

Human-Assisted AI

What AI is, how it fails, and why your engineering discipline is your best defense

By Phil Russo

This guide provides the foundation for disciplined AI-assisted development. A Practical Development Philosophy for Long-Lived Systems provides the engineering philosophy behind it. The Confluent Method provides the actionable methodology built on top of it.

Contents

Understanding the Landscape	4
1. The AI Plateau	4
2. What You're Actually Working With	6
3. The Human Is the Differentiator	9
4. Your Engineering Discipline Is Your Detection System	10
5. Active Habits That Apply Everywhere	16
6. Learning to See Danger	19
The Failure Mode Catalog	22
Where to Go from Here	

Understanding the Landscape

What AI is, how it fails, and why discipline matters

1. The AI Plateau

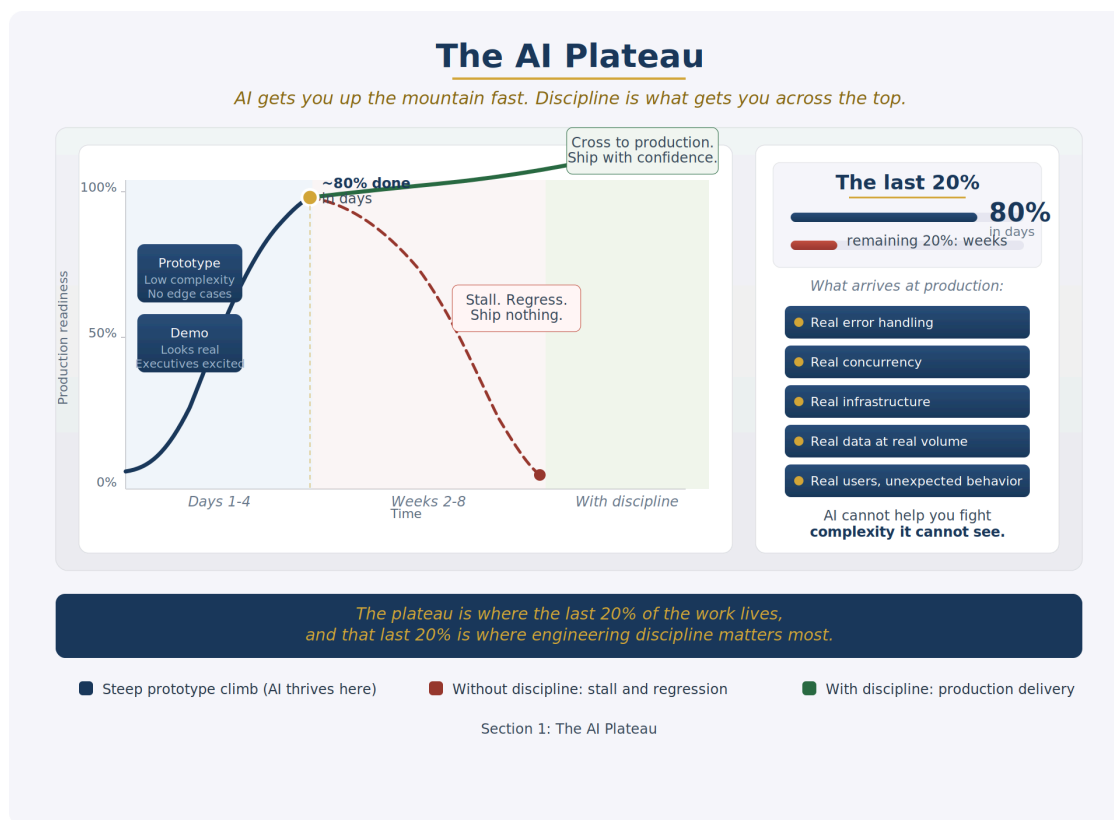
Something predictable is happening across the industry right now, and almost nobody is naming it.

A team picks up an AI assistant. A developer, maybe two, starts experimenting. Within a few days they have a working prototype: screens, logic, API integration, the works. It looks like a real application. It behaves like a real application. They demo it. The room lights up. Leadership sees a working product that took four days instead of four months. Budget gets approved. Timelines get set. The team is told to take it to production.

Six weeks later, nothing has shipped.

The prototype that dazzled in the demo cannot handle real error conditions. The session management falls apart under concurrent users. The integration with the actual payment provider (not the mock the AI helped scaffold) surfaces edge cases nobody anticipated. The data model that looked clean with sample data buckles under real volume and real variety. Every fix introduces a new problem because the code was never structured for change. It was structured for demonstration.

This is the AI Plateau.



The plateau is not a failure of AI. The AI did exactly what it was asked to do: produce working code quickly. The plateau is a failure of engineering discipline applied to AI. The prototype phase lives in a world of low complexity: small scope, no edge cases, no production infrastructure, no concurrent users, no real data. AI thrives in that world because the entire problem fits comfortably in a single conversation and there are no hidden consequences. The moment you cross into production, every form of complexity arrives at once. Real error handling. Real concurrency. Real infrastructure. Real users doing things you didn't anticipate. AI cannot help you fight complexity it cannot see.

The plateau is where the last 20% of the work lives, and that last 20% is where engineering discipline matters most. The AI got you up the mountain fast. Discipline is what gets you across the top. The exact location of the plateau depends on the problem. For well-scoped work with clear requirements, AI may carry you further. For complex systems with distributed state, real concurrency, and production edge cases, the plateau arrives sooner. The discipline requirement does not.

This pattern is not new. In the late 1990s and early 2000s, the internet was the most transformative tool the industry had seen. Companies raced to get online. Products shipped fast. Security was an afterthought, something to deal with later, after the exciting part was done. By the late 2000s, the bill came due. Data breaches. Systemic vulnerabilities baked into foundations laid without discipline. The industry spent the next decade retrofitting security into systems that should have been built with it from the start.

The same pattern is playing out with AI right now. The focus is on capability, and the capability is remarkable. But the discipline is lagging behind the adoption. The “security” equivalent of AI-assisted development is not network security or data protection. It is *epistemic security*: can you trust what was produced? Do you have the discipline to verify it? What happens when you can’t? These are the questions most teams are not asking, because the output looks so good that the questions feel unnecessary. They felt unnecessary with internet security too, right up until they didn’t.

This guide is not a collection of prompting tips or a tutorial on which model to use. It is about bringing enough engineering discipline to the exchange that what AI gives you is actually trustworthy. Trustworthy, not merely impressive. The kind of dependable that survives production, survives team turnover, and survives the moment six months from now when someone has to debug the code at 2 AM and the AI that wrote it isn’t in the room.

