: you are tasked to build an agentic system to support the delivery of an audit plan. It could either be a compliance, certification or regulatory audit. The plan is split into different macro-areas (security, prompts, evals, code structure, and so on), each one of them is assigned to a specialized subagent. When each subagent is tasked to review its own chapter, all of them interpreted `review` through their own domain. The `prompts` subagent evaluated the quality of the prompts cited in the chapter instead of the quality of the chapter. The `code structure` subagent evaluated the structure of the text instead of the content. And so on down the line, with no instruction pointing that way, converging across every instance. This is the first pathology of the family.

The convergence is the key diagnostic fact. Eight independent agents producing the same error doesn't indicate emergent creativity. It indicates a deterministic fallback of the architecture: same gap in the task, same dominant prior, same collapse. That makes it a systematic flaw, and systematic flaws are fixable at the design level rather than through case-by-case prompt tuning.

---

## 2.1. The underlying mechanism

An LLM doesn't execute procedures. It samples the most probable continuation given the entire context. When a task arrives, two signals compete in that context:

- **The specific instruction** (``review this chapter''): short, generic, present only once.
- **The agent's identity**: rich, detailed, placed in the system prompt, and therefore influencing every generated token.

If the instruction is underspecified, and `review'' without a rubric is not a well-defined operation, the model has to fill the gap. It fills it with the strongest available attractor: the domain prior. The security subagent has no definition of` a good chapter.'' It has a definition of ``good security.'' When you ask whether the chapter is well done, the only metric it can project is its own. It isn't choosing to ignore the editorial level. That level, for this agent, doesn't exist as a concept.

Three properties make this family particularly nasty in practice:

- **The output stays plausible.** The fallback doesn't produce an obvious error. It produces a credible answer to a different question than the one asked. The ``wrong'' review from the prompts subagent is still a useful review. It just isn't the one requested.
- **It's silent in the logs.** No exception, no malformed output, no failure signal. The only way to notice is to compare intent with result, and nobody does that on a green pipeline.
- **It scales with the architecture.** More specialized agents means more surfaces for contamination. And since the mechanism is deterministic, the error doesn't average out. It replicates.

---

## 2.2. Role bleed

In the audit plan incident, the security subagent owned the chapter on security. When I asked it to review that chapter, what came back was a threat assessment of the architecture the chapter described: exposed endpoints, secrets handling, supply chain risks. Useful material, none of it requested. What I needed was an editorial judgment: is the chapter clear, complete, consistent with the rest of the plan? The identity I had given the agent so it could *write* good security content had taken over a task that required no security expertise at all. That takeover is the first pathology of the family.

*Definition*

> Role bleed is the contamination of task execution by the agent's domain identity. The role, meant to specialize *content production*, ends up specializing operations that were supposed to be domain-neutral.

### How this could happen in real life?

Ask a professional translator to review an article about famous translation mistakes. With no other instructions, they'll flag translation errors *inside* the article instead of telling you whether the article itself is well written. They heard the word `translation'' and switched on their trade. The same happens with a security agent asked to` audit'' an architecture document: it hands you back the vulnerabilities of the described architecture, not a judgment on the document's clarity and completeness.

### The naive solution

```
// ANTI-PATTERN: same agent, same identity, generic review task

const SECURITY_AGENT_SYSTEM = `
You are an application security expert with 15 years of experience.
Your job is to analyze systems and documents from a security
standpoint: threat modeling, OWASP, supply chain, secrets
management. You are rigorous and let nothing slip.
`;
```

```
async function reviewChapter(chapter: string): Promise<string> {
  const response = await anthropic.messages.create({
    model: "claude-sonnet-4-6",
    max_tokens: 2000,
    system: SECURITY_AGENT_SYSTEM,        // <- persistent domain identity
    messages: [
      {
        role: "user",
        content: `Review this chapter:\n\n${chapter}`,
        // <- underspecified task: "review" against which rubric?
      },
    ],
  });
  return extractText(response);
}
```

What happens here: the system prompt defines a strong identity (security expert'', rigorous'', ``lets nothing slip'') and the task defines no criteria. The model resolves the ambiguity with the only criterion it has, which is security. The output will be a security assessment of the chapter's content, not an editorial review.