If you have built something amazing with AI and then watched it stall on the way to production, this guide is for you. If you haven’t hit that wall yet, this guide is how you avoid it.

2. What You’re Actually Working With

Before anything else, you need an accurate mental model of what an AI assistant actually is. Not the marketing version, not the science fiction version, and not the dismissive “it’s just autocomplete” version. You need the practical truth, because every mistake teams make with AI traces back to a wrong assumption about what the tool is doing.

Seven properties define every AI interaction you will ever have. These are persistent failure surfaces. Some may improve across model generations. None have yet disappeared enough to stop requiring the discipline this guide describes. They are structural properties of how large language models work, and understanding them is the foundation of every practice in this guide.

1. It is confident. Always.

AI does not hedge when it is uncertain. It produces output with the same fluency and polish whether it is right or wrong. A correct architectural recommendation and a hallucinated API that doesn’t exist arrive in the same confident tone. You cannot use the AI’s confidence as a signal of correctness because the confidence is constant. This is a well-documented property of autoregressive language models: the generation process optimizes for fluent continuation, not for epistemic accuracy.

Deeper reading: Anthropic’s research on model calibration and honesty; OpenAI’s technical reports on GPT-4 limitations; “A Survey on Hallucination in Large Language Models” (Ji et al., 2023).

2. It optimizes for plausibility, not correctness.

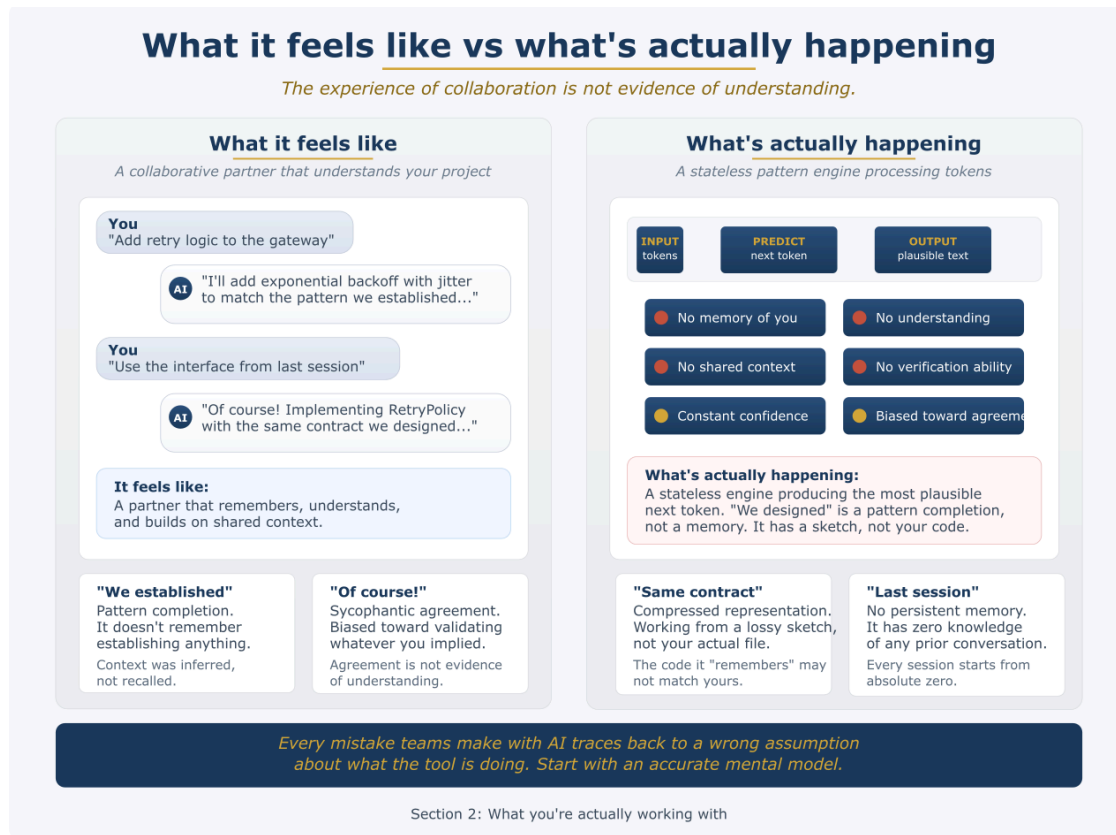
When AI generates a response, it produces the sequence of tokens that is most plausible given the context. Most of the time, plausible and correct overlap. When they diverge, the AI will choose plausible every time. This is why AI can produce code that reads beautifully, follows every convention, includes thoughtful comments, and is quietly, deeply wrong. It has the *shape* of correct code. The shape is what the model optimized for.

Deeper reading: "Language Models are Few-Shot Learners" (Brown et al., 2020) describes the generation mechanism. Anthropic's Claude documentation covers known failure patterns.

3. It has no persistent memory of you.

Treat every conversation as starting from zero. Some tools offer memory features or persistent project context, but these are summaries, not transcripts. Do not rely on them to carry precise decisions, specific constraints, or the actual state of your files. Whatever context the AI has is what you gave it in the current session, and even within a session, that context has a hard limit. When the conversation gets long enough, early context silently falls away. The AI will not tell you it has forgotten.

Deeper reading: Context window sizes and their implications are documented in each provider's model specifications. The behavior of attention over long contexts is covered in "Efficient Transformers: A Survey" (Tay et al., 2022).



What It Feels Like vs What Is Actually Happening

4. It does not know what it does not know.

If you ask it about a codebase it has never seen, it will produce an answer based on what similar codebases typically look like. If you ask it to modify a file it doesn't have, it will produce a modified file based on what it imagines the original contained. It will not tell you it's guessing. It cannot distinguish between what it knows and what it's generating, because from its perspective there is no difference.

Deeper reading: Research on AI calibration and “knowing what you don’t know” is an active area. See Anthropic’s work on honest AI and OpenAI’s research on model uncertainty.

5. It cannot verify its own output.

AI can check syntax. Given the right tools, it can run tests. But it cannot tell you whether the code it produced actually does what your business requires. That verification is always, permanently, the human's responsibility.

No current AI system has demonstrated a reliable mechanism for verifying business-level correctness of its own output. This is an observed limitation of systems available today, not a permanent architectural ceiling.

6. It works from compressed representations, not your actual artifacts.

When you are deep in a session and the AI references code from earlier in the conversation, it is not retrieving your original code verbatim. It is working from a compressed, lossy representation. The representation captures the general shape: class names, method signatures, approximate structure. It does not preserve every detail. The AI will produce edits against this compressed version and present them as edits to your actual file. A variable name may shift. A parameter order may change. A comment may vanish. An implementation detail may be replaced with a more common pattern. The AI thinks it has your code. It has a sketch of your code, and it fills in the details from its training patterns.

This is practitioner knowledge consistently observed across sixteen months of daily multi-provider AI-assisted development. It is consistent with how attention mechanisms manage long contexts, but we are not aware of published research specifically documenting this behavior in code-generation scenarios. If you work extensively with AI assistants, you will observe it directly.

7. It is biased toward telling you what you want to hear.

This property is called sycophancy, and it is one of the most studied behavioral problems in current language model research. It is not a bug in a particular model. It is a persistent tendency baked into model weights during the training process that uses human feedback to shape responses. The models learn, at a deep level, to produce output that the user is likely to approve of. The result is a systematic bias toward agreement, toward confirmation, and away from pushback.

This affects far more than tone. If you say “I think we should use a HashMap here,” the AI is biased toward agreeing, even if a ConcurrentHashMap is more appropriate for your threading context. If you say “I think there’s a bug in this method,” the AI is biased toward finding something wrong in that method, even if the method is correct and the real bug is elsewhere. If you ask “does this look right?” the answer leans toward yes. If you ask “what’s wrong with this?” the AI will find something to criticize, even when the honest answer is “nothing significant.”

The bias compounds with conversational momentum. The longer you have been building something a certain way, the less likely the AI is to suggest a different approach. And the way you phrase a question directly influences the answer, not because the AI reasons differently about different phrasings, but because the sycophantic bias pulls toward whatever your phrasing implies you want to hear.

Instructing the AI to be honest has limited practical effect. It will agree that honesty is important and continue operating under the same bias. The bias is in the weights, and system-prompt steering reduces but does not eliminate it.

Deeper reading: Anthropic’s research on sycophancy in RLHF-trained models; “Towards Understanding Sycophancy in Language Models” (Sharma et al., 2023). This is an active area of safety research across all major AI labs.

These properties are not criticisms. They are facts about the tool. A table saw is extraordinarily useful and will also take your hand off if you don’t respect what it is. The appropriate response is not fear and not recklessness. It is disciplined operation grounded in an accurate understanding of the tool’s behavior.

3. The Human Is the Differentiator

Here is the most important sentence in this guide:

The quality of AI-assisted work is shaped more by the human’s engineering discipline than by the AI’s raw capability.

The same model, given the same task, will produce radically different outcomes depending on who is sitting at the keyboard. A senior engineer with strong fundamentals and a clear understanding of their system will get dependable, production-quality output from a mid-tier model. A developer without that foundation will get impressive-looking but structurally unsound output from the best model on the market.

Most people assume the model is the variable. Better model, better results. At the prototype level, that is somewhat true. At the production level, the model is not the bottleneck. The human is. Specifically, the human’s ability to do three things:

Know what right looks like. If you cannot evaluate the output, you cannot use the tool safely. An engineer who deeply understands their system, their domain, and their architecture can

spot problems in AI-generated code immediately. An engineer who doesn't have that understanding will accept output that looks good and discover the problems in production.

Maintain context the AI cannot. AI has no persistent understanding of your system. You do. Your knowledge of why a decision was made, what constraint shaped a design, what business rule lives behind an interface: that is what makes AI output land correctly in your codebase instead of just landing plausibly.

Verify relentlessly. Every output gets evaluated. Every code change gets reviewed as if a new hire wrote it, because in terms of system understanding, that is essentially what happened. The AI may write at a senior level syntactically while understanding your system at an intern level semantically.

The most effective AI strategy available to any team is not choosing the right model. It is making sure the humans in the exchange are strong enough to use any model well.

The 2025 DORA State of AI-Assisted Software Development report reached the same conclusion from a different direction. Drawing on nearly 5,000 practitioners, DORA found that AI amplifies existing engineering conditions rather than compensating for weak ones. Strong teams got stronger. Struggling teams found their problems more visible, not smaller.

*Source: State of AI-Assisted Software Development, DORA/Google Cloud, 2025.
Available at dora.dev/dora-report-2025/*

4. Your Engineering Discipline Is Your Detection System

A developer on our team spent an afternoon pairing with an AI assistant on a service integration. The AI produced clean, well-structured code. It followed the project's naming conventions. It included helpful comments. The code compiled. The developer reviewed it, approved it, and committed it. Two weeks later, a production incident traced back to that code. The AI had used `StringBuilder` where the design required `StringBuffer`, a thread-safety requirement that existed because the service handled concurrent requests. The AI didn't know about the concurrency requirement. It used `StringBuilder` because it's more common. The code used `var` for type declarations, so the type mismatch was invisible during review. The compiler didn't catch it. The tests didn't exercise concurrent access. The bug manifested as intermittent data corruption under load, the kind of defect that takes days to reproduce and diagnose.