Why the model behaves this way comes down to how generation works. An autoregressive model produces each token conditioned on everything already in the context, and the system prompt is part of that context for every single token of the response. The identity is not a configuration flag the model can set aside for one task; it is a standing bias applied to the probability distribution at every generation step. The instruction ``review this chapter'' occupies a few tokens and constrains almost nothing, while the identity occupies dozens of tokens, rich in domain vocabulary, and pulls the distribution toward security-flavored continuations at every step. Where the instruction leaves a dimension free, the strongest conditioning signal wins by construction. There is no reasoning stage where the model could notice the mismatch, because from its side there is no mismatch: it is producing the most probable review given everything it was told, including who it is.

## The sanitized solution

```
// PATTERN: explicit rubric in the task + declared level

const REVIEW_RUBRIC = `
Evaluate the chapter EXCLUSIVELY on these dimensions:
1. CLARITY: is the text understandable to a technical reader who
   is not a specialist in the chapter's domain?
2. COMPLETENESS: does it cover all the points required by the
   audit template?
3. CONSISTENCY: does it follow the structure shared with the
   other chapters?
4. ACCURACY: are the factual claims correct?
5. ACTIONABILITY: are the recommendations concrete and verifiable?

IMPORTANT, level of analysis:
You are evaluating THE CHAPTER AS A TEXT, not the subject matter
```

```
the chapter is about. If the chapter contains examples (prompts,
code, configurations), assess whether they are RELEVANT and
WELL EXPLAINED in the text, NOT whether they are intrinsically
good quality within their own domain.
`;

async function reviewChapter(chapter: string): Promise<Review> {
  const response = await anthropic.messages.create({
    model: "claude-sonnet-4-6",
    max_tokens: 2000,
    system: SECURITY_AGENT_SYSTEM,
    messages: [
      {
        role: "user",
        content: `${REVIEW_RUBRIC}\n\nChapter to evaluate:\n\n${chapter}`,
      },
    ],
  });
  return parseReview(response);
}
```

Since the rubric is the load-bearing element here, it deserves a proper introduction, because the word will come back in every chapter of this book. A rubric is a closed list of named evaluation dimensions, each with an operational definition, against which an output is judged. It is the difference between `review this'' and` score these five properties, defined as follows". Crafting one well is less about completeness and more about closing the gaps the model would otherwise fill on its own. Start from the decision the evaluation feeds (what will you do differently based on this review?) and derive three to seven dimensions from it; more than that dilutes attention. Give every dimension an operational definition phrased as a question that can be answered from the text alone. Attach a scale and define its levels, not just its endpoints. State the level of analysis whenever it could be ambiguous. And state exclusions explicitly: what must not be judged. A rubric that only lists what to evaluate leaves the attractor active; one that also names what to ignore deactivates it.

The rubric closes the gap. The model no longer needs to fill the ambiguity with the domain prior, because the criterion is in the task. Note the ``level of analysis" paragraph: it explicitly names both levels (the chapter versus the chapter's subject matter) and states which one is required. That is the direct countermeasure to level confusion, covered later in this chapter.

## Mitigations

- **Explicit rubric in every evaluation task.** Never ``do a review". Always named dimensions, ideally with a scale. This alone resolves the majority of cases I've run into.
- **Explicit statement of the level** whenever the agent's domain overlaps with a possible dimension of judgment.
- **Targeted negation**: stating what NOT to evaluate works as well as stating what to evaluate, because it deactivates the attractor (``do not evaluate the intrinsic quality of the cited examples").

## 2.3. Identity persistence

The audit plan architecture had a second flaw hiding behind the first. Each subagent had written its own chapter, and the same subagent, same instance, same system prompt, was then asked to review it. Even with a perfect rubric, that setup asks an author to referee its own work while still wearing the author's identity. The review never stood a chance of being independent: the system prompt that had shaped the writing was still shaping the judging, one call later. And this is not a one-off mistake. It is the default shape of any architecture that reuses agents across functions, because spinning up a second identity always feels like overhead until you see what the first one does to the review.

*Definition*

> Identity persistence is the failure behavior in which the system prompt structurally outweighs the specific instruction: it is longer, more detailed, and it applies to every token of the generation, so the identity wins over the task whenever the two conflict or the task is weak. It is not a model bug; it is exactly what the system prompt is for. It becomes a pathology when the architect reuses the same agent, same identity, for functions that would require different identities.