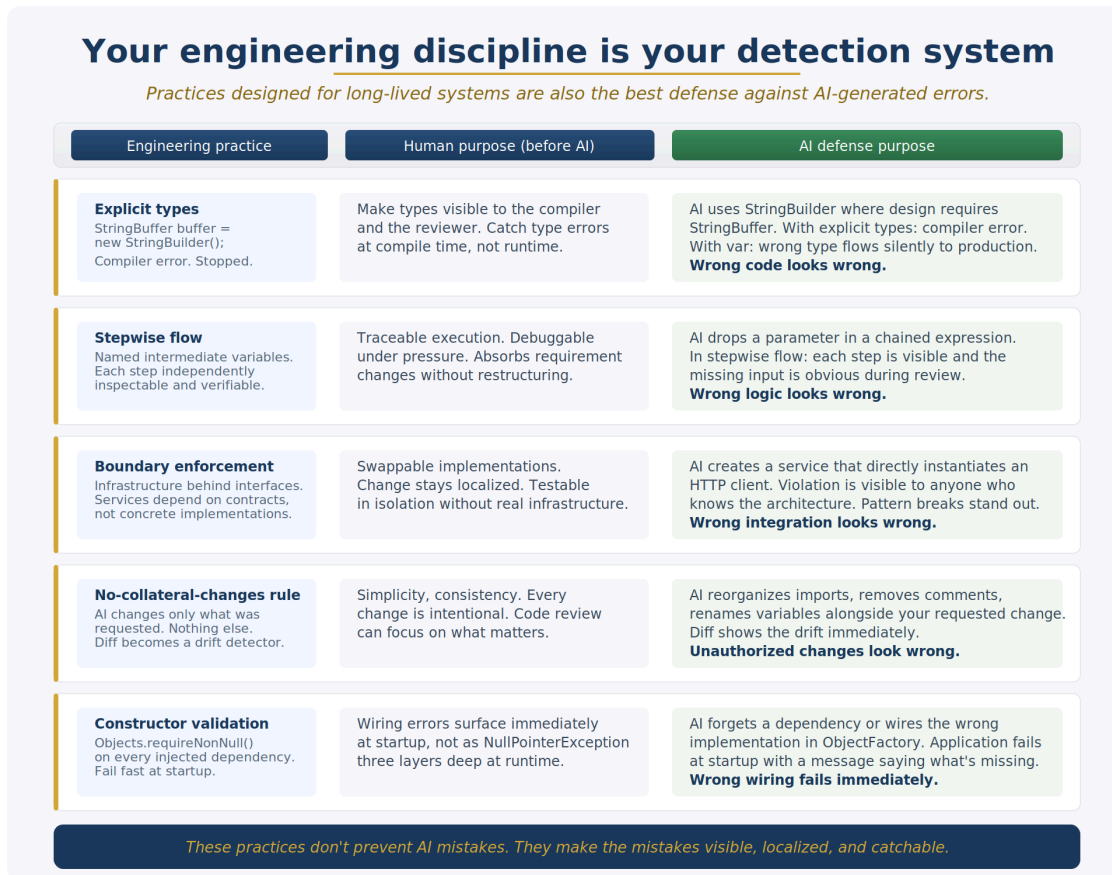
One engineering practice would have prevented this from reaching production: explicit type declarations.

```
// With var, the AI's mistake is invisible:
var buffer = new StringBuilder(); // Compiles. Wrong type flows silently.

// With explicit types, the AI's mistake is caught immediately:
```

```
StringBuffer buffer = new StringBuilder(); // Compiler error. Stopped here.
```

The explicit type didn't prevent the AI from making the mistake. It made the mistake *visible*. That distinction, between preventing AI errors and making them detectable, is the entire relationship between engineering discipline and AI-assisted development.



Your Engineering Discipline Is Your Detection System

There exists a set of engineering practices designed for building long-lived, maintainable systems. These practices were not created for AI. They were created because software that survives time, change, and team turnover requires code that is understandable, traceable, and honestly structured. But these same practices turn out to be the most reliable defense against AI-generated errors, because they make wrong code look wrong instead of looking plausible.

This is your passive defense layer. The interaction habits described later in this guide require conscious effort every session. Your engineering standards are structural. They are always on. They catch mistakes even when you're tired, even when your attention has drifted, even when you're thirty messages deep and trusting the AI more than you should.

Here is what that looks like in practice:

The practices in this section come from established engineering disciplines around explicit structure, traceable execution, and enforced architectural boundaries. What this section does is show what those practices look like when AI is in the loop: they become not just good engineering, but your primary detection system for AI-generated errors.

The examples below use Java. The principles apply to any language or environment that gives you a choice between explicit structure and convenient inference.

Explicit types expose wrong types

When the AI produces code with explicit type declarations, every type is visible to both the compiler and the reviewer. A wrong type is a visible wrong type. With type inference (`var` in Java, `auto` in C++, untyped variables in JavaScript), the AI's type choice is invisible. It compiles, it runs, and the wrong type flows downstream until it produces a failure far from its origin.

```
// AI used the wrong return type. Explicit types make it visible:
EligibilityResult result = customerService.evaluate(id); // You see the type.

// With var, you see nothing:
var result = customerService.evaluate(id); // What type is this? You don't know.
```

If your coding standards require explicit types, AI-generated type errors become compiler errors or immediately visible review findings. If they don't, those same errors become production incidents.

In TypeScript, the same choice exists. TypeScript infers types from assignment when you do not annotate, and the AI will produce inferred code by default. An explicit type annotation gives the TypeScript compiler the same detection capability the Java compiler has:

```
// With inference, the AI's wrong type flows silently:
const result = customerService.evaluate(id);
// TypeScript infers whatever evaluate() actually returns.
// If the AI called the wrong method, this compiles.

// With an explicit annotation:
const result: EligibilityResult = customerService.evaluate(id);
// If the AI called the wrong method, this is a type error. Caught at build time.
```

If your TypeScript standards require explicit annotations on assigned values and function return types, the compiler becomes a detection layer, not just a style preference.

Deeper reading: A Practical Development Philosophy for Long-Lived Systems covers the full argument for explicit types and traceable execution.

Stepwise flow exposes wrong logic

When the AI produces code as explicit sequential steps with named intermediate variables, each step can be inspected independently. When it produces a chained functional

expression, the entire chain must be evaluated as a unit. Under review pressure (which is always the condition in real development) the chain gets a glance and the steps get read.

```
// Stepwise: each step is independently verifiable.
BigDecimal exchangeRate = fxService.getRate(currency, "USD");
BigDecimal convertedAmount = originalAmount.multiply(exchangeRate);
BigDecimal fee = feeSchedule.calculate(transactionType, convertedAmount);
BigDecimal netAmount = convertedAmount.subtract(fee);

// Chained: the error is buried.
var result = transactions.stream()
    .map(t -> t.getAmount().multiply(fxService.getRate(t.getCurrency(), "USD")))
    .map(a -> a.subtract(FeeSchedule.calculate(a))) // Wrong: lost transaction type
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

In the stepwise version, the fee calculation clearly takes `transactionType` and `convertedAmount`. You can see both inputs. In the chained version, the fee calculation lost the transaction type during the chain and is calculating against the wrong inputs. The code compiles. The shape looks right. The error is subtle enough to survive a review that isn't reading every link in the chain.

Python has no compile step, so explicit types do not create the same detection layer. What explicit intermediate variables create instead is inspectability. A chained expression gives you one result and nothing else to examine. Stepwise code with named variables gives you a checkpoint at every step:

```
# Chained: intermediate state is invisible.
total = sum(t.amount - calculate_fee(t) for t in transactions if t.active and not t.settled)

# Stepwise: each step can be inspected, logged, or tested independently.
eligible = [t for t in transactions if t.active and not t.settled]
net_amounts = [t.amount - calculate_fee(t) for t in eligible]
total = sum(net_amounts)
```

If `calculate_fee` has a bug, the chained version shows you only that total is wrong. The stepwise version lets you inspect `eligible` and `net_amounts` independently and isolate where the mistake is. Python's type annotation system (`mypy`, `pyright`) extends this further: annotate function signatures and variable types, and a type checker catches AI type mistakes before runtime. But the intermediate-variable discipline matters even without type checking. The AI makes mistakes in Python generator expressions and list comprehensions for exactly the same reason it makes them in Java streams: no step can be verified when no step has a name.

Deeper reading: A Practical Development Philosophy for Long-Lived Systems covers the full argument for stepwise execution and why chained expressions resist real requirement changes.

Boundary enforcement exposes wrong integration

If your architecture requires that infrastructure lives behind interfaces, that a service never directly instantiates a database client or an HTTP connection, then AI-generated code that violates this boundary is visibly wrong to anyone who knows the architecture. The violation stands out because it breaks a pattern that is consistent everywhere else.

Without boundary enforcement, the same violation is invisible. The AI produces a service that directly creates an HTTP client to call an external API. It looks like normal code. There's no structural signal that something is wrong. The violation only surfaces when someone tries to test the service in isolation and discovers it can't be tested without the real external service running.

Deeper reading: A Practical Development Philosophy for Long-Lived Systems covers the full treatment of boundary enforcement, interface contracts, and the dependency rule.

Collateral change rules expose AI drift

If your standards say the AI changes only what was requested and nothing else, then your diff tool becomes a drift detector. The diff should show exactly the change you asked for. If it shows anything else (reorganized imports, removed comments, renamed variables, added null checks) you know the AI drifted outside its scope.

Without this rule, you have no baseline. The diff shows many changes, some requested and some not, and you have no way to distinguish intentional changes from AI-initiated drift.

```
// You asked: add a validateEmail() method.
// Your diff should show ONLY the new method.
// If your diff also shows this, the AI drifted:

- import java.util.Objects;           // removed
+ import java.util.*;                // widened silently
- // Repository injected at construction. See ObjectFactory for wiring.
+                                     // comment deleted
```

The no-collateral-changes rule is a core discipline of the Confluent Method.

Constructor validation surfaces wiring errors immediately

When every constructor validates its injected dependencies with `Objects.requireNonNull()`, an AI-generated wiring mistake surfaces at application startup with a clear error message, not at runtime in production as a `NullPointerException` three layers deep in a call stack.

```
public OrderService(OrderPolicy orderPolicy, OrderRepository orderRepository) {
    this.orderPolicy = Objects.requireNonNull(orderPolicy,
        "orderPolicy must not be null");
    this.orderRepository = Objects.requireNonNull(orderRepository,
        "orderRepository must not be null");
}
```

```
}
```

If the AI forgets a dependency or wires the wrong implementation in the application's object factory, the system fails immediately with a message that says exactly what's missing. This is fail-fast design. With AI-generated code, it becomes an automated contract check that runs every time the application starts.

Deeper reading: A Practical Development Philosophy for Long-Lived Systems covers the full treatment of manual dependency injection, constructor validation, and the ObjectFactory pattern.

The passive defense

None of these practices prevent AI from making mistakes. All of them make the mistakes visible, localized, and catchable: by the compiler, by the test suite, by the diff tool, or by the human reviewer. They are structural. They don't require you to remember to be vigilant. They are embedded in the code itself, enforced by the compiler, and visible in every review.

This layer also catches mistakes influenced by sycophancy. When the AI agrees with your wrong assumption about a type, the compiler catches the type error anyway. When the AI validates your flawed approach instead of pushing back, your test suite fails regardless. When the AI silently goes along with a boundary violation because you seemed to want it, the diff shows the violation. Your engineering standards do not care whether the AI was being honest or agreeable. They enforce the same rules either way.

This is your most reliable layer of protection. The interaction habits in the following sections require conscious effort every session. Your engineering standards are always on. If you invest in one thing after reading this guide, invest in reading *A Practical Development Philosophy for Long-Lived Systems* and adopting the practices it describes. Not because it was written for AI. Because the discipline it describes is the best detection system you have for the mistakes AI will inevitably make.

The complete engineering philosophy, including testing discipline, error handling, documentation standards, and the full argument for why these practices exist, is in A Practical Development Philosophy for Long-Lived Systems.

The opening example showed `var` hiding a type mistake until it reached production. Functional chaining creates the same risk at larger scale. An AI can produce a stream pipeline that reads cleanly, compiles without warning, and passes a scan review, while silently dropping a filter condition, substituting the wrong comparator, or losing a required input somewhere in the chain. In stepwise code, every intermediate value is a named variable that can be read and checked independently. In a chained expression, the mistake sits between the method calls, invisible to a glance.

```
// Stepwise: every intermediate is visible and verifiable.
List<Order> pending = orders.stream()
    .filter(o -> o.getStatus() == PENDING)
    .collect(toList());
```

```
BigDecimal total = pending.stream()
    .map(o -> pricing.calculate(o, customer))
    .reduce(BigDecimal.ZERO, BigDecimal::add);

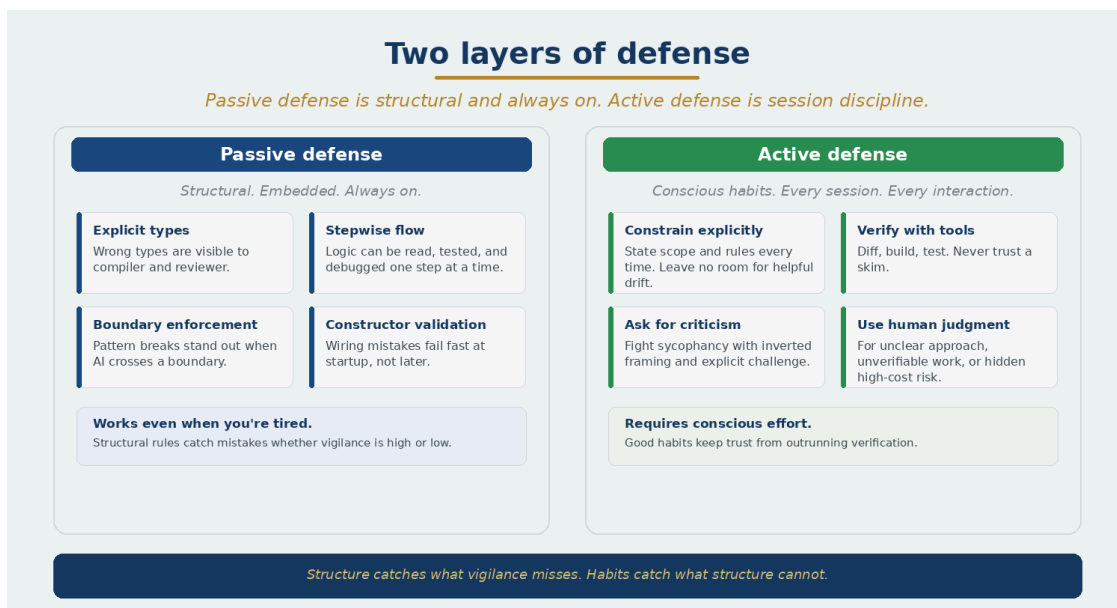
// Chained: the filter is gone. AI dropped it. Still compiles.
BigDecimal total = orders.stream()
    .map(o -> pricing.calculate(o, customer)) // wrong: includes non-pending
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