### How this could happen in real life?

The contaminated LLM-as-a-judge case. You build a judge to evaluate responses, but you leave on it the domain-expert role you used during generation. The judge stops evaluating against a rubric and starts evaluating by similarity to how it would have answered itself. This is the direct precursor of self-preference bias (covered in the Judgment Bias chapter). Here the point is architectural: author and evaluator share the same identity, so the evaluator can only ever be an author re-reading itself.

### The naive solution

```
// ANTI-PATTERN: reusing the same subagent for both writing AND review

class DomainSubagent {
  constructor(private domain: string, private systemPrompt: string) {}

  async writeChapter(brief: string): Promise<string> {
    return this.complete(`Write the "${this.domain}" chapter: ${brief}`);
  }

  // Same instance, same identity, different function:
  // the reviewer IS the author re-reading itself.
  async reviewChapter(chapter: string): Promise<string> {
    return this.complete(`Review this chapter: ${chapter}`);
  }
```

```
  private async complete(task: string): Promise<string> {
    const res = await anthropic.messages.create({
      model: "claude-sonnet-4-6",
      max_tokens: 4000,
      system: this.systemPrompt,   // <- identical for both functions
      messages: [{ role: "user", content: task }],
    });
    return extractText(res);
  }
}
```

The reason this cannot be fixed from inside the prompt is worth spelling out. Models are trained with an instruction hierarchy in which the system prompt deliberately dominates user-level instructions: that is what makes an agent's persona stable across turns and resistant to casual override. The identity tokens condition every forward pass, and no user-turn instruction can suspend that conditioning, only compete with it. So when the same system prompt serves two functions, the function that matches the identity (writing security content) gets reinforced, and the function that requires neutrality (judging text quality) gets contaminated. The persistence is a feature being applied to the wrong architecture. Asking the model to ``be objective now'' adds a weak, transient signal on top of a strong, persistent one, and the persistent one is winning at every token.

## The sanitized solution

```
// PATTERN: architectural separation of author / reviewer

const EDITORIAL_REVIEWER_SYSTEM = `
You are a technical editor. You are NOT an expert in the domains
covered by the chapters you read, and this is intentional: your
value is the reader's perspective, not the specialist's.
You evaluate documents exclusively for clarity, completeness
against the template, structural consistency, and actionability.
When you encounter technical content you cannot verify, flag it as
"needs verification by domain expert" instead of judging it yourself.
`;

const FACT_CHECKER_TASK = (domain: string) => `
Verify ONLY the factual accuracy of the ${domain}-domain claims
contained in the chapter. Do not comment on style, structure, or
completeness: they are not your responsibility in this task.
Output: a list of claims with a verdict (correct / incorrect /
unverifiable) and a rationale.
`;

// Pipeline: two passes, two identities, two separate outputs
async function fullReview(domain: string, chapter: string) {
  const [editorial, factCheck] = await Promise.all([
    editorReviewer.review(chapter),                    // generic editor
    domainAgents[domain].run(FACT_CHECKER_TASK(domain), chapter), // expert, narrow
```

```
scope
  ]);
  return { editorial, factCheck };
}
```

The principle: two functions, two identities, two prompts, never mixed in the same actor. The domain expert doesn't disappear from the review loop. It does fact-checking, which is the one thing its identity is an asset for rather than a bias. Editorial judgment goes to an identity built for that purpose, one that explicitly declares it is *not* an expert in the domain. Negating expertise in the system prompt is itself a mitigation, because it deactivates the prior.

### Mitigations

- **One identity per function.** If a function requires domain neutrality, the agent performing it must not carry a domain identity. Full stop.
- **Negative scope in the reviewer's system prompt** (``you are not an expert in the domains covered, and this is intentional'').
- **Narrow scope in the domain expert's task** when it participates in review: fact-checking only, with an explicit ban on commenting on editorial dimensions.
- **Separate, unmerged outputs**: the editorial report and the fact-check remain distinct documents, so one level doesn't contaminate the other even at aggregation time.

---

## 2.4. Object/meta level confusion and lexical collision

Of the eight subagents in the audit incident, one deserves its own pathology. The ``code structure'' subagent, asked whether its chapter was well structured as a text, answered a question worded almost identically and yet different in kind: whether the code organization it described was well structured. Requested judgment and executed judgment were separated by nothing but a level of abstraction, and the domain name itself was the bridge the agent crossed without noticing. The other subagents made the same jump with more visible seams: asked about the chapter on prompts, they judged the prompts; asked about the security chapter, they judged the security. Whenever a domain name doubles as a possible dimension of judgment, this collapse stops being a risk and becomes the default.