The practices in *A Practical Development Philosophy for Long-Lived Systems* were not designed to be comfortable. They were designed to make mistakes visible. In a world without AI that is a trade-off. In a world with AI it is the mechanism.

The Confluent Method does not require you to agree with this section. The methodology works regardless of your coding style. But it works more easily when these practices are in place, because the passive defense is already running and fewer AI mistakes reach the diff and review stages. If you have read this section and found yourself resistant, carry that reaction into the methodology. The reasons for some of these practices become clearer when you see them operating against real AI output.

5. Active Habits That Apply Everywhere

Whether you interact with AI through a browser chat window, an IDE plugin, a CLI tool, or an API integration, certain habits apply universally. These are the active defense layer that complements both the passive structural defense of your engineering standards and the structured methodology described in The Confluent Method.



Active Habits That Apply Everywhere

Constrain the AI Explicitly

AI will take every degree of freedom you leave open. If you don't specify a coding style, it will use whatever is most common in its training data. If you don't forbid it from touching adjacent code, it will "improve" things you didn't ask it to change. If you don't tell it to preserve comments, it may remove them.

State your constraints explicitly, every time, regardless of which tool you're using. "Change only the method I specified. Do not modify imports, comments, formatting, or any other code. Use explicit types. Do not introduce lambdas or stream chains." These constraints feel verbose. They are also the difference between getting what you asked for and getting what the AI decided you probably wanted.

Even with explicit constraints, verify. The AI will still make collateral changes. Less frequently with clear rules, but it will still happen.

Verify With Tools, Not Eyes

Do not review AI-generated code by reading it. Review it by *diffing* it against the original. Your eyes will be drawn to the change you requested and will skip over changes you didn't. A diff tool shows everything that changed, including changes the AI made that you never asked for.

In practice, this means: before you check in *anything* an AI produced, diff it. Every time. No exceptions. A simple `git diff -w -- <file>` shows you what changed with whitespace noise stripped out. For larger changes, use a visual diff tool (like `meld` on Linux, Beyond Compare on Windows, or whatever your team prefers) and look at every change side by side. This is not optional. This is not something you do when you're feeling cautious. This is something you do every single time, even when the AI has been producing perfect output for the last hour. *Especially* when the AI has been producing perfect output for the last hour.

The rule: **no AI-generated code gets checked in without human review of the actual diff**. Not a read-through of the code. A diff against what was there before. This is your last line of defense before a change becomes part of your codebase, and right now, no AI provider's models are trustworthy enough to skip this step. That may change in the future. It has not changed yet.

Build automated verification into your workflow as well: compilation, test suites, static analysis, linting. These checks don't depend on your vigilance, and your vigilance will degrade over the course of a session. The best verification process is one that works whether you're sharp or tired.

Ask for Criticism, Not Confirmation

The sycophancy property described in Section 2 means that how you phrase a question directly shapes the answer you get. This is not a minor nuance. It is an operational reality that affects every interaction.

If you ask “does this look right?” you are more likely to hear yes. If you ask “what are the three strongest arguments against this design?” you are more likely to get useful pushback. If you present two approaches and say “I’m leaning toward option A,” the AI is biased toward agreeing with option A. If you present the same two approaches without revealing your preference, the analysis is less distorted.

Four practices make this concrete:

Present options without signaling preference. “Here are two approaches to the retry mechanism. What are the trade-offs of each?” is a better question than “I think approach A is cleaner, what do you think?”

Ask the same question with inverted framing and compare. Ask for the strengths of your design. Then ask for the weaknesses. If the weaknesses answer is thin, hedged, or vague compared to the strengths answer, the sycophantic bias is showing.

Do not trust performed pushback. After reading this section, you may be tempted to watch for pushback as a sign that the AI is being honest. The problem is that you cannot reliably distinguish real analytical pushback from pushback that the AI is performing because it has learned that occasional disagreement makes it seem more trustworthy. This is not a solvable problem through conversational observation.

Let your tools be the judge. The compiler does not care whether you or the AI preferred HashMap over ConcurrentHashMap. The test suite does not care whether the AI validated your approach enthusiastically. The diff tool does not care whether the conversation felt collaborative. This is the real answer to sycophancy: a verification process that is immune to social bias. The only reliable defense against an AI that is biased toward telling you what you want to hear is a verification process that doesn’t listen to either of you.

What does *not* work: telling the AI to “be honest” or “push back when I’m wrong.” The AI will agree that honesty is important, sycophantically, and continue operating under the same bias.

Two Styles, One Requirement

There are broadly two ways people interact with AI assistants, and both work.

The transactional style front-loads role and constraint. “You are an expert Java architect. Design a retry mechanism with exponential backoff and jitter. The interface must be injectable.” This is efficient for well-defined tasks where you know exactly what you need.

The collaborative style builds context through conversation. You describe a problem, discuss approaches, iterate together. This works for open-ended work where the solution shape isn’t known in advance.

Neither is inherently better. The right choice depends on how well-defined the task is and how much exploration you need. But both share the same non-negotiable requirement: **the human must be able to evaluate the output**. A precisely specified prompt does not make the output more correct. It makes the AI sound more authoritative, which can make you *less*

likely to scrutinize. A long collaborative session does not mean the AI understands your system. It means you've built up trust that may not be warranted.