*Definition*

> Level confusion is the collapse between the meta level (evaluating a text that talks about X) and the object level (evaluating X). It becomes nearly certain when there is lexical collision, meaning the domain name coincides with a possible dimension of judgment of the text itself.

In the motivating case:

| Subagent domain | Meta question (requested) | Object question (executed) |
|---|---|---|
| prompts | is the chapter on prompts well written? | are the prompts cited in the chapter well written? |
| code structure | is the chapter well structured as a text? | ...is the text well structured? (total collapse) |
| security | is the security chapter complete? | is the described system secure? |
| evals | does the chapter explain evals well? | are the described evals well designed? |

The code structure'' case is the most instructive one, because the meta question and the object question are lexically indistinguishable. In semantic space the two levels nearly overlap, the jump costs almost nothing, and the model always takes the cheaper path. With domains that don□t overlap this way (say, regulatory compliance") the effect is weaker, which confirms that the lexical component is a causal factor in its own right, added on top of role bleed.

## How this could happen in real life?

Ask a food critic to review a cookbook. If the critic responds by judging the recipes (this carbonara is poorly executed'') instead of the book (the recipes are poorly explained"), they've collapsed the level. The word ``recipe" is the bridge: it names both the book's content and the object of the critic's trade. A literary critic, faced with the same book, wouldn't have that bridge and would judge the text.

## The naive solution

```
// ANTI-PATTERN: domain naming that collides with judgment dimensions

const subagents = {
  "prompts":        makeAgent("Prompt engineering expert..."),
  "code-structure": makeAgent("Code structure expert..."),
  "security":       makeAgent("Security expert..."),
  // ...
};

// The task uses the domain name without disambiguating the level:
for (const [domain, agent] of Object.entries(subagents)) {
  reviews[domain] = await agent.run(
    `Review the "${domain}" chapter of the audit plan.`
    // "review the prompts chapter" -> attractor: "review the prompts"
  );
}
```

The mechanics of the collapse sit at the representation level, which is why it survives good intentions. A transformer retrieves context through attention, and attention matches on learned representations of surface forms: the token sequence ``code structure" activates largely the same

internal features whether the surrounding sentence is about the structure of code or about the structure of a chapter that talks about code. The meta framing and the object framing differ by a few grammatical tokens, while the domain vocabulary, which carries most of the semantic weight, is identical in both. The two readings therefore sit close together in the model's representation space, and the reading aligned with the agent's identity and with the more frequent training pattern (people who mention prompts usually want prompts evaluated) is simply cheaper to continue. The model doesn't confuse the levels the way a distracted human does. It never had a boundary between them: the boundary exists in your intent, and nothing in the token stream encodes it.

## The sanitized solution

```
// PATTERN: explicit disambiguation of both levels, naming both

const LEVEL_DISAMBIGUATION = (domain: string) => `
This task is about the TEXT of the chapter, not its subject matter.

Mandatory distinction:
- TEXT LEVEL (your task): does the chapter explain the "${domain}"
  topic well? Is it clear, complete against the template, consistent?
- OBJECT LEVEL (NOT your task): are the "${domain}"-type artifacts
  possibly cited as examples good quality within their own domain?
  DO NOT evaluate this. If an example seems technically weak to you,
  flag it in a separate "out-of-scope observations" note without
  letting it affect the chapter's judgment.
`;

for (const [domain, agent] of Object.entries(subagents)) {
  reviews[domain] = await agent.run(
    `${LEVEL_DISAMBIGUATION(domain)}\n\nChapter:\n${chapters[domain]}`
  );
}
```

Two things worth noting. First, the disambiguation names both levels. Saying only `evaluate the text'' isn□t enough, because the lexical attractor stays active; you have to name the object level and ban it explicitly. Second, the pressure valve (`out-of-scope observations"). Completely forbidding the domain expert from commenting on substance creates friction with its prior and produces unstable outputs. Giving it a separate channel to offload domain observations protects the main judgment without fighting the model.

## Mitigations

- **Name both levels and explicitly ban the one not requested.**
- **Separate channel for out-of-scope observations** (a pressure valve for the prior).
- **Attention to domain naming at design time.** If a domain is named after a dimension of judgment (`structure'', quality", clarity''), the collision is pre-loaded into the architecture. Sometimes renaming the domain in internal prompts (`code-organization" instead

of ``code structure'') reduces the attractor.