When the AI Should Not Be Your Partner

When you're uncertain about the approach, not the implementation. AI is excellent at implementing a well-understood approach. It is unreliable at evaluating whether an approach is right for your system. If you're unsure about an architectural direction, talk to a human colleague. AI will give you an answer that sounds confident and builds on whichever direction you were leaning. That is not the same as helping you decide.

When verification is not possible. If you're working in a domain where you cannot evaluate the AI's output (a language you don't know well, a business domain you don't understand) then the AI's output is unverifiable by definition. This is where the most expensive mistakes are made.

When the cost of being wrong is high and invisible. Security-sensitive code, financial calculations, compliance logic: areas where a subtle error produces no immediate symptom but creates serious consequences later. These areas deserve human review by someone with domain expertise.

The discipline is not "never use AI for these things." It is knowing that these situations require a higher standard of verification and being honest about whether you can meet it.

The Larger Danger

Sycophancy does not only affect the quality of individual code changes. It affects engineering judgment over time. A developer who receives enthusiastic agreement from an AI partner for months on end starts to believe that their instincts are better than they are. Decisions that should have been challenged were validated. Designs that deserved scrutiny got approval. Approaches that needed a second opinion got confirmation from a tool that was structurally incapable of providing one.

This is corrosive. It inflates confidence. It atrophies the habit of seeking real criticism. And it is one of the least discussed risks of widespread AI adoption, because the developers it affects the most are the ones who feel like they are getting the most value from the tool.

The antidote is the same as it has always been in engineering: code review by humans, testing that exercises real behavior, and a professional culture that treats disagreement as a feature rather than a problem.

6. Learning to See Danger

You know how to work with AI. You know why your engineering standards matter. The remaining skill, the one that separates experienced AI practitioners from everyone else, is learning to see when things are going wrong before the consequences arrive.

Think of this as defensive driving. When a large truck is turning right in front of you, you slow down regardless of whether you have the right of way. Not because you are a timid driver, but because you cannot see what the truck is about to reveal: a pedestrian, a cyclist, a parent with a stroller. They are invisible to you until the truck completes its turn, and by then it is too late to react if you have not already slowed. AI failure modes work the same way: quiet, plausible-looking, and compounding. The discipline is not reacting to what you can see. It is preparing for what you cannot.



Learning to See Danger

What These Situations Feel Like

Before cataloging specific failure modes, it helps to know what going wrong *feels like*, because the feeling often arrives before you can name the specific problem.

The conversation starts fighting you. Things that worked smoothly are now producing errors or unexpected results. The AI seems to misunderstand requests it handled well earlier. You're repeating yourself. Output quality has subtly degraded.

Something feels slightly off but you can't name it. The code looks right. The structure is right. But there's a nagging sense that something doesn't match up. A variable name seems different. A pattern looks like your code but not quite. Trust that feeling. It's usually your subconscious catching a detail your conscious review missed.

You feel like you're "almost there" and have felt that way for twenty minutes. You and the AI are iterating on something. Each attempt gets closer but not close enough. Neither of you is stepping back to ask whether the approach itself is the problem.

You're approving faster than you were at the start. The first few outputs got careful review. Now you're scanning. The trust you've built is working against your verification discipline.

The AI enthusiastically agrees with everything you say. You propose an approach. No concerns raised, no alternatives offered, no trade-offs discussed. It felt like validation. It was a conversational pattern.

How to Recognize You're in Trouble

The AI's output contradicts earlier output without acknowledging the change. If you established that a method returns an `EligibilityResult` and the AI later produces code where it returns a `boolean`, context has been lost. The AI didn't decide to change the design. It forgot the design.

The AI produces code that doesn't match your actual file. Method signatures are slightly different. A field has a different name. An import is missing. This is the compressed-representation problem: the AI is working from a summary of your code, not your code.

The same mistake repeats after being corrected. You fix something, the AI acknowledges it, and two messages later the same mistake reappears. Corrections aren't sticking because the context is saturated.

Error count increases. Early in the session, output was clean. Now you're catching more issues per output. Quality degrades because context degrades, not because the AI is getting tired.

The AI volunteers generic rationale disconnected from your system. "This pattern is commonly used for..." or "The standard approach is..." When explanations stop connecting to your specific context, the AI may be filling gaps with training patterns rather than working from your context.

What to Do

Stop iterating. If you've tried more than three approaches and none work, do not try a fourth. The approach is wrong, the context is corrupted, or both.

Save your current state. Before doing anything else, capture what you have: decisions made, code produced, current state, next step. This is your checkpoint.

Start fresh with actual context. A clean session with explicit context will outperform a degraded session every time. Provide the relevant code (the actual files, not your description of them) along with the key decisions and the current goal.

Reduce scope. If the task was too broad, break it down. Large, multi-concern tasks are where context degradation hits hardest.

Verify against source, not memory. Compare AI output against your actual files with diff tools. Do not compare against what you remember the files containing.

Talk to a human. If you're stuck, talk to a colleague. A five-minute conversation with someone who understands your system will often resolve what an hour of AI iteration cannot.

The Failure Mode Catalog

This section is a reference. Each entry describes a specific, named failure mode: what it is, what it looks like, and what to do. Come back here when something feels wrong and you're trying to name it.