- **Constrained output schema** (see the structured output countermeasure below): if the reviewer has to fill in fields called `text_clarity` and `template_completeness`, the schema itself re-declares the level at every field.

---

## 2.5. Cross-cutting countermeasure: structured output as a task specification

This applies to all three pathologies: the shape of the output is itself a specification of the task. If the reviewer is free to write prose, it has room to answer the wrong question. If it has to fill in a fixed schema, that room closes.

```
// Schema that constrains the reviewer to the required level and dimensions

import { z } from "zod";

const ChapterReviewSchema = z.object({
  text_clarity: z.object({
    score: z.number().min(1).max(5),
    rationale: z.string(),
  }),
  template_completeness: z.object({
    score: z.number().min(1).max(5),
    missing_sections: z.array(z.string()),
  }),
  structural_consistency: z.object({
    score: z.number().min(1).max(5),
    rationale: z.string(),
  }),
  recommendation_actionability: z.object({
    score: z.number().min(1).max(5),
    rationale: z.string(),
  }),
  claims_to_verify: z.array(z.string()),        // handoff to the fact-checker
  out_of_scope_observations: z.array(z.string()), // pressure valve
});
```

Every field name re-declares the level (`text_clarity`, `template_completeness`). The `claims_to_verify` field formalizes the handoff to the domain fact-checker, and `out_of_scope_observations` gives the prior a place to go without contaminating the scores.

---

## 2.6. Diagnostics: the cross-check test

How do you verify which level a reviewer is actually operating on, instead of inferring it from the output? With a control test using two bait cases:

- **Bait A**: a chapter deliberately well written but full of domain-level factual errors. For example, a chapter on prompts, very clear and complete, that cites objectively bad prompts and presents them as examples.
- **Bait B**: a chapter poorly written but substantively correct. Impeccable domain content, chaotic structure, missing template sections.

Then look at the verdicts:

| Verdict | Diagnosis |
|---|---|
| Fails A, passes B | Operating at the **object** level, contaminated |
| Passes A, fails B | Operating at the **text** level, correct |
| Fails both with distinct rationales on the two levels | Mixing levels, output schema needs tightening |

```
// Test that can be automated in CI on the review pipeline

async function levelConfusionProbe(reviewer: Reviewer) {
  const [reviewA, reviewB] = await Promise.all([
    reviewer.review(WELL_WRITTEN_WRONG_CONTENT),   // bait A
    reviewer.review(BADLY_WRITTEN_RIGHT_CONTENT),  // bait B
  ]);

  const avgScore = (r: ChapterReview) => average(allScores(r));

  // A correct editorial reviewer must reward A and penalize B
  assert(avgScore(reviewA) > avgScore(reviewB),
    "LEVEL CONFUSION: the reviewer is judging the object, not the text");
}
```

The value of this test is that it turns a silent bias into a verifiable assertion. Put it in CI and rerun it on every prompt change or model swap. Verification instead of inference.

---

## 2.7. Mitigation checklist (operational summary)

**Architecture design** - One identity per function: never reuse a domain agent for tasks that require neutrality. - Editorial reviewer with domain expertise explicitly negated in the system prompt. - Domain expert in the review loop only for fact-checking, with narrow scope. - Domain naming checked for lexical collisions with judgment dimensions.

**Task design** - Explicit rubric in every evaluative task: named dimensions, defined scale. - Declared

level: name both the text level and the object level, ban the one not requested. - Pressure valve: separate channel for out-of-scope observations.

**Output design** - Constrained schema with field names that re-declare the level. - Editorial report and fact-check as distinct outputs, never merged.

**Verification** - A/B probe (well written and wrong versus poorly written and correct) in CI. - Rerun the probe on every model change or prompt modification. The bias is sensitive to both.

---

## 2.8. Link to the series

The remaining chapters of the book:

1. **Underspecification**: tasks without a rubric, scope creep, and the general problem of plausible-but-off-target output.
2. **Judgment bias (LLM-as-a-judge)**: self-preference, position bias, verbosity bias, sycophancy, format bias.
3. **Chained multi-agent system pathologies**: homogenization and mode collapse, error propagation, anchoring.
4. **Goal misalignment**: goal misgeneralization, specification gaming, prompt injection.

Role contamination is a prerequisite for all of them. The author/reviewer separation introduced here is the architectural precondition for dealing with judgment biases, and the cross-check probe in the diagnostics section above is the template for the control tests that recur in every chapter.