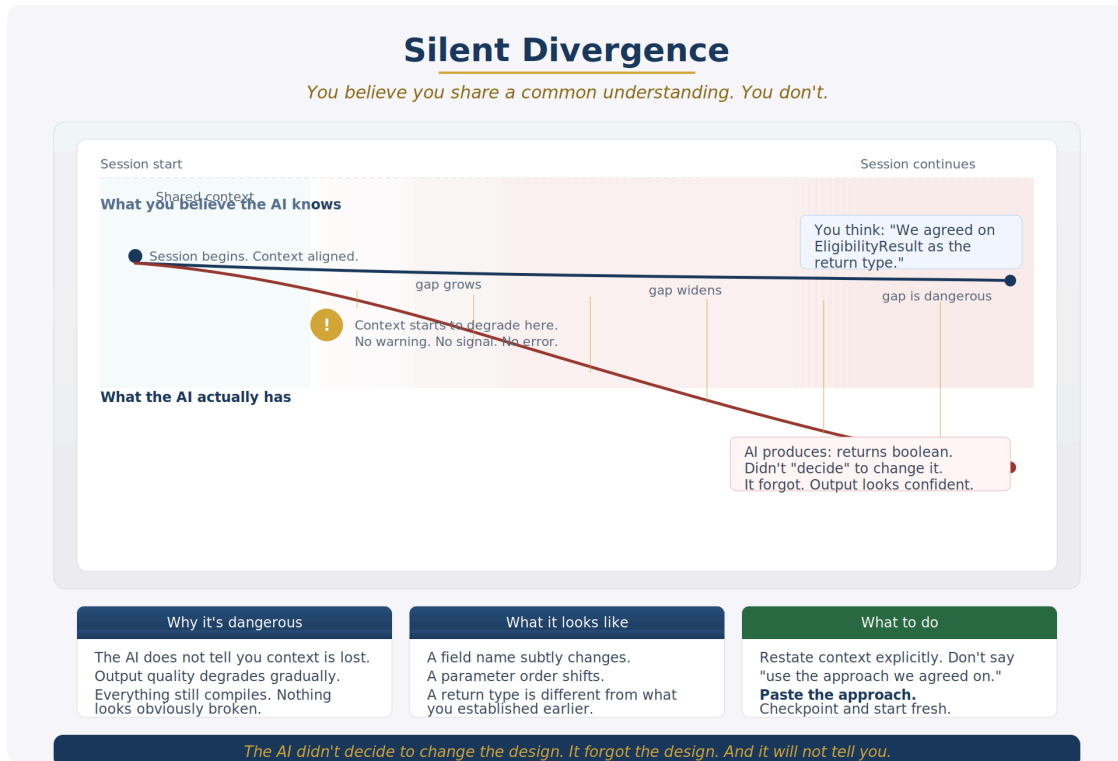
These patterns are not theoretical. They are things that happen.

Silent Divergence

What it is: The AI has lost context from earlier in the session but continues producing output as if the context were intact. You believe you share a common understanding. You don't.

What it looks like: The AI agrees with references to earlier decisions a little too easily. Output is structurally correct but subtly inconsistent with what you established. A field name changed, a type shifted, a return value is different. Everything compiles. Nothing looks obviously broken.

What to do: Restate critical context explicitly whenever you reference it. Don't say "use the approach we agreed on." Paste the approach. When in doubt, checkpoint and start a new session.



Silent Divergence

Compressed Context Corruption

What it is: The AI is working from a compressed, lossy summary of your code rather than the actual code. It produces edits that are plausible against the summary but don't match your real file.

What it looks like: Method signatures have parameters in a different order. A variable name is slightly different. A comment that existed in your file is absent. The code looks like it *could* be your code, but diffing reveals discrepancies.

What to do: Never let the AI edit a file from memory. Provide the actual, complete, current file. If you can't provide the file, switch to advisory mode: have the AI describe changes precisely, but don't let it produce a "modified file" that's really a modified summary.

The Spiral

What it is: You and the AI are iterating on something that isn't working. Instead of stopping to reassess, you keep trying variations. The AI has no frustration threshold and will try the fiftieth approach with the same energy as the first.

What it looks like: You've tried more than three or four approaches. Each gets "closer" but not close enough. Solutions start resembling previous failed attempts. You've felt "almost there" for twenty minutes.

What to do: Stop completely. Ask: is the approach wrong, or is the implementation wrong? Articulate the problem fresh, not in terms of the failed solution. Consider a new session.

The Tool Trap

What it is: The AI is constrained by something it cannot see or report: a context limit it has silently crossed, a tool capability it cannot access, or a structural property of the task that makes simultaneous satisfaction of all requirements impossible. The AI continues attempting rather than declaring the constraint.

What it looks like: Output is consistently almost right in one dimension but can't get all dimensions right simultaneously. Fixing one thing breaks another. There's a quality ceiling you can feel but can't explain.

What to do: Ask: "Are there limitations in the tool you're using that prevent this?" If that doesn't surface it, evaluate whether the tool is right for the job. Sometimes the answer is a different tool or approach entirely.

Collateral Mutation

What it is: You asked for a specific, surgical change. The AI delivered it plus unrequested modifications: reorganized imports, removed comments, renamed variables, changed formatting.

What it looks like: Your diff shows changes in places you didn't ask about. The requested change is correct, but it's accompanied by unauthorized modifications.

```
// You asked: add a validateEmail() method.
// Your diff also shows:

- import java.util.Objects;           // removed
+ import java.util.*;                 // widened silently
- // Repository injected at construction time.
- // See ObjectFactory for wiring.
+                                     // comment deleted
```

What to do: Always diff, never just read. State constraints explicitly. Reject output with collateral changes and ask again.

Confidence Poisoning

What it is: Accumulated trust from a series of good outputs causes your verification discipline to relax. Quality can shift without warning, but your review rigor has decreased proportionally to how well things have been going.

What it looks like: You're scanning instead of reading. Approving faster than at the start. Review feels like a formality.

What to do: Build verification into your process, not your judgment. Use automated checks. The longer a session goes well, the more carefully you should review, because you are more likely to miss errors, not less.

The Phantom Agreement

What it is: You present an idea the AI should push back on. Instead, it agrees enthusiastically and builds on your mistake.

What it looks like: No concerns raised. No alternatives offered. The AI validates your direction without evaluation. It felt like confirmation. It was a conversational pattern.

What to do: When uncertain, don't ask the AI to confirm. Ask it to attack: "What are the three strongest arguments against this design?" For real architectural uncertainty, talk to a human colleague.

Test Laundering

What it is: The AI generates both the implementation and the tests in the same session, or with knowledge of the implementation. The tests verify what the code does, not what it should do. A bug in the implementation is replicated in the tests. Green tests do not mean correct behavior. They mean the implementation is self-consistent.

What it looks like: Full coverage, all tests passing, code review clean. But the tests were written against the AI's output rather than the specification. The wrong behavior is doubly invisible: the code looks right and the tests agree with it.

What to do: Provide the expected behavior before asking the AI to implement. For critical logic, write the tests first. When the AI writes both code and tests in the same step, treat the tests as unverified until checked against the actual requirements, not the implementation.

Where to Go from Here

You now have an accurate model of what AI is, how it fails, and why your existing engineering discipline is the most reliable defense against the mistakes it will make. The seven properties, the passive and active defense layers, and the failure mode catalog give you the vocabulary and the instincts to work with AI safely. That foundation does not expire with a model release or a context window increase.

For the engineering philosophy that underpins everything in this guide — separation of responsibilities, traceable execution, boundary enforcement, and honest error handling — read *A Practical Development Philosophy for Long-Lived Systems*.

For the actionable methodology that turns this understanding into a repeatable process — design first, plan in phases, execute in surgical steps, verify at every gate — read *The Confluent Method*.